AWS Data Wrangler

Release 2.10.0

Igor Tavares

Aug 11, 2021

CONTENTS

1 Read The Docs **3**
- 1.1 What is AWS Data Wrangler? . . . 3
- 1.2 Install . . . 3
- 1.3 Tutorials . . . 7
- 1.4 API Reference . . . 96

Index **263**

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

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

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

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

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

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

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

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

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

# Amazon Timestream Query
wr.timestream.query("""
SELECT time, measure_value::double, my_dimension
FROM "sampleDB"."sampleTable" ORDER BY time DESC LIMIT 3
""")
```

CHAPTER

ONE

READ THE DOCS

1.1 What is AWS Data Wrangler?

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

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

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

Check our tutorials or the list of functionalities.

1.2 Install

AWS Data Wrangler runs with Python `3.6`, `3.7`, `3.8` and `3.9` and on several platforms (AWS Lambda, AWS Glue Python Shell, EMR, EC2, on-premises, Amazon SageMaker, local, etc).

Some good practices for most of the methods below are:

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

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

1.2.1 PyPI (pip)

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

1.2.2 Conda

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

1.2.3 AWS Lambda Layer

1 - Go to GitHub's release section and download the layer zip related to the desired version.

2 - Go to the AWS Lambda Panel, open the layer section (left side) and click **create layer**.

3 - Set name and python version, upload your fresh downloaded zip file and press **create** to create the layer.

4 - Go to your Lambda and select your new layer!

1.2.4 AWS Glue Python Shell Jobs

1 - Go to GitHub's release page and download the wheel file (.whl) related to the desired version.

2 - Upload the wheel file to any Amazon S3 location.

3 - Go to your Glue Python Shell job and point to the wheel file on S3 in the *Python library path* field.

Official Glue Python Shell Reference

1.2.5 AWS Glue PySpark Jobs

Note: AWS Data Wrangler has compiled dependencies (C/C++) so there is only support for `Glue PySpark Jobs >= 2.0`.

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

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

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

- Value: `pyarrow==2,awswrangler==2.10.0`

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

Official Glue PySpark Reference

1.2.6 Public Artifacts

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

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

Here is an example of how to reference the Lambda layer in your CDK app:

```
wrangler_layer = LayerVersion(
  self,
  "wrangler-layer",
  compatible_runtimes=[Runtime.PYTHON_3_8],
  code=S3Code(
    bucket=Bucket.from_bucket_arn(
        self,
        "wrangler-bucket",
        bucket_arn="arn:aws:s3:::aws-data-wrangler-public-artifacts",
    ),
    key="releases/2.10.0/awswrangler-layer-2.10.0-py3.8.zip",
  ),
  layer_version_name="aws-data-wrangler"
)
```

1.2.7 Amazon SageMaker Notebook

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

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

1.2.8 Amazon SageMaker Notebook Lifecycle

Open SageMaker console, go to the lifecycle section and use the follow snippet to configure AWS Data Wrangler 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.
```

(continues on next page)

(continued from previous page)

```
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
```

1.2.9 EMR Cluster

Even not being a distributed library, AWS Data Wrangler could be a good helper 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": [
       {
         "Classification": "export",
         "Properties": {
            "PYSPARK_PYTHON": "/usr/bin/python3"
          }
       }
    ]
  }
]
```

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

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

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

  - For EMR Release >= 6

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

sudo pip install pyarrow==2 awswrangler
```

Note: Make sure to freeze the Wrangler version in the bootstrap for productive environments (e.g. awswrangler==2.10.0)

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

1.2.10 From Source

```
>>> git clone https://github.com/awslabs/aws-data-wrangler.git
>>> cd aws-data-wrangler
>>> pip install .
```

1.2.11 Notes for Microsoft SQL Server

`awswrangler` is using the pyodbc for interacting with Microsoft SQL Server. For installing this package you need the ODBC header files, which can be installed, for example, 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.

1.3 Tutorials

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

1.3.1 1 - Introduction

What is AWS Data Wrangler?

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?

The Wrangler runs almost anywhere over Python 3.6, 3.7, 3.8 and 3.9, so there are several different ways to install it in the desired enviroment.

- 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'
```

1.3.2 2 - Sessions

How Wrangler handle Sessions and AWS credentials?

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

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

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

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

Using the default Boto3 Session

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

[2]:
```
False
```

Customizing and using the default Boto3 Session

```
[3]: boto3.setup_default_session(region_name="us-east-2")

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

Using a new custom Boto3 Session

```
[4]: my_session = boto3.Session(region_name="us-east-2")

     wr.s3.does_object_exist("s3://noaa-ghcn-pds/fake", boto3_session=my_session)
[4]: False
```

1.3.3 3 - Amazon S3

Table of Contents

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

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

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

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

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

Enter your bucket name:

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

1. CSV files

1.1 Writing CSV files

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

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

1.2 Reading single CSV file

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

[4]:

	id	name
0	1	foo
1	2	boo

1.3 Reading multiple CSV files

1.3.1 Reading CSV by list

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

[5]:

	id	name
0	1	foo
1	2	boo
2	3	bar

1.3.2 Reading CSV by prefix

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

[6]:

	id	name
0	1	foo
1	2	boo
2	3	bar

2. JSON files

2.1 Writing JSON files

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

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

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

2.2 Reading single JSON file

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

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

2.3 Reading multiple JSON files

2.3.1 Reading JSON by list

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

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

2.3.2 Reading JSON by prefix

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

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

3. Parquet files

For more complex features 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\n"\
          "2    Lambusart 14-06-18\n"\
          "3 Spormaggiore 15-04-18"
boto3.client("s3").put_object(Body=content, Bucket=bucket, Key="fwf/file1.txt")

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

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

4.1 Reading single FWF file

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

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

4.2 Reading multiple FWF files

4.2.1 Reading FWF by list

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

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

4.2.2 Reading FWF by prefix

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

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

5. Excel files

5.1 Writing Excel file

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

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

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

5.2 Reading Excel file

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

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

6. Reading with lastModified filter

Specify the filter by LastModified Date.

The filter needs to be specified as 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}/")
```

1.3.4 4 - Parquet Datasets

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

- **append** (Default)

  Only adds new files without any delete.

- **overwrite**

  Deletes everything in the target directory and then add new files.

- **overwrite_partitions** (Partition Upsert)

  Only deletes the paths of partitions that should be updated and then writes the new partitions files. It's like a "partition Upsert".

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

Enter your bucket name:

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

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

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 Partitoned Dataset

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

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

(continues on next page)

(continued from previous page)

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

[6]:
```
   id value        date
0   1   foo  2020-01-01
1   2   boo  2020-01-02
```

Upserting partitions (overwrite_partitions)

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

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

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

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

BONUS - Glue/Athena integration

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

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

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

[8]:
```
   id value        date
0   1   foo  2020-01-01
1   2   boo  2020-01-02
```

1.3.5 5 - Glue Catalog

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

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

Enter your bucket name:

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

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

Creating a Pandas DataFrame

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

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

Checking Glue Catalog Databases

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

```
            Database                                    Description
0  aws_data_wrangler  AWS Data Wrangler 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_data_wrangler  AWS Data Wrangler 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)

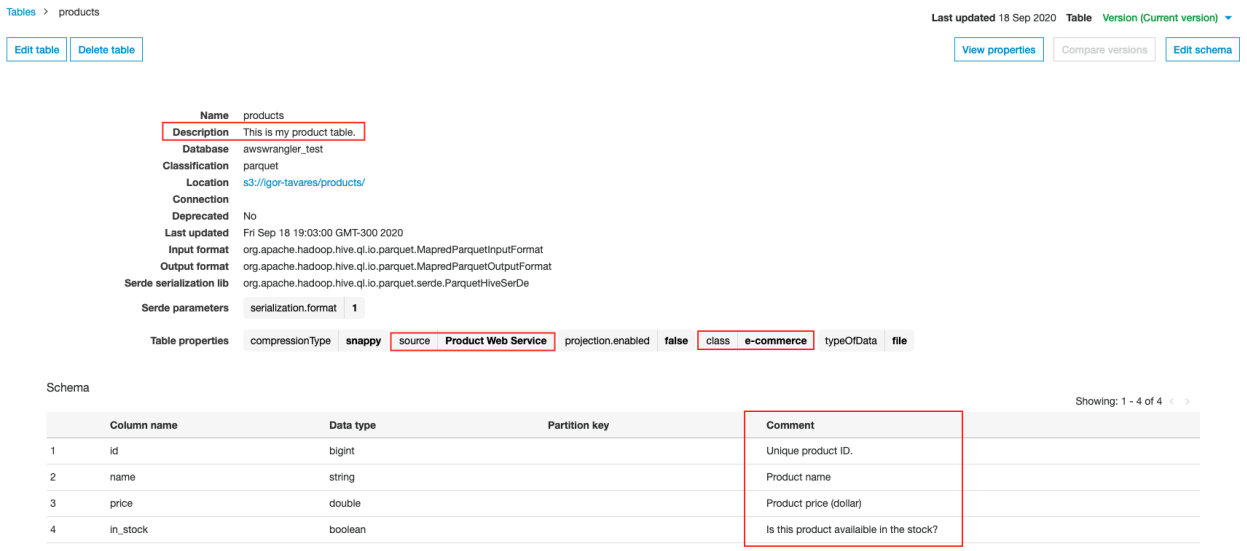
[7]:
```
desc = "This is my product table."

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/",
```

(continues on next page)

(continued from previous page)

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

Checking Glue Catalog (AWS Console)



Looking Up for the new table!

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

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

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

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

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

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

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

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

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

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

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

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

Getting tables details

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

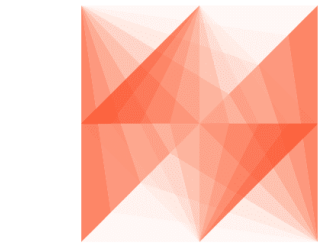
[12]:
```
  Column Name     Type  Partition                                     Comment
0          id   bigint      False                        Unique product ID.
1        name   string      False                             Product name
2       price   double      False                    Product price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



1.3.6 6 - Amazon Athena

Wrangler has two 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.

- **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

[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/189",
    names=cols,
    parse_dates=["dt", "obs_time"])  # Read 10 files from the 1890 decade (~1GB)

df
```

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

[29240019 rows x 8 columns]
```

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

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

[6]:
```
  Column Name       Type  Partition Comment
0          id     string      False
1          dt  timestamp      False
2     element     string      False
3       value     bigint      False
4      m_flag     string      False
5      q_flag     string      False
```

(continues on next page)

(continued from previous page)

```
6      s_flag     string      False
7    obs_time     string      False
```

Reading with ctas_approach=False

[7]:
```
%%time

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

```
CPU times: user 8min 45s, sys: 6.52 s, total: 8min 51s
Wall time: 11min 3s
```

[7]:

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

[29240019 rows x 8 columns]

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

[8]:
```
%%time

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

```
CPU times: user 28 s, sys: 6.07 s, total: 34.1 s
Wall time: 50.5 s
```

[8]:

	id	dt	element	value	m_flag	q_flag	s_flag	obs_time
0	ASN00017088	1890-06-11	PRCP	0	<NA>	<NA>	a	<NA>
1	ASN00017087	1890-06-11	PRCP	0	<NA>	<NA>	a	<NA>
2	ASN00017089	1890-06-11	PRCP	71	<NA>	<NA>	a	<NA>
3	ASN00017095	1890-06-11	PRCP	0	<NA>	<NA>	a	<NA>
4	ASN00017094	1890-06-11	PRCP	0	<NA>	<NA>	a	<NA>
...	...	...	...	...	...	...	...	...
29240014	USC00461260	1899-12-31	SNOW	0	<NA>	<NA>	6	<NA>
29240015	USC00461515	1899-12-31	TMAX	-89	<NA>	<NA>	6	<NA>
29240016	USC00461515	1899-12-31	TMIN	-189	<NA>	<NA>	6	<NA>
29240017	USC00461515	1899-12-31	PRCP	0	<NA>	<NA>	6	<NA>
29240018	USC00461515	1899-12-31	SNOW	0	<NA>	<NA>	6	<NA>

[29240019 rows x 8 columns]

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

[9]:
```
%%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"])
```

```
CPU times: user 6.89 s, sys: 2.27 s, total: 9.16 s
Wall time: 27.3 s
```

[9]:

	id	dt	element	value	m_flag	q_flag	s_flag	obs_time
0	GME00102348	1890-08-03	TMAX	172	NaN	NaN	E	NaN
1	GME00102348	1890-08-03	TMIN	117	NaN	NaN	E	NaN
2	GME00102348	1890-08-03	PRCP	63	NaN	NaN	E	NaN
3	GME00102348	1890-08-03	SNWD	0	NaN	NaN	E	NaN
4	GME00121126	1890-08-03	PRCP	32	NaN	NaN	E	NaN
...	...	...	...	...	...	...	...	...
29240014	USC00461260	1899-12-31	SNOW	0	NaN	NaN	6	NaN
29240015	USC00461515	1899-12-31	TMAX	-89	NaN	NaN	6	NaN
29240016	USC00461515	1899-12-31	TMIN	-189	NaN	NaN	6	NaN
29240017	USC00461515	1899-12-31	PRCP	0	NaN	NaN	6	NaN
29240018	USC00461515	1899-12-31	SNOW	0	NaN	NaN	6	NaN

[29240019 rows x 8 columns]

Batching (Good for restricted memory environments)

[10]:
```
%%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))
```

```
1024
8086528
1024
1024
1024
1024
1024
15360
1024
10090496
2153472
8886995
CPU times: user 22.7 s, sys: 5.41 s, total: 28.1 s
Wall time: 48 s
```

[11]:
```
%%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))
```

```
29240019
CPU times: user 34.8 s, sys: 8.54 s, total: 43.4 s
Wall time: 1min 1s
```

Cleaning Up S3

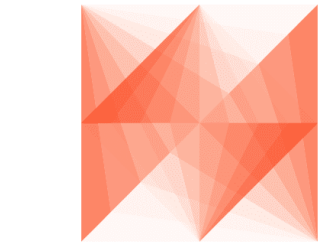
[12]:
```
wr.s3.delete_objects(path)
```

Delete table

[13]:
```
wr.catalog.delete_table_if_exists(database="awswrangler_test", table="noaa");
```

Delete Database

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



AWS Data Wrangler

1.3.7 7 - Redshift, MySQL, PostgreSQL and SQL Server

Wrangler's Redshift, MySQL and PostgreSQL have two basic function in common that tries to follow the 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()

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

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()

[2]:
```
con_redshift = wr.redshift.connect("aws-data-wrangler-redshift")
con_mysql = wr.mysql.connect("aws-data-wrangler-mysql")
con_postgresql = wr.postgresql.connect("aws-data-wrangler-postgresql")
con_sqlserver = wr.sqlserver.connect("aws-data-wrangler-sqlserver")
```

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")
```

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)
```

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

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

1.3.8 8 - Redshift - COPY & UNLOAD

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

1 - COPY

2 - UNLOAD

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

[1]:
```
import awswrangler as wr

con = wr.redshift.connect("aws-data-wrangler-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/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()
```

1.3.9 9 - Redshift - Append, Overwrite and Upsert

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

1 - `append`

2 - `overwrite`

3 - `upsert`

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

con = wr.redshift.connect("aws-data-wrangler-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

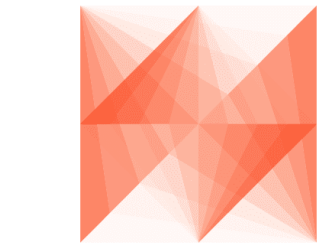
[12]:
```
df = pd.DataFrame({
    "id": [2, 3],
    "value": ["xoo", "bar"],
    "date": [date(2020, 1, 2), date(2020, 1, 3)]
})

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 Data Wrangler

1.3.10 10 - Parquet Crawler

Wrangler 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/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

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"])
```

1.3.11 11 - CSV Datasets

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

- **append** (Default)

  Only adds new files without any delete.

- **overwrite**

  Deletes everything in the target directory and then add new files.

- **overwrite_partitions** (Partition Upsert)

  Only deletes the paths of partitions that should be updated and then writes the new partitions files. It's like a "partition Upsert".

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

Enter your bucket name:

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

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

Checking/Creating Glue Catalog Databases

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

Creating the Dataset

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

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

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

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

Appending

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

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

(continues on next page)

(continued from previous page)

```
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 Partitoned 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_data_wrangler",
    table="my_table",
    compression="gzip"
)

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

[9]:
```
   id value        date
0   1   foo  2020-01-01
1   2   boo  2020-01-02
```

1.3.12 12 - CSV Crawler

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

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

Enter your bucket name:

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

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

Creating a Pandas DataFrame

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

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

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
```

(continues on next page)

(continued from previous page)

```
6        par0     bigint       True
7        par1     string       True
```

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

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

df
```

[9]:
```
   id string  float       date   timestamp  bool  par0 par1
0   1      1      1 2020-01-01  2020-01-02     1     1    a
```

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

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

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

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

Checking out

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

df
```

[13]:
```
   id string  float  date  timestamp  bool  par0 par1
0   1      1    1.0  None 2020-01-02  True     1    a
1   1      1    1.0  None 2020-01-02  True     1    a
```

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

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

Cleaning Up S3

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

Cleaning Up the Database

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

[16]:
```
True
```

1.3.13 13 - Merging Datasets on S3

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

- **append** (Default)

  Only adds new files without any delete.

- **overwrite**

  Deletes everything in the target directory and then add new files.

- **overwrite_partitions** (Partition Upsert)

  Only deletes the paths of partitions that should be updated and then writes the new partitions files. It's like a "partition Upsert".

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

Enter your bucket name:

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

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

Creating Dataset 1

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

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

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

[3]:

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

Creating Dataset 2

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Cleaning Up

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

1.3.14 14 - Schema Evolution

Wrangler support new **columns** on Parquet Dataset through:

- wr.s3.to_parquet()
- wr.s3.store_parquet_metadata() i.e. "Crawler"

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

Enter your bucket name:

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

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

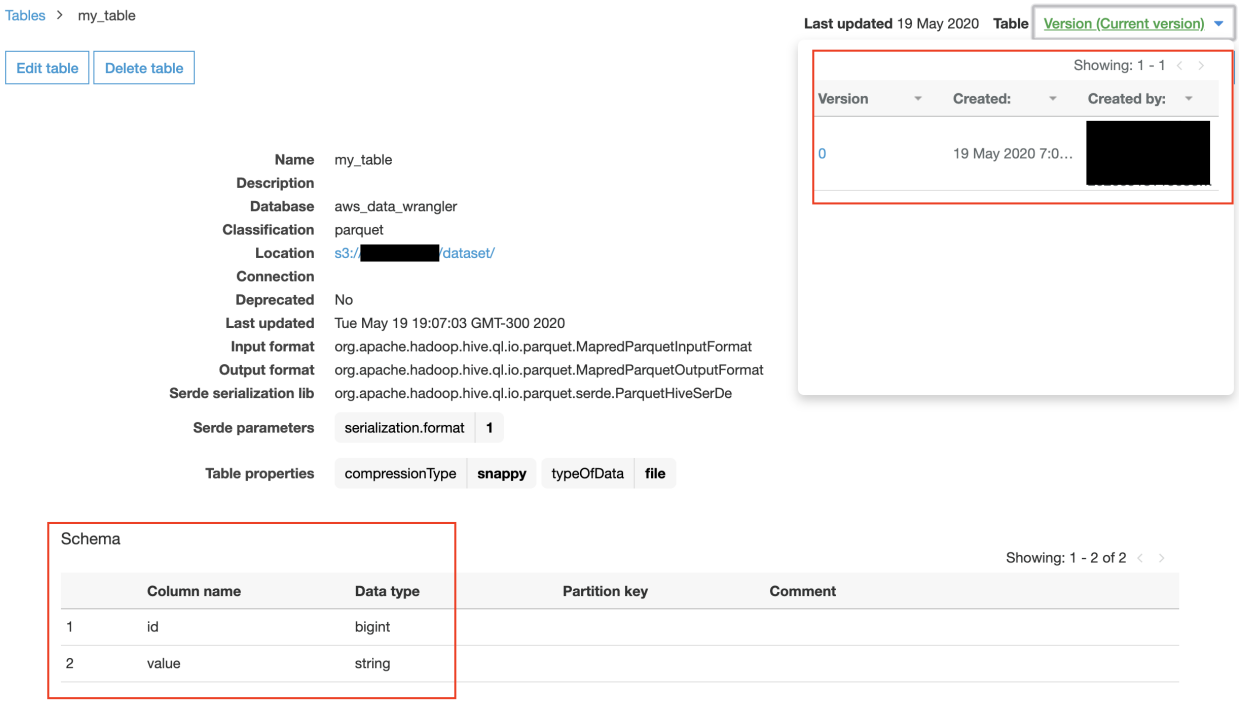
Creating the Dataset

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

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

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

[3]:

	id	value
0	1	foo
1	2	boo

Schema Version 0 on Glue Catalog (AWS Console)

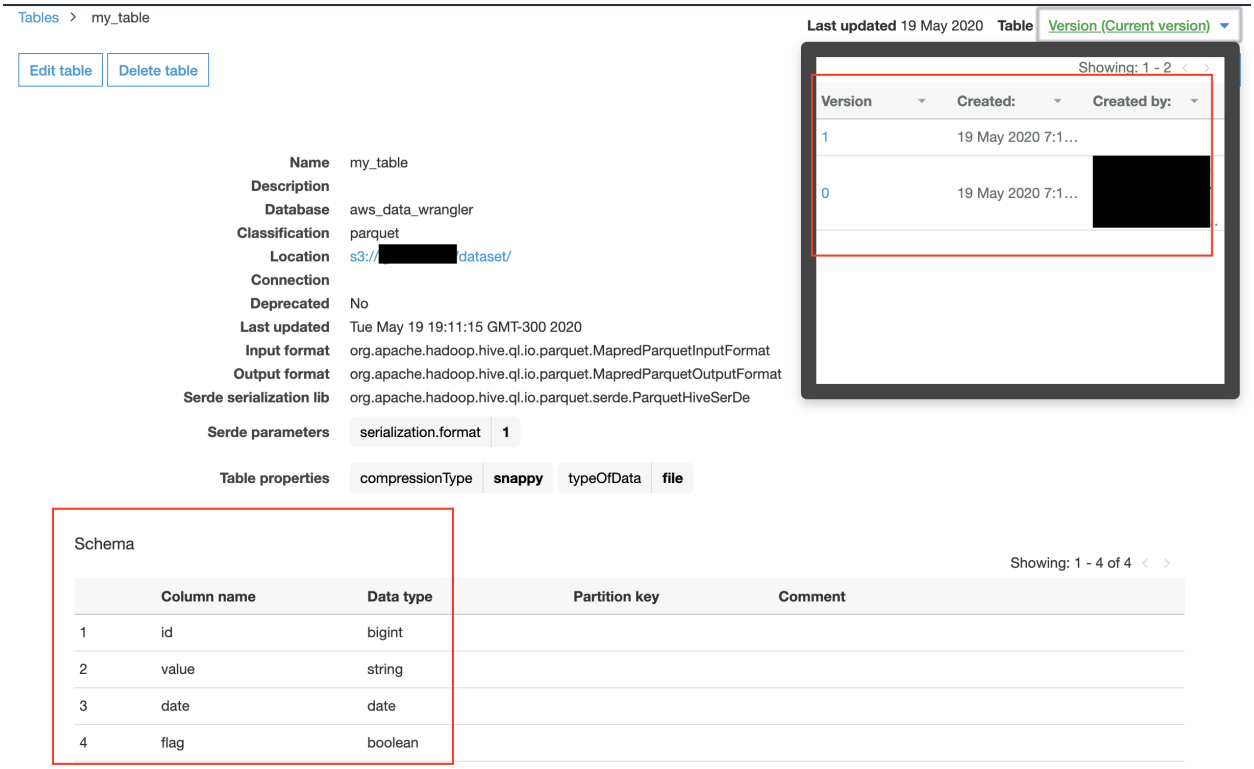


Appending with NEW COLUMNS

[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_data_wrangler",
    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
2   1   foo         NaN    NaN
3   2   boo         NaN    NaN
```

Schema Version 1 on Glue Catalog (AWS Console)



Reading from Athena

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

```
[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_data_wrangler")
```

[6]:
```
True
```

1.3.15 15 - EMR

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

Enter your bucket name:

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

```
 ·····································
```

Enter your Subnet ID:

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

```
 ·····················
```

Creating EMR Cluster

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

Uploading our PySpark script to Amazon S3

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

print("Spark Initialized")
"""

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

Submit PySpark step

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

Wait Step

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

Terminate Cluster

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

[ ]:
```

```

1.3.16 16 - EMR & Docker

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

Enter your bucket name:

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

Enter your Subnet ID:

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

Build and Upload Docker Image to ECR repository

Replace the {ACCOUNT_ID} placeholder.

[ ]:
```
%%writefile Dockerfile

FROM amazoncorretto:8

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

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

RUN python -V
RUN python3 -V

ENV PYSPARK_DRIVER_PYTHON python3
ENV PYSPARK_PYTHON python3

RUN pip3 install --upgrade pip
RUN pip3 install awswrangler

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

[ ]:
```
%%bash

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

(continues on next page)

(continued from previous page)

```
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"Wrangler 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-6.0.0",
    subnet_id=subnet,
    emr_ec2_role="EMR_EC2_DefaultRole",
    emr_role="EMR_DefaultRole",
    instance_type_master="m5.2xlarge",
    instance_type_core="m5.2xlarge",
    instance_ebs_size_master=50,
    instance_ebs_size_core=50,
    instance_num_on_demand_master=0,
    instance_num_on_demand_core=0,
    instance_num_spot_master=1,
    instance_num_spot_core=2,
    spot_bid_percentage_of_on_demand_master=100,
    spot_bid_percentage_of_on_demand_core=100,
    spot_provisioning_timeout_master=5,
    spot_provisioning_timeout_core=5,
    spot_timeout_to_on_demand_master=False,
    spot_timeout_to_on_demand_core=False,
    python3=True,
    docker=True,
    spark_glue_catalog=True,
    hive_glue_catalog=True,
    presto_glue_catalog=True,
    debugging=True,
    applications=["Hadoop", "Spark", "Hive", "Zeppelin", "Livy"],
    visible_to_all_users=True,
    maximize_resource_allocation=True,
    keep_cluster_alive_when_no_steps=True,
    termination_protected=False,
    spark_pyarrow=True
)

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

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

(continues on next page)

(continued from previous page)

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

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

wr.emr.terminate_cluster(cluster_id)
```

[ ]:
```
```

1.3.17 17 - Partition Projection

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

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

Enter your bucket name:

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

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

Integer projection

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

df
```

[3]:

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

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

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

[5]:

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

Enum projection

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

df
```

[6]:
```
   value       city
0      1  São Paulo
1      2      Tokio
2      3    Seattle
```

[7]:
```
wr.s3.to_parquet(
    df=df,
    path=f"s3://{bucket}/table_enum/",
    dataset=True,
    partition_cols=["city"],
    database="default",
    table="table_enum",
    projection_enabled=True,
    projection_types={
        "city": "enum",
    },
    projection_values={
        "city": "São Paulo,Tokio,Seattle"
    },
);
```

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

[8]:
```
   value       city
0      1  São Paulo
1      3    Seattle
2      2      Tokio
```

Date projection

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

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

df
```

[9]:
```
   value          dt                  ts
0      1  2020-01-01 2020-01-01 00:00:00
1      2  2020-01-02 2020-01-01 00:00:01
2      3  2020-01-03 2020-01-01 00:00:02
```

[10]:
```
wr.s3.to_parquet(
    df=df,
    path=f"s3://{bucket}/table_date/",
    dataset=True,
    partition_cols=["dt", "ts"],
    database="default",
    table="table_date",
    projection_enabled=True,
    projection_types={
        "dt": "date",
        "ts": "date",
    },
    projection_ranges={
        "dt": "2020-01-01,2020-01-03",
        "ts": "2020-01-01 00:00:00,2020-01-01 00:00:02"
    },
);
```

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

[11]:
```
   value          dt                  ts
0      1  2020-01-01 2020-01-01 00:00:00
1      2  2020-01-02 2020-01-01 00:00:01
2      3  2020-01-03 2020-01-01 00:00:02
```

Injected projection

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

df
```

[12]:
```
   value                                  uuid
0      1  761e2488-a078-11ea-bb37-0242ac130002
1      2  b89ed095-8179-4635-9537-88592c0f6bc3
2      3  87adc586-ce88-4f0a-b1c8-bf8e00d32249
```

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

(continues on next page)

(continued from previous page)

```
    table="table_injected",
    projection_enabled=True,
    projection_types={
        "uuid": "injected",
    }
);
```

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

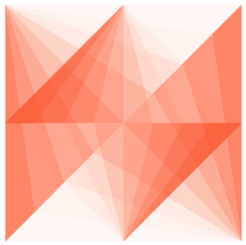
[14]:
```
   value                                  uuid
0      2  b89ed095-8179-4635-9537-88592c0f6bc3
```

Cleaning Up

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

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

[ ]:
```

```



AWS Data Wrangler

1.3.18 18 - QuickSight

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

References:

- A public data lake for analysis of COVID-19 data
- Exploring the public AWS COVID-19 data lake
- CloudFormation template

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

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

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

List users of QuickSight account

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

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

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

[3]:
```
            Database                                        Description
0  aws_data_wrangler  AWS Data Wrangler 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
```

(continues on next page)

(continued from previous page)

```
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 ...
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, ...
```

(continues on next page)

(continued from previous page)

```
16  uid, fips, iso2, iso3, code3, admin2, latitude...
17  objectid, hospital_name, hospital_type, hq_add...
18           date, county, state, fips, cases, deaths
19                    date, state, fips, cases, deaths
20  countyfips, countyname, statename, severity_co...
21  severity_1-day, severity_2-day, severity_3-day...
22  country_short_name, country_alpha_3_code, coun...
23  case_type, cases, difference, date, country_re...
24                                 state, abbreviation
25  iso_code, location, date, total_cases, new_cas...
```

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

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

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

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

                                        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,
```

(continues on next page)

(continued from previous page)

```
  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
        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
```

(continues on next page)

(continued from previous page)

```
    )
  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
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",
```

(continues on next page)

(continued from previous page)

```
    sql=sql,
    sql_name='CustomSQL',
    data_source_name="covid-19",
    import_mode='SPICE',
    allowed_to_manage=["dev"]
)
```

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

Wait ingestion

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

Describe last ingestion

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

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

List all ingestions

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

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

Create new dataset from a table directly

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

Cleaning up

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

1.3.19 19 - Amazon Athena Cache

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

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

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

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

[1]:
```
import 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

[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/189",
    names=cols,
    parse_dates=["dt", "obs_time"])  # Read 10 files from the 1890 decade (~1GB)

df
```

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

[29240019 rows x 8 columns]
```

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

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

[6]:
```
  Column Name       Type  Partition Comment
0          id     string      False
1          dt  timestamp      False
2     element     string      False
3       value     bigint      False
4      m_flag     string      False
5      q_flag     string      False
```

(continues on next page)

(continued from previous page)

```
6      s_flag     string      False
7    obs_time     string      False
```

The test query

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

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

First execution...

