AWS Data Wrangler

Release 1.0.4

Igor Tavares

May 09, 2020

CONTENTS

Note: We just released a new major version *1.0* with breaking changes. Please make sure that all your old projects has dependencies frozen on the desired version (e.g. *pip install awswrangler==0.3.2*).

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

```
import awswrangler as wr
import pandas as pd

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")

# Getting Redshift connection (SQLAlchemy) from Glue Catalog Connections
engine = wr.catalog.get_engine("my-redshift-connection")

# Retrieving the data from Amazon Redshift Spectrum
df = wr.db.read_sql_query("SELECT * FROM external_schema.my_table", con=engine)
```

CHAPTER

ONE

READ THE DOCS

Note: We just released a new major version *1.0* with breaking changes. Please make sure that all your old projects has dependencies frozen on the desired version (e.g. *pip install awswrangler==0.3.2*).

1.1 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 EMR**, etc).

Built on top of other open-source projects like Pandas, Apache Arrow, Boto3, s3fs, SQLAlchemy, Psycopg2 and PyMySQL, 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.

Note: We just released a new major version *1.0* with breaking changes. Please make sure that all your old projects has dependencies frozen on the desired version (e.g. *pip install awswrangler==0.3.2*).

1.2 Install

AWS Data Wrangler runs with Python `3.6`, `3.7` and `3.8` 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 bellow are:

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

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 Wheel

Note: AWS Data Wrangler has compiled dependencies (C/C++) so there is only support for `Glue Python Shell`, **not** for `Glue PySpark`.

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 - Got to your Glue Python Shell job and point to the new file on s3.

1.2.5 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
```

1.2.6 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.
```

(continues on next page)

(continued from previous page)

```
# Note this may timeout if the package installations in all environments take longer␣
↪than 5 mins, consider using
# "nohup" to run this as a background process in that case.

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

# PARAMETERS
PACKAGE=awswrangler

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

1.2.7 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 under your cluster configuration

```
[
  {
     "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.

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

sudo pip-3.6 install awswrangler
```

Note: Make sure to freeze the Wrangler version in the bootstrap for productive environments (e.g. awswrangler==1.0.0)

1.2.8 From Source

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

Note: We just released a new major version *1.0* with breaking changes. Please make sure that all your old projects has the dependencies frozen on the desired version (e.g. *pip install awswrangler==0.3.2*).

1.3 API Reference

1.3.1 Amazon S3

delete_objects(path[, use_threads, ...])	Delete Amazon S3 objects from a received S3 prefix or list of S3 objects paths.
describe_objects(path[, wait_time, ...])	Describe Amazon S3 objects from a received S3 prefix or list of S3 objects paths.
does_object_exist(path[, boto3_session])	Check if object exists on S3.
get_bucket_region(bucket[, boto3_session])	Get bucket region name.
list_objects(path[, boto3_session])	List Amazon S3 objects from a prefix.
read_csv(path[, use_threads, boto3_session, ...])	Read CSV file(s) from from a received S3 prefix or list of S3 objects paths.
read_fwf(path[, use_threads, boto3_session, ...])	Read fixed-width formatted file(s) from from a received S3 prefix or list of S3 objects paths.
read_json(path[, use_threads, ...])	Read JSON file(s) from from a received S3 prefix or list of S3 objects paths.
read_parquet(path[, filters, columns, ...])	Read Apache Parquet file(s) 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.
read_parquet_metadata(path[, filters, ...])	Read Apache Parquet file(s) metadata from from a received S3 prefix or list of S3 objects paths.
size_objects(path[, wait_time, use_threads, ...])	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_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.
wait_objects_exist(paths[, delay, ...])	Wait Amazon S3 objects exist.
wait_objects_not_exist(paths[, delay, ...])	Wait Amazon S3 objects not exist.
copy_objects(paths, source_path, target_path)	Copy a list of S3 objects to another S3 directory.
merge_datasets(source_path, target_path[, ...])	Merge a source dataset into a target dataset.

awswrangler.s3.delete_objects

awswrangler.s3.**delete_objects**(*path: Union[str, List[str]]*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*) → None

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

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

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (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.
- **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]]*, *wait_time: Union[int, float, None] = None*, *use_threads: bool = True*, *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, LastModified, 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

---

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

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **wait_time** (*Union[int,float], optional*) – How much time (seconds) should Wrangler try to reach this objects. Very useful to overcome eventual consistence issues. *None* means only a single try will be done.

- **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 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
>>> descs2 = wr.s3.describe_objects('s3://bucket/prefix', wait_time=30)  #␣
↪Overcoming eventual consistence issues
```

awswrangler.s3.does_object_exist

awswrangler.s3.**does_object_exist**(*path: str*, *boto3_session: Optional[boto3.session.Session] = None*) → bool

Check if object exists on S3.

Parameters

- **path** (*str*) – S3 path (e.g. s3://bucket/key).
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns True if exists, False otherwise.

Return type bool

Examples

Using the default boto3 session

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

Using a custom boto3 session

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

(continues on next page)

(continued from previous page)

```
>>> wr.s3.does_object_exist('s3://bucket/key_unreal', boto3_session=boto3.
↪Session())
False
```

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_objects

awswrangler.s3.**list_objects**(*path: str*, *boto3_session: Optional[boto3.session.Session] = None*) → List[str]

List Amazon S3 objects from a prefix.

