

---

# **AWS Data Wrangler**

***Release 2.0.1***

**Igor Tavares**

**Dec 11, 2020**



# CONTENTS

<b>1</b>	<b>Read The Docs</b>	<b>3</b>
1.1	What is AWS Data Wrangler? . . . . .	3
1.2	Install . . . . .	3
1.3	API Reference . . . . .	6
	<b>Index</b>	<b>149</b>



An AWS Professional Service open source initiative | [aws-proserve-opensource@amazon.com](mailto:aws-proserve-opensource@amazon.com)

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

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

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

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

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

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

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

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

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



## READ THE DOCS

### 1.1 What is AWS Data Wrangler?

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

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

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

### 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 Python Shell Jobs

- 1 - Go to [GitHub's release page](#) and download the wheel file (.whl) related to the desired version.
- 2 - Upload the wheel file to any Amazon S3 location.
- 3 - Go to your Glue Python Shell job and point to the wheel file on S3 in the *Python library path* field.

[Official Glue Python Shell Reference](#)

### 1.2.5 AWS Glue PySpark Jobs

---

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

---

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

- Key: `--additional-python-modules`
- Value: `awswrangler`

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

- Value: `awswrangler==1.10.0`

[Official Glue PySpark Reference](#)

### 1.2.6 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
```

### 1.2.7 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
```

(continues on next page)



(continued from previous page)