[8]:
```
%%time

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

```
CPU times: user 5.31 s, sys: 232 ms, total: 5.54 s
Wall time: 6min 42s
```

[8]:

	element	cnt
0	WDMV	49755046
1	SNWD	5089486328
2	DATN	10817510
3	DAPR	102579666
4	MDTN	10817510
5	WT03	71184687
6	WT09	584412
7	TOBS	146984266
8	DASF	7764526
9	WT04	9648963
10	WT18	92635444
11	WT01	87526136
12	WT16	323354156
13	PRCP	71238907298
14	SNOW	21950890838
15	WT06	307339
16	TAVG	2340863803
17	TMIN	41450979633

(continues on next page)

(continued from previous page)

```
18    MDTX     11210687
19    WT07      4486872
20    WT10       137873
21    EVAP       970404
22    WT14      8073701
23    DATX     11210687
24    WT08     33933005
25    WT05      8211491
26    TMAX  39876132467
27    MDPR    114320989
28    WT11     22212890
29    DWPR     69005655
30    MDSF     12004843
```

Second execution with CACHE (400x faster)

[9]:
```
%%time

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

```
CPU times: user 493 ms, sys: 34.9 ms, total: 528 ms
Wall time: 975 ms
```

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

(continues on next page)

(continued from previous page)

```
27    MDPR    114320989
28    WT11     22212890
29    DWPR     69005655
30    MDSF     12004843
```

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

[10]:
```
%%time

wr.athena.read_sql_query(query, database="awswrangler_test", max_cache_seconds=900, max_
↪cache_query_inspections=500)
```

```
CPU times: user 504 ms, sys: 44 ms, total: 548 ms
Wall time: 1.19 s
```

[10]:

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

Cleaning Up S3

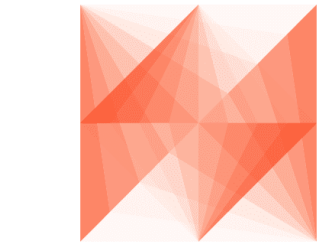
```
[11]: wr.s3.delete_objects(path)
```

Delete table

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

Delete Database

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



AWS Data Wrangler

1.3.20 20 - Spark Table Interoperability

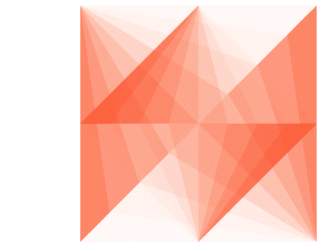
Wrangler has no difficults to insert, overwrite or do any other kind of interaction with a Table created by Apache Spark.

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

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

Or just move forward using the `insertInto` alternative:

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



AWS Data Wrangler

1.3.21 21 - Global Configurations

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

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

P.S. Check 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``. (seeBoto3 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 awswrangler as wr
import botocore
```

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

[3]:

	foo
0	1

Resetting

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

wr.config

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

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

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

Visualizing

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

[7]:
```
<awswrangler._config._Config at 0x2d3a2c63df0>
```

1.3.22 22 - Writing Partitions Concurrently

- concurrent_partitioning argument:

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

[ ]:

1.3.23 23 - Flexible Partitions Filter (PUSH-DOWN)

- `partition_filter` argument:

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

*P.S. Check the*function API doc*to see it has some argument that can be configured through Global configurations.*

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

Enter your bucket name:

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

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

Creating the Dataset (PARQUET)

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

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

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

[3]:

	id	value
0	3	bar
1	2	boo
2	1	foo

Example 1

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

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

[4]:

	id	value
0	2	boo
1	1	foo

Example 2

[5]:
```
from Levenshtein import distance


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


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

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

Creating the Dataset (CSV)

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

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

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

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

Example 1

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

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

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

Example 2

[8]:
```
from Levenshtein import distance


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


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

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

1.3.24 24 - Athena Query Metadata

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

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

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

Environment Variables

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

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

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

df
```

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

Getting statistics from query metadata

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

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

1.3.25 25 - Redshift - Loading Parquet files with Spectrum

Enter your bucket name:

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

```
 ··········
```

Mocking some Parquet Files on S3

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

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

(continues on next page)

(continued from previous page)

```
df
```

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

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

Crawling the metadata and adding into Glue Catalog

[4]:
```
wr.s3.store_parquet_metadata(
    path=PATH,
    database="aws_data_wrangler",
    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-data-wrangler-redshift")
```

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

1.3.26 26 - Amazon Timestream

Creating resources

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

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

Write

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

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

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

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

Query

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

[12]:

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

Deleting resources

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

1.3.27 27 - Amazon Timestream - Example 2

Reading test data

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

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

df
```

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

(continues on next page)

(continued from previous page)

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

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

[126000 rows x 7 columns]
```

Creating resources

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

Write CPU_UTILIZATION records

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

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

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

(continues on next page)

(continued from previous page)

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

[63000 rows x 7 columns]
```

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

assert len(rejected_records) == 0
```

Write MEMORY_UTILIZATION records

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

df_memory
```

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

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

(continues on next page)

(continued from previous page)

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

[63000 rows x 7 columns]
```

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

assert len(rejected_records) == 0
```

Querying CPU_UTILIZATION

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

[7]:
```
     hostname      region           az     measure_name  \
0  host-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
9  host-8EBHm   us-west-2   us-west-2c  cpu_utilization

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

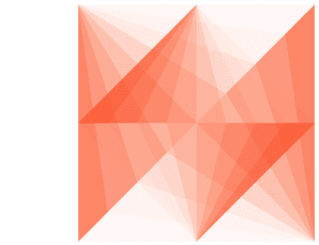
Querying MEMORY_UTILIZATION

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

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

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

Deleting resources

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



AWS Data Wrangler

1.3.28 28 - Amazon DynamoDB

Writing Data

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

Writing DataFrame

[2]:
```
df = pd.DataFrame({
    "key": [1, 2],
    "value": ["foo", "boo"]
})
wr.dynamodb.put_df(df=df, table_name="table")
```

Writing CSV file

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

Writing list of items

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

Deleting items

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

1.3.29 29 - S3 Select

AWS Data Wrangler 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 Wrangler operations such as `read_parquet` where the S3 object is downloaded and filtered on the client-side.

This feature has a number of limitations however, and should be used for specific scenarios only: * It operates on a single S3 object * The maximum length of a record in the input or result is 1 MB * The maximum uncompressed row group size is 256 MB (Parquet only) * It can only emit nested data in JSON format * Certain SQL operations are not supported (e.g. ORDER BY)

Read full CSV file

[1]:
```
import awswrangler as wr

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

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

  DOLocationID SR_Flag
0          264
```

(continues on next page)

(continued from previous page)

```
1          264
2          264
3          264
4          264
```

Filter JSON file

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

[2]:
```
  family_name                                contact_details               name  \
0       Biden  [{'type': 'twitter', 'value': 'joebiden'}]  Joseph Biden, Jr.

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

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

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

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

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

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

Read Snappy compressed Parquet

[3]:
```
df = wr.s3.select_query(
        sql="SELECT * FROM s3object s where s.\"star_rating\" >= 5",
        path="s3://amazon-reviews-pds/parquet/product_category=Gift_Card/part-00000-
↪495c48e6-96d6-4650-aa65-3c36a3516ddd.c000.snappy.parquet",
        input_serialization="Parquet",
        input_serialization_params={},
        use_threads=True,
)
df.loc[:, df.columns != "product_title"].head()
```

[3]:
```
  marketplace customer_id       review_id  product_id product_parent  \
0           US    52670295   RGPOFKORD8RTU  B0002CZPPG      867256265
1           US    29964102  R2U8X8V5KPB4J3  B00H5BMF00      373287760
2           US    25173351  R15XV3LXUMLTXL  B00PG40CO4      137115061
3           US    12516181  R3G6G7H8TX4H0T  B0002CZPPG      867256265
4           US    38355314  R2NJ7WNBU16YTQ  B00B2TFSO6       89375983

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

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

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

1.4 API Reference

- *Amazon S3*
- *AWS Glue Catalog*
- *Amazon Athena*
- *Amazon Redshift*
- *PostgreSQL*
- *MySQL*
- *Microsoft SQL Server*
- *Data API Redshift*
- *Data API RDS*
- *DynamoDB*
- *Amazon Timestream*
- *Amazon EMR*
- *Amazon CloudWatch Logs*
- *Amazon QuickSight*

- *AWS STS*
- *AWS Secrets Manager*
- *Global Configurations*

1.4.1 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 from a received S3 path to local file.
get_bucket_region(bucket[, boto3_session])	Get bucket region name.
list_directories(path[, ...])	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.
merge_upsert_table(delta_df, database, ...)	Perform Upsert (Update else Insert) onto an existing Glue table.
read_csv(path[, path_suffix, ...])	Read CSV file(s) from from a received S3 prefix or list of S3 objects paths.
read_excel(path[, version_id, use_threads, ...])	Read EXCEL file(s) from from a received S3 path.
read_fwf(path[, path_suffix, ...])	Read fixed-width formatted file(s) from from a received S3 prefix or list of S3 objects paths.
read_json(path[, path_suffix, ...])	Read JSON file(s) from from a received S3 prefix or list of S3 objects paths.
read_parquet(path[, path_suffix, ...])	Read Apache Parquet file(s) from from a received S3 prefix or list of S3 objects paths.
read_parquet_metadata(path[, version_id, ...])	Read Apache Parquet file(s) metadata from from a received S3 prefix or list of S3 objects paths.
read_parquet_table(table, database[, ...])	Read Apache Parquet table registered on AWS Glue Catalog.
select_query(sql, path, input_serialization, ...)	Filter contents of an Amazon S3 object 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[, boto3_session, ...])	Write JSON file on Amazon S3.
to_parquet(df[, path, index, compression, ...])	Write Parquet file or dataset on Amazon S3.
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: Optional[Dict[str, str]] = None*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*) → List[str]

Copy a list of S3 objects to another S3 directory.

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

Parameters

- **paths** (*List[str]*) – List of S3 objects paths (e.g. [s3://bucket/dir0/key0, s3://bucket/dir0/key1]).
- **source_path** (*str,*) – S3 Path for the source directory.
- **target_path** (*str,*) – S3 Path for the target directory.
- **replace_filenames** (*Dict[str, str], optional*) – e.g. {"old_name.csv": "new_name.csv", "old_name2.csv": "new_name2.csv"}
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}

Returns List of new objects paths.

Return type List[str]

Examples

Copying

```
>>> import awswrangler as wr
>>> wr.s3.copy_objects(
...     paths=["s3://bucket0/dir0/key0", "s3://bucket0/dir0/key1"],
...     source_path="s3://bucket0/dir0/",
...     target_path="s3://bucket1/dir1/"
... )
["s3://bucket1/dir1/key0", "s3://bucket1/dir1/key1"]
```

Copying with a KMS key

```
>>> import awswrangler as wr
>>> wr.s3.copy_objects(
...     paths=["s3://bucket0/dir0/key0", "s3://bucket0/dir0/key1"],
...     source_path="s3://bucket0/dir0/",
...     target_path="s3://bucket1/dir1/",
```

(continues on next page)

(continued from previous page)

```
...     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: Union[str, List[str]]*, *use_threads: bool = True*, *last_modified_begin: Optional[datetime.datetime] = None*, *last_modified_end: Optional[datetime.datetime] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Delete Amazon S3 objects from a received S3 prefix or list of S3 objects paths.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (**, ?, []*), you can use *glob.escape(path)* before passing the path to this function.

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

Note: The filter by last_modified begin last_modified end is applied after list all S3 files

---

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.s3.delete_objects(['s3://bucket/key0', 's3://bucket/key1'])  # Delete both␣
↪objects
>>> wr.s3.delete_objects('s3://bucket/prefix')  # Delete all objects under the␣
↪received prefix
```

awswrangler.s3.describe_objects

awswrangler.s3.**describe_objects**(*path: Union[str, List[str]]*, *version_id: Optional[Union[str, Dict[str, str]]] = None*, *use_threads: bool = True*, *last_modified_begin: Optional[datetime.datetime] = None*, *last_modified_end: Optional[datetime.datetime] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Dict[str, Any]]

Describe Amazon S3 objects from a received S3 prefix or list of S3 objects paths.

Fetch attributes like ContentLength, DeleteMarker, last_modified, ContentType, etc The full list of attributes can be explored under the boto3 head_object documentation: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.head_object

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use *glob.escape(path)* before passing the path to this function.

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

Note: The filter by last_modified begin last_modified end is applied after list all S3 files

---

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. {'s3://bucket/key0': '121212', 's3://bucket/key1': '343434'})
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}

- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Return a dictionary of objects returned from head_objects where the key is the object path. The response object can be explored here: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.head_object

Return type Dict[str, Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> 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: Optional[Dict[str, Any]] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *version_id: Optional[str] = None*) → bool

Check if object exists on S3.

Parameters

- **path** (*str*) – S3 path (e.g. s3://bucket/key).
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **version_id** (*str, optional*) – Specific version of the object that should exist.

Returns True if exists, False otherwise.

Return type bool

Examples

Using the default boto3 session

```
>>> import awswrangler as wr
>>> wr.s3.does_object_exist('s3://bucket/key_real')
True
>>> wr.s3.does_object_exist('s3://bucket/key_unreal')
False
```

Using a custom boto3 session

```
>>> import boto3
>>> import awswrangler as wr
>>> wr.s3.does_object_exist('s3://bucket/key_real', boto3_session=boto3.Session())
```

(continues on next page)

(continued from previous page)

```
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: Union[str, Any]*, *version_id: Optional[str] = None*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*) → None

Download file from from a received S3 path to local file.

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

Parameters

- **path** (*str*) – S3 path (e.g. `s3://bucket/key0`).
- **local_file** (*Union[str, Any]*) – A file-like object in binary mode or a path to local file (e.g. `./local/path/to/key0`).
- **version_id** (*Optional[str]*) – Version id of the object.
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only "SSECustomerAlgorithm", "SSECustomerKey" and "RequestPayer" arguments will be considered.

Returns

Return type None

Examples

Downloading a file using a path to local file

```
>>> import awswrangler as wr
>>> wr.s3.download(path='s3://bucket/key', local_file='./key')
```

Downloading a file using a file-like object

```
>>> import awswrangler as wr
>>> with open(file='./key', mode='wb') as local_f:
>>>     wr.s3.download(path='s3://bucket/key', local_file=local_f)
```

awswrangler.s3.get_bucket_region

awswrangler.s3.**get_bucket_region**(*bucket: str*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Get bucket region name.

Parameters

- **bucket** (*str*) – Bucket name.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

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

Return type str

Examples

Using the default boto3 session

```
>>> import awswrangler as wr
>>> region = wr.s3.get_bucket_region('bucket-name')
```

Using a custom boto3 session

```
>>> import boto3
>>> import awswrangler as wr
>>> region = wr.s3.get_bucket_region('bucket-name', boto3_session=boto3.Session())
```

awswrangler.s3.list_directories

awswrangler.s3.**list_directories**(*path: str*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → 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).
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns List of objects paths.

Return type List[str]

Examples

Using the default boto3 session

```
>>> import awswrangler as wr
>>> wr.s3.list_directories('s3://bucket/prefix/')
['s3://bucket/prefix/dir0/', 's3://bucket/prefix/dir1/', 's3://bucket/prefix/dir2/']
```

Using a custom boto3 session

```
>>> import boto3
>>> import awswrangler as wr
>>> wr.s3.list_directories('s3://bucket/prefix/', boto3_session=boto3.Session())
['s3://bucket/prefix/dir0/', 's3://bucket/prefix/dir1/', 's3://bucket/prefix/dir2/']
```

awswrangler.s3.list_objects

awswrangler.s3.**list_objects**(*path: str*, *suffix: Optional[Union[str, List[str]]] = None*, *ignore_suffix: Optional[Union[str, List[str]]] = None*, *last_modified_begin: Optional[datetime.datetime] = None*, *last_modified_end: Optional[datetime.datetime] = None*, *ignore_empty: bool = False*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[str]

List Amazon S3 objects from a prefix.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use *glob.escape(path)* before passing the path to this function.

Note: The filter by last_modified begin last_modified end is applied after list all S3 files

Parameters

- **path** (*str*) – S3 path (e.g. s3://bucket/prefix).
- **suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for filtering S3 keys.
- **ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns List of objects paths.

Return type List[str]

Examples

Using the default boto3 session

```
>>> import awswrangler as wr
>>> wr.s3.list_objects('s3://bucket/prefix')
['s3://bucket/prefix0', 's3://bucket/prefix1', 's3://bucket/prefix2']
```

Using a custom boto3 session

```
>>> import boto3
>>> import awswrangler as wr
>>> wr.s3.list_objects('s3://bucket/prefix', boto3_session=boto3.Session())
['s3://bucket/prefix0', 's3://bucket/prefix1', 's3://bucket/prefix2']
```

awswrangler.s3.merge_datasets

awswrangler.s3.**merge_datasets**(*source_path: str*, *target_path: str*, *mode: str = 'append'*, *ignore_empty: bool = False*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*) → List[str]

Merge a source dataset into a target dataset.

This function accepts Unix shell-style wildcards in the source_path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use *glob.escape(source_path)* before passing the path to this function.

Note: If you are merging tables (S3 datasets + Glue Catalog metadata), remember that you will also need to update your partitions metadata in some cases. (e.g. wr.athena.repair_table(table='...', database='...'))

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

Parameters

- **source_path** (*str,*) – S3 Path for the source directory.
- **target_path** (*str,*) – S3 Path for the target directory.
- **mode** (*str, optional*) – append (Default), overwrite, overwrite_partitions.
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}

Returns List of new objects paths.

Return type List[str]

Examples

Merging

```
>>> import awswrangler as wr
>>> wr.s3.merge_datasets(
...     source_path="s3://bucket0/dir0/",
...     target_path="s3://bucket1/dir1/",
...     mode="append"
... )
["s3://bucket1/dir1/key0", "s3://bucket1/dir1/key1"]
```

Merging with a KMS key

```
>>> import awswrangler as wr
>>> wr.s3.merge_datasets(
...     source_path="s3://bucket0/dir0/",
...     target_path="s3://bucket1/dir1/",
...     mode="append",
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'
...     }
... )
["s3://bucket1/dir1/key0", "s3://bucket1/dir1/key1"]
```

awswrangler.s3.merge_upsert_table

awswrangler.s3.**merge_upsert_table**(*delta_df: pandas.core.frame.DataFrame*, *database: str*, *table: str*, *primary_key: List[str]*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Perform Upsert (Update else Insert) onto an existing Glue table.

Parameters

- **delta_df** (*pandas.DataFrame*) – The delta dataframe has all the data which needs to be merged on the primary key
- **database** (*Str*) – An existing database name
- **table** (*Str*) – An existing table name
- **primary_key** (*List[str]*) – Pass the primary key as a List of string columns List['column_a', 'column_b']
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Return type None

Examples

Reading all Parquet files under a prefix >>> import awswrangler as wr >>> import pandas as pd >>> delta_df = pd.DataFrame({"id": [1], "cchar": ["foo"], "date": [datetime.date(2021, 1, 2)]}) >>> primary_key = ["id", "cchar"] >>> wr.s3.merge_upsert_table(delta_df=delta_df, database='database', table='table', primary_key=primary_key)

awswrangler.s3.read_csv

awswrangler.s3.**read_csv**(*path: Union[str, List[str]]*, *path_suffix: Optional[Union[str, List[str]]] = None*, *path_ignore_suffix: Optional[Union[str, List[str]]] = None*, *version_id: Optional[Union[str, Dict[str, str]]] = None*, *ignore_empty: bool = True*, *use_threads: Union[bool, int] = True*, *last_modified_begin: Optional[datetime.datetime] = None*, *last_modified_end: Optional[datetime.datetime] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *chunksize: Optional[int] = None*, *dataset: bool = False*, *partition_filter: Optional[Callable[[Dict[str, str]], bool]] = None*, ***pandas_kwargs: Any*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Read CSV file(s) from from a received S3 prefix or list of S3 objects paths.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (**, ?, []*), you can use *glob.escape(path)* before passing the path to this function.

Note: For partial and gradual reading use the argument `chunksize` instead of `iterator`.

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

Note: The filter by last_modified begin last_modified end is applied after list all S3 files

---

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. [".csv"]). If None, will try to read all files. (default)
- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. ["_SUCCESS"]). If None, will try to read all files. (default)
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. {'s3://bucket/key0': '121212', 's3://bucket/key1': '343434'})

- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads. If given an int will use the given amount of threads.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **dataset** (*bool*) – If *True* read a CSV dataset instead of simple file(s) loading all the related partitions as columns.
- **partition_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (Dict[str, str]) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False to ignore it. Ignored if *dataset=False*. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False` https://aws-data-wrangler.readthedocs.io/en/2.10.0/tutorials/023%20-%20Flexible%20Partitions%20Filter.html
- **pandas_kwargs** – KEYWORD arguments forwarded to pandas.read_csv(). You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. wr.s3.read_csv('s3://bucket/prefix/', sep='|', na_values=['null', 'none'], skip_blank_lines=True) https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

Returns Pandas DataFrame or a Generator in case of *chunksize != None*.

Return type Union[pandas.DataFrame, Generator[pandas.DataFrame, None, None]]

Examples

Reading all CSV files under a prefix

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

Reading all CSV files under a prefix and using pandas_kwargs

```
>>> import awswrangler as wr
>>> df = wr.s3.read_csv('s3://bucket/prefix/', sep='|', na_values=['null', 'none'],␣
↪skip_blank_lines=True)
```

Reading all CSV files from a list

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

Reading in chunks of 100 lines

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_csv(path=['s3://bucket/filename0.csv', 's3://bucket/filename1.
↪csv'], chunksize=100)
>>> for df in dfs:
>>>     print(df)  # 100 lines Pandas DataFrame
```

Reading CSV Dataset with PUSH-DOWN filter over partitions

```
>>> import awswrangler as wr
>>> my_filter = lambda x: True if x["city"].startswith("new") else False
>>> df = wr.s3.read_csv(path, dataset=True, partition_filter=my_filter)
```

awswrangler.s3.read_excel

awswrangler.s3.**read_excel**(*path: str*, *version_id: Optional[str] = None*, *use_threads: Union[bool, int] = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, ***pandas_kwargs: Any*) → pandas.core.frame.DataFrame

Read EXCEL file(s) from from a received S3 path.

Note: This function accepts any Pandas's read_excel() argument. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

---

Note: Depending on the file extension ('xlsx', 'xls', 'odf'...), an additional library might have to be installed first (e.g. xlrd).

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

Parameters

- **path** (*str*) – S3 path (e.g. `s3://bucket/key.xlsx`).
- **version_id** (*Optional[str]*) – Version id of the object.
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads. If given an int will use the given amount of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.

- **pandas_kwargs** – KEYWORD arguments forwarded to pandas.read_excel(). You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. wr.s3.read_excel("s3://bucket/key.xlsx", na_rep="", verbose=True) https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

Returns Pandas DataFrame.

Return type pandas.DataFrame

Examples

Reading an EXCEL file

```
>>> import awswrangler as wr
>>> df = wr.s3.read_excel('s3://bucket/key.xlsx')
```

awswrangler.s3.read_fwf

awswrangler.s3.**read_fwf**(*path: Union[str, List[str]]*, *path_suffix: Optional[Union[str, List[str]]] = None*, *path_ignore_suffix: Optional[Union[str, List[str]]] = None*, *version_id: Optional[Union[str, Dict[str, str]]] = None*, *ignore_empty: bool = True*, *use_threads: Union[bool, int] = True*, *last_modified_begin: Optional[datetime.datetime] = None*, *last_modified_end: Optional[datetime.datetime] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *chunksize: Optional[int] = None*, *dataset: bool = False*, *partition_filter: Optional[Callable[[Dict[str, str]], bool]] = None*, ***pandas_kwargs: Any*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Read fixed-width formatted file(s) from from a received S3 prefix or list of S3 objects paths.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (**, ?, []*), you can use *glob.escape(path)* before passing the path to this function.

Note: For partial and gradual reading use the argument `chunksize` instead of `iterator`.

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

Note: The filter by last_modified begin last_modified end is applied after list all S3 files

---

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. [".txt"]). If None, will try to read all files. (default)

- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. ["_SUCCESS"]). If None, will try to read all files. (default)
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. {'s3://bucket/key0': '121212', 's3://bucket/key1': '343434'})
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads. If given an int will use the given amount of threads.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **dataset** (*bool*) – If *True* read a FWF dataset instead of simple file(s) loading all the related partitions as columns.
- **partition_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (Dict[str, str]) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False to ignore it. Ignored if *dataset=False*. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False` https://aws-data-wrangler.readthedocs.io/en/2.10.0/tutorials/023%20-%20Flexible%20Partitions%20Filter.html
- **pandas_kwargs** – KEYWORD arguments forwarded to pandas.read_fwf(). You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. wr.s3.read_fwf(path='s3://bucket/prefix/', widths=[1, 3], names=["c0", "c1"]) https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_fwf.html

Returns Pandas DataFrame or a Generator in case of *chunksize != None*.

Return type Union[pandas.DataFrame, Generator[pandas.DataFrame, None, None]]

Examples

Reading all fixed-width formatted (FWF) files under a prefix

```
>>> import awswrangler as wr
>>> df = wr.s3.read_fwf(path='s3://bucket/prefix/', widths=[1, 3], names=['c0', 'c1
↪'])
```

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: Union[str, List[str]]*, *path_suffix: Optional[Union[str, List[str]]] = None*, *path_ignore_suffix: Optional[Union[str, List[str]]] = None*, *version_id: Optional[Union[str, Dict[str, str]]] = None*, *ignore_empty: bool = True*, *orient: str = 'columns'*, *use_threads: Union[bool, int] = True*, *last_modified_begin: Optional[datetime.datetime] = None*, *last_modified_end: Optional[datetime.datetime] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *chunksize: Optional[int] = None*, *dataset: bool = False*, *partition_filter: Optional[Callable[[Dict[str, str]], bool]] = None*, ***pandas_kwargs: Any*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Read JSON file(s) from from a received S3 prefix or list of S3 objects paths.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use *glob.escape(path)* before passing the path to this function.