Parameters

- **path** (*str*) – S3 path (e.g. s3://bucket/prefix).
- **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.read_csv

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

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 get from os.cpu_count().

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (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.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** – Forward to s3fs, useful for server side encryption https://s3fs.readthedocs.io/en/latest/#serverside-encryption
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **pandas_kwargs** – keyword arguments forwarded to pandas.read_csv(). 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 encrypted with a KMS key

```
>>> import awswrangler as wr
>>> df = wr.s3.read_csv(
...     path='s3://bucket/prefix/',
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMY_KEY_ARN'
...     }
... )
```

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
```

awswrangler.s3.read_fwf

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

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 get from os.cpu_count().

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (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.

- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** – Forward to s3fs, useful for server side encryption https://s3fs.readthedocs.io/en/latest/#serverside-encryption
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **pandas_kwargs** – keyword arguments forwarded to pandas.read_fwf(). 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/')
```

Reading all fixed-width formatted (FWF) files under a prefix encrypted with a KMS key

```
>>> import awswrangler as wr
>>> df = wr.s3.read_fwf(
...     path='s3://bucket/prefix/',
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMY_KEY_ARN'
...     }
... )
```

Reading all fixed-width formatted (FWF) files from a list

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

Reading in chunks of 100 lines

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

awswrangler.s3.read_json

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

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 get from os.cpu_count().

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (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.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** – Forward to s3fs, useful for server side encryption https://s3fs.readthedocs.io/en/latest/#serverside-encryption
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **pandas_kwargs** – keyword arguments forwarded to pandas.read_json(). 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 JSON files under a prefix encrypted with a KMS key

```
>>> import awswrangler as wr
>>> df = wr.s3.read_json(
...     path='s3://bucket/prefix/',
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMY_KEY_ARN'
...     }
... )
```

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/filename0.json', 's3://bucket/
↪filename1.json'], chunksize=100)
>>> for df in dfs:
>>>     print(df)  # 100 lines Pandas DataFrame
```

awswrangler.s3.read_parquet

awswrangler.s3.**read_parquet**(*path: Union[str, List[str]]*, *filters: Union[List[Tuple], List[List[Tuple]], None] = None*, *columns: Optional[List[str]] = None*, *validate_schema: bool = True*, *chunked: bool = False*, *dataset: bool = False*, *categories: List[str] = None*, *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]]

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).

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

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **filters** (*Union[List[Tuple], List[List[Tuple]]], optional*) – List of filters to apply, like `[[('x', '=', 0), ...], ...]`.
- **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** (*bool*) – If True will break the data in smaller DataFrames (Non deterministic number of lines). Otherwise return a single DataFrame with the whole data.
- **dataset** (*bool*) – If True read a parquet dataset instead of simple file(s) loading all the related partitions as columns.
- **categories** (*List[str], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **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 s3fs, useful for server side encryption https://s3fs.readthedocs.io/en/latest/#serverside-encryption

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 under a prefix encrypted with a KMS key

```
>>> import awswrangler as wr
>>> df = wr.s3.read_parquet(
...     path='s3://bucket/prefix/',
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMY_KEY_ARN'
...     }
... )
```

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

```
>>> 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
```

awswrangler.s3.read_parquet_table

awswrangler.s3.**read_parquet_table**(*table: str*, *database: str*, *filters: Union[List[Tuple], List[List[Tuple]], None] = None*, *columns: Optional[List[str]] = None*, *categories: List[str] = None*, *chunked: 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]]

Read Apache Parquet table registered on AWS Glue Catalog.

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

Parameters

- **table** (*str*) – AWS Glue Catalog table name.
- **database** (*str*) – AWS Glue Catalog database name.
- **filters** (*Union[List[Tuple], List[List[Tuple]]], optional*) – List of filters to apply, like `[[('x', '=', 0), ...], ...]`.
- **columns** (*List[str], optional*) – Names of columns to read from the file(s).
- **categories** (*List[str], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **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** (*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 s3fs, useful for server side encryption https://s3fs.readthedocs.io/en/latest/#serverside-encryption

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

Reading Parquet Table in chunks

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

awswrangler.s3.read_parquet_metadata

awswrangler.s3.**read_parquet_metadata**(*path: Union[str, List[str]]*, *filters: Union[List[Tuple], List[List[Tuple]], None] = None*, *dataset: bool = False*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*) → Tuple[Dict[str, str], Optional[Dict[str, str]]]

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).

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

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **filters** (*Union[List[Tuple], List[List[Tuple]], optional*) – List of filters to apply, like `[[('x', '=', 0), ...], ...]`.
- **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.

Returns columns_types: Dictionary with keys as column names and vales 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.size_objects

awswrangler.s3.**size_objects**(*path: Union[str, List[str]], wait_time: Union[int, float, None] = None, use_threads: bool = True, 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.

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

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **wait_time** (*Union[int,float], optional*) – How much time (seconds) should Wrangler try to reach this objects. Very useful to overcome eventual consistence issues. *None* means only a single try will be done.
- **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 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
>>> sizes2 = wr.s3.size_objects('s3://bucket/prefix', wait_time=30)  # Overcoming␣
↪eventual consistence issues
```

awswrangler.s3.store_parquet_metadata

awswrangler.s3.**store_parquet_metadata**(*path: str, database: str, table: str, filters: Union[List[Tuple], List[List[Tuple]], None] = None, 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, boto3_session: Optional[boto3.session.Session] = None*) → Tuple[Dict[str, str], Optional[Dict[str, str]], Optional[Dict[str, List[str]]]]

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).