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

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

# PARAMETERS
PACKAGE=aws wrangler

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

## 1.2.8 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 aws wrangler
```

**Note:** Make sure to freeze the Wrangler version in the bootstrap for productive environments (e.g. `aws wrangler==1.8.1`)

## 1.2.9 From Source

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

## 1.3 API Reference

- *Amazon S3*
- *AWS Glue Catalog*
- *Amazon Athena*
- *Amazon Redshift*
- *PostgreSQL*
- *MySQL*
- *Amazon Timestream*
- *Amazon EMR*
- *Amazon CloudWatch Logs*
- *Amazon QuickSight*
- *AWS STS*
- *Global Configurations*

### 1.3.1 Amazon S3

<code>copy_objects(paths, source_path, target_path)</code>	Copy a list of S3 objects to another S3 directory.
<code>delete_objects(path[, use_threads, ...])</code>	Delete Amazon S3 objects from a received S3 prefix or list of S3 objects paths.
<code>describe_objects(path[, use_threads, ...])</code>	Describe Amazon S3 objects from a received S3 prefix or list of S3 objects paths.
<code>does_object_exist(path[, boto3_session])</code>	Check if object exists on S3.
<code>get_bucket_region(bucket[, boto3_session])</code>	Get bucket region name.
<code>list_directories(path[, boto3_session])</code>	List Amazon S3 objects from a prefix.
<code>list_objects(path[, suffix, ignore_suffix, ...])</code>	List Amazon S3 objects from a prefix.
<code>merge_datasets(source_path, target_path[, ...])</code>	Merge a source dataset into a target dataset.
<code>read_csv(path[, path_suffix, ...])</code>	Read CSV file(s) from from a received S3 prefix or list of S3 objects paths.
<code>read_fwf(path[, path_suffix, ...])</code>	Read fixed-width formatted file(s) from from a received S3 prefix or list of S3 objects paths.
<code>read_json(path[, path_suffix, ...])</code>	Read JSON file(s) from from a received S3 prefix or list of S3 objects paths.
<code>read_parquet(path[, path_suffix, ...])</code>	Read Apache Parquet file(s) from from a received S3 prefix or list of S3 objects paths.
<code>read_parquet_metadata(path[, path_suffix, ...])</code>	Read Apache Parquet file(s) metadata from from a received S3 prefix or list of S3 objects paths.

continues on next page

Table 1 – continued from previous page

<code>read_parquet_table(table, database[, ...])</code>	Read Apache Parquet table registered on AWS Glue Catalog.
<code>size_objects(path[, use_threads, boto3_session])</code>	Get the size (ContentLength) in bytes of Amazon S3 objects from a received S3 prefix or list of S3 objects paths.
<code>store_parquet_metadata(path, database, table)</code>	Infer and store parquet metadata on AWS Glue Catalog.
<code>to_csv(df, path[, sep, index, columns, ...])</code>	Write CSV file or dataset on Amazon S3.
<code>to_json(df, path[, boto3_session, ...])</code>	Write JSON file on Amazon S3.
<code>to_parquet(df, path[, index, compression, ...])</code>	Write Parquet file or dataset on Amazon S3.
<code>wait_objects_exist(paths[, delay, ...])</code>	Wait Amazon S3 objects exist.
<code>wait_objects_not_exist(paths[, delay, ...])</code>	Wait Amazon S3 objects not exist.

### aws wrangler.s3.copy\_objects

`aws wrangler.s3.copy_objects` (*paths*: *List[str]*, *source\_path*: *str*, *target\_path*: *str*, *replace\_filenames*: *Optional[Dict[str, str]] = None*, *use\_threads*: *bool = True*, *boto3\_session*: *Optional[boto3.session.Session] = None*, *s3\_additional\_kwargs*: *Optional[Dict[str, Any]] = None*) → *List[str]*

Copy a list of S3 objects to another S3 directory.

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

#### Parameters

- **paths** (*List[str]*) – List of S3 objects paths (e.g. `[s3://bucket/dir0/key0, s3://bucket/dir0/key1]`).
- **source\_path** (*str*,) – S3 Path for the source directory.
- **target\_path** (*str*,) – S3 Path for the target directory.
- **replace\_filenames** (*Dict[str, str]*, *optional*) – e.g. `{“old_name.csv”: “new_name.csv”, “old_name2.csv”: “new_name2.csv”}`
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if *boto3\_session* receive None.
- **s3\_additional\_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests. Valid parameters: “ACL”, “Metadata”, “ServerSideEncryption”, “StorageClass”, “SSECustomerAlgorithm”, “SSECustomerKey”, “SSEKMSKeyId”, “SSEKMSEncryption-Context”, “Tagging”. e.g. `s3_additional_kwargs={‘ServerSideEncryption’: ‘aws:kms’, ‘SSEKMSKeyId’: ‘YOUR_KMS_KEY_ARN’}`

**Returns** List of new objects paths.

**Return type** *List[str]*

## Examples

### Copying

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

### Copying with a KMS key

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

## awswrangler.s3.delete\_objects

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

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

This function accepts Unix shell-style wildcards in the path argument. \* (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq).

---

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

---

---

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

---

### Parameters

- **path** (`Union[str, List[str]]`) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **use\_threads** (`bool`) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **last\_modified\_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.

- **last\_modified\_end**(*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **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]], use\_threads: bool = True, last\_modified\_begin: Optional[datetime.datetime] = None, last\_modified\_end: Optional[datetime.datetime] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Dict[str, Dict[str, Any]]

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

Fetch attributes like ContentLength, DeleteMarker, last\_modified, ContentType, etc The full list of attributes can be explored under the boto3 head\_object documentation: [https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.head\\_object](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.head_object)

This function accepts Unix shell-style wildcards in the path argument. \* (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq).

---

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

---



---

**Note:** The filter by last\_modified begin last\_modified end is applied after list all S3 files

---

## Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. s3://bucket/prefix) or list of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **last\_modified\_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last\_modified\_end**(*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.

- **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](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.head_object)

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

## Examples

```
>>> import awswrangler as wr
>>> descs0 = wr.s3.describe_objects(['s3://bucket/key0', 's3://bucket/key1']) # Describe both objects
>>> descs1 = wr.s3.describe_objects('s3://bucket/prefix') # Describe all objects under the prefix
```

## awswrangler.s3.does\_object\_exist

**awswrangler.s3.does\_object\_exist** (*path: str*, *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
>>> wr.s3.does_object_exist('s3://bucket/key_unreal', boto3_session=boto3.Session())
False
```