Note: For partial and gradual reading use the argument `chunksize` instead of `iterator`.

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

Note: The filter by last_modified begin last_modified end is applied after list all S3 files

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. [".json"]). If None, will try to read all files. (default)
- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. ["_SUCCESS"]). If None, will try to read all files. (default)
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. {'s3://bucket/key0': '121212', 's3://bucket/key1': '343434'})
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **orient** (*str*) – Same as Pandas: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads. If given an int will use the given amount of threads.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **dataset** (*bool*) – If *True* read a JSON dataset instead of simple file(s) loading all the related partitions as columns. If *True*, the *lines=True* will be assumed by default.
- **partition_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (Dict[str, str]) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False

to ignore it. Ignored if *dataset=False*. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False` https://aws-data-wrangler.readthedocs.io/en/2.10.0/tutorials/023%20-%20Flexible%20Partitions%20Filter.html

- **pandas_kwargs** – KEYWORD arguments forwarded to pandas.read_json(). You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. wr.s3.read_json('s3://bucket/prefix/', lines=True, keep_default_dates=True) https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html

Returns Pandas DataFrame or a Generator in case of *chunksize != None*.

Return type Union[pandas.DataFrame, Generator[pandas.DataFrame, None, None]]

Examples

Reading all JSON files under a prefix

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

Reading all CSV files under a prefix and using pandas_kwargs

```
>>> import awswrangler as wr
>>> df = wr.s3.read_json('s3://bucket/prefix/', lines=True, keep_default_dates=True)
```

Reading all JSON files from a list

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

Reading in chunks of 100 lines

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_json(path=['s3://bucket/0.json', 's3://bucket/1.json'],␣
↪chunksize=100, lines=True)
>>> for df in dfs:
>>>     print(df)  # 100 lines Pandas DataFrame
```

Reading JSON Dataset with PUSH-DOWN filter over partitions

```
>>> import awswrangler as wr
>>> my_filter = lambda x: True if x["city"].startswith("new") else False
>>> df = wr.s3.read_json(path, dataset=True, partition_filter=my_filter)
```

awswrangler.s3.read_parquet

awswrangler.s3.**read_parquet**(*path: Union[str, List[str]]*, *path_suffix: Optional[Union[str, List[str]]] = None*, *path_ignore_suffix: Optional[Union[str, List[str]]] = None*, *version_id: Optional[Union[str, Dict[str, str]]] = None*, *ignore_empty: bool = True*, *ignore_index: Optional[bool] = None*, *partition_filter: Optional[Callable[[Dict[str, str]], bool]] = None*, *columns: Optional[List[str]] = None*, *validate_schema: bool = False*, *chunked: Union[bool, int] = False*, *dataset: bool = False*, *categories: Optional[List[str]] = None*, *safe: bool = True*, *map_types: bool = True*, *use_threads: Union[bool, int] = True*, *last_modified_begin: Optional[datetime.datetime] = None*, *last_modified_end: Optional[datetime.datetime] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

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

The concept of Dataset goes beyond the simple idea of files and enable more complex features like partitioning and catalog integration (AWS Glue Catalog).

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (**, ?, []*), you can use *glob.escape(path)* before passing the path to this function.

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

Will anable the function to return a Iterable of DataFrames instead of a regular DataFrame.

There are two batching strategies on Wrangler:

- If **chunked=True**, a new DataFrame will be returned for each file in your path/dataset.
- If **chunked=INTEGER**, Wrangler will iterate on the data by number of rows igual the received INTEGER.

P.S. chunked=True if faster and uses less memory while *chunked=INTEGER* is more precise in number of rows for each Dataframe.

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

Note: The filter by last_modified begin last_modified end is applied after list all S3 files

---

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. [".gz.parquet", ".snappy.parquet"]). If None, will try to read all files. (default)
- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. [".csv", "_SUCCESS"]). If None, will try to read all files. (default)

- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. {'s3://bucket/key0': '121212', 's3://bucket/key1': '343434'})
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **ignore_index** (*Optional[bool]*) – Ignore index when combining multiple parquet files to one DataFrame.
- **partition_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (Dict[str, str]) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False to ignore it. Ignored if *dataset=False*. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False`
- **columns** (*List[str], optional*) – Names of columns to read from the file(s).
- **validate_schema** – Check that individual file schemas are all the same / compatible. Schemas within a folder prefix should all be the same. Disable if you have schemas that are different and want to disable this check.
- **chunked** (*Union[int, bool]*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* wrangler will iterate on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed Wrangler will iterate on the data by number of rows igual the received INTEGER.
- **dataset** (*bool*) – If *True* read a parquet dataset instead of simple file(s) loading all the related partitions as columns.
- **categories** (*Optional[List[str]], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **safe** (*bool, default True*) – For certain data types, a cast is needed in order to store the data in a pandas DataFrame or Series (e.g. timestamps are always stored as nanoseconds in pandas). This option controls whether it is a safe cast or not.
- **map_types** (*bool, default True*) – True to convert pyarrow DataTypes to pandas ExtensionDtypes. It is used to override the default pandas type for conversion of built-in pyarrow types or in absence of pandas_metadata in the Table schema.
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads. If given an int will use the given amount of threads.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.

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

Return type Union[pandas.DataFrame, Generator[pandas.DataFrame, None, None]]

Examples

Reading all Parquet files under a prefix

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

Reading all Parquet files from a list

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

Reading in chunks (Chunk by file)

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_parquet(path=['s3://bucket/filename0.csv', 's3://bucket/
↪filename1.csv'], 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.csv', 's3://bucket/
↪filename1.csv'], 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: Union[str, List[str]]*, *version_id: Optional[Union[str, Dict[str, str]]] = None*, *path_suffix: Optional[str] = None*, *path_ignore_suffix: Optional[str] = None*, *ignore_empty: bool = True*, *dtype: Optional[Dict[str, str]] = None*, *sampling: float = 1.0*, *dataset: bool = False*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*) → Any

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

The concept of Dataset goes beyond the simple idea of files and enable more complex features like partitioning and catalog integration (AWS Glue Catalog).

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use *glob.escape(path)* before passing the path to this function.

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

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

Check out the Global Configurations Tutorial for details.

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. {'s3://bucket/key0': '121212', 's3://bucket/key1': '343434'})
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. [".gz.parquet", ".snappy.parquet"]). If None, will try to read all files. (default)
- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. [".csv", "_SUCCESS"]). If None, will try to read all files. (default)
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined data types as partitions columns. (e.g. {'col name': 'bigint', 'col2 name': 'int'})
- **sampling** (*float*) – Random sample ratio of files that will have the metadata inspected. Must be *0.0 < sampling <= 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*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.

Returns columns_types: Dictionary with keys as column names and values as data types (e.g. {'col0': 'bigint', 'col1': 'double'}). / partitions_types: Dictionary with keys as partition names and values as data types (e.g. {'col2': 'date'}).

Return type Tuple[Dict[str, str], Optional[Dict[str, str]]]

Examples

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

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

Reading all Parquet files metadata from a list

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

awswrangler.s3.read_parquet_table

awswrangler.s3.**read_parquet_table**(*table: str*, *database: str*, *filename_suffix: Optional[Union[str, List[str]]] = None*, *filename_ignore_suffix: Optional[Union[str, List[str]]] = None*, *catalog_id: Optional[str] = None*, *partition_filter: Optional[Callable[[Dict[str, str]], bool]] = None*, *columns: Optional[List[str]] = None*, *validate_schema: bool = True*, *categories: Optional[List[str]] = None*, *safe: bool = True*, *map_types: bool = True*, *chunked: Union[bool, int] = False*, *use_threads: Union[bool, int] = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*) → Any

Read Apache Parquet table registered on AWS Glue Catalog.

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

Will anable the function to return a Iterable of DataFrames instead of a regular DataFrame.

There are two batching strategies on Wrangler:

- If **chunked=True**, a new DataFrame will be returned for each file in your path/dataset.
- If **chunked=INTEGER**, Wrangler will paginate through files slicing and concatenating to return DataFrames with the number of row igual the received INTEGER.

P.S. chunked=True if faster and uses less memory while *chunked=INTEGER* is more precise in number of rows for each Dataframe.

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

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **table** (*str*) – AWS Glue Catalog table name.
- **database** (*str*) – AWS Glue Catalog database name.
- **filename_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. [".gz.parquet", ".snappy.parquet"]). If None, will try to read all files. (default)
- **filename_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. [".csv", "_SUCCESS"]). If None, will try to read all files. (default)
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **partition_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (Dict[str, str]) where keys are 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-data-wrangler.readthedocs.io/en/2.10.0/tutorials/023%20-%20Flexible%20Partitions%20Filter.html
- **columns** (*List[str], optional*) – Names of columns to read from the file(s).
- **validate_schema** – Check that individual file schemas are all the same / compatible. Schemas within a folder prefix should all be the same. Disable if you have schemas that are different and want to disable this check.
- **categories** (*Optional[List[str]], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **safe** (*bool, default True*) – For certain data types, a cast is needed in order to store the data in a pandas DataFrame or Series (e.g. timestamps are always stored as nanoseconds in pandas). This option controls whether it is a safe cast or not.
- **map_types** (*bool, default True*) – True to convert pyarrow DataTypes to pandas ExtensionDtypes. It is used to override the default pandas type for conversion of built-in pyarrow types or in absence of pandas_metadata in the Table schema.
- **chunked** (*bool*) – If True will break the data in smaller DataFrames (Non deterministic number of lines). Otherwise return a single DataFrame with the whole data.
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads. If given an int will use the given amount of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.

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

Return type Union[pandas.DataFrame, Generator[pandas.DataFrame, None, None]]

Examples

Reading Parquet Table

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

Reading Parquet Table encrypted

```
>>> import awswrangler as wr
>>> df = wr.s3.read_parquet_table(
...     database='...',
...     table='...'
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'
...     }
... )
```

Reading Parquet Table in chunks (Chunk by file)

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_parquet_table(database='...', table='...', chunked=True)
>>> for df in dfs:
>>>     print(df)  # Smaller Pandas DataFrame
```

Reading Parquet Dataset with PUSH-DOWN filter over partitions

```
>>> import awswrangler as wr
>>> my_filter = lambda x: True if x["city"].startswith("new") else False
>>> df = wr.s3.read_parquet_table(path, dataset=True, partition_filter=my_filter)
```

awswrangler.s3.select_query

awswrangler.s3.**select_query**(*sql: str*, *path: str*, *input_serialization: str*, *input_serialization_params: Dict[str, Union[bool, str]]*, *compression: Optional[str] = None*, *use_threads: Union[bool, int] = False*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*) → pandas.core.frame.DataFrame

Filter contents of an Amazon S3 object based on SQL statement.

Note: Scan ranges are only supported for uncompressed CSV/JSON, CSV (without quoted delimiters) and JSON objects (in LINES mode only). It means scanning cannot be split across threads if the latter conditions are not met, leading to lower performance.

Parameters

- **sql** (*str*) – SQL statement used to query the object.
- **path** (*str*) – S3 path to the object (e.g. s3://bucket/key).
- **input_serialization** (*str,*) – Format of the S3 object queried. Valid values: "CSV", "JSON", or "Parquet". Case sensitive.
- **input_serialization_params** (*Dict[str, Union[bool, str]]*) – Dictionary describing the serialization of the S3 object.

- **compression** (*Optional[str]*) – Compression type of the S3 object. Valid values: None, "gzip", or "bzip2". gzip and bzip2 are only valid for CSV and JSON objects.
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() is used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session is used if none is provided.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. Valid values: "SSECustomerAlgorithm", "SSECustomerKey", "ExpectedBucketOwner". e.g. s3_additional_kwargs={'SSECustomerAlgorithm': 'md5'}

Returns Pandas DataFrame with results from query.

Return type pandas.DataFrame

Examples

Reading a gzip compressed JSON document

```
>>> import awswrangler as wr
>>> df = wr.s3.select_query(
...     sql='SELECT * FROM s3object[*][*]',
...     path='s3://bucket/key.json.gzip',
...     input_serialization='JSON',
...     input_serialization_params={
...         'Type': 'Document',
...     },
...     compression="gzip",
... )
```

Reading an entire CSV object using threads

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

Reading a single column from Parquet object with pushdown filter

```
>>> import awswrangler as wr
>>> df = wr.s3.select_query(
...     sql='SELECT s.\"id\" FROM s3object s where s.\"id\" = 1.0',
...     path='s3://bucket/key.snappy.parquet',
...     input_serialization='Parquet',
... )
```

awswrangler.s3.size_objects

awswrangler.s3.**size_objects**(*path: Union[str, List[str]]*, *version_id: Optional[Union[str, Dict[str, str]]] = None*, *use_threads: bool = True*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Optional[int]]

Get the size (ContentLength) in bytes of Amazon S3 objects from a received S3 prefix or list of S3 objects paths.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []*), you can use *glob.escape(path)* before passing the path to this function.

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

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. {'s3://bucket/key0': '121212', 's3://bucket/key1': '343434'})
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

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

Return type Dict[str, Optional[int]]

Examples

```
>>> import awswrangler as wr
>>> sizes0 = wr.s3.size_objects(['s3://bucket/key0', 's3://bucket/key1'])  # Get␣
↪the sizes of both objects
>>> sizes1 = wr.s3.size_objects('s3://bucket/prefix')  # Get the sizes of all␣
↪objects under the received prefix
```

awswrangler.s3.store_parquet_metadata

awswrangler.s3.**store_parquet_metadata**(*path: str*, *database: str*, *table: str*, *catalog_id: Optional[str] = None*, *path_suffix: Optional[str] = None*, *path_ignore_suffix: Optional[str] = None*, *ignore_empty: bool = True*, *dtype: Optional[Dict[str, str]] = None*, *sampling: float = 1.0*, *dataset: bool = False*, *use_threads: bool = True*, *description: Optional[str] = None*, *parameters: Optional[Dict[str, str]] = None*, *columns_comments: Optional[Dict[str, str]] = None*, *compression: Optional[str] = None*, *mode: str = 'overwrite'*, *catalog_versioning: bool = False*, *regular_partitions: bool = True*, *projection_enabled: bool = False*, *projection_types: Optional[Dict[str, str]] = None*, *projection_ranges: Optional[Dict[str, str]] = None*, *projection_values: Optional[Dict[str, str]] = None*, *projection_intervals: Optional[Dict[str, str]] = None*, *projection_digits: Optional[Dict[str, str]] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Infer and store parquet metadata on AWS Glue Catalog.

Infer Apache Parquet file(s) metadata from from a received S3 prefix or list of S3 objects paths And then stores it on AWS Glue Catalog including all inferred partitions (No need of 'MCSK REPAIR TABLE')

The concept of Dataset goes beyond the simple idea of files and enable more complex features like partitioning and catalog integration (AWS Glue Catalog).

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (**, ?, []*), you can use *glob.escape(path)* before passing the path to this function.

Note: On *append* mode, the *parameters* will be upsert on an existing table.

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

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]). database : str Glue/Athena catalog: Database name.
- **table** (*str*) – Glue/Athena catalog: Table name.
- **database** (*str*) – AWS Glue Catalog database name.

- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for filtering S3 keys.
- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined data types as partitions columns. (e.g. {'col name': 'bigint', 'col2 name': 'int'})
- **sampling** (*float*) – Random sample ratio of files that will have the metadata inspected. Must be *0.0 < sampling <= 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*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **description** (*str, optional*) – Glue/Athena catalog: Table description
- **parameters** (*Dict[str, str], optional*) – Glue/Athena catalog: Key/value pairs to tag the table.
- **columns_comments** (*Dict[str, str], optional*) – Glue/Athena catalog: Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **compression** (*str, optional*) – Compression style (`None`, `snappy`, `gzip`, etc).
- **mode** (*str*) – 'overwrite' to recreate any possible existing table or 'append' to keep any possible existing table.
- **catalog_versioning** (*bool*) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **regular_partitions** (*bool*) – Create regular partitions (Non projected partitions) on Glue Catalog. Disable when you will work only with Partition Projection. Keep enabled even when working with projections is useful to keep Redshift Spectrum working with the regular partitions.
- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: "enum", "integer", "date", "injected" https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': 'enum', 'col2_name': 'integer'})
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '0,10', 'col2_name': '-1,8675309'})
- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': 'A,B,Unknown', 'col2_name': 'foo,boo,bar'})

- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '1', 'col2_name': '5'})
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '1', 'col2_name': '2'})
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns The metadata used to create the Glue Table. columns_types: Dictionary with keys as column names and values as data types (e.g. {'col0': 'bigint', 'col1': 'double'}). / partitions_types: Dictionary with keys as partition names and values as data types (e.g. {'col2': 'date'}). / partitions_values: Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. {'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).

Return type Tuple[Dict[str, str], Optional[Dict[str, str]], Optional[Dict[str, List[str]]]]

Examples

Reading all Parquet files metadata under a prefix

```
>>> import awswrangler as wr
>>> columns_types, partitions_types, partitions_values = wr.s3.store_parquet_
↪metadata(
...     path='s3://bucket/prefix/',
...     database='...',
...     table='...',
...     dataset=True
... )
```

awswrangler.s3.to_csv

awswrangler.s3.**to_csv**(*df: pandas.core.frame.DataFrame*, *path: Optional[str] = None*, *sep: str = ','*, *index: bool = True*, *columns: Optional[List[str]] = None*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *sanitize_columns: bool = False*, *dataset: bool = False*, *filename_prefix: Optional[str] = None*, *partition_cols: Optional[List[str]] = None*, *bucketing_info: Optional[Tuple[List[str], int]] = None*, *concurrent_partitioning: bool = False*, *mode: Optional[str] = None*, *catalog_versioning: bool = False*, *schema_evolution: bool = False*, *database: Optional[str] = None*, *table: Optional[str] = None*, *dtype: Optional[Dict[str, str]] = None*, *description: Optional[str] = None*, *parameters: Optional[Dict[str, str]] = None*, *columns_comments: Optional[Dict[str, str]] = None*, *regular_partitions: bool = True*, *projection_enabled: bool = False*, *projection_types: Optional[Dict[str, str]] = None*, *projection_ranges: Optional[Dict[str, str]] = None*, *projection_values: Optional[Dict[str, str]] = None*, *projection_intervals: Optional[Dict[str, str]] = None*, *projection_digits: Optional[Dict[str, str]] = None*, *catalog_id: Optional[str] = None*, ***pandas_kwargs: Any*) → Any

Write CSV file or dataset on Amazon S3.

The concept of Dataset goes beyond the simple idea of ordinary files and enable more complex features like partitioning and catalog integration (Amazon Athena/AWS Glue Catalog).

Note: If database` and *table* arguments are passed, the table name and all column names will be automatically sanitized using *wr.catalog.sanitize_table_name* and *wr.catalog.sanitize_column_name*. Please, pass *sanitize_columns=True* to enforce this behaviour always.

Note: If *table* and *database* arguments are passed, *pandas_kwargs* will be ignored due restrictive quoting, date_format, escapechar and encoding required by Athena/Glue Catalog.

Note: Compression: The minimum acceptable version to achive it is Pandas 1.2.0 that requires Python >= 3.7.1.

Note: On *append* mode, the *parameters* will be upsert on an existing table.

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

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

- catalog_id
- concurrent_partitioning
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **path** (*str, optional*) – Amazon S3 path (e.g. s3://bucket/prefix/filename.csv) (for dataset e.g. `s3://bucket/prefix`). Required if dataset=False or when creating a new dataset
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **index** (*bool*) – Write row names (index).
- **columns** (*Optional[List[str]]*) – Columns to write.
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}
- **sanitize_columns** (*bool*) – True to sanitize columns names or False to keep it as is. True value is forced if *dataset=True*.
- **dataset** (*bool*) – If True store as a dataset instead of ordinary file(s) If True, enable all follow arguments: partition_cols, mode, database, table, description, parameters, columns_comments, concurrent_partitioning, catalog_versioning, projection_enabled, projection_types, projection_ranges, projection_values, projection_intervals, projection_digits, catalog_id, schema_evolution.
- **filename_prefix** (*str, optional*) – If dataset=True, add a filename prefix to the output files.
- **partition_cols** (*List[str], optional*) – List of column names that will be used to create partitions. Only takes effect if dataset=True.
- **bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **concurrent_partitioning** (*bool*) – If True will increase the parallelism level during the partitions writing. It will decrease the writing time and increase the memory usage. https://aws-data-wrangler.readthedocs.io/en/2.10.0/tutorials/022%20-%20Writing%20Partitions%20Concurrently.html
- **mode** (*str, optional*) – `append` (Default), `overwrite`, `overwrite_partitions`. Only takes effect if dataset=True. For details check the related tutorial: https://aws-data-wrangler.readthedocs.io/en/2.10.0/stubs/awswrangler.s3.to_parquet.html#awswrangler.s3.to_parquet
- **catalog_versioning** (*bool*) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **schema_evolution** (*bool*) – If True allows schema evolution (new or missing columns), otherwise a exception will be raised. (Only considered if dataset=True and mode in ("append", "overwrite_partitions")) Related tutorial: https://aws-data-wrangler.readthedocs.io/en/2.10.0/tutorials/014%20-%20Schema%20Evolution.html
- **database** (*str, optional*) – Glue/Athena catalog: Database name.
- **table** (*str, optional*) – Glue/Athena catalog: Table name.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'bigint', 'col2 name': 'int'})
- **description** (*str, optional*) – Glue/Athena catalog: Table description
- **parameters** (*Dict[str, str], optional*) – Glue/Athena catalog: Key/value pairs to tag the table.
- **columns_comments** (*Dict[str, str], optional*) – Glue/Athena catalog: Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **regular_partitions** (*bool*) – Create regular partitions (Non projected partitions) on Glue Catalog. Disable when you will work only with Partition Projection. Keep enabled even when working with projections is useful to keep Redshift Spectrum working with the regular partitions.

- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: "enum", "integer", "date", "injected" https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': 'enum', 'col2_name': 'integer'})
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '0,10', 'col2_name': '-1,8675309'})
- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': 'A,B,Unknown', 'col2_name': 'foo,boo,bar'})
- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '1', 'col2_name': '5'})
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '1', 'col2_name': '2'})
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **pandas_kwargs** – KEYWORD arguments forwarded to pandas.DataFrame.to_csv(). You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. wr.s3.to_csv(df, path, sep='|', na_rep='NULL', decimal=',') https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html

Returns Dictionary with: 'paths': List of all stored files paths on S3. 'partitions_values': Dictionary of partitions added with keys as S3 path locations and values as a list of partitions values as str.

Return type Dict[str, Union[List[str], Dict[str, List[str]]]]

Examples

Writing single file

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

Writing single file with pandas_kwargs

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_csv(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/prefix/my_file.csv',
...     sep='|',
...     na_rep='NULL',
...     decimal=','
... )
{
    'paths': ['s3://bucket/prefix/my_file.csv'],
    'partitions_values': {}
}
```

Writing single file encrypted with a KMS key

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

Writing partitioned dataset

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

Writing bucketed dataset

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_csv(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B']
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     bucketing_info=(["col2"], 2)
... )
{
    'paths': ['s3://.../x_bucket-00000.csv', 's3://.../col2=B/x_bucket-00001.csv'],
    'partitions_values: {}
}
```

Writing dataset to S3 with metadata on Athena/Glue Catalog.

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_csv(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B']
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     partition_cols=['col2'],
...     database='default',  # Athena/Glue database
...     table='my_table'  # Athena/Glue table
... )
{
    'paths': ['s3://.../col2=A/x.csv', 's3://.../col2=B/y.csv'],
    'partitions_values: {
        's3://.../col2=A/': ['A'],
        's3://.../col2=B/': ['B']
    }
}
```

Writing dataset 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
...     table='my_table'  # Athena/Glue table
```

(continues on next page)

(continued from previous page)

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

awswrangler.s3.to_excel

awswrangler.s3.**to_excel**(*df: pandas.core.frame.DataFrame*, *path: str*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *use_threads: bool = True*, ***pandas_kwargs: Any*) → str

Write EXCEL file on Amazon S3.

Note: This function accepts any Pandas's read_excel() argument. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

---

Note: Depending on the file extension ('xlsx', 'xls', 'odf'...), an additional library might have to be installed first (e.g. xlrd).

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

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **path** (*str*) – Amazon S3 path (e.g. s3://bucket/filename.xlsx).
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **pandas_kwargs** – KEYWORD arguments forwarded to pandas.DataFrame.to_excel(). You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. wr.s3.to_excel(df, path, na_rep="", index=False) https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_excel.html

Returns Written S3 path.

Return type str

Examples

Writing EXCEL file

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_excel(df, 's3://bucket/filename.xlsx')
```

awswrangler.s3.to_json

awswrangler.s3.**to_json**(*df: pandas.core.frame.DataFrame*, *path: str*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *use_threads: bool = True*, ***pandas_kwargs: Any*) → List[str]

Write JSON file on Amazon S3.

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

Note: Compression: The minimum acceptable version to achive it is Pandas 1.2.0 that requires Python >= 3.7.1.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **path** (*str*) – Amazon S3 path (e.g. s3://bucket/filename.csv).
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **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 Wrangler will accept it. e.g. wr.s3.to_json(df, path, lines=True, date_format='iso') https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_json.html

Returns List of written files.

Return type List[str]

Examples

Writing JSON file

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_json(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/filename.json',
... )
```

Writing JSON file using pandas_kwargs

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_json(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/filename.json',
...     lines=True,
...     date_format='iso'
... )
```

Writing CSV file encrypted with a KMS key

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_json(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/filename.json',
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'
...     }
... )
```

awswrangler.s3.to_parquet

awswrangler.s3.**to_parquet**(*df: pandas.core.frame.DataFrame*, *path: Optional[str] = None*, *index: bool = False*, *compression: Optional[str] = 'snappy'*, *max_rows_by_file: Optional[int] = None*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*, *sanitize_columns: bool = False*, *dataset: bool = False*, *filename_prefix: Optional[str] = None*, *partition_cols: Optional[List[str]] = None*, *bucketing_info: Optional[Tuple[List[str], int]] = None*, *concurrent_partitioning: bool = False*, *mode: Optional[str] = None*, *catalog_versioning: bool = False*, *schema_evolution: bool = True*, *database: Optional[str] = None*, *table: Optional[str] = None*, *dtype: Optional[Dict[str, str]] = None*, *description: Optional[str] = None*, *parameters: Optional[Dict[str, str]] = None*, *columns_comments: Optional[Dict[str, str]] = None*, *regular_partitions: bool = True*, *projection_enabled: bool = False*, *projection_types: Optional[Dict[str, str]] = None*, *projection_ranges: Optional[Dict[str, str]] = None*, *projection_values: Optional[Dict[str, str]] = None*, *projection_intervals: Optional[Dict[str, str]] = None*, *projection_digits: Optional[Dict[str, str]] = None*, *catalog_id: Optional[str] = None*) → Any

Write Parquet file or dataset on Amazon S3.

The concept of Dataset goes beyond the simple idea of ordinary files and enable more complex features like partitioning and catalog integration (Amazon Athena/AWS Glue Catalog).

Note: This operation may mutate the original pandas dataframe in-place. To avoid this behaviour please pass in a deep copy instead (i.e. *df.copy()*)

Note: If *database* and *table* arguments are passed, the table name and all column names will be automatically sanitized using *wr.catalog.sanitize_table_name* and *wr.catalog.sanitize_column_name*. Please, pass *sanitize_columns=True* to enforce this behaviour always.

Note: On *append* mode, the *parameters* will be upsert on an existing table.