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

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (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.
- **filters** (*Union[List[Tuple], List[List[Tuple]], optional*) – List of filters to apply, like `[[('x', '=', 0), ...], ...]`.
- **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).
- **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 vales 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: str*, *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, str]] = None*, *dataset: bool = False*, *partition_cols: Optional[List[str]] = None*, *mode: Optional[str] = None*, *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*, ***pandas_kwargs*) → Dict[str, Union[List[str], Dict[str, List[str]]]]

Write CSV file or dataset on Amazon S3.

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

Note: The table name and all column names will be automatically sanitize using *wr.catalog.sanitize_table_name* and *wr.catalog.sanitize_column_name*.

Note: In case of *use_threads=True* the number of threads that will be spawned will be get 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.csv).
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **index** (*bool*) – Write row names (index).
- **columns** (*List[str], optional*) – 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** – Forward to s3fs, useful for server side encryption https://s3fs.readthedocs.io/en/latest/#serverside-encryption
- **dataset** (*bool*) – If True store a parquet dataset instead of a single file. If True, enable all follow arguments: partition_cols, mode, database, table, description, parameters, columns_comments, .
- **partition_cols** (*List[str], optional*) – List of column names that will be used to create partitions. Only takes effect if dataset=True.
- **mode** (*str, optional*) – `append` (Default), `overwrite`, `overwrite_partitions`. Only takes effect if dataset=True.
- **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. Only takes effect if dataset=True. (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.'}).
- **pandas_kwargs** – keyword arguments forwarded to pandas.DataFrame.to_csv() https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html

Returns None.

Return type None

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 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_KMY_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(
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
    'partitions_values: {}
}
```

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, str]] = None*, ***pandas_kwargs*) → None

Write JSON file on Amazon S3.

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** – Forward to s3fs, useful for server side encryption https://s3fs.readthedocs.io/en/latest/#serverside-encryption
- **pandas_kwargs** – keyword arguments forwarded to pandas.DataFrame.to_csv() https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_json.html

Returns None.

Return type None

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

awswrangler.s3.to_parquet

awswrangler.s3.**to_parquet**(*df: pandas.core.frame.DataFrame*, *path: str*, *index: bool = False*, *compression: Optional[str] = 'snappy'*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, str]] = None*, *dataset: bool = False*, *partition_cols: Optional[List[str]] = None*, *mode: Optional[str] = None*, *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*) → Dict[str, Union[List[str], Dict[str, List[str]]]]

Write Parquet file or dataset on Amazon S3.

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

Note: The table name and all column names will be automatically sanitize using *wr.catalog.sanitize_table_name* and *wr.catalog.sanitize_column_name*.

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

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **path** (*str*) – S3 path (for file e.g. `s3://bucket/prefix/filename.parquet`) (for dataset e.g. `s3://bucket/prefix`).
- **index** (*bool*) – True to store the DataFrame index in file, otherwise False to ignore it.
- **compression** (*str, optional*) – Compression style (`None`, `snappy`, `gzip`).
- **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 s3fs, useful for server side encryption https://s3fs.readthedocs.io/en/latest/#serverside-encryption
- **dataset** (*bool*) – If True store a parquet dataset instead of a single file. If True, enable all follow arguments: partition_cols, mode, database, table, description, parameters, columns_comments, .
- **partition_cols** (*List[str], optional*) – List of column names that will be used to create partitions. Only takes effect if dataset=True.
- **mode** (*str, optional*) – `append` (Default), `overwrite`, `overwrite_partitions`. Only takes effect if dataset=True.
- **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. Only takes effect if dataset=True. (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.'}).

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_KMY_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],
```

(continues on next page)

(continued from previous page)

```
...         '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 dataset to S3 with metadata on Athena/Glue Catalog.

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

Writing dataset casting empty column data type

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_parquet(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B'],
...         'col3': [None, None, None]
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     database='default',  # Athena/Glue database
...     table='my_table'  # Athena/Glue table
...     dtype={'col3': 'date'}
... )
{
    'paths': ['s3://.../x.parquet'],
    'partitions_values: {}
}
```

awswrangler.s3.wait_objects_exist

awswrangler.s3.**wait_objects_exist**(*paths: List[str], delay: Union[int, float, None] = 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 get 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: Union[int, float, None] = 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 get 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
```

awswrangler.s3.copy_objects

awswrangler.s3.**copy_objects**(*paths: List[str], source_path: str, target_path: str, use_threads: bool = True, boto3_session: Optional[boto3.session.Session] = 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 get 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.
- **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 List of new objects paths.

Return type List[str]

Examples

```
>>> 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"]
```

awswrangler.s3.merge_datasets

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

Merge a source dataset into a target dataset.

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 get 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`.
- **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 List of new objects paths.

Return type List[str]

Examples