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

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

- catalog_id
- concurrent_partitioning
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **path** (*str, optional*) – S3 path (for file e.g. `s3://bucket/prefix/filename.parquet`) (for dataset e.g. `s3://bucket/prefix`). Required if dataset=False or when dataset=True and creating a new dataset
- **index** (*bool*) – True to store the DataFrame index in file, otherwise False to ignore it.
- **compression** (*str, optional*) – Compression style (`None`, `snappy`, `gzip`).
- **max_rows_by_file** (*int*) – Max number of rows in each file. Default is None i.e. dont split the files. (e.g. 33554432, 268435456)
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}
- **sanitize_columns** (*bool*) – True to sanitize columns names (using *wr.catalog.sanitize_table_name* and *wr.catalog.sanitize_column_name*) or False to keep it as is. True value behaviour is enforced if *database* and *table* arguments are passed.
- **dataset** (*bool*) – If True store a parquet dataset instead of a ordinary file(s) If True, enable all follow arguments: partition_cols, mode, database, table, description, parameters, columns_comments, concurrent_partitioning, catalog_versioning, projection_enabled, projection_types, projection_ranges, projection_values, projection_intervals, projection_digits, catalog_id, schema_evolution.
- **filename_prefix** (*str, optional*) – If dataset=True, add a filename prefix to the output files.
- **partition_cols** (*List[str], optional*) – List of column names that will be used to create partitions. Only takes effect if dataset=True.
- **bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **concurrent_partitioning** (*bool*) – If True will increase the parallelism level during the partitions writing. It will decrease the writing time and increase the memory usage. https://aws-data-wrangler.readthedocs.io/en/2.10.0/tutorials/022%20-%20Writing%20Partitions%20Concurrently.html
- **mode** (*str, optional*) – `append` (Default), `overwrite`, `overwrite_partitions`. Only takes effect if dataset=True. For details check the related tutorial: https://aws-data-wrangler.readthedocs.io/en/2.10.0/stubs/awswrangler.s3.to_parquet.html#awswrangler.s3.to_parquet
- **catalog_versioning** (*bool*) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **schema_evolution** (*bool*) – If True allows schema evolution (new or missing columns), otherwise a exception will be raised. (Only considered if dataset=True and mode in ("append", "overwrite_partitions")) Related tutorial: https://aws-data-wrangler.readthedocs.io/en/2.10.0/tutorials/014%20-%20Schema%20Evolution.html

- **database** (*str, optional*) – Glue/Athena catalog: Database name.
- **table** (*str, optional*) – Glue/Athena catalog: Table name.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'bigint', 'col2 name': 'int'})
- **description** (*str, optional*) – Glue/Athena catalog: Table description
- **parameters** (*Dict[str, str], optional*) – Glue/Athena catalog: Key/value pairs to tag the table.
- **columns_comments** (*Dict[str, str], optional*) – Glue/Athena catalog: Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **regular_partitions** (*bool*) – Create regular partitions (Non projected partitions) on Glue Catalog. Disable when you will work only with Partition Projection. Keep enabled even when working with projections is useful to keep Redshift Spectrum working with the regular partitions.
- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: "enum", "integer", "date", "injected" https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': 'enum', 'col2_name': 'integer'})
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '0,10', 'col2_name': '-1,8675309'})
- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': 'A,B,Unknown', 'col2_name': 'foo,boo,bar'})
- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '1', 'col2_name': '5'})
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '1', 'col2_name': '2'})
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

Returns Dictionary with: 'paths': List of all stored files paths on S3. 'partitions_values': Dictionary of partitions added with keys as S3 path locations and values as a list of partitions values as str.

Return type Dict[str, Union[List[str], Dict[str, List[str]]]]

Examples

Writing single file

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

Writing single file encrypted with a KMS key

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

Writing partitioned dataset

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

Writing bucketed dataset

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_parquet(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B']
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     bucketing_info=(["col2"], 2)
... )
{
    'paths': ['s3://.../x_bucket-00000.csv', 's3://.../col2=B/x_bucket-00001.csv'],
    'partitions_values: {}
}
```

Writing dataset to S3 with metadata on Athena/Glue Catalog.

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_parquet(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B']
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     partition_cols=['col2'],
...     database='default',  # Athena/Glue database
...     table='my_table'  # Athena/Glue table
... )
{
    'paths': ['s3://.../col2=A/x.parquet', 's3://.../col2=B/y.parquet'],
    'partitions_values: {
        's3://.../col2=A/': ['A'],
        's3://.../col2=B/': ['B']
    }
}
```

Writing dataset 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
```

(continues on next page)

(continued from previous page)

```
...     dtype={'col3': 'date'}
... )
{
    'paths': ['s3://.../x.parquet'],
    'partitions_values: {}
}
```

awswrangler.s3.upload

awswrangler.s3.**upload**(*local_file: Union[str, Any]*, *path: str*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*) → None

Upload file from a local file to received S3 path.

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

Parameters

- **local_file** (*Union[str, Any]*) – A file-like object in binary mode or a path to local file (e.g. `./local/path/to/key0`).
- **path** (*str*) – S3 path (e.g. `s3://bucket/key0`).
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.

Returns

Return type None

Examples

Uploading a file using a path to local file

```
>>> import awswrangler as wr
>>> wr.s3.upload(local_file='./key', path='s3://bucket/key')
```

Uploading a file using a file-like object

```
>>> import awswrangler as wr
>>> with open(file='./key', mode='wb') as local_f:
>>>     wr.s3.upload(local_file=local_f, path='s3://bucket/key')
```

awswrangler.s3.wait_objects_exist

awswrangler.s3.**wait_objects_exist**(*paths: List[str]*, *delay: Optional[float] = None*, *max_attempts: Optional[int] = None*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Wait Amazon S3 objects exist.

Polls S3.Client.head_object() every 5 seconds (default) until a successful state is reached. An error is returned after 20 (default) failed checks. https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Waiter.ObjectExists

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

Parameters

- **paths** (*List[str]*) – List of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **delay** (*Union[int,float], optional*) – The amount of time in seconds to wait between attempts. Default: 5
- **max_attempts** (*int, optional*) – The maximum number of attempts to be made. Default: 20
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.s3.wait_objects_exist(['s3://bucket/key0', 's3://bucket/key1'])  # wait both␣
↪objects
```

awswrangler.s3.wait_objects_not_exist

awswrangler.s3.**wait_objects_not_exist**(*paths: List[str]*, *delay: Optional[float] = None*, *max_attempts: Optional[int] = None*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Wait Amazon S3 objects not exist.

Polls S3.Client.head_object() every 5 seconds (default) until a successful state is reached. An error is returned after 20 (default) failed checks. https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Waiter.ObjectNotExists

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

Parameters

- **paths** (*List[str]*) – List of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **delay** (*Union[int,float], optional*) – The amount of time in seconds to wait between attempts. Default: 5
- **max_attempts** (*int, optional*) – The maximum number of attempts to be made. Default: 20
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

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

1.4.2 AWS Glue Catalog

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_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, ...])	Create 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.
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_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.

continues on next page

Table 2 – continued from previous page

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 tatal number of versions.
get_table_parameters(database, table[, ...])	Get all parameters.
get_table_types(database, table[, boto3_session])	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.
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.
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: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *catalog_id: Optional[str] = None*) → Any

Add a column in a AWS Glue Catalog table.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **column_name** (*str*) – Column name
- **column_type** (*str*) – Column type.
- **column_comment** (*str*) – Column Comment
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

Returns None

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.add_column(
...     database='my_db',
...     table='my_table',
...     column_name='my_col',
...     column_type='int'
... )
```

awswrangler.catalog.add_csv_partitions

awswrangler.catalog.**add_csv_partitions**(*database: str*, *table: str*, *partitions_values: Dict[str, List[str]]*, *bucketing_info: Optional[Tuple[List[str], int]] = None*, *catalog_id: Optional[str] = None*, *compression: Optional[str] = None*, *sep: str = ','*, *serde_library: Optional[str] = None*, *serde_parameters: Optional[Dict[str, str]] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *columns_types: Optional[Dict[str, str]] = None*) → Any

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

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **partitions_values** (*Dict[str, List[str]]*) – Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. {'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).
- **bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **compression** (*str, optional*) – Compression style (`None`, `gzip`, etc).
- **sep** (*str*) – String of length 1. Field delimiter for the output file.

- **serde_library** (*Optional[str]*) – Specifies the SerDe Serialization library which will be used. You need to provide the Class library name as a string. If no library is provided the default is *org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe*.
- **serde_parameters** (*Optional[str]*) – Dictionary of initialization parameters for the SerDe. The default is *{"field.delim": sep, "escape.delim": "\"}*.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **columns_types** (*Optional[Dict[str, str]]*) – Only required for Hive 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.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.add_csv_partitions(
...     database='default',
...     table='my_table',
...     partitions_values={
...         's3://bucket/prefix/y=2020/m=10/': ['2020', '10'],
...         's3://bucket/prefix/y=2020/m=11/': ['2020', '11'],
...         's3://bucket/prefix/y=2020/m=12/': ['2020', '12']
...     }
... )
```

awswrangler.catalog.add_parquet_partitions

awswrangler.catalog.**add_parquet_partitions**(*database: str*, *table: str*, *partitions_values: Dict[str, List[str]]*, *bucketing_info: Optional[Tuple[List[str], int]] = None*, *catalog_id: Optional[str] = None*, *compression: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *columns_types: Optional[Dict[str, str]] = None*) → Any

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

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

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

- **partitions_values** (*Dict[str, List[str]]*) – Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. {'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).
- **bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **compression** (*str, optional*) – Compression style (`None`, `snappy`, `gzip`, etc).
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **columns_types** (*Optional[Dict[str, str]]*) – Only required for Hive compability. Dictionary with keys as column names and values as data types (e.g. {'col0': 'bigint', 'col1': 'double'}). P.S. Only materialized columns please, not partition columns.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.add_parquet_partitions(
...     database='default',
...     table='my_table',
...     partitions_values={
...         's3://bucket/prefix/y=2020/m=10/': ['2020', '10'],
...         's3://bucket/prefix/y=2020/m=11/': ['2020', '11'],
...         's3://bucket/prefix/y=2020/m=12/': ['2020', '12']
...     }
... )
```

awswrangler.catalog.create_csv_table

awswrangler.catalog.**create_csv_table**(*database: str*, *table: str*, *path: str*, *columns_types: Dict[str, str]*, *partitions_types: Optional[Dict[str, str]] = None*, *bucketing_info: Optional[Tuple[List[str], int]] = None*, *compression: Optional[str] = None*, *description: Optional[str] = None*, *parameters: Optional[Dict[str, str]] = None*, *columns_comments: Optional[Dict[str, str]] = None*, *mode: str = 'overwrite'*, *catalog_versioning: bool = False*, *sep: str = ','*, *skip_header_line_count: Optional[int] = None*, *serde_library: Optional[str] = None*, *serde_parameters: Optional[Dict[str, str]] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *projection_enabled: bool = False*, *projection_types: Optional[Dict[str, str]] = None*, *projection_ranges: Optional[Dict[str, str]] = None*, *projection_values: Optional[Dict[str, str]] = None*, *projection_intervals: Optional[Dict[str, str]] = None*, *projection_digits: Optional[Dict[str, str]] = None*, *catalog_id: Optional[str] = None*) → Any

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

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

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **path** (*str*) – Amazon S3 path (e.g. s3://bucket/prefix/).
- **columns_types** (*Dict[str, str]*) – Dictionary with keys as column names and values as data types (e.g. {'col0': 'bigint', 'col1': 'double'}).
- **partitions_types** (*Dict[str, str], optional*) – Dictionary with keys as partition names and values as data types (e.g. {'col2': 'date'}).
- **bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **compression** (*str, optional*) – Compression style (None, gzip, etc).
- **description** (*str, optional*) – Table description
- **parameters** (*Dict[str, str], optional*) – Key/value pairs to tag the table.
- **columns_comments** (*Dict[str, str], optional*) – Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).

- **mode** (*str*) – 'overwrite' to recreate any possible axisting table or 'append' to keep any possible axisting table.
- **catalog_versioning** (*bool*) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **skip_header_line_count** (*Optional[int]*) – Number of Lines to skip regarding to the header.
- **serde_library** (*Optional[str]*) – Specifies the SerDe Serialization library which will be used. You need to provide the Class library name as a string. If no library is provided the default is *org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe*.
- **serde_parameters** (*Optional[str]*) – Dictionary of initialization parameters for the SerDe. The default is *{"field.delim": sep, "escape.delim": "\"}*.
- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: "enum", "integer", "date", "injected" https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': 'enum', 'col2_name': 'integer'})
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '0,10', 'col2_name': '-1,8675309'})
- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': 'A,B,Unknown', 'col2_name': 'foo,boo,bar'})
- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '1', 'col2_name': '5'})
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '1', 'col2_name': '2'})
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.create_csv_table(
...     database='default',
...     table='my_table',
...     path='s3://bucket/prefix/',
...     columns_types={'col0': 'bigint', 'col1': 'double'},
...     partitions_types={'col2': 'date'},
...     compression='gzip',
...     description='My own table!',
...     parameters={'source': 'postgresql'},
...     columns_comments={'col0': 'Column 0.', 'col1': 'Column 1.', 'col2':
↪'Partition.'}
... )
```

awswrangler.catalog.create_database

awswrangler.catalog.**create_database**(*name: str*, *description: Optional[str] = None*, *catalog_id: Optional[str] = None*, *exist_ok: bool = False*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Create a database in AWS Glue Catalog.

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

- catalog_id

Check out the Global Configurations Tutorial for details.

Parameters

- **name** (*str*) – Database name.
- **description** (*str, optional*) – A Descrption for the Database.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **exist_ok** (*bool*) – If set to True will not raise an Exception if a Database with the same already exists. In this case the description will be updated if it is different from the current one.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.create_database(
...     name='awswrangler_test'
... )
```

awswrangler.catalog.create_parquet_table

awswrangler.catalog.**create_parquet_table**(*database: str*, *table: str*, *path: str*, *columns_types: Dict[str, str]*, *partitions_types: Optional[Dict[str, str]] = None*, *bucketing_info: Optional[Tuple[List[str], int]] = None*, *catalog_id: Optional[str] = None*, *compression: Optional[str] = None*, *description: Optional[str] = None*, *parameters: Optional[Dict[str, str]] = None*, *columns_comments: Optional[Dict[str, str]] = None*, *mode: str = 'overwrite'*, *catalog_versioning: bool = False*, *projection_enabled: bool = False*, *projection_types: Optional[Dict[str, str]] = None*, *projection_ranges: Optional[Dict[str, str]] = None*, *projection_values: Optional[Dict[str, str]] = None*, *projection_intervals: Optional[Dict[str, str]] = None*, *projection_digits: Optional[Dict[str, str]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

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

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

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **path** (*str*) – Amazon S3 path (e.g. s3://bucket/prefix/).
- **columns_types** (*Dict[str, str]*) – Dictionary with keys as column names and values as data types (e.g. {'col0': 'bigint', 'col1': 'double'}).
- **partitions_types** (*Dict[str, str], optional*) – Dictionary with keys as partition names and values as data types (e.g. {'col2': 'date'}).
- **bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.

- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **compression** (*str, optional*) – Compression style (None, snappy, gzip, etc).
- **description** (*str, optional*) – Table description
- **parameters** (*Dict[str, str], optional*) – Key/value pairs to tag the table.
- **columns_comments** (*Dict[str, str], optional*) – Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **mode** (*str*) – 'overwrite' to recreate any possible existing table or 'append' to keep any possible existing table.
- **catalog_versioning** (*bool*) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: "enum", "integer", "date", "injected" https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': 'enum', 'col2_name': 'integer'})
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '0,10', 'col2_name': '-1,8675309'})
- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': 'A,B,Unknown', 'col2_name': 'foo,boo,bar'})
- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '1', 'col2_name': '5'})
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html (e.g. {'col_name': '1', 'col2_name': '2'})
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.create_parquet_table(
...     database='default',
...     table='my_table',
...     path='s3://bucket/prefix/',
...     columns_types={'col0': 'bigint', 'col1': 'double'},
...     partitions_types={'col2': 'date'},
...     compression='snappy',
...     description='My own table!',
...     parameters={'source': 'postgresql'},
...     columns_comments={'col0': 'Column 0.', 'col1': 'Column 1.', 'col2':
↪'Partition.'}
... )
```

awswrangler.catalog.databases

awswrangler.catalog.**databases**(*limit: int = 100*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Get a Pandas DataFrame with all listed databases.

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

- catalog_id

Check out the Global Configurations Tutorial for details.

Parameters

- **limit** (*int, optional*) – Max number of tables to be returned.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Pandas DataFrame filled by formatted infos.

Return type pandas.DataFrame

Examples

```
>>> import awswrangler as wr
>>> df_dbs = wr.catalog.databases()
```

awswrangler.catalog.delete_column

awswrangler.catalog.**delete_column**(*database: str*, *table: str*, *column_name: str*, *boto3_session: Optional[boto3.session.Session] = None*, *catalog_id: Optional[str] = None*) → Any

Delete a column in a AWS Glue Catalog table.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **column_name** (*str*) – Column name
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

Returns None

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.delete_column(
...     database='my_db',
...     table='my_table',
...     column_name='my_col',
... )
```

awswrangler.catalog.delete_database

awswrangler.catalog.**delete_database**(*name: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Create a database in AWS Glue Catalog.

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

- catalog_id

Check out the Global Configurations Tutorial for details.

Parameters

- **name** (*str*) – Database name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.delete_database(
...     name='awswrangler_test'
... )
```

awswrangler.catalog.delete_partitions

awswrangler.catalog.**delete_partitions**(*table: str*, *database: str*, *partitions_values: List[List[str]]*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Delete specified partitions in a AWS Glue Catalog table.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **table** (*str*) – Table name.
- **database** (*str*) – Table name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **partitions_values** (*List[List[str]]*) – List of lists of partitions values as strings. (e.g. [[‘2020’, ‘10’, ‘25’], [‘2020’, ‘11’, ‘16’], [‘2020’, ‘12’, ‘19’]]).
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.delete_partitions(
...     table='my_table',
...     database='awswrangler_test',
...     partitions_values=[['2020', '10', '25'], ['2020', '11', '16'], ['2020', '12
↪', '19']]
... )
```

awswrangler.catalog.delete_all_partitions

awswrangler.catalog.**delete_all_partitions**(*table: str*, *database: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Delete all partitions in a AWS Glue Catalog table.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **table** (*str*) – Table name.
- **database** (*str*) – Table name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Partitions values.

Return type List[List[str]]

Examples

```
>>> import awswrangler as wr
>>> partitions = wr.catalog.delete_all_partitions(
...     table='my_table',
...     database='awswrangler_test',
... )
```

awswrangler.catalog.delete_table_if_exists

awswrangler.catalog.**delete_table_if_exists**(*database: str*, *table: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Delete Glue table if exists.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns True if deleted, otherwise False.

Return type bool

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.delete_table_if_exists(database='default', table='my_table')  #␣
↪deleted
True
>>> wr.catalog.delete_table_if_exists(database='default', table='my_table')  #␣
↪Nothing to be deleted
False
```

awswrangler.catalog.does_table_exist

awswrangler.catalog.**does_table_exist**(*database: str*, *table: str*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Check if the table exists.

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

- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns True if exists, otherwise False.

Return type bool

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.does_table_exist(database='default', table='my_table')
```

awswrangler.catalog.drop_duplicated_columns

awswrangler.catalog.**drop_duplicated_columns**(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Drop all repeated columns (duplicated names).

Note: This transformation will run *inplace* and will make changes in the original DataFrame.

Note: It is different from Panda's drop_duplicates() function which considers the column values. wr.catalog.drop_duplicated_columns() will deduplicate by column name.

Parameters **df** (*pandas.DataFrame*) – Original Pandas DataFrame.

Returns Pandas DataFrame without duplicated columns.

Return type pandas.DataFrame

Examples

```
>>> import awswrangler as wr
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4]})
>>> df.columns = ["A", "A"]
>>> wr.catalog.drop_duplicated_columns(df=df)
   A
0  1
1  2
```

awswrangler.catalog.extract_athena_types

awswrangler.catalog.**extract_athena_types**(*df: pandas.core.frame.DataFrame*, *index: bool = False*, *partition_cols: Optional[List[str]] = None*, *dtype: Optional[Dict[str, str]] = None*, *file_format: str = 'parquet'*) → Tuple[Dict[str, str], Dict[str, str]]

Extract columns and partitions types (Amazon Athena) from Pandas DataFrame.

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

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame.
- **index** (*bool*) – Should consider the DataFrame index as a column?.
- **partition_cols** (*List[str], optional*) – List of partitions names.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'bigint', 'col2 name': 'int'})
- **file_format** (*str, optional*) – File format to be consided to place the index column: "parquet" | "csv".

Returns columns_types: Dictionary with keys as column names and values as data types (e.g. {'col0': 'bigint', 'col1': 'double'}). / partitions_types: Dictionary with keys as partition names and values as data types (e.g. {'col2': 'date'}).

Return type Tuple[Dict[str, str], Dict[str, str]]

Examples

```
>>> import awswrangler as wr
>>> columns_types, partitions_types = wr.catalog.extract_athena_types(
...     df=df, index=False, partition_cols=["par0", "par1"], file_format="csv"
... )
```

awswrangler.catalog.get_columns_comments

awswrangler.catalog.**get_columns_comments**(*database: str*, *table: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Get all columns comments.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

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

- **table** (*str*) – Table name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Columns comments. e.g. {"col1": "foo boo bar"}.

Return type Dict[str, str]

Examples

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

awswrangler.catalog.get_csv_partitions

awswrangler.catalog.**get_csv_partitions**(*database: str*, *table: str*, *expression: Optional[str] = None*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Get all partitions from a Table in the AWS Glue Catalog.

Expression argument instructions: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_partitions

---

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **expression** (*str, optional*) – An expression that filters the partitions to be returned.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns partitions_values: Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. {'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).

Return type Dict[str, List[str]]

Examples

Fetch all partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_csv_partitions(
...     database='default',
...     table='my_table',
... )
{
    's3://bucket/prefix/y=2020/m=10/': ['2020', '10'],
    's3://bucket/prefix/y=2020/m=11/': ['2020', '11'],
    's3://bucket/prefix/y=2020/m=12/': ['2020', '12']
}
```

Filtering partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_csv_partitions(
...     database='default',
...     table='my_table',
...     expression='m=10'
... )
{
    's3://bucket/prefix/y=2020/m=10/': ['2020', '10']
}
```

awswrangler.catalog.get_databases

awswrangler.catalog.**get_databases**(*catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Iterator[Dict[str, Any]]

Get an iterator of databases.

Parameters

- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Iterator of Databases.

Return type Iterator[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> dbs = wr.catalog.get_databases()
```

awswrangler.catalog.get_parquet_partitions

awswrangler.catalog.**get_parquet_partitions**(*database: str*, *table: str*, *expression: Optional[str] = None*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Get all partitions from a Table in the AWS Glue Catalog.

Expression argument instructions: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_partitions

---

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **expression** (*str, optional*) – An expression that filters the partitions to be returned.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns partitions_values: Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. {'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).

Return type Dict[str, List[str]]

Examples

Fetch all partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_parquet_partitions(
...     database='default',
...     table='my_table',
... )
{
    's3://bucket/prefix/y=2020/m=10/': ['2020', '10'],
    's3://bucket/prefix/y=2020/m=11/': ['2020', '11'],
    's3://bucket/prefix/y=2020/m=12/': ['2020', '12']
}
```

Filtering partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_parquet_partitions(
...     database='default',
...     table='my_table',
...     expression='m=10'
... )
{
    's3://bucket/prefix/y=2020/m=10/': ['2020', '10']
}
```

awswrangler.catalog.get_partitions

awswrangler.catalog.**get_partitions**(*database: str*, *table: str*, *expression: Optional[str] = None*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Get all partitions from a Table in the AWS Glue Catalog.

Expression argument instructions: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_partitions

---

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **expression** (*str, optional*) – An expression that filters the partitions to be returned.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns partitions_values: Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. {'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).

Return type Dict[str, List[str]]

Examples

Fetch all partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_partitions(
...     database='default',
...     table='my_table',
... )
{
    's3://bucket/prefix/y=2020/m=10/': ['2020', '10'],
    's3://bucket/prefix/y=2020/m=11/': ['2020', '11'],
    's3://bucket/prefix/y=2020/m=12/': ['2020', '12']
}
```

Filtering partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_partitions(
...     database='default',
...     table='my_table',
...     expression='m=10'
... )
{
    's3://bucket/prefix/y=2020/m=10/': ['2020', '10']
}
```

awswrangler.catalog.get_table_description

awswrangler.catalog.**get_table_description**(*database: str*, *table: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Optional[str]

Get table description.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Description if exists.

Return type Optional[str]

Examples

```
>>> import awswrangler as wr
>>> desc = wr.catalog.get_table_description(database="...", table="...")
```

awswrangler.catalog.get_table_location

awswrangler.catalog.**get_table_location**(*database: str*, *table: str*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Get table's location on Glue catalog.

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

- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Table's location.

Return type str

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.get_table_location(database='default', table='my_table')
's3://bucket/prefix/'
```

awswrangler.catalog.get_table_number_of_versions

awswrangler.catalog.**get_table_number_of_versions**(*database: str*, *table: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Get tatal number of versions.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Total number of versions.

Return type int

Examples

```
>>> import awswrangler as wr
>>> num = wr.catalog.get_table_number_of_versions(database="...", table="...")
```

awswrangler.catalog.get_table_parameters

awswrangler.catalog.**get_table_parameters**(*database: str*, *table: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, str]

Get all parameters.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Dictionary of parameters.

Return type Dict[str, str]

Examples

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

awswrangler.catalog.get_table_types

awswrangler.catalog.**get_table_types**(*database: str*, *table: str*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Get all columns and types from a table.

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

- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

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

Return type Optional[Dict[str, str]]

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.get_table_types(database='default', table='my_table')
{'col0': 'int', 'col1': double}
```

awswrangler.catalog.get_table_versions

awswrangler.catalog.**get_table_versions**(*database: str*, *table: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Get all versions.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

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

Return type List[Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> tables_versions = wr.catalog.get_table_versions(database="...", table="...")
```

awswrangler.catalog.get_tables

awswrangler.catalog.**get_tables**(*catalog_id: Optional[str] = None*, *database: Optional[str] = None*, *name_contains: Optional[str] = None*, *name_prefix: Optional[str] = None*, *name_suffix: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Get an iterator of tables.

Note: Please, does not filter using name_contains and name_prefix/name_suffix at the same time. Only name_prefix and name_suffix can be combined together.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **database** (*str, optional*) – Database name.
- **name_contains** (*str, optional*) – Select by a specific string on table name
- **name_prefix** (*str, optional*) – Select by a specific prefix on table name
- **name_suffix** (*str, optional*) – Select by a specific suffix on table name
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Iterator of tables.

Return type Iterator[Dict[str, Any]]

Examples

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

awswrangler.catalog.overwrite_table_parameters

awswrangler.catalog.**overwrite_table_parameters**(*parameters: Dict[str, str]*, *database: str*, *table: str*, *catalog_versioning: bool = False*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Overwrite all existing parameters.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **parameters** (*Dict[str, str]*) – e.g. {"source": "mysql", "destination": "datalake"}
- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **catalog_versioning** (*bool*) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

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

Return type Dict[str, str]

Examples

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

awswrangler.catalog.sanitize_column_name

awswrangler.catalog.**sanitize_column_name**(*column: str*) → str

Convert the column name to be compatible with Amazon Athena.

https://docs.aws.amazon.com/athena/latest/ug/tables-databases-columns-names.html

Possible transformations: - Strip accents - Remove non alphanumeric characters - Convert CamelCase to snake_case

Parameters **column** (*str*) – Column name.

Returns Normalized column name.

Return type str

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.sanitize_column_name('MyNewColumn')
'my_new_column'
```

awswrangler.catalog.sanitize_dataframe_columns_names

awswrangler.catalog.**sanitize_dataframe_columns_names**(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.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 - Convert CamelCase to snake_case

---

Note: After transformation, some column names might not be unique anymore. Example: the columns ["A", "a"] will be sanitized to ["a", "a"]

Parameters **df** (*pandas.DataFrame*) – Original Pandas DataFrame.

Returns Original Pandas DataFrame with columns names normalized.

Return type pandas.DataFrame

Examples

```
>>> import awswrangler as wr
>>> df_normalized = wr.catalog.sanitize_dataframe_columns_names(df=pd.DataFrame({'A
↪': [1, 2]}))
```

awswrangler.catalog.sanitize_table_name

awswrangler.catalog.**sanitize_table_name**(*table: str*) → str

Convert the table name 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 - Convert CamelCase to snake_case

Parameters **table** (*str*) – Table name.

Returns Normalized table name.

Return type str

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.sanitize_table_name('MyNewTable')
'my_new_table'
```

awswrangler.catalog.search_tables

awswrangler.catalog.**search_tables**(*text: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Iterator[Dict[str, Any]]

Get Pandas DataFrame of tables filtered by a search string.

Parameters

- **text** (*str, optional*) – Select only tables with the given string in table's properties.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Iterator of tables.

Return type Iterator[Dict[str, Any]]

Examples

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

awswrangler.catalog.table

awswrangler.catalog.**table**(*database: str*, *table: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Get table details as Pandas DataFrame.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Pandas DataFrame filled by formatted infos.

Return type pandas.DataFrame

Examples

```
>>> import awswrangler as wr
>>> df_table = wr.catalog.table(database='default', table='my_table')
```

awswrangler.catalog.tables

awswrangler.catalog.**tables**(*limit: int = 100*, *catalog_id: Optional[str] = None*, *database: Optional[str] = None*, *search_text: Optional[str] = None*, *name_contains: Optional[str] = None*, *name_prefix: Optional[str] = None*, *name_suffix: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

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

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **limit** (*int, optional*) – Max number of tables to be returned.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **database** (*str, optional*) – Database name.
- **search_text** (*str, optional*) – Select only tables with the given string in table's properties.
- **name_contains** (*str, optional*) – Select by a specific string on table name
- **name_prefix** (*str, optional*) – Select by a specific prefix on table name
- **name_suffix** (*str, optional*) – Select by a specific suffix on table name
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Pandas Dataframe filled by formatted infos.

Return type pandas.DataFrame

Examples

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

awswrangler.catalog.upsert_table_parameters

awswrangler.catalog.**upsert_table_parameters**(*parameters: Dict[str, str]*, *database: str*, *table: str*, *catalog_versioning: bool = False*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Insert or Update the received parameters.

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

- catalog_id
- database

Check out the Global Configurations Tutorial for details.

Parameters

- **parameters** (*Dict[str, str]*) – e.g. {"source": "mysql", "destination": "datalake"}
- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **catalog_versioning** (*bool*) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns All parameters after the upsert.

Return type Dict[str, str]

Examples

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

1.4.3 Amazon Athena

create_athena_bucket([boto3_session])	Create the default Athena bucket if it doesn't exist.
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_work_group(workgroup[, boto3_session])	Return information about the workgroup with the specified name.
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, s3_output, ...])	Run the Hive's metastore consistency check: 'MSCK REPAIR TABLE table;'.
start_query_execution(sql[, database, ...])	Start a SQL Query against AWS Athena.
stop_query_execution(query_execution_id[, ...])	Stop a query execution.
wait_query(query_execution_id[, boto3_session])	Wait for the query end.

awswrangler.athena.create_athena_bucket

awswrangler.athena.**create_athena_bucket**(*boto3_session: Optional[boto3.session.Session] = None*) → str

Create the default Athena bucket if it doesn't exist.

Parameters **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

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

Return type str

Examples

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

awswrangler.athena.get_query_columns_types

awswrangler.athena.**get_query_columns_types**(*query_execution_id: str*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, str]

Get the data type of all columns queried.

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

Parameters

- **query_execution_id** (*str*) – Athena query execution ID.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Dictionary with all data types.

Return type Dict[str, str]

Examples

```
>>> import awswrangler as wr
>>> wr.athena.get_query_columns_types('query-execution-id')
{'col0': 'int', 'col1': 'double'}
```

awswrangler.athena.get_query_execution

awswrangler.athena.**get_query_execution**(*query_execution_id: str*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Fetch query execution details.

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

Parameters

- **query_execution_id** (*str*) – Athena query execution ID.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Dictionary with the get_query_execution response.

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.get_query_execution(query_execution_id='query-execution-id')
```

awswrangler.athena.get_work_group

awswrangler.athena.**get_work_group**(*workgroup: str*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Return information about the workgroup with the specified name.

Parameters

- **workgroup** (*str*) – Work Group name.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

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

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.get_work_group(workgroup='workgroup_name')
```

awswrangler.athena.read_sql_query

awswrangler.athena.**read_sql_query**(*sql: str*, *database: str*, *ctas_approach: bool = True*, *categories: Optional[List[str]] = None*, *chunksize: Optional[Union[int, bool]] = None*, *s3_output: Optional[str] = None*, *workgroup: Optional[str] = None*, *encryption: Optional[str] = None*, *kms_key: Optional[str] = None*, *keep_files: bool = True*, *ctas_database_name: Optional[str] = None*, *ctas_temp_table_name: Optional[str] = None*, *ctas_bucketing_info: Optional[Tuple[List[str], int]] = None*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *max_cache_seconds: int = 0*, *max_cache_query_inspections: int = 50*, *max_remote_cache_entries: int = 50*, *max_local_cache_entries: int = 100*, *data_source: Optional[str] = None*, *params: Optional[Dict[str, Any]] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*) → Any

Execute any SQL query on AWS Athena and return the results as a Pandas DataFrame.

Related tutorial:

- Amazon Athena
- Athena Cache
- Global Configurations

There are two approaches to be defined through ctas_approach parameter:

1 - ctas_approach=True (Default):

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

PROS:

- Faster for mid and big result sizes.
- Can handle some level of nested types.

CONS:

- Requires create/delete table permissions on Glue.
- Does not support timestamp with time zone
- Does not support columns with repeated names.
- Does not support columns with undefined data types.
- A temporary table will be created and then deleted immediately.
- Does not support custom data_source/catalog_id.

2 - ctas_approach=False:

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

PROS:

- Faster for small result sizes (less latency).
- Does not require create/delete table permissions on Glue
- Supports timestamp with time zone.
- Support custom data_source/catalog_id.

CONS:

- Slower for big results (But stills faster than other libraries that uses the regular Athena's API)
- Does not handle nested types at all.

Note: The resulting DataFrame (or every DataFrame in the returned Iterator for chunked queries) have a *query_metadata* attribute, which brings the query result metadata returned by Boto3/Athena .

For a practical example check out the related tutorial!

Note: Valid encryption modes: [None, 'SSE_S3', 'SSE_KMS'].

P.S. 'CSE_KMS' is not supported.

Note: Create the default Athena bucket if it doesn't exist and s3_output is None.

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

Note: *chunksize* argument (Memory Friendly) (i.e batching):

Return an Iterable of DataFrames instead of a regular DataFrame.

There are two batching strategies:

- If **chunksize=True**, a new DataFrame will be returned for each file in the query result.

- If **chunksize=INTEGER**, Wrangler will iterate on the data by number of rows igual the received INTEGER.

P.S. chunksize=True is faster and uses less memory while *chunksize=INTEGER* is more precise in number of rows for each Dataframe.

P.P.S. If *ctas_approach=False* and *chunksize=True*, you will always receive an interador with a single DataFrame because regular Athena queries only produces a single output file.

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

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

- ctas_approach
- database
- max_cache_query_inspections
- max_cache_seconds
- max_remote_cache_entries
- max_local_cache_entries
- workgroup
- chunksize

Check out the Global Configurations Tutorial for details.

Parameters

- **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.
- **categories** (*List[str], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **chunksize** (*Union[int, bool], optional*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* wrangler will iterate on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed Wrangler will iterate on the data by number of rows igual the received INTEGER.
- **s3_output** (*str, optional*) – Amazon S3 path.
- **workgroup** (*str, optional*) – Athena workgroup.
- **encryption** (*str, optional*) – Valid values: [None, 'SSE_S3', 'SSE_KMS']. Notice: 'CSE_KMS' is not supported.
- **kms_key** (*str, optional*) – For SSE-KMS, this is the KMS key ARN or ID.

- **keep_files** (*bool*) – Should Wrangler delete or keep the staging files produced by Athena?
- **ctas_database_name** (*str, optional*) – The name of the alternative database where the CTAS temporary table is stored. If None, the default *database* is used.
- **ctas_temp_table_name** (*str, optional*) – The name of the temporary table and also the directory name on S3 where the CTAS result is stored. If None, it will use the follow random pattern: *f"temp_table_{uuid.uuid4().hex()}"*. On S3 this directory will be under under the pattern: *f"{s3_output}/{ctas_temp_table_name}/"*.
- **ctas_bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **max_cache_seconds** (*int*) – Wrangler can look up in Athena's history if this query has been run before. If so, and its completion time is less than *max_cache_seconds* before now, wrangler skips query execution and just returns the same results as last time. If cached results are valid, wrangler ignores the *ctas_approach*, *s3_output*, *encryption*, *kms_key*, *keep_files* and *ctas_temp_table_name* params. If reading cached data fails for any reason, execution falls back to the usual query run path.
- **max_cache_query_inspections** (*int*) – Max number of queries that will be inspected from the history to try to find some result to reuse. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if max_cache_seconds > 0.
- **max_remote_cache_entries** (*int*) – Max number of queries that will be retrieved from AWS for cache inspection. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if max_cache_seconds > 0 and default value is 50.
- **max_local_cache_entries** (*int*) – Max number of queries for which metadata will be cached locally. This will reduce the latency and also enables keeping more than *max_remote_cache_entries* available for the cache. This value should not be smaller than max_remote_cache_entries. Only takes effect if max_cache_seconds > 0 and default value is 100.
- **data_source** (*str, optional*) – Data Source / Catalog name. If None, 'AwsDataCatalog' will be used by default.
- **params** (*Dict[str, any], optional*) – Dict of parameters that will be used for constructing the SQL query. Only named parameters are supported. The dict needs to contain the information in the form {'name': 'value'} and the SQL query needs to contain *:name;*. Note that for varchar columns and similar, you must surround the value in single quotes.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}

Returns Pandas DataFrame or Generator of Pandas DataFrames if chunksize is passed.

Return type Union[pd.DataFrame, Iterator[pd.DataFrame]]

Examples

```
>>> import awswrangler as wr
>>> df = wr.athena.read_sql_query(sql="...", database="...")
>>> scanned_bytes = df.query_metadata["Statistics"]["DataScannedInBytes"]
```

```
>>> import awswrangler as wr
>>> df = wr.athena.read_sql_query(
...     sql="SELECT * FROM my_table WHERE name=:name; AND city=:city;",
...     params={"name": "'filtered_name'", "city": "'filtered_city'"}
... )
```

awswrangler.athena.read_sql_table

awswrangler.athena.**read_sql_table**(*table: str*, *database: str*, *ctas_approach: bool = True*, *categories: Optional[List[str]] = None*, *chunksize: Optional[Union[int, bool]] = None*, *s3_output: Optional[str] = None*, *workgroup: Optional[str] = None*, *encryption: Optional[str] = None*, *kms_key: Optional[str] = None*, *keep_files: bool = True*, *ctas_database_name: Optional[str] = None*, *ctas_temp_table_name: Optional[str] = None*, *ctas_bucketing_info: Optional[Tuple[List[str], int]] = None*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *max_cache_seconds: int = 0*, *max_cache_query_inspections: int = 50*, *max_remote_cache_entries: int = 50*, *max_local_cache_entries: int = 100*, *data_source: Optional[str] = None*, *s3_additional_kwargs: Optional[Dict[str, Any]] = None*) → Any

Extract the full table AWS Athena and return the results as a Pandas DataFrame.

Related tutorial:

- Amazon Athena
- Athena Cache
- Global Configurations

There are two approaches to be defined through ctas_approach parameter:

1 - ctas_approach=True (Default):

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

PROS:

- Faster for mid and big result sizes.
- Can handle some level of nested types.

CONS:

- Requires create/delete table permissions on Glue.
- Does not support timestamp with time zone
- Does not support columns with repeated names.
- Does not support columns with undefined data types.
- A temporary table will be created and then deleted immediately.

2 - ctas_approach=False:

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

PROS:

- Faster for small result sizes (less latency).
- Does not require create/delete table permissions on Glue
- Supports timestamp with time zone.

CONS:

- Slower for big results (But stills faster than other libraries that uses the regular Athena's API)
- Does not handle nested types at all.

Note: The resulting DataFrame (or every DataFrame in the returned Iterator for chunked queries) have a *query_metadata* attribute, which brings the query result metadata returned by Boto3/Athena .

For a practical example check out the related tutorial!

Note: Valid encryption modes: [None, 'SSE_S3', 'SSE_KMS'].

P.S. 'CSE_KMS' is not supported.

Note: Create the default Athena bucket if it doesn't exist and s3_output is None.

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

Note: *chunksize* argument (Memory Friendly) (i.e batching):

Return an Iterable of DataFrames instead of a regular DataFrame.

There are two batching strategies:

- If **chunksize=True**, a new DataFrame will be returned for each file in the query result.
- If **chunksize=INTEGER**, Wrangler will iterate on the data by number of rows igual the received INTEGER.

P.S. chunksize=True is faster and uses less memory while *chunksize=INTEGER* is more precise in number of rows for each Dataframe.

P.P.S. If *ctas_approach=False* and *chunksize=True*, you will always receive an interador with a single DataFrame because regular Athena queries only produces a single output file.

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

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

- ctas_approach

- database
- max_cache_query_inspections
- max_cache_seconds
- max_remote_cache_entries
- max_local_cache_entries
- workgroup
- chunksize

Check out the Global Configurations Tutorial for details.

Parameters

- **table** (*str*) – Table name.
- **database** (*str*) – AWS Glue/Athena database name.
- **ctas_approach** (*bool*) – Wraps the query using a CTAS, and read the resulted parquet data on S3. If false, read the regular CSV on S3.
- **categories** (*List[str], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **chunksize** (*Union[int, bool], optional*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* wrangler will iterate on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed Wrangler will iterate on the data by number of rows igual the received INTEGER.
- **s3_output** (*str, optional*) – AWS S3 path.
- **workgroup** (*str, optional*) – Athena workgroup.
- **encryption** (*str, optional*) – Valid values: [None, 'SSE_S3', 'SSE_KMS']. Notice: 'CSE_KMS' is not supported.
- **kms_key** (*str, optional*) – For SSE-KMS, this is the KMS key ARN or ID.
- **keep_files** (*bool*) – Should Wrangler delete or keep the staging files produced by Athena?
- **ctas_database_name** (*str, optional*) – The name of the alternative database where the CTAS temporary table is stored. If None, the default *database* is used.
- **ctas_temp_table_name** (*str, optional*) – The name of the temporary table and also the directory name on S3 where the CTAS result is stored. If None, it will use the follow random pattern: *f"temp_table_{uuid.uuid4().hex}"*. On S3 this directory will be under under the pattern: *f"{s3_output}/{ctas_temp_table_name}/"*.
- **ctas_bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **max_cache_seconds** (*int*) – Wrangler 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,

wrangler skips query execution and just returns the same results as last time. If cached results are valid, wrangler ignores the *ctas_approach*, *s3_output*, *encryption*, *kms_key*, *keep_files* and *ctas_temp_table_name* params. If reading cached data fails for any reason, execution falls back to the usual query run path.

- **max_cache_query_inspections** (*int*) – Max number of queries that will be inspected from the history to try to find some result to reuse. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if max_cache_seconds > 0.
- **max_remote_cache_entries** (*int*) – Max number of queries that will be retrieved from AWS for cache inspection. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if max_cache_seconds > 0 and default value is 50.
- **max_local_cache_entries** (*int*) – Max number of queries for which metadata will be cached locally. This will reduce the latency and also enables keeping more than *max_remote_cache_entries* available for the cache. This value should not be smaller than max_remote_cache_entries. Only takes effect if max_cache_seconds > 0 and default value is 100.
- **data_source** (*str, optional*) – Data Source / Catalog name. If None, 'AwsDataCatalog' will be used by default.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}

Returns Pandas DataFrame or Generator of Pandas DataFrames if chunksize is passed.

Return type Union[pd.DataFrame, Iterator[pd.DataFrame]]

Examples

```
>>> import awswrangler as wr
>>> df = wr.athena.read_sql_table(table="...", database="...")
>>> scanned_bytes = df.query_metadata["Statistics"]["DataScannedInBytes"]
```

awswrangler.athena.repair_table

awswrangler.athena.**repair_table**(*table: str*, *database: Optional[str] = None*, *s3_output: Optional[str] = None*, *workgroup: Optional[str] = None*, *encryption: Optional[str] = None*, *kms_key: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Any

Run the Hive's metastore consistency check: 'MSCK REPAIR TABLE table;'.

Recovers partitions and data associated with partitions. Use this statement when you add partitions to the catalog. It is possible it will take some time to add all partitions. If this operation times out, it will be in an incomplete state where only a few partitions are added to the catalog.

Note: Create the default Athena bucket if it doesn't exist and s3_output is None. (E.g. s3://aws-athena-query-results-ACCOUNT-REGION/)

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

- database
- workgroup

Check out the Global Configurations Tutorial for details.

Parameters

- **table** (*str*) – Table name.
- **database** (*str, optional*) – AWS Glue/Athena database name.
- **s3_output** (*str, optional*) – AWS S3 path.
- **workgroup** (*str, optional*) – Athena workgroup.
- **encryption** (*str, optional*) – None, 'SSE_S3', 'SSE_KMS', 'CSE_KMS'.
- **kms_key** (*str, optional*) – For SSE-KMS and CSE-KMS , this is the KMS key ARN or ID.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Query final state ('SUCCEEDED', 'FAILED', 'CANCELLED').

Return type str

Examples

```
>>> import awswrangler as wr
>>> query_final_state = wr.athena.repair_table(table='...', database='...')
```

awswrangler.athena.start_query_execution

awswrangler.athena.**start_query_execution**(*sql: str*, *database: Optional[str] = None*, *s3_output: Optional[str] = None*, *workgroup: Optional[str] = None*, *encryption: Optional[str] = None*, *kms_key: Optional[str] = None*, *params: Optional[Dict[str, Any]] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *data_source: Optional[str] = None*) → Any

Start a SQL Query against AWS Athena.

Note: Create the default Athena bucket if it doesn't exist and s3_output is None. (E.g. s3://aws-athena-query-results-ACCOUNT-REGION/)

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

- database
- workgroup

Check out the Global Configurations Tutorial for details.

Parameters

- **sql** (*str*) – SQL query.
- **database** (*str, optional*) – AWS Glue/Athena database name.
- **s3_output** (*str, optional*) – AWS S3 path.
- **workgroup** (*str, optional*) – Athena workgroup.
- **encryption** (*str, optional*) – None, 'SSE_S3', 'SSE_KMS', 'CSE_KMS'.
- **kms_key** (*str, optional*) – For SSE-KMS and CSE-KMS , this is the KMS key ARN or ID.
- **params** (*Dict[str, any], optional*) – Dict of parameters that will be used for constructing the SQL query. Only named parameters are supported. The dict needs to contain the information in the form {'name': 'value'} and the SQL query needs to contain *:name;*. Note that for varchar columns and similar, you must surround the value in single quotes.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **data_source** (*str, optional*) – Data Source / Catalog name. If None, 'AwsDataCatalog' will be used by default.

Returns Query execution ID

Return type str

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: Optional[boto3.session.Session] = None*) → None

Stop a query execution.

Requires you to have access to the workgroup in which the query ran.

Parameters

- **query_execution_id** (*str*) – Athena query execution ID.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.athena.stop_query_execution(query_execution_id='query-execution-id')
```

awswrangler.athena.wait_query

awswrangler.athena.**wait_query**(*query_execution_id: str*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Wait for the query end.

Parameters

- **query_execution_id** (*str*) – Athena query execution ID.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Dictionary with the get_query_execution response.

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.wait_query(query_execution_id='query-execution-id')
```

1.4.4 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 Parquet 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: Optional[str] = None*, *secret_id: Optional[str] = None*, *catalog_id: Optional[str] = None*, *dbname: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *ssl: bool = True*, *timeout: Optional[int] = None*, *max_prepared_statements: int = 1000*, *tcp_keepalive: bool = True*) → 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** (*Optional[str]*) – Glue Catalog Connection name.
- **secret_id** (*Optional[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.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (*Optional[str]*) – Optional database name to overwrite the stored one.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **ssl** (*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** (*Optional[int]*) – This is the time in seconds before the connection to the server will time out. The default is None which means no timeout. This parameter is forward to 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

Returns redshift_connector connection.

Return type redshift_connector.Connection

Examples

Fetching Redshift connection from Glue Catalog

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

Fetching Redshift connection from Secrets Manager

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect(secret_id="MY_SECRET")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

awswrangler.redshift.connect_temp

awswrangler.redshift.**connect_temp**(*cluster_identifier: str*, *user: str*, *database: Optional[str] = None*, *duration: int = 900*, *auto_create: bool = True*, *db_groups: Optional[List[str]] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *ssl: bool = True*, *timeout: Optional[int] = None*, *max_prepared_statements: int = 1000*, *tcp_keepalive: bool = True*) → 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, optional*) – The name of a database user.
- **database** (*str, optional*) – Database name. If None, the default Database is used.
- **duration** (*int, optional*) – The number of seconds until the returned temporary password expires. Constraint: minimum 900, maximum 3600. Default: 900
- **auto_create** (*bool*) – Create a database user with the name specified for the user named in user if one does not exist.
- **db_groups** (*List[str], optional*) – A list of the names of existing database groups that the user named in user will join for the current session, in addition to any group memberships for an existing user. If not specified, a new user is added only to PUBLIC.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **ssl** (*bool*) – This governs SSL encryption for TCP/IP sockets. This parameter is forward to redshift_connector. https://github.com/aws/amazon-redshift-python-driver
- **timeout** (*Optional[int]*) – This is the time in seconds before the connection to the server will time out. The default is None which means no timeout. This parameter is forward to 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

Returns redshift_connector connection.

Return type redshift_connector.Connection

Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect_temp(cluster_identifier="my-cluster", user="test")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

awswrangler.redshift.copy

awswrangler.redshift.**copy**(*df: pandas.core.frame.DataFrame*, *path: str*, *con: redshift_connector.core.Connection*, *table: str*, *schema: str*, *iam_role: Optional[str] = None*, *aws_access_key_id: Optional[str] = None*, *aws_secret_access_key: Optional[str] = None*, *aws_session_token: Optional[str] = None*, *index: bool = False*, *dtype: Optional[Dict[str, str]] = None*, *mode: str = 'append'*, *overwrite_method: str = 'drop'*, *diststyle: str = 'AUTO'*, *distkey: Optional[str] = None*, *sortstyle: str = 'COMPOUND'*, *sortkey: Optional[List[str]] = None*, *primary_keys: Optional[List[str]] = None*, *varchar_lengths_default: int = 256*, *varchar_lengths: Optional[Dict[str, int]] = None*, *serialize_to_json: bool = False*, *keep_files: bool = False*, *use_threads: bool = True*, *lock: bool = False*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, str]] = None*, *max_rows_by_file: Optional[int] = 10000000*) → None

Load Pandas DataFrame as a Table on Amazon Redshift using parquet files on S3 as stage.

This is a **HIGH** latency and **HIGH** throughput alternative to *wr.redshift.to_sql()* to load large DataFrames into Amazon Redshift through the ** SQL COPY command**.

This strategy has more overhead and requires more IAM privileges than the regular *wr.redshift.to_sql()* function, so it is only recommended to inserting +1K rows at once.

https://docs.aws.amazon.com/redshift/latest/dg/r_COPY.html

Note: If the table does not exist yet, it will be automatically created for you using the Parquet metadata to infer the columns data types.

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

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame.
- **path** (*str*) – S3 path to write stage files (e.g. s3://bucket_name/any_name/). Note: This path must be empty.
- **con** (*redshift_connector.Connection*) – Use redshift_connector.connect() to use " "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name

- **iam_role** (*str, optional*) – AWS IAM role with the related permissions.
- **aws_access_key_id** (*str, optional*) – The access key for your AWS account.
- **aws_secret_access_key** (*str, optional*) – The secret key for your AWS account.
- **aws_session_token** (*str, optional*) – The session key for your AWS account. This is only needed when you are using temporary credentials.
- **index** (*bool*) – True to store the DataFrame index in file, otherwise False to ignore it.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. Only takes effect if dataset=True. (e.g. {'col name': 'bigint', 'col2 name': 'int'})
- **mode** (*str*) – Append, overwrite or upsert.
- **overwrite_method** (*str*) – Drop, cascade, truncate, or delete. Only applicable in overwrite mode.

  "drop" - `DROP ... RESTRICT` - drops the table. Fails if there are any views that depend on it. "cascade" - `DROP ... CASCADE` - drops the table, and all views that depend on it. "truncate" - `TRUNCATE ...` - truncates the table, but immediatly commits current transaction & starts a new one, hence the overwrite happens in two transactions and is not atomic. "delete" - `DELETE FROM ...` - deletes all rows from the table. Slow relative to the other methods.
- **diststyle** (*str*) – Redshift distribution styles. Must be in ["AUTO", "EVEN", "ALL", "KEY"]. https://docs.aws.amazon.com/redshift/latest/dg/t_Distributing_data.html
- **distkey** (*str, optional*) – Specifies a column name or positional number for the distribution key.
- **sortstyle** (*str*) – Sorting can be "COMPOUND" or "INTERLEAVED". https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html
- **sortkey** (*List[str], optional*) – List of columns to be sorted.
- **primary_keys** (*List[str], optional*) – Primary keys.
- **varchar_lengths_default** (*int*) – The size that will be set for all VARCHAR columns not specified with varchar_lengths.
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. {"col1": 10, "col5": 200}).
- **keep_files** (*bool*) – Should keep stage files?
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **lock** (*bool*) – True to execute LOCK command inside the transaction to force serializable isolation.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}
- **max_rows_by_file** (*int*) – Max number of rows in each file. Default is None i.e. dont split the files. (e.g. 33554432, 268435456)

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> wr.db.copy(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path="s3://bucket/my_parquet_files/",
...     con=con,
...     table="my_table",
...     schema="public"
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
>>> con.close()
```

awswrangler.redshift.copy_from_files

awswrangler.redshift.**copy_from_files**(*path: str*, *con: redshift_connector.core.Connection*, *table: str*, *schema: str*, *iam_role: Optional[str] = None*, *aws_access_key_id: Optional[str] = None*, *aws_secret_access_key: Optional[str] = None*, *aws_session_token: Optional[str] = None*, *parquet_infer_sampling: float = 1.0*, *mode: str = 'append'*, *overwrite_method: str = 'drop'*, *diststyle: str = 'AUTO'*, *distkey: Optional[str] = None*, *sortstyle: str = 'COMPOUND'*, *sortkey: Optional[List[str]] = None*, *primary_keys: Optional[List[str]] = None*, *varchar_lengths_default: int = 256*, *varchar_lengths: Optional[Dict[str, int]] = None*, *serialize_to_json: bool = False*, *path_suffix: Optional[str] = None*, *path_ignore_suffix: Optional[str] = None*, *use_threads: bool = True*, *lock: bool = False*, *commit_transaction: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, str]] = None*) → None

Load Parquet files from S3 to a Table on Amazon Redshift (Through COPY command).

https://docs.aws.amazon.com/redshift/latest/dg/r_COPY.html

Note: If the table does not exist yet, it will be automatically created for you using the Parquet metadata to infer the columns data types.

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

Parameters

- **path** (*str*) – S3 prefix (e.g. s3://bucket/prefix/)
- **con** (*redshift_connector.Connection*) – Use redshift_connector.connect() to use " "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.

- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **iam_role** (*str, optional*) – AWS IAM role with the related permissions.
- **aws_access_key_id** (*str, optional*) – The access key for your AWS account.
- **aws_secret_access_key** (*str, optional*) – The secret key for your AWS account.
- **aws_session_token** (*str, optional*) – The session key for your AWS account. This is only needed when you are using temporary credentials.
- **parquet_infer_sampling** (*float*) – Random sample ratio of files that will have the metadata inspected. Must be *0.0 < sampling <= 1.0*. The higher, the more accurate. The lower, the faster.
- **mode** (*str*) – Append, overwrite or upsert.
- **overwrite_method** (*str*) – Drop, cascade, truncate, or delete. Only applicable in overwrite mode.

  ”drop” - `DROP ... RESTRICT` - drops the table. Fails if there are any views that depend on it. “cascade” - `DROP ... CASCADE` - drops the table, and all views that depend on it. “truncate” - `TRUNCATE ...` - truncates the table, but immediatly commits current transaction & starts a new one, hence the overwrite happens in two transactions and is not atomic. “delete” - `DELETE FROM ...` - deletes all rows from the table. Slow relative to the other methods.
- **diststyle** (*str*) – Redshift distribution styles. Must be in [“AUTO”, “EVEN”, “ALL”, “KEY”]. https://docs.aws.amazon.com/redshift/latest/dg/t_Distributing_data.html
- **distkey** (*str, optional*) – Specifies a column name or positional number for the distribution key.
- **sortstyle** (*str*) – Sorting can be “COMPOUND” or “INTERLEAVED”. https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html
- **sortkey** (*List[str], optional*) – List of columns to be sorted.
- **primary_keys** (*List[str], optional*) – Primary keys.
- **varchar_lengths_default** (*int*) – The size that will be set for all VARCHAR columns not specified with varchar_lengths.
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. {“col1”: 10, “col5”: 200}).
- **serialize_to_json** (*bool*) – Should Wrangler add SERIALIZETOJSON parameter into the COPY command? SERIALIZETOJSON is necessary to load nested data https://docs.aws.amazon.com/redshift/latest/dg/ingest-super.html#copy_json
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be scanned on s3 for the schema extraction (e.g. [“.gz.parquet”, “.snappy.parquet”]). Only has effect during the table creation. If None, will try to read all files. (default)
- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored during the schema extraction. (e.g. [“.csv”, “_SUCCESS”]). Only has effect during the table creation. If None, will try to read all files. (default)
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **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.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> wr.db.copy_from_files(
...     path="s3://bucket/my_parquet_files/",
...     con=con,
...     table="my_table",
...     schema="public"
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
>>> con.close()
```

awswrangler.redshift.read_sql_query

awswrangler.redshift.**read_sql_query**(*sql: str*, *con: redshift_connector.core.Connection*, *index_col: Optional[Union[str, List[str]]] = None*, *params: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None*, *chunksize: Optional[int] = None*, *dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None*, *safe: bool = True*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Return a DataFrame corresponding to the result set of the query string.

Note: For large extractions (1K+ rows) consider the function **wr.redshift.unload()**.

Parameters

- **sql** (*str*) – SQL query.
- **con** (*redshift_connector.Connection*) – Use redshift_connector.connect() to use " "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.
- **index_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.

- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

Returns Result as Pandas DataFrame(s).

Return type Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> df = wr.redshift.read_sql_query(
...     sql="SELECT * FROM public.my_table",
...     con=con
... )
>>> con.close()
```

awswrangler.redshift.read_sql_table

awswrangler.redshift.**read_sql_table**(*table: str*, *con: redshift_connector.core.Connection*, *schema: Optional[str] = None*, *index_col: Optional[Union[str, List[str]]] = None*, *params: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None*, *chunksize: Optional[int] = None*, *dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None*, *safe: bool = True*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Return a DataFrame corresponding the table.