```
>>> 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"]
```

1.3.2 AWS Glue Catalog

add_parquet_partitions(database, table, ...)	Add partitions (metadata) to a Parquet Table in the AWS Glue Catalog.
create_parquet_table(database, table, path, ...)	Create a Parquet Table (Metadata Only) in the AWS Glue Catalog.
add_csv_partitions(database, table, ... [, ...])	Add partitions (metadata) to a CSV Table in the AWS Glue Catalog.
create_csv_table(database, table, path, ...)	Create a CSV Table (Metadata Only) in the AWS Glue Catalog.
databases([limit, catalog_id, boto3_session])	Get a Pandas DataFrame with all listed databases.
delete_table_if_exists(database, table[, ...])	Delete Glue table if exists.
does_table_exist(database, table[, ...])	Check if the table exists.
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_csv_partitions(database, table[, ...])	Get all partitions from a Table in the AWS Glue Catalog.
get_table_location(database, table[, ...])	Get table's location on Glue catalog.
get_table_types(database, table[, boto3_session])	Get all columns and types from a table.
get_tables([catalog_id, database, ...])	Get an iterator of tables.
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.
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.
drop_duplicated_columns(df)	Drop all repeated columns (duplicated names).
get_engine(connection[, catalog_id, ...])	Return a SQLAlchemy Engine from a Glue Catalog Connection.
extract_athena_types(df[, index, ...])	Extract columns and partitions types (Amazon Athena) from Pandas DataFrame.

awswrangler.catalog.add_parquet_partitions

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

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

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **partitions_values** (*Dict[str, List[str]]*) – Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g.

{'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).

- **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.

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_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, compression: Optional[str] = None, description: Optional[str] = None, parameters: Optional[Dict[str, str]] = None, columns_comments: Optional[Dict[str, str]] = None, mode: str = 'overwrite', boto3_session: Optional[boto3.session.Session] = None*) → None

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

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

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **path** (*str*) – Amazon S3 path (e.g. s3://bucket/prefix/).
- **columns_types** (*Dict[str, str]*) – Dictionary with keys as column names and vales 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'}).
- **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 axisting table or 'append' to keep any possible axisting 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

```
>>> 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.add_csv_partitions

awswrangler.catalog.**add_csv_partitions**(*database: str, table: str, partitions_values: Dict[str, List[str]], compression: Optional[str] = None, sep: str = ',', boto3_session: Optional[boto3.session.Session] = None*) → None

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

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **partitions_values** (*Dict[str, List[str]]*) – Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. {'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).
- **compression** (*str, optional*) – Compression style (`None`, `gzip`, etc).
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **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.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.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*, *compression: Optional[str] = None*, *description: Optional[str] = None*, *parameters: Optional[Dict[str, str]] = None*, *columns_comments: Optional[Dict[str, str]] = None*, *mode: str = 'overwrite'*, *sep: str = ','*, *boto3_session: Optional[boto3.session.Session] = None*) → None

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

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

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **path** (*str*) – Amazon S3 path (e.g. s3://bucket/prefix/).
- **columns_types** (*Dict[str, str]*) – Dictionary with keys as column names and vales 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'}).
- **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.
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **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_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.databases

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

Get a Pandas DataFrame with all listed databases.

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_table_if_exists

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

Delete Glue table if exists.

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 deleted, otherwise False.

Return type bool

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.delete_table_if_exists(database='default', name='my_table')  #␣
↪deleted
True
>>> wr.catalog.delete_table_if_exists(database='default', name='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*)

Check if the table exists.

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', name='my_table')
```

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*) → Dict[str, List[str]]

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

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

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **expression** (*str, 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',
```

(continues on next page)

(continued from previous page)

```
...     table='my_table',
...     expression='m=10'
... )
{
    's3://bucket/prefix/y=2020/m=10/': ['2020', '10']
}
```

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*) → Dict[str, List[str]]

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

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

Parameters

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

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

Get table's location on Glue catalog.

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', name='my_table')
's3://bucket/prefix/'
```

awswrangler.catalog.get_table_types

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

Get all columns and types from a table.

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 A dictionary as {'col name': 'col data type'}.

Return type Dict[str, str]

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.get_table_types(database='default', name='my_table')
{'col0': 'int', 'col1': double}
```

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*) → Iterator[Dict[str, Any]]

Get an iterator of tables.

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.search_tables

awswrangler.catalog.**search_tables**(*text: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*)

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*) → pandas.core.frame.DataFrame

Get table details as Pandas DataFrame.

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', name='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*) → pandas.core.frame.DataFrame

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

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 Iterator[Dict[str, Any]]

Examples

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

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.drop_duplicated_columns

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

Drop all repeated columns (duplicated names).

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.get_engine

awswrangler.catalog.**get_engine**(*connection: str*, *catalog_id: Optional[str] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → sqlalchemy.engine.base.Engine

Return a SQLAlchemy Engine from a Glue Catalog Connection.

Only Redshift, PostgreSQL and MySQL are supported.

Parameters

- **connection** (*str*) – Connection 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 SQLAlchemy Engine.

Return type sqlalchemy.engine.Engine

Examples

```
>>> import awswrangler as wr
>>> res = wr.catalog.get_engine(name='my_connection')
```

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 vales 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

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

1.3.3 Amazon Athena

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.
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_work_group(workgroup[, boto3_session])	Return information about the workgroup with the specified name.

awswrangler.athena.read_sql_query

awswrangler.athena.**read_sql_query**(*sql: str*, *database: str*, *ctas_approach: bool = True*, *categories: List[str] = None*, *chunksize: Optional[int] = None*, *s3_output: Optional[str] = None*, *workgroup: Optional[str] = None*, *encryption: Optional[str] = None*, *kms_key: Optional[str] = None*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Execute any SQL query on AWS Athena and return the results as a Pandas DataFrame.