Note: For large extractions (1K+ rows) consider the function **wr.redshift.unload()**.

Parameters

- **table** (*str*) – Table name.
- **con** (*redshift_connector.Connection*) – Use redshift_connector.connect() to use " "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.
- **schema** (*str, optional*) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).
- **index_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.

- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

Returns Result as Pandas DataFrame(s).

Return type Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> df = wr.redshift.read_sql_table(
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

awswrangler.redshift.to_sql

awswrangler.redshift.**to_sql**(*df: pandas.core.frame.DataFrame*, *con: redshift_connector.core.Connection*, *table: str*, *schema: str*, *mode: str = 'append'*, *overwrite_method: str = 'drop'*, *index: bool = False*, *dtype: Optional[Dict[str, str]] = None*, *diststyle: str = 'AUTO'*, *distkey: Optional[str] = None*, *sortstyle: str = 'COMPOUND'*, *sortkey: Optional[List[str]] = None*, *primary_keys: Optional[List[str]] = None*, *varchar_lengths_default: int = 256*, *varchar_lengths: Optional[Dict[str, int]] = None*, *use_column_names: bool = False*, *lock: bool = False*, *chunksize: int = 200*, *commit_transaction: bool = True*) → Any

Write records stored in a DataFrame into Redshift.

Note: For large DataFrames (1K+ rows) consider the function **wr.redshift.copy**().

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

- chunksize

Check out the Global Configurations Tutorial for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

- **con** (*redshift_connector.Connection*) – Use redshift_connector.connect() to use " "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **mode** (*str*) – Append, overwrite or upsert.
- **overwrite_method** (*str*) – Drop, cascade, truncate, or delete. Only applicable in overwrite mode.

  "drop" - `DROP ... RESTRICT` - drops the table. Fails if there are any views that depend on it. "cascade" - `DROP ... CASCADE` - drops the table, and all views that depend on it. "truncate" - `TRUNCATE ...` - truncates the table, but immediatly commits current transaction & starts a new one, hence the overwrite happens in two transactions and is not atomic. "delete" - `DELETE FROM ...` - deletes all rows from the table. Slow relative to the other methods.
- **index** (*bool*) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Redshift types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'VARCHAR(10)', 'col2 name': 'FLOAT'}) diststyle : str Redshift distribution styles. Must be in ["AUTO", "EVEN", "ALL", "KEY"]. https://docs.aws.amazon.com/redshift/latest/dg/t_Distributing_data.html
- **distkey** (*str, optional*) – Specifies a column name or positional number for the distribution key.
- **sortstyle** (*str*) – Sorting can be "COMPOUND" or "INTERLEAVED". https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html
- **sortkey** (*List[str], optional*) – List of columns to be sorted.
- **primary_keys** (*List[str], optional*) – Primary keys.
- **varchar_lengths_default** (*int*) – The size that will be set for all VARCHAR columns not specified with varchar_lengths.
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. {"col1": 10, "col5": 200}).
- **use_column_names** (*bool*) – If set to True, will use the column names of the DataFrame for generating the INSERT SQL Query. E.g. If the DataFrame has two columns *col1* and *col3* and *use_column_names* is True, data will only be inserted into the database columns *col1* and *col3*.
- **lock** (*bool*) – True to execute LOCK command inside the transaction to force serializable isolation.
- **chunksize** (*int*) – Number of rows which are inserted with each SQL query. Defaults to inserting 200 rows per query.
- **commit_transaction** (*bool*) – Whether to commit the transaction. True by default.

Returns None.

Return type None

Examples

Writing to Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> wr.redshift.to_sql(
...     df=df,
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

awswrangler.redshift.unload

awswrangler.redshift.**unload**(*sql: str*, *path: str*, *con: redshift_connector.core.Connection*, *iam_role: Optional[str] = None*, *aws_access_key_id: Optional[str] = None*, *aws_secret_access_key: Optional[str] = None*, *aws_session_token: Optional[str] = None*, *region: Optional[str] = None*, *max_file_size: Optional[float] = None*, *kms_key_id: Optional[str] = None*, *categories: Optional[List[str]] = None*, *chunked: Union[bool, int] = False*, *keep_files: bool = False*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, str]] = None*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.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 Wrangler:

- If **chunked=True**, a new DataFrame will be returned for each file in your path/dataset.
- If **chunked=INTEGER**, Wrangler will iterate on the data by number of rows (equal to the received INTEGER).

P.S. chunked=True is faster and uses less memory while *chunked=INTEGER* is more precise in the number of rows for each Dataframe.

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

Parameters

- **sql** (*str*) – SQL query.
- **path** (*Union[str, List[str]]*) – S3 path to write stage files (e.g. s3://bucket_name/any_name/)
- **con** (*redshift_connector.Connection*) – Use redshift_connector.connect() to use " "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.
- **iam_role** (*str, optional*) – AWS IAM role with the related permissions.
- **aws_access_key_id** (*str, optional*) – The access key for your AWS account.
- **aws_secret_access_key** (*str, optional*) – The secret key for your AWS account.
- **aws_session_token** (*str, optional*) – The session key for your AWS account. This is only needed when you are using temporary credentials.
- **region** (*str, optional*) – Specifies the AWS Region where the target Amazon S3 bucket is located. REGION is required for UNLOAD to an Amazon S3 bucket that isn't in the same AWS Region as the Amazon Redshift cluster. By default, UNLOAD assumes that the target Amazon S3 bucket is located in the same AWS Region as the Amazon Redshift cluster.
- **max_file_size** (*float, optional*) – Specifies the maximum size (MB) of files that UNLOAD creates in Amazon S3. Specify a decimal value between 5.0 MB and 6200.0 MB. If None, the default maximum file size is 6200.0 MB.
- **kms_key_id** (*str, optional*) – Specifies the key ID for an AWS Key Management Service (AWS KMS) key to be used to encrypt data files on Amazon S3.
- **categories** (*List[str], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **keep_files** (*bool*) – Should keep stage files?
- **chunked** (*Union[int, bool]*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* wrangler will iterate on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed Wrangler will iterate on the data by number of rows igual the received INTEGER.
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.

Returns Result as Pandas DataFrame(s).

Return type Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> df = wr.redshift.unload(
...     sql="SELECT * FROM public.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     con=con,
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
>>> con.close()
```

awswrangler.redshift.unload_to_files

awswrangler.redshift.**unload_to_files**(*sql: str*, *path: str*, *con: redshift_connector.core.Connection*, *iam_role: Optional[str] = None*, *aws_access_key_id: Optional[str] = None*, *aws_secret_access_key: Optional[str] = None*, *aws_session_token: Optional[str] = None*, *region: Optional[str] = None*, *unload_format: Optional[str] = None*, *max_file_size: Optional[float] = None*, *kms_key_id: Optional[str] = None*, *manifest: bool = False*, *use_threads: bool = True*, *partition_cols: Optional[List[str]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Unload Parquet files on s3 from a Redshift query result (Through the UNLOAD command).

https://docs.aws.amazon.com/redshift/latest/dg/r_UNLOAD.html

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

Parameters

- **sql** (*str*) – SQL query.
- **path** (*Union[str, List[str]]*) – S3 path to write stage files (e.g. s3://bucket_name/any_name/)
- **con** (*redshift_connector.Connection*) – Use redshift_connector.connect() to use " "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.
- **iam_role** (*str, optional*) – AWS IAM role with the related permissions.
- **aws_access_key_id** (*str, optional*) – The access key for your AWS account.
- **aws_secret_access_key** (*str, optional*) – The secret key for your AWS account.
- **aws_session_token** (*str, optional*) – The session key for your AWS account. This is only needed when you are using temporary credentials.
- **region** (*str, optional*) – Specifies the AWS Region where the target Amazon S3 bucket is located. REGION is required for UNLOAD to an Amazon S3 bucket that isn't in the same AWS Region as the Amazon Redshift cluster. By default, UNLOAD assumes that the target Amazon S3 bucket is located in the same AWS Region as the Amazon Redshift cluster.
- **unload_format** (*str, optional*) – Format of the unloaded S3 objects from the query. Valid values: "CSV", "PARQUET". Case sensitive. Defaults to PARQUET.

- **max_file_size** (*float, optional*) – Specifies the maximum size (MB) of files that UNLOAD creates in Amazon S3. Specify a decimal value between 5.0 MB and 6200.0 MB. If None, the default maximum file size is 6200.0 MB.
- **kms_key_id** (*str, optional*) – Specifies the key ID for an AWS Key Management Service (AWS KMS) key to be used to encrypt data files on Amazon S3.
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads.
- **manifest** (*bool*) – Unload a manifest file on S3.
- **partition_cols** (*List[str], optional*) – Specifies the partition keys for the unload operation.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Return type None

Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> wr.redshift.unload_to_files(
...     sql="SELECT * FROM public.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     con=con,
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
>>> con.close()
```

1.4.5 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: Optional[str] = None*, *secret_id: Optional[str] = None*, *catalog_id: Optional[str] = None*, *dbname: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *ssl_context: Optional[Union[bool, ssl.SSLContext]] = None*, *timeout: Optional[int] = None*, *tcp_keepalive: bool = True*) → pg8000.legacy.Connection

Return a pg8000 connection from a Glue Catalog Connection.

https://github.com/tlocke/pg8000

Parameters

- **connection** (*Optional[str]*) – Glue Catalog Connection name.
- **secret_id** (*Optional[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.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (*Optional[str]*) – Optional database name to overwrite the stored one.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **ssl_context** (*Optional[Union[bool, SSLContext]]*) – This governs SSL encryption for TCP/IP sockets. This parameter is forward to pg8000. https://github.com/tlocke/pg8000#functions
- **timeout** (*Optional[int]*) – This is the time in seconds before the connection to the server will time out. The default is None which means no timeout. This parameter is forward to pg8000. https://github.com/tlocke/pg8000#functions
- **tcp_keepalive** (*bool*) – If True then use TCP keepalive. The default is True. This parameter is forward to pg8000. https://github.com/tlocke/pg8000#functions

Returns pg8000 connection.

Return type pg8000.Connection

Examples

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

awswrangler.postgresql.read_sql_query

awswrangler.postgresql.**read_sql_query**(*sql: str*, *con: pg8000.legacy.Connection*, *index_col: Optional[Union[str, List[str]]] = None*, *params: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None*, *chunksize: Optional[int] = None*, *dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None*, *safe: bool = True*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Return a DataFrame corresponding to the result set of the query string.

Parameters

- **sql** (*str*) – SQL query.
- **con** (*pg8000.Connection*) – Use pg8000.connect() to use credentials directly or wr.postgresql.connect() to fetch it from the Glue Catalog.

- **index_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

Returns Result as Pandas DataFrame(s).

Return type Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> df = wr.postgresql.read_sql_query(
...     sql="SELECT * FROM public.my_table",
...     con=con
... )
>>> con.close()
```

awswrangler.postgresql.read_sql_table

awswrangler.postgresql.**read_sql_table**(*table: str*, *con: pg8000.legacy.Connection*, *schema: Optional[str] = None*, *index_col: Optional[Union[str, List[str]]] = None*, *params: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None*, *chunksize: Optional[int] = None*, *dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None*, *safe: bool = True*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Return a DataFrame corresponding the table.

Parameters

- **table** (*str*) – Table name.
- **con** (*pg8000.Connection*) – Use pg8000.connect() to use credentials directly or wr.postgresql.connect() to fetch it from the Glue Catalog.
- **schema** (*str, optional*) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).
- **index_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).

- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

Returns Result as Pandas DataFrame(s).

Return type Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> df = wr.postgresql.read_sql_table(
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

awswrangler.postgresql.to_sql

awswrangler.postgresql.**to_sql**(*df: pandas.core.frame.DataFrame*, *con: pg8000.legacy.Connection*, *table: str*, *schema: str*, *mode: str = 'append'*, *index: bool = False*, *dtype: Optional[Dict[str, str]] = None*, *varchar_lengths: Optional[Dict[str, int]] = None*, *use_column_names: bool = False*, *chunksize: int = 200*, *upsert_conflict_columns: Optional[List[str]] = None*) → Any

Write records stored in a DataFrame into PostgreSQL.

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

- chunksize

Check out the Global Configurations Tutorial for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **con** (*pg8000.Connection*) – Use pg8000.connect() to use credentials directly or wr.postgresql.connect() to fetch it from the Glue Catalog.
- **table** (*str*) – Table name

- **schema** (*str*) – Schema name
- **mode** (*str*) –

  Append, overwrite or upsert. append: Inserts new records into table. overwrite: Drops table and recreates. upsert: Perform an upsert which checks for conflicts on columns given by *upsert_conflict_columns* and sets the new values on conflicts. Note that *upsert_conflict_columns* is required for this mode.
- **index** (*bool*) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and PostgreSQL types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'TEXT', 'col2 name': 'FLOAT'})
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. {"col1": 10, "col5": 200}).
- **use_column_names** (*bool*) – If set to True, will use the column names of the DataFrame for generating the INSERT SQL Query. E.g. If the DataFrame has two columns *col1* and *col3* and *use_column_names* is True, data will only be inserted into the database columns *col1* and *col3*.
- **chunksize** (*int*) – Number of rows which are inserted with each SQL query. Defaults to inserting 200 rows per query.
- **upsert_conflict_columns** (*List[str], optional*) – This parameter is only supported if *mode* is set top *upsert*. In this case conflicts for the given columns are checked for evaluating the upsert.

Returns None.

Return type None

Examples

Writing to PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> wr.postgresql.to_sql(
...     df=df,
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

1.4.6 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: Optional[str] = None*, *secret_id: Optional[str] = None*, *catalog_id: Optional[str] = None*, *dbname: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *read_timeout: Optional[int] = None*, *write_timeout: Optional[int] = None*, *connect_timeout: int = 10*) → pymysql.connections.Connection[Any]

Return a pymysql connection from a Glue Catalog Connection or Secrets Manager.

https://pymysql.readthedocs.io

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

Parameters

- **connection** (*str*) – Glue Catalog Connection name.
- **secret_id** (*Optional[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.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (*Optional[str]*) – Optional database name to overwrite the stored one.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **read_timeout** (*Optional[int]*) – The timeout for reading from the connection in seconds (default: None - no timeout). This parameter is forward to pymysql. https://pymysql.readthedocs.io/en/latest/modules/connections.html
- **write_timeout** (*Optional[int]*) – The timeout for writing to the connection in seconds (default: None - no timeout) This parameter is forward to pymysql. https://pymysql.readthedocs.io/en/latest/modules/connections.html
- **connect_timeout** (*int*) – Timeout before throwing an exception when connecting. (default: 10, min: 1, max: 31536000) This parameter is forward to pymysql. https://pymysql.readthedocs.io/en/latest/modules/connections.html

Returns pymysql connection.

Return type pymysql.connections.Connection

Examples

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

awswrangler.mysql.read_sql_query

awswrangler.mysql.**read_sql_query**(*sql: str*, *con: pymysql.connections.Connection[Any]*, *index_col: Optional[Union[str, List[str]]] = None*, *params: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None*, *chunksize: Optional[int] = None*, *dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None*, *safe: bool = True*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Return a DataFrame corresponding to the result set of the query string.

Parameters

- **sql** (*str*) – SQL query.
- **con** (*pymysql.connections.Connection*) – Use pymysql.connect() to use credentials directly or wr.mysql.connect() to fetch it from the Glue Catalog.
- **index_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

Returns Result as Pandas DataFrame(s).

Return type Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> df = wr.mysql.read_sql_query(
...     sql="SELECT * FROM test.my_table",
...     con=con
... )
>>> con.close()
```

awswrangler.mysql.read_sql_table

awswrangler.mysql.**read_sql_table**(*table: str*, *con: pymysql.connections.Connection[Any]*, *schema: Optional[str] = None*, *index_col: Optional[Union[str, List[str]]] = None*, *params: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None*, *chunksize: Optional[int] = None*, *dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None*, *safe: bool = True*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Return a DataFrame corresponding the table.

Parameters

- **table** (*str*) – Table name.
- **con** (*pymysql.connections.Connection*) – Use pymysql.connect() to use credentials directly or wr.mysql.connect() to fetch it from the Glue Catalog.
- **schema** (*str, optional*) – Name of SQL schema in database to query. Uses default schema if None.
- **index_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

Returns Result as Pandas DataFrame(s).

Return type Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> df = wr.mysql.read_sql_table(
...     table="my_table",
...     schema="test",
...     con=con
... )
>>> con.close()
```

awswrangler.mysql.to_sql

awswrangler.mysql.**to_sql**(*df: pandas.core.frame.DataFrame*, *con: pymysql.connections.Connection[Any]*, *table: str*, *schema: str*, *mode: str = 'append'*, *index: bool = False*, *dtype: Optional[Dict[str, str]] = None*, *varchar_lengths: Optional[Dict[str, int]] = None*, *use_column_names: bool = False*, *chunksize: int = 200*) → Any

Write records stored in a DataFrame into MySQL.

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

- chunksize

Check out the Global Configurations Tutorial for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **con** (*pymysql.connections.Connection*) – Use pymysql.connect() to use credentials directly or wr.mysql.connect() to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **mode** (*str*) –

  Append, overwrite, upsert_duplicate_key, upsert_replace_into, upsert_distinct.
  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.

- **index** (*bool*) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.

- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and MySQL types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'TEXT', 'col2 name': 'FLOAT'})
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. {"col1": 10, "col5": 200}).
- **use_column_names** (*bool*) – If set to True, will use the column names of the DataFrame for generating the INSERT SQL Query. E.g. If the DataFrame has two columns *col1* and *col3* and *use_column_names* is True, data will only be inserted into the database columns *col1* and *col3*.
- **chunksize** (*int*) – Number of rows which are inserted with each SQL query. Defaults to inserting 200 rows per query.

Returns None.

Return type None

Examples

Writing to MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> wr.mysql.to_sql(
...     df=df,
...     table="my_table",
...     schema="test",
...     con=con
... )
>>> con.close()
```

Microsoft SQL Server

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: Optional[str] = None*, *secret_id: Optional[str] = None*, *catalog_id: Optional[str] = None*, *dbname: Optional[str] = None*, *odbc_driver_version: int = 17*, *boto3_session: Optional[boto3.session.Session] = None*, *timeout: Optional[int] = 0*) → Any

Return a pyodbc connection from a Glue Catalog Connection.

https://github.com/mkleehammer/pyodbc

Parameters

- **connection** (*Optional[str]*) – Glue Catalog Connection name.
- **secret_id** (*Optional[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.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (*Optional[str]*) – Optional database name to overwrite the stored one.
- **odbc_driver_version** (*int*) – Major version of the OBDC Driver version that is installed and should be used.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **timeout** (*Optional[int]*) – This is the time in seconds before the connection to the server will time out. The default is None which means no timeout. This parameter is forwarded to pyodbc. https://github.com/mkleehammer/pyodbc/wiki/The-pyodbc-Module#connect

Returns pyodbc connection.

Return type pyodbc.Connection

Examples

```
>>> import awswrangler as wr
>>> con = wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_
↪version=17)
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

awswrangler.sqlserver.read_sql_query

awswrangler.sqlserver.**read_sql_query**(*sql: str*, *con: pyodbc.Connection*, *index_col: Optional[Union[str, List[str]]] = None*, *params: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None*, *chunksize: Optional[int] = None*, *dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None*, *safe: bool = True*) → Any

Return a DataFrame corresponding to the result set of the query string.

Parameters

- **sql** (*str*) – SQL query.
- **con** (*pyodbc.Connection*) – Use pyodbc.connect() to use credentials directly or wr.sqlserver.connect() to fetch it from the Glue Catalog.
- **index_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.

- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

Returns Result as Pandas DataFrame(s).

Return type Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from Microsoft SQL Server using a Glue Catalog Connections >>> import awswrangler as wr >>> con = wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_version=17) >>> df = wr.sqlserver.read_sql_query(... sql="SELECT * FROM dbo.my_table", ... con=con ...) >>> con.close()

awswrangler.sqlserver.read_sql_table

awswrangler.sqlserver.**read_sql_table**(*table: str*, *con: pyodbc.Connection*, *schema: Optional[str] = None*, *index_col: Optional[Union[str, List[str]]] = None*, *params: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None*, *chunksize: Optional[int] = None*, *dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None*, *safe: bool = True*) → Any

Return a DataFrame corresponding the table.

Parameters

- **table** (*str*) – Table name.
- **con** (*pyodbc.Connection*) – Use pyodbc.connect() to use credentials directly or wr.sqlserver.connect() to fetch it from the Glue Catalog.
- **schema** (*str, optional*) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).
- **index_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

Returns Result as Pandas DataFrame(s).

Return type Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from Microsoft SQL Server using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_
↪version=17)
>>> df = wr.sqlserver.read_sql_table(
...     table="my_table",
...     schema="dbo",
...     con=con
... )
>>> con.close()
```

awswrangler.sqlserver.to_sql

awswrangler.sqlserver.**to_sql**(*df: pandas.core.frame.DataFrame*, *con: pyodbc.Connection*, *table: str*, *schema: str*, *mode: str = 'append'*, *index: bool = False*, *dtype: Optional[Dict[str, str]] = None*, *varchar_lengths: Optional[Dict[str, int]] = None*, *use_column_names: bool = False*, *chunksize: int = 200*) → Any

Write records stored in a DataFrame into Microsoft SQL Server.

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

- chunksize

Check out the Global Configurations Tutorial for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **con** (*pyodbc.Connection*) – Use pyodbc.connect() to use credentials directly or wr.sqlserver.connect() to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **mode** (*str*) – Append or overwrite.
- **index** (*bool*) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Microsoft SQL Server types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'TEXT', 'col2 name': 'FLOAT'})
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. {"col1": 10, "col5": 200}).
- **use_column_names** (*bool*) – If set to True, will use the column names of the DataFrame for generating the INSERT SQL Query. E.g. If the DataFrame has two columns *col1* and *col3* and *use_column_names* is True, data will only be inserted into the database columns *col1* and *col3*.

- **chunksize** (*int*) – Number of rows which are inserted with each SQL query. Defaults to inserting 200 rows per query.

Returns None.

Return type None

Examples

Writing to Microsoft SQL Server using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_
↪version=17)
>>> wr.sqlserver.to_sql(
...     df=df,
...     table="table",
...     schema="dbo",
...     con=con
... )
>>> con.close()
```

1.4.7 Data API Redshift

connect(cluster_id, database[, secret_arn, ...])	Create a Redshift Data API connection.
read_sql_query(sql, con[, database])	Run an SQL query on a RedshiftDataApi connection and return the result as a dataframe.

awswrangler.data_api.redshift.connect

awswrangler.data_api.redshift.**connect**(*cluster_id: str*, *database: str*, *secret_arn: str = ''*, *db_user: str = ''*, ***kwargs: Any*) → awswrangler.data_api.redshift.RedshiftDataApi

Create a Redshift Data API connection.

Parameters

- **cluster_id** (*str*) – Id for the target Redshift cluster.
- **database** (*str*) – Target database name.
- **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.
- ****kwargs** – Any additional kwargs are passed to the underlying RedshiftDataApi class.

Returns

Return type A RedshiftDataApi connection instance that can be used with *wr.redshift.data_api.read_sql_query*.

awswrangler.data_api.redshift.read_sql_query

awswrangler.data_api.redshift.**read_sql_query**(*sql: str*, *con: awswrangler.data_api.redshift.RedshiftDataApi*, *database: Optional[str] = None*) → pandas.core.frame.DataFrame

Run an SQL query on a RedshiftDataApi connection and return the result as a dataframe.

Parameters

- **sql** (*str*) – SQL query to run.
- **database** (*str*) – Database to run query on - defaults to the database specified by *con*.

Returns

Return type A Pandas dataframe containing the query results.

1.4.8 Data API RDS

connect(resource_arn, database[, secret_arn])	Create a RDS Data API connection.
read_sql_query(sql, con[, database])	Run an SQL query on an RdsDataApi connection and return the result as a dataframe.

awswrangler.data_api.rds.connect

awswrangler.data_api.rds.**connect**(*resource_arn: str*, *database: str*, *secret_arn: str = ''*, ***kwargs: Any*) → awswrangler.data_api.rds.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.
- ****kwargs** – Any additional kwargs are passed to the underlying RdsDataApi class.

Returns

Return type A RdsDataApi connection instance that can be used with *wr.rds.data_api.read_sql_query*.

awswrangler.data_api.rds.read_sql_query

awswrangler.data_api.rds.**read_sql_query**(*sql: str*, *con: awswrangler.data_api.rds.RdsDataApi*, *database: Optional[str] = None*) → pandas.core.frame.DataFrame

Run an SQL query on an RdsDataApi connection and return the result as a dataframe.

Parameters

- **sql** (*str*) – SQL query to run.
- **database** (*str*) – Database to run query on - defaults to the database specified by *con*.

Returns

Return type A Pandas dataframe containing the query results.

1.4.9 DynamoDB

delete_items(items, table_name[, boto3_session])	Delete all items in the specified 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[, boto3_session])	Insert all items to the specified DynamoDB table.
put_json(path, table_name[, boto3_session])	Write all items from JSON file to a DynamoDB.

awswrangler.dynamodb.delete_items

awswrangler.dynamodb.**delete_items**(*items: List[Dict[str, Any]]*, *table_name: str*, *boto3_session: Optional[boto3.session.Session] = 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** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

Writing rows of DataFrame

```
>>> import awswrangler as wr
>>> wr.dynamodb.delete_items(
...     items=[{'key': 1}, {'key': 2, 'value': 'Hello'}],
...     table_name='table'
... )
```

awswrangler.dynamodb.get_table

awswrangler.dynamodb.**get_table**(*table_name: str*, *boto3_session: Optional[boto3.session.Session] = None*) → boto3.resource

Get DynamoDB table object for specified table name.

Parameters

- **table_name** (*str*) – Name of the Amazon DynamoDB table.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns **dynamodb_table** – Boto3 DynamoDB.Table object. https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/dynamodb.html#DynamoDB.Table

Return type boto3.resources.dynamodb.Table

awswrangler.dynamodb.put_csv

awswrangler.dynamodb.**put_csv**(*path: Union[str, pathlib.Path]*, *table_name: str*, *boto3_session: Optional[boto3.session.Session] = None*, ***pandas_kwargs: Any*) → None

Write all items from a CSV file to a DynamoDB.

Parameters

- **path** (*Union[str, Path]*) – Path as str or Path object to the CSV file which contains the items.
- **table_name** (*str*) – Name of the Amazon DynamoDB table.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.
- **pandas_kwargs** – KEYWORD arguments forwarded to pandas.read_csv(). You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. wr.dynamodb.put_csv('items.csv', 'my_table', sep='|', na_values=['null', 'none'], skip_blank_lines=True) https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

Returns None.

Return type None

Examples

Writing contents of CSV file

```
>>> import awswrangler as wr
>>> wr.dynamodb.put_csv(
...     path='items.csv',
...     table_name='table'
... )
```

Writing contents of CSV file using pandas_kwargs

```
>>> import awswrangler as wr
>>> wr.dynamodb.put_csv(
...     path='items.csv',
...     table_name='table',
...     sep='|',
...     na_values=['null', 'none']
... )
```

awswrangler.dynamodb.put_df

awswrangler.dynamodb.**put_df**(*df: pandas.core.frame.DataFrame*, *table_name: str*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Write all items from a DataFrame to a DynamoDB.

Parameters

- **df** (*pd.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **table_name** (*str*) – Name of the Amazon DynamoDB table.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

Writing rows of DataFrame

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.dynamodb.put_df(
...     df=pd.DataFrame({'key': [1, 2, 3]}),
...     table_name='table'
... )
```

awswrangler.dynamodb.put_items

awswrangler.dynamodb.**put_items**(*items: Union[List[Dict[str, Any]], List[Mapping[str, Any]]]*, *table_name: str*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Insert all items to the specified DynamoDB table.