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 and can handle some level of nested types. CONS: Requires create/delete table permissions on Glue and Does not support timestamp with time zone (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: Does not require create/delete table permissions on Glue and supports timestamp with time zone. CONS: Slower (But stills faster than other libraries that uses the regular Athena API) and does not handle nested types at all.

Note: If *chunksize* is passed, then a Generator of DataFrames is returned.

Note: If *ctas_approach* is True, *chunksize* will return non deterministic chunks sizes, but it still useful to overcome memory limitation.

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

Parameters

- **sql** (*str*) – SQL query.
- **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** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **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.
- **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 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='...')
```

awswrangler.athena.read_sql_table

awswrangler.athena.**read_sql_table**(*table: str*, *database: str*, *ctas_approach: bool = True*, *categories: List[str] = None*, *chunksize: Optional[int] = None*, *s3_output: Optional[str] = None*, *workgroup: Optional[str] = None*, *encryption: Optional[str] = None*, *kms_key: Optional[str] = None*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Extract the full table AWS Athena and return the results as a Pandas DataFrame.

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 and can handle some level of nested types CONS: Requires create/delete table permissions on Glue and Does not support timestamp with time zone (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: Does not require create/delete table permissions on Glue and give support timestamp with time zone. CONS: Slower (But stills faster than other libraries that uses the regular Athena API) and does not handle nested types at all

---

Note: If *chunksize* is passed, then a Generator of DataFrames is returned.

Note: If *ctas_approach* is True, *chunksize* will return non deterministic chunks sizes, but it still useful to overcome memory limitation.

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

Parameters

- **table** (*str*) – Table name.
- **database** (*str*) – AWS Glue/Athena database name.
- **ctas_approach** (*bool*) – Wraps the query using a CTAS, and read the resulted parquet data on S3. If false, read the regular CSV on S3.
- **categories** (*List[str], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.

- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **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.
- **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 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='...')
```

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*) → str

Run the Hive's metastore consistency check: 'MSCK REPAIR TABLE table;'.

Recovers partitions and data associated with partitions. Use this statement when you add partitions to the catalog. It is possible it will take some time to add all partitions. If this operation times out, it will be in an incomplete state where only a few partitions are added to the catalog.

Note: Create the default Athena bucket if it doesn't exist and s3_output is None. (E.g. s3://aws-athena-query-results-ACCOUNT-REGION/)

Parameters

- **table** (*str*) – Table name.
- **database** (*str, 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*, *boto3_session: Optional[boto3.session.Session] = None*) → str

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/)

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

Returns Query execution ID

Return type str

Examples

```
>>> import awswrangler as wr
>>> query_exec_id = wr.athena.start_query_execution(sql='...', database='...')
```

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')
```

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_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')
```

1.3.4 Databases (Redshift, PostgreSQL, MySQL)

to_sql(df, con, **pandas_kwargs)	Write records stored in a DataFrame to a SQL database.
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 to the result set of the query string.
get_engine(db_type, host, port, database, ...)	Return a SQLAlchemy Engine from the given arguments.
get_redshift_temp_engine(cluster_identifier, ...)	Get Glue connection details.
copy_to_redshift(df, path, con, table, ...)	Load Pandas DataFrame as a Table on Amazon Redshift using parquet files on S3 as stage.
copy_files_to_redshift(path, ...[, mode, ...])	Load Parquet files from S3 to a Table on Amazon Redshift (Through COPY command).
unload_redshift(sql, path, con, iam_role[, ...])	Load Pandas DataFrame from a Amazon Redshift query result using Parquet files on s3 as stage.
unload_redshift_to_files(sql, path, con, ...)	Unload Parquet files from a Amazon Redshift query result to parquet files on s3 (Through UNLOAD command).
write_redshift_copy_manifest(manifest_path, ...)	Write Redshift copy manifest and return its structure.

awswrangler.db.to_sql

awswrangler.db.**to_sql**(*df: pandas.core.frame.DataFrame*, *con: sqlalchemy.engine.base.Engine*, ***pandas_kwargs*) → None

Write records stored in a DataFrame to a SQL database.

Support for **Redshift**, **PostgreSQL** and **MySQL**.

Support for all pandas to_sql() arguments: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_sql.html

Note: Redshift: For large DataFrames (1MM+ rows) consider the function **wr.db.copy_to_redshift()**.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

- **con** (*sqlalchemy.engine.Engine*) – SQLAlchemy Engine. Please use, wr.db.get_engine(), wr.db.get_redshift_temp_engine() or wr.catalog.get_engine()
- **pandas_kwargs** – keyword arguments forwarded to pandas.DataFrame.to_csv() https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_sql.html

Returns None.

Return type None

Examples

Writing to Redshift with temporary credentials

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.db.to_sql(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     con=wr.db.get_redshift_temp_engine(cluster_identifier="...", user="..."),
...     name="table_name",
...     schema="schema_name"
... )
```

Writing to Redshift from Glue Catalog Connections

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.db.to_sql(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     con=wr.catalog.get_engine(connection="..."),
...     name="table_name",
...     schema="schema_name"
... )
```

awswrangler.db.read_sql_query

awswrangler.db.**read_sql_query**(*sql: str*, *con: sqlalchemy.engine.base.Engine*, *index_col: Union[str, List[str], None] = None*, *params: Union[List, Tuple, Dict, None] = None*, *chunksize: Optional[int] = None*, *dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Return a DataFrame corresponding to the result set of the query string.

Support for **Redshift**, **PostgreSQL** and **MySQL**.

Note: Redshift: For large extractions (1MM+ rows) consider the function **wr.db.unload_redshift**().

Parameters

- **sql** (*str*) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **con** (*sqlalchemy.engine.Engine*) – SQLAlchemy Engine. Please use, wr.db.get_engine(), wr.db.get_redshift_temp_engine() or wr.catalog.get_engine()

- **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. Eg. for psycopg2, uses %(name)s so use params={'name' : 'value'}.
- **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.

Returns Result as Pandas DataFrame(s).

Return type Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from Redshift with temporary credentials

```
>>> import awswrangler as wr
>>> df = wr.db.read_sql_query(
...     sql="SELECT * FROM public.my_table",
...     con=wr.db.get_redshift_temp_engine(cluster_identifier="...", user="...")
... )
```

Reading from Redshift from Glue Catalog Connections

```
>>> import awswrangler as wr
>>> df = wr.db.read_sql_query(
...     sql="SELECT * FROM public.my_table",
...     con=wr.catalog.get_engine(connection="...")
... )
```

awswrangler.db.read_sql_table

awswrangler.db.**read_sql_table**(*table: str, con: sqlalchemy.engine.base.Engine, schema: Optional[str] = None, index_col: Union[str, List[str], None] = None, params: Union[List, Tuple, Dict, None] = None, chunksize: Optional[int] = None, dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Return a DataFrame corresponding to the result set of the query string.

Support for **Redshift**, **PostgreSQL** and **MySQL**.

Note: Redshift: For large extractions (1MM+ rows) consider the function *wr.db.unload_redshift()*.

Parameters

- **table** (*str*) – Nable name.
- **con** (*sqlalchemy.engine.Engine*) – SQLAlchemy Engine. Please use, wr.db.get_engine(), wr.db.get_redshift_temp_engine() or wr.catalog.get_engine()
- **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. Eg. for psycopg2, uses %(name)s so use params={'name' : 'value'}.
- **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.

Returns Result as Pandas DataFrame(s).

Return type Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from Redshift with temporary credentials

```
>>> import awswrangler as wr
>>> df = wr.db.read_sql_table(
...     table="my_table",
...     schema="public",
...     con=wr.db.get_redshift_temp_engine(cluster_identifier="...", user="...")
... )
```

Reading from Redshift from Glue Catalog Connections

```
>>> import awswrangler as wr
>>> df = wr.db.read_sql_table(
...     table="my_table",
...     schema="public",
...     con=wr.catalog.get_engine(connection="...")
... )
```

awswrangler.db.get_engine

awswrangler.db.**get_engine**(*db_type: str*, *host: str*, *port: int*, *database: str*, *user: str*, *password: str*) → sqlalchemy.engine.base.Engine

Return a SQLAlchemy Engine from the given arguments.

Only Redshift, PostgreSQL and MySQL are supported.

Parameters

- **db_type** (*str*) – Database type: "redshift", "mysql" or "postgresql".
- **host** (*str*) – Host address.
- **port** (*str*) – Port number.
- **database** (*str*) – Database name.
- **user** (*str*) – Username.
- **password** (*str*) – Password.

Returns SQLAlchemy Engine.

Return type sqlalchemy.engine.Engine

Examples

```
>>> import awswrangler as wr
>>> engine = wr.db.get_engine(
...     db_type="postgresql",
...     host="...",
...     port=1234,
...     database="...",
...     user="...",
...     password="..."
... )
```

awswrangler.db.get_redshift_temp_engine

awswrangler.db.**get_redshift_temp_engine**(*cluster_identifier: str*, *user: str*, *database: Optional[str] = None*, *duration: int = 900*, *boto3_session: Optional[boto3.session.Session] = None*) → sqlalchemy.engine.base.Engine

Get Glue connection details.

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

Returns SQLAlchemy Engine.

Return type sqlalchemy.engine.Engine

Examples

```
>>> import awswrangler as wr
>>> engine = wr.db.get_redshift_temp_engine('my_cluster', 'my_user')
```

awswrangler.db.copy_to_redshift

awswrangler.db.**copy_to_redshift**(*df: pandas.core.frame.DataFrame*, *path: str*, *con: sqlalchemy.engine.base.Engine*, *table: str*, *schema: str*, *iam_role: str*, *index: bool = False*, *dtype: Optional[Dict[str, str]] = None*, *mode: str = 'append'*, *diststyle: str = 'AUTO'*, *distkey: Optional[str] = None*, *sortstyle: str = 'COMPOUND'*, *sortkey: Optional[str] = None*, *primary_keys: Optional[List[str]] = None*, *varchar_lengths_default: int = 256*, *varchar_lengths: Optional[Dict[str, int]] = None*, *keep_files: bool = False*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*, *s3_additional_kwargs: Optional[Dict[str, str]] = None*) → None

Load Pandas DataFrame as a Table on Amazon Redshift using parquet files on S3 as stage.