Parameters

- **items** (*Union[List[Dict[str, Any]], List[Mapping[str, Any]]]*) – List which contains the items that will be inserted.
- **table_name** (*str*) – Name of the Amazon DynamoDB table.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

Writing items

```
>>> import awswrangler as wr
>>> wr.dynamodb.put_items(
...     items=[{'key': 1}, {'key': 2, 'value': 'Hello'}],
...     table_name='table'
... )
```

awswrangler.dynamodb.put_json

awswrangler.dynamodb.**put_json**(*path: Union[str, pathlib.Path]*, *table_name: str*, *boto3_session: Optional[boto3.session.Session] = None*) → 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** (*Union[str, Path]*) – Path as str or Path object to the JSON file which contains the items.
- **table_name** (*str*) – Name of the Amazon DynamoDB table.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

Writing contents of JSON file

```
>>> import awswrangler as wr
>>> wr.dynamodb.put_json(
...     path='items.json',
...     table_name='table'
... )
```

1.4.10 Amazon Timestream

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.
query(sql[, chunked, pagination_config, ...])	Run a query and retrieve the result as a Pandas DataFrame.
write(df, database, table, time_col, ...[, ...])	Store a Pandas DataFrame into a Amazon Timestream table.

awswrangler.timestream.create_database

awswrangler.timestream.**create_database**(*database: str*, *kms_key_id: Optional[str] = None*, *tags: Optional[Dict[str, str]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Create a new Timestream database.

Note: If the KMS key is not specified, the database will be encrypted with a Timestream managed KMS key located in your account.

Parameters

- **database** (*str*) – Database name.
- **kms_key_id** (*Optional[str]*) – The KMS key for the database. If the KMS key is not specified, the database will be encrypted with a Timestream managed KMS key located in your account.
- **tags** (*Optional[Dict[str, str]]*) – Key/Value dict to put on the database. Tags enable you to categorize databases and/or tables, for example, by purpose, owner, or environment. e.g. {"foo": "boo", "bar": "xoo"})
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns The Amazon Resource Name that uniquely identifies this database. (ARN)

Return type str

Examples

Creating a database.

```
>>> import awswrangler as wr
>>> arn = wr.timestream.create_database("MyDatabase")
```

awswrangler.timestream.create_table

awswrangler.timestream.**create_table**(*database: str*, *table: str*, *memory_retention_hours: int*, *magnetic_retention_days: int*, *tags: Optional[Dict[str, str]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Create a new Timestream database.

Note: If the KMS key is not specified, the database will be encrypted with a Timestream managed KMS key located in your account.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **memory_retention_hours** (*int*) – The duration for which data must be stored in the memory store.

- **magnetic_retention_days** (*int*) – The duration for which data must be stored in the magnetic store.
- **tags** (*Optional[Dict[str, str]]*) – Key/Value dict to put on the table. Tags enable you to categorize databases and/or tables, for example, by purpose, owner, or environment. e.g. {"foo": "boo", "bar": "xoo"})
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns The Amazon Resource Name that uniquely identifies this database. (ARN)

Return type str

Examples

Creating a table.

```
>>> import awswrangler as wr
>>> arn = wr.timestream.create_table(
...     database="MyDatabase",
...     table="MyTable",
...     memory_retention_hours=3,
...     magnetic_retention_days=7
... )
```

awswrangler.timestream.delete_database

awswrangler.timestream.**delete_database**(*database: str*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Delete a given Timestream database. This is an irreversible operation.

After a database is deleted, the time series data from its tables cannot be recovered.

All tables in the database must be deleted first, or a ValidationException error will be thrown.

Due to the nature of distributed retries, the operation can return either success or a ResourceNotFoundException. Clients should consider them equivalent.

Parameters

- **database** (*str*) – Database name.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

Deleting a database

```
>>> import awswrangler as wr
>>> arn = wr.timestream.delete_database("MyDatabase")
```

awswrangler.timestream.delete_table

awswrangler.timestream.**delete_table**(*database: str*, *table: str*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Delete a given Timestream table.

This is an irreversible operation.

After a Timestream database table is deleted, the time series data stored in the table cannot be recovered.

Due to the nature of distributed retries, the operation can return either success or a ResourceNotFoundException. Clients should consider them equivalent.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

Deleting a table

```
>>> import awswrangler as wr
>>> arn = wr.timestream.delete_table("MyDatabase", "MyTable")
```

awswrangler.timestream.query

awswrangler.timestream.**query**(*sql: str*, *chunked: bool = False*, *pagination_config: Optional[Dict[str, Any]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Run a query and retrieve the result as a Pandas DataFrame.

Parameters

- **sql** (*str*) – SQL query.
- **chunked** (*bool*) – If True returns dataframe iterator, and a single dataframe otherwise. False by default.
- **pagination_config** (*Dict[str, Any], optional*) – Pagination configuration dictionary of a form {'MaxItems': 10, 'PageSize': 10, 'StartingToken': '...'}

- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

Return type pd.DataFrame

Examples

Run a query and return the result as a Pandas DataFrame or an iterable.

```
>>> import awswrangler as wr
>>> df = wr.timestream.query('SELECT * FROM "sampleDB"."sampleTable" ORDER BY time␣
↪DESC LIMIT 10')
```

awswrangler.timestream.write

awswrangler.timestream.**write**(*df: pandas.core.frame.DataFrame*, *database: str*, *table: str*, *time_col: str*, *measure_col: str*, *dimensions_cols: List[str]*, *version: int = 1*, *num_threads: int = 32*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, str]]

Store a Pandas DataFrame into a Amazon Timestream table.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **database** (*str*) – Amazon Timestream database name.
- **table** (*str*) – Amazon Timestream table name.
- **time_col** (*str*) – DataFrame column name to be used as time. MUST be a timestamp column.
- **measure_col** (*str*) – DataFrame column name to be used as measure.
- **dimensions_cols** (*List[str]*) – List of DataFrame column names to be used as dimensions.
- **version** (*int*) – Version number used for upserts. Documentation https://docs.aws.amazon.com/timestream/latest/developerguide/API_WriteRecords.html.
- **num_threads** (*str*) – Number of thread to be used for concurrent writing.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns Rejected records.

Return type List[Dict[str, str]]

Examples

Store a Pandas DataFrame into a Amazon Timestream table.

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> df = pd.DataFrame(
>>>     {
>>>         "time": [datetime.now(), datetime.now(), datetime.now()],
>>>         "dim0": ["foo", "boo", "bar"],
>>>         "dim1": [1, 2, 3],
>>>         "measure": [1.0, 1.1, 1.2],
>>>     }
>>> )
>>> rejected_records = wr.timestream.write(
>>>     df=df,
>>>     database="sampleDB",
>>>     table="sampleTable",
>>>     time_col="time",
>>>     measure_col="measure",
>>>     dimensions_cols=["dim0", "dim1"],
>>> )
>>> assert len(rejected_records) == 0
```

1.4.11 Amazon EMR

build_spark_step(path[, 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[, ...])	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*, *deploy_mode: str = 'cluster'*, *docker_image: Optional[str] = None*, *name: str = 'my-step'*, *action_on_failure: str = 'CONTINUE'*, *region: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Build the Step structure (dictionary).

Parameters

- **path** (*str*) – Script path. (e.g. s3://bucket/app.py)
- **deploy_mode** (*str*) – "cluster" | "client"

- **docker_image** (*str, optional*) – e.g. "{ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{IMAGE_NAM
- **name** (*str, optional*) – Step name.
- **action_on_failure** (*str*) – 'TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'
- **region** (*str, optional*) – Region name to not get it from boto3.Session. (e.g. *us-east-1*)
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Step structure.

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> step_id = wr.emr.submit_steps(
>>>     cluster_id="cluster-id",
>>>     steps=[
>>>         wr.emr.build_spark_step(path="s3://bucket/app.py")
>>>     ]
>>> )
```

awswrangler.emr.build_step

awswrangler.emr.**build_step**(*command: str*, *name: str = 'my-step'*, *action_on_failure: str = 'CONTINUE'*, *script: bool = False*, *region: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Build the Step structure (dictionary).

Parameters

- **command** (*str*) – e.g. 'echo "Hello!"' e.g. for script 's3://.../script.sh arg1 arg2'
- **name** (*str, optional*) – Step name.
- **action_on_failure** (*str*) – 'TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'
- **script** (*bool*) – False for raw command or True for script runner. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-commandrunner.html
- **region** (*str, optional*) – Region name to not get it from boto3.Session. (e.g. *us-east-1*)
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Step structure.

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> steps = []
>>> for cmd in ['echo "Hello"', "ls -la"]:
...     steps.append(wr.emr.build_step(name=cmd, command=cmd))
>>> wr.emr.submit_steps(cluster_id="cluster-id", steps=steps)
```

awswrangler.emr.create_cluster

awswrangler.emr.**create_cluster**(*subnet_id: str*, *cluster_name: str = 'my-emr-cluster'*, *logging_s3_path: Optional[str] = None*, *emr_release: str = 'emr-6.0.0'*, *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: Optional[List[str]] = None*, *debugging: bool = True*, *applications: Optional[List[str]] = None*, *visible_to_all_users: bool = True*, *key_pair_name: Optional[str] = None*, *security_group_master: Optional[str] = None*, *security_groups_master_additional: Optional[List[str]] = None*, *security_group_slave: Optional[str] = None*, *security_groups_slave_additional: Optional[List[str]] = None*, *security_group_service_access: Optional[str] = None*, *docker: bool = False*, *extra_public_registries: Optional[List[str]] = None*, *spark_log_level: str = 'WARN'*, *spark_jars_path: Optional[List[str]] = None*, *spark_defaults: Optional[Dict[str, str]] = None*, *spark_pyarrow: bool = False*, *custom_classifications: Optional[List[Dict[str, Any]]] = None*, *maximize_resource_allocation: bool = False*, *steps: Optional[List[Dict[str, Any]]] = None*, *keep_cluster_alive_when_no_steps: bool = True*, *termination_protected: bool = False*, *tags: Optional[Dict[str, str]] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Create a EMR cluster with instance fleets configuration.

https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-instance-fleet.html

Parameters

- **subnet_id** (*str*) – VPC subnet ID.

- **cluster_name** (*str*) – Cluster name.
- **logging_s3_path** (*str, optional*) – Logging s3 path (e.g. s3://BUCKET_NAME/DIRECTORY_NAME/). If None, the default is *s3://aws-logs-{AccountId}-{RegionId}/elasticmapreduce/*
- **emr_release** (*str*) – EMR release (e.g. emr-5.28.0).
- **emr_ec2_role** (*str*) – IAM role name.
- **emr_role** (*str*) – IAM role name.
- **instance_type_master** (*str*) – EC2 instance type.
- **instance_type_core** (*str*) – EC2 instance type.
- **instance_type_task** (*str*) – EC2 instance type.
- **instance_ebs_size_master** (*int*) – Size of EBS in GB.
- **instance_ebs_size_core** (*int*) – Size of EBS in GB.
- **instance_ebs_size_task** (*int*) – Size of EBS in GB.
- **instance_num_on_demand_master** (*int*) – Number of on demand instances.
- **instance_num_on_demand_core** (*int*) – Number of on demand instances.
- **instance_num_on_demand_task** (*int*) – Number of on demand instances.
- **instance_num_spot_master** (*int*) – Number of spot instances.
- **instance_num_spot_core** (*int*) – Number of spot instances.
- **instance_num_spot_task** (*int*) – Number of spot instances.
- **spot_bid_percentage_of_on_demand_master** (*int*) – The bid price, as a percentage of On-Demand price.
- **spot_bid_percentage_of_on_demand_core** (*int*) – The bid price, as a percentage of On-Demand price.
- **spot_bid_percentage_of_on_demand_task** (*int*) – The bid price, as a percentage of On-Demand price.
- **spot_provisioning_timeout_master** (*int*) – The spot provisioning timeout period in minutes. If Spot instances are not provisioned within this time period, the TimeOutAction is taken. Minimum value is 5 and maximum value is 1440. The timeout applies only during initial provisioning, when the cluster is first created.
- **spot_provisioning_timeout_core** (*int*) – The spot provisioning timeout period in minutes. If Spot instances are not provisioned within this time period, the TimeOutAction is taken. Minimum value is 5 and maximum value is 1440. The timeout applies only during initial provisioning, when the cluster is first created.
- **spot_provisioning_timeout_task** (*int*) – The spot provisioning timeout period in minutes. If Spot instances are not provisioned within this time period, the TimeOutAction is taken. Minimum value is 5 and maximum value is 1440. The timeout applies only during initial provisioning, when the cluster is first created.
- **spot_timeout_to_on_demand_master** (*bool*) – After a provisioning timeout should the cluster switch to on demand or shutdown?
- **spot_timeout_to_on_demand_core** (*bool*) – After a provisioning timeout should the cluster switch to on demand or shutdown?

- **spot_timeout_to_on_demand_task** (*bool*) – After a provisioning timeout should the cluster switch to on demand or shutdown?
- **python3** (*bool*) – Python 3 Enabled?
- **spark_glue_catalog** (*bool*) – Spark integration with Glue Catalog?
- **hive_glue_catalog** (*bool*) – Hive integration with Glue Catalog?
- **presto_glue_catalog** (*bool*) – Presto integration with Glue Catalog?
- **consistent_view** (*bool*) – Consistent view allows EMR clusters to check for list and read-after-write consistency for Amazon S3 objects written by or synced with EMRFS. https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-consistent-view.html
- **consistent_view_retry_seconds** (*int*) – Delay between the tries (seconds).
- **consistent_view_retry_count** (*int*) – Number of tries.
- **consistent_view_table_name** (*str*) – Name of the DynamoDB table to store the consistent view data.
- **bootstraps_paths** (*List[str], optional*) – Bootstraps paths (e.g ["s3://BUCKET_NAME/script.sh"]).
- **debugging** (*bool*) – Debugging enabled?
- **applications** (*List[str], optional*) – List of applications (e.g ["Hadoop", "Spark", "Ganglia", "Hive"]). If None, ["Spark"] will be considered.
- **visible_to_all_users** (*bool*) – True or False.
- **key_pair_name** (*str, optional*) – Key pair name.
- **security_group_master** (*str, optional*) – The identifier of the Amazon EC2 security group for the master node.
- **security_groups_master_additional** (*str, optional*) – A list of additional Amazon EC2 security group IDs for the master node.
- **security_group_slave** (*str, optional*) – The identifier of the Amazon EC2 security group for the core and task nodes.
- **security_groups_slave_additional** (*str, optional*) – A list of additional Amazon EC2 security group IDs for the core and task nodes.
- **security_group_service_access** (*str, optional*) – The identifier of the Amazon EC2 security group for the Amazon EMR service to access clusters in VPC private subnets.
- **docker** (*bool*) – Enable Docker Hub and ECR registries access.
- **extra_public_registries** (*List[str], optional*) – Additional docker registries.
- **spark_log_level** (*str*) – log4j.rootCategory log level (ALL, DEBUG, INFO, WARN, ERROR, FATAL, OFF, TRACE).
- **spark_jars_path** (*List[str], optional*) – spark.jars e.g. [s3://.../foo.jar, s3://.../boo.jar] https://spark.apache.org/docs/latest/configuration.html
- **spark_defaults** (*Dict[str, str], optional*) – https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html#spark-defaults
- **spark_pyarrow** (*bool*) – Enable PySpark to use PyArrow behind the scenes. P.S. You must install pyarrow by your self via bootstrap

- **custom_classifications** (*List[Dict[str, Any]], optional*) – Extra classifications.
- **maximize_resource_allocation** (*bool*) – Configure your executors to utilize the maximum resources possible https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html#emr-spark-maximizeresourceallocation
- **steps** (*List[Dict[str, Any]], optional*) – Steps definitions (Obs : str Use EMR.build_step() to build it)
- **keep_cluster_alive_when_no_steps** (*bool*) – Specifies whether the cluster should remain available after completing all steps
- **termination_protected** (*bool*) – Specifies whether the Amazon EC2 instances in the cluster are protected from termination by API calls, user intervention, or in the event of a job-flow error.
- **tags** (*Dict[str, str], optional*) – Key/Value collection to put on the Cluster. e.g. {"foo": "boo", "bar": "xoo"})
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Cluster ID.

Return type str

Examples

Minimal Example

```
>>> import awswrangler as wr
>>> cluster_id = wr.emr.create_cluster("SUBNET_ID")
```

Minimal Example With Custom Classification

```
>>> import awswrangler as wr
>>> cluster_id = wr.emr.create_cluster(
>>> subnet_id="SUBNET_ID",
>>> custom_classifications=[
>>>         {
>>>             "Classification": "livy-conf",
>>>             "Properties": {
>>>                 "livy.spark.master": "yarn",
>>>                 "livy.spark.deploy-mode": "cluster",
>>>                 "livy.server.session.timeout": "16h",
>>>             },
>>>         }
>>>     ],
>>> )
```

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/",
```

(continues on next page)

(continued from previous page)

```
...     emr_release="emr-5.28.0",
...     subnet_id="SUBNET_ID",
...     emr_ec2_role="EMR_EC2_DefaultRole",
...     emr_role="EMR_DefaultRole",
...     instance_type_master="m5.xlarge",
...     instance_type_core="m5.xlarge",
...     instance_type_task="m5.xlarge",
...     instance_ebs_size_master=50,
...     instance_ebs_size_core=50,
...     instance_ebs_size_task=50,
...     instance_num_on_demand_master=1,
...     instance_num_on_demand_core=1,
...     instance_num_on_demand_task=1,
...     instance_num_spot_master=0,
...     instance_num_spot_core=1,
...     instance_num_spot_task=1,
...     spot_bid_percentage_of_on_demand_master=100,
...     spot_bid_percentage_of_on_demand_core=100,
...     spot_bid_percentage_of_on_demand_task=100,
...     spot_provisioning_timeout_master=5,
...     spot_provisioning_timeout_core=5,
...     spot_provisioning_timeout_task=5,
...     spot_timeout_to_on_demand_master=True,
...     spot_timeout_to_on_demand_core=True,
...     spot_timeout_to_on_demand_task=True,
...     python3=True,
...     spark_glue_catalog=True,
...     hive_glue_catalog=True,
...     presto_glue_catalog=True,
...     bootstraps_paths=None,
...     debugging=True,
...     applications=["Hadoop", "Spark", "Ganglia", "Hive"],
...     visible_to_all_users=True,
...     key_pair_name=None,
...     spark_jars_path=[f"s3://...jar"],
...     maximize_resource_allocation=True,
...     keep_cluster_alive_when_no_steps=True,
...     termination_protected=False,
...     spark_pyarrow=True,
...     tags={
...         "foo": "boo"
...     })
```

awswrangler.emr.get_cluster_state

awswrangler.emr.**get_cluster_state**(*cluster_id: str*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Get the EMR cluster state.

Possible states: 'STARTING', 'BOOTSTRAPPING', 'RUNNING', 'WAITING', 'TERMINATING', 'TERMINATED', 'TERMINATED_WITH_ERRORS'

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns State.

Return type str

Examples

```
>>> import 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: Optional[boto3.session.Session] = None*) → str

Get EMR step state.

Possible states: 'PENDING', 'CANCEL_PENDING', 'RUNNING', 'COMPLETED', 'CANCELLED', 'FAILED', 'INTERRUPTED'

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **step_id** (*str*) – Step ID.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns State.

Return type str

Examples

```
>>> import awswrangler as wr
>>> state = wr.emr.get_step_state("cluster-id", "step-id")
```

awswrangler.emr.submit_ecr_credentials_refresh

awswrangler.emr.**submit_ecr_credentials_refresh**(*cluster_id: str*, *path: str*, *action_on_failure: str = 'CONTINUE'*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Update internal ECR credentials.

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **path** (*str*) – Amazon S3 path where Wrangler will stage the script ecr_credentials_refresh.py (e.g. s3://bucket/emr/)
- **action_on_failure** (*str*) – 'TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Step ID.

Return type str

Examples

```
>>> import awswrangler as wr
>>> step_id = wr.emr.submit_ecr_credentials_refresh("cluster_id", "s3://bucket/emr/
↪")
```

awswrangler.emr.submit_spark_step

awswrangler.emr.**submit_spark_step**(*cluster_id: str*, *path: str*, *deploy_mode: str = 'cluster'*, *docker_image: Optional[str] = None*, *name: str = 'my-step'*, *action_on_failure: str = 'CONTINUE'*, *region: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Submit Spark Step.

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **path** (*str*) – Script path. (e.g. s3://bucket/app.py)
- **deploy_mode** (*str*) – "cluster" | "client"
- **docker_image** (*str, optional*) – e.g. "{ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{IMAGE_NAM
- **name** (*str, optional*) – Step name.
- **action_on_failure** (*str*) – 'TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'
- **region** (*str, optional*) – Region name to not get it from boto3.Session. (e.g. *us-east-1*)
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Step ID.

Return type str

Examples

```
>>> import awswrangler as wr
>>> step_id = wr.emr.submit_spark_step(
>>>     cluster_id="cluster-id",
>>>     path="s3://bucket/emr/app.py"
>>> )
```

awswrangler.emr.submit_step

awswrangler.emr.**submit_step**(*cluster_id: str*, *command: str*, *name: str = 'my-step'*, *action_on_failure: str = 'CONTINUE'*, *script: bool = False*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Submit new job in the EMR Cluster.

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **command** (*str*) – e.g. 'echo "Hello!"' e.g. for script 's3://.../script.sh arg1 arg2'
- **name** (*str, optional*) – Step name.
- **action_on_failure** (*str*) – 'TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'
- **script** (*bool*) – True for raw command or False for script runner. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-commandrunner.html
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Step ID.

Return type str

Examples

```
>>> import awswrangler as wr
>>> step_id = wr.emr.submit_step(
...     cluster_id=cluster_id,
...     name="step_test",
...     command="s3://...script.sh arg1 arg2",
...     script=True)
```

awswrangler.emr.submit_steps

awswrangler.emr.**submit_steps**(*cluster_id: str*, *steps: List[Dict[str, Any]]*, *boto3_session: Optional[boto3.session.Session] = None*) → List[str]

Submit a list of steps.

Parameters

- **cluster_id** (*str*) – Cluster ID.

- **steps** (*List[Dict[str, Any]]*) – Steps definitions (Obs: Use EMR.build_step() to build it).
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns List of step IDs.

Return type List[str]

Examples

```
>>> import awswrangler as wr
>>> for cmd in ['echo "Hello"', "ls -la"]:
...     steps.append(wr.emr.build_step(name=cmd, command=cmd))
>>> wr.emr.submit_steps(cluster_id="cluster-id", steps=steps)
```

awswrangler.emr.terminate_cluster

awswrangler.emr.**terminate_cluster**(*cluster_id: str*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Terminate EMR cluster.

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.emr.terminate_cluster("cluster-id")
```

1.4.12 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.

awswrangler.cloudwatch.read_logs

awswrangler.cloudwatch.**read_logs**(*query: str*, *log_group_names: List[str]*, *start_time: datetime.datetime = datetime.datetime(1970, 1, 1, 0, 0, tzinfo=datetime.timezone.utc)*, *end_time: datetime.datetime = datetime.datetime(2021, 8, 11, 18, 25, 42, 24250)*, *limit: Optional[int] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → pandas.core.frame.DataFrame

Run a query against AWS CloudWatchLogs Insights and convert the results to Pandas DataFrame.

https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html

Parameters

- **query** (*str*) – The query string.
- **log_group_names** (*str*) – The list of log groups to be queried. You can include up to 20 log groups.
- **start_time** (*datetime.datetime*) – The beginning of the time range to query.
- **end_time** (*datetime.datetime*) – The end of the time range to query.
- **limit** (*Optional[int]*) – The maximum number of log events to return in the query.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Result as a Pandas DataFrame.

Return type pandas.DataFrame

Examples

```
>>> import awswrangler as wr
>>> df = wr.cloudwatch.read_logs(
...     log_group_names=["loggroup"],
...     query="fields @timestamp, @message | sort @timestamp desc | limit 5",
... )
```

awswrangler.cloudwatch.run_query

awswrangler.cloudwatch.**run_query**(*query: str*, *log_group_names: List[str]*, *start_time: datetime.datetime = datetime.datetime(1970, 1, 1, 0, 0, tzinfo=datetime.timezone.utc)*, *end_time: datetime.datetime = datetime.datetime(2021, 8, 11, 18, 25, 42, 24215)*, *limit: Optional[int] = None*, *boto3_session: Optional[boto3.session.Session] = 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** (*str*) – The list of log groups to be queried. You can include up to 20 log groups.
- **start_time** (*datetime.datetime*) – The beginning of the time range to query.

- **end_time** (*datetime.datetime*) – The end of the time range to query.
- **limit** (*Optional[int]*) – The maximum number of log events to return in the query.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Result.

Return type List[List[Dict[str, str]]]

Examples

```
>>> import awswrangler as wr
>>> result = wr.cloudwatch.run_query(
...     log_group_names=["loggroup"],
...     query="fields @timestamp, @message | sort @timestamp desc | limit 5",
... )
```

awswrangler.cloudwatch.start_query

awswrangler.cloudwatch.**start_query**(*query: str*, *log_group_names: List[str]*, *start_time: datetime.datetime = datetime.datetime(1970, 1, 1, 0, 0, tzinfo=datetime.timezone.utc)*, *end_time: datetime.datetime = datetime.datetime(2021, 8, 11, 18, 25, 42, 24201)*, *limit: Optional[int] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Run a query against AWS CloudWatchLogs Insights.

https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html

Parameters

- **query** (*str*) – The query string.
- **log_group_names** (*str*) – The list of log groups to be queried. You can include up to 20 log groups.
- **start_time** (*datetime.datetime*) – The beginning of the time range to query.
- **end_time** (*datetime.datetime*) – The end of the time range to query.
- **limit** (*Optional[int]*) – The maximum number of log events to return in the query.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Query ID.

Return type str

Examples

```
>>> import awswrangler as wr
>>> query_id = wr.cloudwatch.start_query(
...     log_group_names=["loggroup"],
...     query="fields @timestamp, @message | sort @timestamp desc | limit 5",
... )
```

awswrangler.cloudwatch.wait_query

awswrangler.cloudwatch.**wait_query**(*query_id: str*, *boto3_session: Optional[boto3.session.Session] = None*) → 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** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Query result payload.

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> query_id = wr.cloudwatch.start_query(
...     log_group_names=["loggroup"],
...     query="fields @timestamp, @message | sort @timestamp desc | limit 5",
... )
... response = wr.cloudwatch.wait_query(query_id=query_id)
```

1.4.13 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, boto3_session])	Delete all datasets.
delete_all_templates([account_id, boto3_session])	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.

continues on next page

Table 14 – continued from previous page

delete_template([name, template_id, ...])	Delete a tamplate.
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_name[, ...])	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: Optional[str] = None*, *dataset_id: Optional[str] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Cancel an ongoing ingestion of data into SPICE.

Note: You must pass a not None value for `dataset_name` or `dataset_id` argument.

Parameters

- **ingestion_id** (*str*) – Ingestion ID.

- **dataset_name** (*str, optional*) – Dataset name.
- **dataset_id** (*str, optional*) – Dataset ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>>  wr.quicksight.cancel_ingestion(ingestion_id="...", dataset_name="...")
```

awswrangler.quicksight.create_athena_data_source

awswrangler.quicksight.**create_athena_data_source**(*name: str*, *workgroup: str = 'primary'*, *allowed_to_use: Optional[List[str]] = None*, *allowed_to_manage: Optional[List[str]] = None*, *tags: Optional[Dict[str, str]] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *namespace: str = 'default'*) → None

Create a QuickSight data source pointing to an Athena/Workgroup.

Note: You will not be able to see the the data source in the console if you not pass your user to one of the `allowed_*` arguments.