This is a **HIGH** latency and **HIGH** throughput alternative to *wr.db.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.db.to_sql()* function, so it is only recommended to inserting +1MM 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 get from os.cpu_count().

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame.
- **path** (*Union[str, List[str]]*) – S3 path to write stage files (e.g. s3://bucket_name/any_name/)
- **con** (*sqlalchemy.engine.Engine*) – SQLAlchemy Engine. Please use, wr.db.get_engine(), wr.db.get_redshift_temp_engine() or wr.catalog.get_engine()
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **iam_role** (*str*) – AWS IAM role with the related permissions.
- **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.
- **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** (*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 the 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.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** – Forward to s3fs, useful for server side encryption https://s3fs.readthedocs.io/en/latest/#serverside-encryption

Returns None.

Return type None

Examples

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.db.copy_to_redshift(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path="s3://bucket/my_parquet_files/",
...     con=wr.catalog.get_engine(connection="my_glue_conn_name"),
...     table="my_table",
...     schema="public"
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
```

awswrangler.db.copy_files_to_redshift

awswrangler.db.**copy_files_to_redshift**(*path: Union[str, List[str]]*, *manifest_directory: str*, *con: sqlalchemy.engine.base.Engine*, *table: str*, *schema: str*, *iam_role: str*, *mode: str = 'append'*, *diststyle: str = 'AUTO'*, *distkey: Optional[str] = None*, *sortstyle: str = 'COMPOUND'*, *sortkey: Optional[str] = None*, *primary_keys: Optional[List[str]] = None*, *varchar_lengths_default: int = 256*, *varchar_lengths: Optional[Dict[str, int]] = None*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = 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 get from os.cpu_count().

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **manifest_directory** (*str*) – S3 prefix (e.g. s3://bucket/prefix)
- **con** (*sqlalchemy.engine.Engine*) – SQLAlchemy Engine. Please use, wr.db.get_engine(), wr.db.get_redshift_temp_engine() or wr.catalog.get_engine()
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **iam_role** (*str*) – AWS IAM role with the related permissions.
- **mode** (*str*) – Append, overwrite or upsert.
- **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** (*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_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.db.copy_files_to_redshift(
...     path="s3://bucket/my_parquet_files/",
...     con=wr.catalog.get_engine(connection="my_glue_conn_name"),
...     table="my_table",
...     schema="public"
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
```

awswrangler.db.unload_redshift

awswrangler.db.**unload_redshift**(*sql: str*, *path: str*, *con: sqlalchemy.engine.base.Engine*, *iam_role: str*, *categories: List[str] = None*, *chunked: bool = 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.db.read_sql_query()/wr.db.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.db.read_sql_query()/wr.db.read_sql_table()* function, so it is only recommended to fetch +1MM rows at once.

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 get 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** (*sqlalchemy.engine.Engine*) – SQLAlchemy Engine. Please use, wr.db.get_engine(), wr.db.get_redshift_temp_engine() or wr.catalog.get_engine()
- **iam_role** (*str*) – AWS IAM role with the related permissions.
- **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 the stage files?
- **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** (*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 s3fs, useful for server side encryption https://s3fs.readthedocs.io/en/latest/#serverside-encryption

Returns Pandas DataFrame

Return type pandas.DataFrame

Examples

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> df = wr.db.unload_redshift(
...     sql="SELECT * FROM public.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     con=wr.catalog.get_engine(connection="my_glue_connection"),
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
```

awswrangler.db.unload_redshift_to_files

awswrangler.db.**unload_redshift_to_files**(*sql: str*, *path: str*, *con: sqlalchemy.engine.base.Engine*, *iam_role: str*, *use_threads: bool = True*, *manifest: bool = False*, *partition_cols: Optional[List] = None*, *boto3_session: Optional[boto3.session.Session] = None*) → List[str]

Unload Parquet files from a Amazon Redshift query result to parquet files on s3 (Through 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 get 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** (*sqlalchemy.engine.Engine*) – SQLAlchemy Engine. Please use, wr.db.get_engine(), wr.db.get_redshift_temp_engine() or wr.catalog.get_engine()
- **iam_role** (*str*) – AWS IAM role with the related permissions.
- **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 Paths list with all unloaded files.

Return type List[str]

Examples

```
>>> import awswrangler as wr
>>> paths = wr.db.unload_redshift_to_files(
...     sql="SELECT * FROM public.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     con=wr.catalog.get_engine(connection="my_glue_connection"),
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
```

awswrangler.db.write_redshift_copy_manifest

awswrangler.db.**write_redshift_copy_manifest**(*manifest_path: str*, *paths: List[str]*, *use_threads: bool = True*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, List[Dict[str, Union[str, bool, Dict[str, int]]]]]

Write Redshift copy manifest and return its structure.

Only Parquet files are supported.

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

Parameters

- **manifest_path** (*str*) – Amazon S3 manifest path (e.g. s3://...)
- **paths** (*List[str]*) – List of S3 paths (Parquet Files) to be copied.
- **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 Manifest content.

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

Examples

Copying two files to Redshift cluster.

```
>>> import awswrangler as wr
>>> wr.db.write_redshift_copy_manifest(
...     path="s3://bucket/my.manifest",
...     paths=["s3://...parquet", "s3://...parquet"]
... )
```

1.3.5 EMR

create_cluster(cluster_name, ..., [, ...])	Create a EMR cluster with instance fleets configuration.
get_cluster_state(cluster_id[, boto3_session])	Get the EMR cluster state.
terminate_cluster(cluster_id[, boto3_session])	Terminate EMR cluster.
submit_step(cluster_id, name, command[, ...])	Submit new job in the EMR Cluster.
submit_steps(cluster_id, steps[, boto3_session])	Submit a list of steps.
build_step(name, command[, ...])	Build the Step structure (dictionary).
get_step_state(cluster_id, step_id[, ...])	Get EMR step state.

awswrangler.emr.create_cluster

awswrangler.emr.**create_cluster**(*cluster_name: str*, *logging_s3_path: str*, *emr_release: str*, *subnet_id: str*, *emr_ec2_role: str*, *emr_role: str*, *instance_type_master: str*, *instance_type_core: str*, *instance_type_task: str*, *instance_ebs_size_master: int*, *instance_ebs_size_core: int*, *instance_ebs_size_task: int*, *instance_num_on_demand_master: int*, *instance_num_on_demand_core: int*, *instance_num_on_demand_task: int*, *instance_num_spot_master: int*, *instance_num_spot_core: int*, *instance_num_spot_task: int*, *spot_bid_percentage_of_on_demand_master: int*, *spot_bid_percentage_of_on_demand_core: int*, *spot_bid_percentage_of_on_demand_task: int*, *spot_provisioning_timeout_master: int*, *spot_provisioning_timeout_core: int*, *spot_provisioning_timeout_task: int*, *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*, *spark_log_level: str = 'WARN'*, *spark_jars_path: Optional[List[str]] = None*, *spark_defaults: Optional[Dict[str, str]] = None*, *spark_pyarrow: bool = False*, *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

- **cluster_name** (*str*) – Cluster name.
- **logging_s3_path** (*str*) – Logging s3 path (e.g. s3://BUCKET_NAME/DIRECTORY_NAME/).
- **emr_release** (*str*) – EMR release (e.g. emr-5.28.0).
- **emr_ec2_role** (*str*) – IAM role name.
- **emr_role** (*str*) – IAM role name.

- **subnet_id** (*str*) – VPC subnet ID.
- **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"]).
- **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.
- **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
- **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

```
>>> import awswrangler as wr
>>> cluster_id = wr.emr.create_cluster(
...     cluster_name="wrangler_cluster",
...     logging_s3_path=f"s3://BUCKET_NAME/emr-logs/",
...     emr_release="emr-5.28.0",
...     subnet_id="SUBNET_ID",
...     emr_ec2_role="EMR_EC2_DefaultRole",
...     emr_role="EMR_DefaultRole",
...     instance_type_master="m5.xlarge",
...     instance_type_core="m5.xlarge",
...     instance_type_task="m5.xlarge",
...     instance_ebs_size_master=50,
...     instance_ebs_size_core=50,
...     instance_ebs_size_task=50,
...     instance_num_on_demand_master=1,
...     instance_num_on_demand_core=1,
...     instance_num_on_demand_task=1,
...     instance_num_spot_master=0,
...     instance_num_spot_core=1,
...     instance_num_spot_task=1,
...     spot_bid_percentage_of_on_demand_master=100,
...     spot_bid_percentage_of_on_demand_core=100,
...     spot_bid_percentage_of_on_demand_task=100,
...     spot_provisioning_timeout_master=5,
...     spot_provisioning_timeout_core=5,
...     spot_provisioning_timeout_task=5,
...     spot_timeout_to_on_demand_master=True,
...     spot_timeout_to_on_demand_core=True,
...     spot_timeout_to_on_demand_task=True,
...     python3=True,
...     spark_glue_catalog=True,
...     hive_glue_catalog=True,
...     presto_glue_catalog=True,
...     bootstraps_paths=None,
...     debugging=True,
...     applications=["Hadoop", "Spark", "Ganglia", "Hive"],
...     visible_to_all_users=True,
...     key_pair_name=None,
...     spark_jars_path=[f"s3://...jar"],
...     maximize_resource_allocation=True,
...     keep_cluster_alive_when_no_steps=True,
...     termination_protected=False,
```

(continues on next page)

(continued from previous page)

```
...     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.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")
```

awswrangler.emr.submit_step

awswrangler.emr.**submit_step**(*cluster_id: str*, *name: str*, *command: str*, *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.
- **name** (*str*) – Step name.
- **command** (*str*) – e.g. 'echo "Hello!"' e.g. for script 's3://.../script.sh arg1 arg2'
- **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.build_step

awswrangler.emr.**build_step**(*name: str*, *command: str*, *action_on_failure: str = 'CONTINUE'*, *script: bool = False*, *boto3_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Build the Step structure (dictionary).

Parameters

- **name** (*str*) – Step name.
- **command** (*str*) – e.g. 'echo "Hello!"' e.g. for script 's3://.../script.sh arg1 arg2'
- **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 structure.

Return type Dict[str, Any]

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.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")
```

1.3.6 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)*, *end_time: datetime.datetime = datetime.datetime(2020, 5, 9, 9, 37, 22, 179102)*, *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)*, *end_time: datetime.datetime = datetime.datetime(2020, 5, 9, 9, 37, 22, 179060)*, *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)*, *end_time: datetime.datetime = datetime.datetime(2020, 5, 9, 9, 37, 22, 179042)*, *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)
```

INDEX

S

T

U

W