Parameters

- **name** (*str*) – Data source name.
- **workgroup** (*str*) – Athena workgroup.
- **tags** (*Dict[str, str], optional*) – Key/Value collection to put on the Cluster. e.g. {"foo": "boo", "bar": "xoo"})
- **allowed_to_use** (*optional*) – List of principals that will be allowed to see and use the data source. e.g. ["John"]
- **allowed_to_manage** (*optional*) – List of principals that will be allowed to see, use, update and delete the data source. e.g. ["Mary"]
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **namespace** (*str*) – The namespace. Currently, you should set this to default.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.create_athena_data_source(
...     name="...",
...     allowed_to_manage=["john"]
... )
```

awswrangler.quicksight.create_athena_dataset

awswrangler.quicksight.**create_athena_dataset**(*name: str*, *database: Optional[str] = None*, *table: Optional[str] = None*, *sql: Optional[str] = None*, *sql_name: str = 'CustomSQL'*, *data_source_name: Optional[str] = None*, *data_source_arn: Optional[str] = None*, *import_mode: str = 'DIRECT_QUERY'*, *allowed_to_use: Optional[List[str]] = None*, *allowed_to_manage: Optional[List[str]] = None*, *logical_table_alias: str = 'LogicalTable'*, *rename_columns: Optional[Dict[str, str]] = None*, *cast_columns_types: Optional[Dict[str, str]] = None*, *tags: Optional[Dict[str, str]] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*, *namespace: str = 'default'*) → str

Create a QuickSight dataset.

Note: You will not be able to see the the dataset in the console if you not pass your username to one of the `allowed_*` arguments.

Note: You must pass `database`/`table` OR `sql` argument.

Note: You must pass `data_source_name` OR `data_source_arn` argument.

Parameters

- **name** (*str*) – Dataset name.
- **database** (*str*) – Athena's database name.
- **table** (*str*) – Athena's table name.
- **sql** (*str*) – Use a SQL query to define your table.
- **sql_name** (*str*) – Query name.
- **data_source_name** (*str, optional*) – QuickSight data source name.
- **data_source_arn** (*str, optional*) – QuickSight data source ARN.

- **import_mode** (*str*) – Indicates whether you want to import the data into SPICE. 'SPICE'|'DIRECT_QUERY'
- **tags** (*Dict[str, str], optional*) – Key/Value collection to put on the Cluster. e.g. {"foo": "boo", "bar": "xoo"})
- **allowed_to_use** (*optional*) – List of usernames that will be allowed to see and use the data source. e.g. ["john", "Mary"]
- **allowed_to_manage** (*optional*) – List of usernames that will be allowed to see, use, update and delete the data source. e.g. ["Mary"]
- **logical_table_alias** (*str*) – A display name for the logical table.
- **rename_columns** (*Dict[str, str], optional*) – Dictionary to map column renames. e.g. {"old_name": "new_name", "old_name2": "new_name2"}
- **cast_columns_types** (*Dict[str, str], optional*) – Dictionary to map column casts. e.g. {"col_name": "STRING", "col_name2": "DECIMAL"} Valid types: 'STRING'|'INTEGER'|'DECIMAL'|'DATETIME'
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **namespace** (*str*) – The namespace. Currently, you should set this to default.

Returns Dataset ID.

Return type str

Examples

```
>>> import awswrangler as wr
>>> dataset_id = wr.quicksight.create_athena_dataset(
...     name="...",
...     database="..."
...     table="..."
...     data_source_name="..."
...     allowed_to_manage=["Mary"]
... )
```

awswrangler.quicksight.create_ingestion

awswrangler.quicksight.**create_ingestion**(*dataset_name: Optional[str] = None*, *dataset_id: Optional[str] = None*, *ingestion_id: Optional[str] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Create and starts a new SPICE ingestion on a dataset.

Note: You must pass `dataset_name` OR `dataset_id` argument.

Parameters

- **dataset_name** (*str, optional*) – Dataset name.
- **dataset_id** (*str, optional*) – Dataset ID.
- **ingestion_id** (*str, optional*) – Ingestion ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Ingestion ID

Return type str

Examples

```
>>> import awswrangler as wr
>>> status = wr.quicksight.create_ingestion("my_dataset")
```

awswrangler.quicksight.delete_all_dashboards

awswrangler.quicksight.**delete_all_dashboards**(*account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Delete all dashboards.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns 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: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Delete all data sources.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns 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: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Delete all datasets.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns 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: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Delete all templates.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_all_templates()
```

awswrangler.quicksight.delete_dashboard

awswrangler.quicksight.**delete_dashboard**(*name: Optional[str] = None*, *dashboard_id: Optional[str] = None*, *version_number: Optional[int] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Delete a dashboard.

Note: You must pass a not None `name` or `dashboard_id` argument.

Parameters

- **name** (*str, optional*) – Dashboard name.
- **dashboard_id** (*str, optional*) – The ID for the dashboard.
- **version_number** (*int, optional*) – The version number of the dashboard. If the version number property is provided, only the specified version of the dashboard is deleted.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_dashboard(name="...")
```

awswrangler.quicksight.delete_data_source

awswrangler.quicksight.**delete_data_source**(*name: Optional[str] = None*, *data_source_id: Optional[str] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Delete a data source.

Note: You must pass a not None `name` or `data_source_id` argument.

Parameters

- **name** (*str, optional*) – Dashboard name.

- **data_source_id** (*str, optional*) – The ID for the data source.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_data_source(name="...")
```

awswrangler.quicksight.delete_dataset

awswrangler.quicksight.**delete_dataset**(*name: Optional[str] = None*, *dataset_id: Optional[str] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Delete a dataset.

Note: You must pass a not None `name` or `dataset_id` argument.

Parameters

- **name** (*str, optional*) – Dashboard name.
- **dataset_id** (*str, optional*) – The ID for the dataset.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_dataset(name="...")
```

awswrangler.quicksight.delete_template

awswrangler.quicksight.**delete_template**(*name: Optional[str] = None*, *template_id: Optional[str] = None*, *version_number: Optional[int] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → None

Delete a tamplate.

Note: You must pass a not None `name` or `template_id` argument.

Parameters

- **name** (*str, optional*) – Dashboard name.
- **template_id** (*str, optional*) – The ID for the dashboard.
- **version_number** (*int, optional*) – Specifies the version of the template that you want to delete. If you don't provide a version number, it deletes all versions of the template.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_template(name="...")
```

awswrangler.quicksight.describe_dashboard

awswrangler.quicksight.**describe_dashboard**(*name: Optional[str] = None*, *dashboard_id: Optional[str] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Describe a QuickSight dashboard by name or ID.

Note: You must pass a not None `name` or `dashboard_id` argument.

Parameters

- **name** (*str, optional*) – Dashboard name.
- **dashboard_id** (*str, optional*) – Dashboard ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Dashboad Description.

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_dashboard(name="my-dashboard")
```

awswrangler.quicksight.describe_data_source

awswrangler.quicksight.**describe_data_source**(*name: Optional[str] = None*, *data_source_id: Optional[str] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Describe a QuickSight data source by name or ID.

Note: You must pass a not None `name` or `data_source_id` argument.

Parameters

- **name** (*str, optional*) – Data source name.
- **data_source_id** (*str, optional*) – Data source ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Data source Description.

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_data_source("...")
```

awswrangler.quicksight.describe_data_source_permissions

awswrangler.quicksight.**describe_data_source_permissions**(*name: Optional[str] = None*, *data_source_id: Optional[str] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Describe a QuickSight data source permissions by name or ID.

Note: You must pass a not None `name` or `data_source_id` argument.

Parameters

- **name** (*str, optional*) – Data source name.
- **data_source_id** (*str, optional*) – Data source ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Data source Permissions Description.

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_data_source_permissions("my-data-source")
```

awswrangler.quicksight.describe_dataset

awswrangler.quicksight.**describe_dataset**(*name: Optional[str] = None*, *dataset_id: Optional[str] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Describe a QuickSight dataset by name or ID.

Note: You must pass a not None `name` or `dataset_id` argument.

Parameters

- **name** (*str, optional*) – Dataset name.
- **dataset_id** (*str, optional*) – Dataset ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Dataset Description.

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_dataset("my-dataset")
```

awswrangler.quicksight.describe_ingestion

awswrangler.quicksight.**describe_ingestion**(*ingestion_id: str*, *dataset_name: Optional[str] = None*, *dataset_id: Optional[str] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Describe a QuickSight ingestion by ID.

Note: You must pass a not None value for `dataset_name` or `dataset_id` argument.

Parameters

- **ingestion_id** (*str*) – Ingestion ID.
- **dataset_name** (*str, optional*) – Dataset name.
- **dataset_id** (*str, optional*) – Dataset ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Ingestion Description.

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_dataset(ingestion_id="...", dataset_name="...")
```

awswrangler.quicksight.get_dashboard_id

awswrangler.quicksight.**get_dashboard_id**(*name: str*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Get QuickSight dashboard ID given a name and fails if there is more than 1 ID associated with this name.

Parameters

- **name** (*str*) – Dashboard name.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Dashboad ID.

Return type str

Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_dashboard_id(name="...")
```

awswrangler.quicksight.get_dashboard_ids

awswrangler.quicksight.**get_dashboard_ids**(*name: str*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[str]

Get QuickSight dashboard IDs given a name.

Note: This function returns a list of ID because Quicksight accepts duplicated dashboard names, so you may have more than 1 ID for a given name.

Parameters

- **name** (*str*) – Dashboard name.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Dashboad IDs.

Return type List[str]

Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_dashboard_ids(name="...")
```

awswrangler.quicksight.get_data_source_arn

awswrangler.quicksight.**get_data_source_arn**(*name: str*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Get QuickSight data source ARN given a name and fails if there is more than 1 ARN associated with this name.

Note: This function returns a list of ARNs because Quicksight accepts duplicated data source names, so you may have more than 1 ARN for a given name.

Parameters

- **name** (*str*) – Data source name.

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Data source ARN.

Return type str

Examples

```
>>> import awswrangler as wr
>>> arn = wr.quicksight.get_data_source_arn("...")
```

awswrangler.quicksight.get_data_source_arns

awswrangler.quicksight.**get_data_source_arns**(*name: str*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[str]

Get QuickSight Data source ARNs given a name.

Note: This function returns a list of ARNs because Quicksight accepts duplicated data source names, so you may have more than 1 ARN for a given name.

Parameters

- **name** (*str*) – Data source name.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Data source ARNs.

Return type List[str]

Examples

```
>>> import awswrangler as wr
>>> arns = wr.quicksight.get_data_source_arns(name="...")
```

awswrangler.quicksight.get_data_source_id

awswrangler.quicksight.**get_data_source_id**(*name: str*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Get QuickSight data source ID given a name and fails if there is more than 1 ID associated with this name.

Parameters

- **name** (*str*) – Data source name.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Dataset ID.

Return type str

Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_data_source_id(name="...")
```

awswrangler.quicksight.get_data_source_ids

awswrangler.quicksight.**get_data_source_ids**(*name: str*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[str]

Get QuickSight data source IDs given a name.

Note: This function returns a list of ID because Quicksight accepts duplicated data source names, so you may have more than 1 ID for a given name.

Parameters

- **name** (*str*) – Data source name.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Data source IDs.

Return type List[str]

Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_data_source_ids(name="...")
```

awswrangler.quicksight.get_dataset_id

awswrangler.quicksight.**get_dataset_id**(*name: str*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Get QuickSight Dataset ID given a name and fails if there is more than 1 ID associated with this name.

Parameters

- **name** (*str*) – Dataset name.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Dataset ID.

Return type str

Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_dataset_id(name="...")
```

awswrangler.quicksight.get_dataset_ids

awswrangler.quicksight.**get_dataset_ids**(*name: str*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[str]

Get QuickSight dataset IDs given a name.

Note: This function returns a list of ID because Quicksight accepts duplicated datasets names, so you may have more than 1 ID for a given name.

Parameters

- **name** (*str*) – Dataset name.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Datasets IDs.

Return type List[str]

Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_dataset_ids(name="...")
```

awswrangler.quicksight.get_template_id

awswrangler.quicksight.**get_template_id**(*name: str*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → str

Get QuickSight template ID given a name and fails if there is more than 1 ID associated with this name.

Parameters

- **name** (*str*) – Template name.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Template ID.

Return type str

Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_template_id(name="...")
```

awswrangler.quicksight.get_template_ids

awswrangler.quicksight.**get_template_ids**(*name: str*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[str]

Get QuickSight template IDs given a name.

Note: This function returns a list of ID because Quicksight accepts duplicated templates names, so you may have more than 1 ID for a given name.

Parameters

- **name** (*str*) – Template name.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Tamplate IDs.

Return type List[str]

Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_template_ids(name="...")
```

awswrangler.quicksight.list_dashboards

awswrangler.quicksight.**list_dashboards**(*account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List dashboards in an AWS account.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Dashboards.

Return type List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> dashboards = wr.quicksight.list_dashboards()
```

awswrangler.quicksight.list_data_sources

awswrangler.quicksight.**list_data_sources**(*account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List all QuickSight Data sources summaries.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Data sources summaries.

Return type List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> sources = wr.quicksight.list_data_sources()
```

awswrangler.quicksight.list_datasets

awswrangler.quicksight.**list_datasets**(*account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List all QuickSight datasets summaries.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Datasets summaries.

Return type List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> datasets = wr.quicksight.list_datasets()
```

awswrangler.quicksight.list_groups

awswrangler.quicksight.**list_groups**(*namespace: str = 'default'*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List all QuickSight Groups.

Parameters

- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Groups.

Return type List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> groups = wr.quicksight.list_groups()
```

awswrangler.quicksight.list_group_memberships

awswrangler.quicksight.**list_group_memberships**(*group_name: str*, *namespace: str = 'default'*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List all QuickSight Group memberships.

Parameters

- **group_name** (*str*) – The name of the group that you want to see a membership list of.
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Group memberships.

Return type List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> memberships = wr.quicksight.list_group_memberships()
```

awswrangler.quicksight.list_iam_policy_assignments

awswrangler.quicksight.**list_iam_policy_assignments**(*status: Optional[str] = None*, *namespace: str = 'default'*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List IAM policy assignments in the current Amazon QuickSight account.

Parameters

- **status** (*str, optional*) – The status of the assignments. 'ENABLED'|'DRAFT'|'DISABLED'
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns IAM policy assignments.

Return type List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> assigns = wr.quicksight.list_iam_policy_assignments()
```

awswrangler.quicksight.list_iam_policy_assignments_for_user

awswrangler.quicksight.**list_iam_policy_assignments_for_user**(*user_name: str*, *namespace: str = 'default'*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List all the IAM policy assignments.

Including the Amazon Resource Names (ARNs) for the IAM policies assigned to the specified user and group or groups that the user belongs to.

Parameters

- **user_name** (*str*) – The name of the user.
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns IAM policy assignments.

Return type List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> assigns = wr.quicksight.list_iam_policy_assignments_for_user()
```

awswrangler.quicksight.list_ingestions

awswrangler.quicksight.**list_ingestions**(*dataset_name: Optional[str] = None*, *dataset_id: Optional[str] = None*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List the history of SPICE ingestions for a dataset.

Parameters

- **dataset_name** (*str, optional*) – Dataset name.
- **dataset_id** (*str, optional*) – The ID of the dataset used in the ingestion.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.

- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns IAM policy assignments.

Return type List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> ingestions = wr.quicksight.list_ingestions()
```

awswrangler.quicksight.list_templates

awswrangler.quicksight.**list_templates**(*account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List all QuickSight templates.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Templates summaries.

Return type List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> templates = wr.quicksight.list_templates()
```

awswrangler.quicksight.list_users

awswrangler.quicksight.**list_users**(*namespace: str = 'default'*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

Return a list of all of the Amazon QuickSight users belonging to this account.

Parameters

- **namespace** (*str*) – The namespace. Currently, you should set this to default.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Groups.

Return type List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> users = wr.quicksight.list_users()
```

awswrangler.quicksight.list_user_groups

awswrangler.quicksight.**list_user_groups**(*user_name: str*, *namespace: str = 'default'*, *account_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List the Amazon QuickSight groups that an Amazon QuickSight user is a member of.

Parameters

- **user_name** (*str:*) – The Amazon QuickSight user name that you want to list group memberships for.
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Groups.

Return type List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> groups = wr.quicksight.list_user_groups()
```

1.4.14 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: Optional[boto3.session.Session] = None*) → str

Get Account ID.

Parameters **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Account ID.

Return type str

Examples

```
>>> import awswrangler as wr
>>> account_id = wr.sts.get_account_id()
```

awswrangler.sts.get_current_identity_arn

awswrangler.sts.**get_current_identity_arn**(*boto3_session: Optional[boto3.session.Session] = None*) → str

Get current user/role ARN.

Parameters **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns User/role ARN.

Return type str

Examples

```
>>> import awswrangler as wr
>>> arn = wr.sts.get_current_identity_arn()
```

awswrangler.sts.get_current_identity_name

awswrangler.sts.**get_current_identity_name**(*boto3_session: Optional[boto3.session.Session] = None*) → str

Get current user/role name.

Parameters **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns User/role name.

Return type str

Examples

```
>>> import awswrangler as wr
>>> name = wr.sts.get_current_identity_name()
```

1.4.15 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: Optional[boto3.session.Session] = None*) → Union[str, bytes]

Get secret value.

Parameters

- **name** (*str:*) – Specifies the secret containing the version that you want to retrieve. You can specify either the Amazon Resource Name (ARN) or the friendly name of the secret.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Secret value.

Return type Union[str, bytes]

Examples

```
>>> import awswrangler as wr
>>> value = wr.secretsmanager.get_secret("my-secret")
```

awswrangler.secretsmanager.get_secret_json

awswrangler.secretsmanager.**get_secret_json**(*name: str*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Get JSON secret value.

Parameters

- **name** (*str:*) – Specifies the secret containing the version that you want to retrieve. You can specify either the Amazon Resource Name (ARN) or the friendly name of the secret.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns Secret JSON value parsed as a dictionary.

Return type Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> value = wr.secretsmanager.get_secret_json("my-secret-with-json-content")
```

1.4.16 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*) → Optional[Any]

Send message on an existing Chime Chat rooms.

:param : Webhook: This contains all the authentication information to send the message :type : param webhook : webhook :param : The actual message which needs to be posted on Slack channel :type : param message : message

Returns Represents the response from Chime

Return type dict

1.4.17 Global Configurations

reset([item])	Reset one or all (if None is received) configuration values.
to_pandas()	Load all configurations on a Pandas DataFrame.

awswrangler.config.reset

config.**reset**(*item: Optional[str] = None*) → None

Reset one or all (if None is received) configuration values.

Parameters **item** (*str, optional*) – Configuration item name.

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> wr.config.reset("database")  # Reset one specific configuration
>>> wr.config.reset()  # Reset all
```

awswrangler.config.to_pandas

config.**to_pandas**() → pandas.core.frame.DataFrame

Load all configurations on a Pandas DataFrame.

Returns Configuration DataFrame.

Return type pd.DataFrame

Examples

```
>>> import awswrangler as wr
>>> wr.config.to_pandas()
```

INDEX

A

add_column() (*in module awswrangler.catalog*), 143
add_csv_partitions() (*in module awswrangler.catalog*), 144
add_parquet_partitions() (*in module awswrangler.catalog*), 145

B

build_spark_step() (*in module awswrangler.emr*), 222
build_step() (*in module awswrangler.emr*), 223

C

cancel_ingestion() (*in module awswrangler.quicksight*), 236
connect() (*in module awswrangler.data_api.rds*), 213
connect() (*in module awswrangler.data_api.redshift*), 212
connect() (*in module awswrangler.mysql*), 204
connect() (*in module awswrangler.postgresql*), 199
connect() (*in module awswrangler.redshift*), 186
connect() (*in module awswrangler.sqlserver*), 208
connect_temp() (*in module awswrangler.redshift*), 187
copy() (*in module awswrangler.redshift*), 188
copy_from_files() (*in module awswrangler.redshift*), 190
copy_objects() (*in module awswrangler.s3*), 98
create_athena_bucket() (*in module awswrangler.athena*), 173
create_athena_data_source() (*in module awswrangler.quicksight*), 237
create_athena_dataset() (*in module awswrangler.quicksight*), 238
create_cluster() (*in module awswrangler.emr*), 224
create_csv_table() (*in module awswrangler.catalog*), 147
create_database() (*in module awswrangler.catalog*), 149
create_database() (*in module awswrangler.timestream*), 218
create_ingestion() (*in module awswrangler.quicksight*), 239
create_parquet_table() (*in module awswrangler.catalog*), 150
create_table() (*in module awswrangler.timestream*), 218

D

databases() (*in module awswrangler.catalog*), 152
delete_all_dashboards() (*in module awswrangler.quicksight*), 240
delete_all_data_sources() (*in module awswrangler.quicksight*), 240
delete_all_datasets() (*in module awswrangler.quicksight*), 241
delete_all_partitions() (*in module awswrangler.catalog*), 155
delete_all_templates() (*in module awswrangler.quicksight*), 241
delete_column() (*in module awswrangler.catalog*), 153
delete_dashboard() (*in module awswrangler.quicksight*), 242
delete_data_source() (*in module awswrangler.quicksight*), 242
delete_database() (*in module awswrangler.catalog*), 153
delete_database() (*in module awswrangler.timestream*), 219
delete_dataset() (*in module awswrangler.quicksight*), 243
delete_items() (*in module awswrangler.dynamodb*), 214
delete_objects() (*in module awswrangler.s3*), 99
delete_partitions() (*in module awswrangler.catalog*), 154
delete_table() (*in module awswrangler.timestream*), 220
delete_table_if_exists() (*in module awswrangler.catalog*), 156
delete_template() (*in module awswrangler.quicksight*), 244
describe_dashboard() (*in module awswrangler.quicksight*), 244

describe_data_source() (*in module awswrangler.quicksight*), 245
describe_data_source_permissions() (*in module awswrangler.quicksight*), 245
describe_dataset() (*in module awswrangler.quicksight*), 246
describe_ingestion() (*in module awswrangler.quicksight*), 247
describe_objects() (*in module awswrangler.s3*), 100
does_object_exist() (*in module awswrangler.s3*), 101
does_table_exist() (*in module awswrangler.catalog*), 156
download() (*in module awswrangler.s3*), 102
drop_duplicated_columns() (*in module awswrangler.catalog*), 157

E

extract_athena_types() (*in module awswrangler.catalog*), 158

G

get_account_id() (*in module awswrangler.sts*), 258
get_bucket_region() (*in module awswrangler.s3*), 103
get_cluster_state() (*in module awswrangler.emr*), 229
get_columns_comments() (*in module awswrangler.catalog*), 158
get_csv_partitions() (*in module awswrangler.catalog*), 159
get_current_identity_arn() (*in module awswrangler.sts*), 259
get_current_identity_name() (*in module awswrangler.sts*), 259
get_dashboard_id() (*in module awswrangler.quicksight*), 247
get_dashboard_ids() (*in module awswrangler.quicksight*), 248
get_data_source_arn() (*in module awswrangler.quicksight*), 248
get_data_source_arns() (*in module awswrangler.quicksight*), 249
get_data_source_id() (*in module awswrangler.quicksight*), 250
get_data_source_ids() (*in module awswrangler.quicksight*), 250
get_databases() (*in module awswrangler.catalog*), 160
get_dataset_id() (*in module awswrangler.quicksight*), 251
get_dataset_ids() (*in module awswrangler.quicksight*), 251
get_parquet_partitions() (*in module awswrangler.catalog*), 161
get_partitions() (*in module awswrangler.catalog*), 162
get_query_columns_types() (*in module awswrangler.athena*), 174
get_query_execution() (*in module awswrangler.athena*), 174
get_secret() (*in module awswrangler.secretsmanager*), 260
get_secret_json() (*in module awswrangler.secretsmanager*), 260
get_step_state() (*in module awswrangler.emr*), 229
get_table() (*in module awswrangler.dynamodb*), 214
get_table_description() (*in module awswrangler.catalog*), 163
get_table_location() (*in module awswrangler.catalog*), 164
get_table_number_of_versions() (*in module awswrangler.catalog*), 164
get_table_parameters() (*in module awswrangler.catalog*), 165
get_table_types() (*in module awswrangler.catalog*), 166
get_table_versions() (*in module awswrangler.catalog*), 166
get_tables() (*in module awswrangler.catalog*), 167
get_template_id() (*in module awswrangler.quicksight*), 252
get_template_ids() (*in module awswrangler.quicksight*), 252
get_work_group() (*in module awswrangler.athena*), 175

L

list_dashboards() (*in module awswrangler.quicksight*), 253
list_data_sources() (*in module awswrangler.quicksight*), 253
list_datasets() (*in module awswrangler.quicksight*), 254
list_directories() (*in module awswrangler.s3*), 103
list_group_memberships() (*in module awswrangler.quicksight*), 255
list_groups() (*in module awswrangler.quicksight*), 254
list_iam_policy_assignments() (*in module awswrangler.quicksight*), 255
list_iam_policy_assignments_for_user() (*in module awswrangler.quicksight*), 256
list_ingestions() (*in module awswrangler.quicksight*), 256
list_objects() (*in module awswrangler.s3*), 104

list_templates() (*in module awswrangler.quicksight*), 257
list_user_groups() (*in module awswrangler.quicksight*), 258
list_users() (*in module awswrangler.quicksight*), 257

M

merge_datasets() (*in module awswrangler.s3*), 105
merge_upsert_table() (*in module awswrangler.s3*), 106

O

overwrite_table_parameters() (*in module awswrangler.catalog*), 168

P

post_message() (*in module awswrangler.chime*), 261
put_csv() (*in module awswrangler.dynamodb*), 215
put_df() (*in module awswrangler.dynamodb*), 216
put_items() (*in module awswrangler.dynamodb*), 216
put_json() (*in module awswrangler.dynamodb*), 217

Q

query() (*in module awswrangler.timestream*), 220

R

read_csv() (*in module awswrangler.s3*), 107
read_excel() (*in module awswrangler.s3*), 109
read_fwf() (*in module awswrangler.s3*), 110
read_json() (*in module awswrangler.s3*), 112
read_logs() (*in module awswrangler.cloudwatch*), 233
read_parquet() (*in module awswrangler.s3*), 115
read_parquet_metadata() (*in module awswrangler.s3*), 117
read_parquet_table() (*in module awswrangler.s3*), 119
read_sql_query() (*in module awswrangler.athena*), 175
read_sql_query() (*in module awswrangler.data_api.rds*), 213
read_sql_query() (*in module awswrangler.data_api.redshift*), 213
read_sql_query() (*in module awswrangler.mysql*), 205
read_sql_query() (*in module awswrangler.postgresql*), 200
read_sql_query() (*in module awswrangler.redshift*), 192
read_sql_query() (*in module awswrangler.sqlserver*), 209
read_sql_table() (*in module awswrangler.athena*), 179
read_sql_table() (*in module awswrangler.mysql*), 206
read_sql_table() (*in module awswrangler.postgresql*), 201
read_sql_table() (*in module awswrangler.redshift*), 193
read_sql_table() (*in module awswrangler.sqlserver*), 210
repair_table() (*in module awswrangler.athena*), 182
reset() (*awswrangler.config method*), 261
run_query() (*in module awswrangler.cloudwatch*), 233

S

sanitize_column_name() (*in module awswrangler.catalog*), 169
sanitize_dataframe_columns_names() (*in module awswrangler.catalog*), 169
sanitize_table_name() (*in module awswrangler.catalog*), 170
search_tables() (*in module awswrangler.catalog*), 170
select_query() (*in module awswrangler.s3*), 121
size_objects() (*in module awswrangler.s3*), 123
start_query() (*in module awswrangler.cloudwatch*), 234
start_query_execution() (*in module awswrangler.athena*), 183
stop_query_execution() (*in module awswrangler.athena*), 184
store_parquet_metadata() (*in module awswrangler.s3*), 124
submit_ecr_credentials_refresh() (*in module awswrangler.emr*), 230
submit_spark_step() (*in module awswrangler.emr*), 230
submit_step() (*in module awswrangler.emr*), 231
submit_steps() (*in module awswrangler.emr*), 231

T

table() (*in module awswrangler.catalog*), 171
tables() (*in module awswrangler.catalog*), 171
terminate_cluster() (*in module awswrangler.emr*), 232
to_csv() (*in module awswrangler.s3*), 126
to_excel() (*in module awswrangler.s3*), 132
to_json() (*in module awswrangler.s3*), 133
to_pandas() (*awswrangler.config method*), 261
to_parquet() (*in module awswrangler.s3*), 135
to_sql() (*in module awswrangler.mysql*), 207
to_sql() (*in module awswrangler.postgresql*), 202
to_sql() (*in module awswrangler.redshift*), 194
to_sql() (*in module awswrangler.sqlserver*), 211

U

unload() (*in module awswrangler.redshift*), 196
unload_to_files() (*in module awswrangler.redshift*), 198

upload() (*in module awswrangler.s3*), 140
upsert_table_parameters() (*in module awswrangler.catalog*), 172

W

wait_objects_exist() (*in module awswrangler.s3*), 141
wait_objects_not_exist() (*in module awswrangler.s3*), 141
wait_query() (*in module awswrangler.athena*), 185
wait_query() (*in module awswrangler.cloudwatch*), 235
write() (*in module awswrangler.timestream*), 221