**aws wrangler.s3.get\_bucket\_region**

`aws wrangler.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 aws wrangler as wr
>>> region = wr.s3.get_bucket_region('bucket-name')
```

Using a custom boto3 session

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

**aws wrangler.s3.list\_directories**

`aws wrangler.s3.list_directories (path: str, boto3_session: Optional[boto3.session.Session] = None) → List[str]`

List Amazon S3 objects from a prefix.

This function accepts Unix shell-style wildcards in the path argument. \* (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq).

**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_directories('s3://bucket/prefix/')
['s3://bucket/prefix/dir0/', 's3://bucket/prefix/dir1/', 's3://bucket/prefix/dir2/']
```

Using a custom boto3 session

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

## awswrangler.s3.list\_objects

```
awswrangler.s3.list_objects(path: str, suffix: Optional[Union[str, List[str]]] =
                             None, ignore_suffix: Optional[Union[str, List[str]]] =
                             None, last_modified_begin: Optional[datetime.datetime] =
                             None, last_modified_end: Optional[datetime.datetime] =
                             None, ignore_empty: bool = False, boto3_session: Op-
                             tional[boto3.session.Session] = None) → List[str]
```

List Amazon S3 objects from a prefix.

This function accepts Unix shell-style wildcards in the path argument. \* (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq).

---

**Note:** The filter by last\_modified begin last\_modified end is applied after list all S3 files

---

### Parameters

- **path** (*str*) – S3 path (e.g. s3://bucket/prefix).
- **suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for filtering S3 keys.
- **ignore\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.
- **last\_modified\_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last\_modified\_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **ignore\_empty** (*bool*) – Ignore files with 0 bytes.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** List of objects paths.

**Return type** List[str]



## Examples

Using the default boto3 session

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

Using a custom boto3 session

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

## awswrangler.s3.merge\_datasets

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

Merge a source dataset into a target dataset.

This function accepts Unix shell-style wildcards in the `source_path` argument. `*` (matches everything), `?` (matches any single character), `[seq]` (matches any character in seq), `[!seq]` (matches any character not in seq).

---

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

---



---

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

---

### Parameters

- **source\_path** (*str*,) – S3 Path for the source directory.
- **target\_path** (*str*,) – S3 Path for the target directory.
- **mode** (*str*, optional) – `append` (Default), `overwrite`, `overwrite_partitions`.
- **ignore\_empty** (*bool*) – Ignore files with 0 bytes.
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **boto3\_session** (*boto3.Session()*, optional) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3\_additional\_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests. Valid parameters: “ACL”, “Metadata”, “ServerSideEncryption”, “StorageClass”, “SSECustomerAlgorithm”, “SSECustomerKey”, “SSEKMSKeyId”, “SSEKMSEncryption-Context”, “Tagging”. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`

**Returns** List of new objects paths.

**Return type** List[str]

## Examples

### Merging

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

### Merging with a KMS key

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

## awswrangler.s3.read\_csv

`awswrangler.s3.read_csv` (*path*: Union[str, List[str]], *path\_suffix*: Optional[Union[str, List[str]]] = None, *path\_ignore\_suffix*: Optional[Union[str, List[str]]] = None, *ignore\_empty*: bool = True, *use\_threads*: bool = True, *last\_modified\_begin*: Optional[datetime.datetime] = None, *last\_modified\_end*: Optional[datetime.datetime] = None, *boto3\_session*: Optional[boto3.session.Session] = None, *s3\_additional\_kwargs*: Optional[Dict[str, Any]] = None, *chunksize*: Optional[int] = None, *dataset*: bool = False, *partition\_filter*: Optional[Callable[[Dict[str, str]], bool]] = None, *\*\*pandas\_kwargs*: Any) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

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

This function accepts Unix shell-style wildcards in the path argument. \* (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq).

---

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

---

---

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

---

---

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

---

### Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. [`s3://bucket/key0`, `s3://bucket/key1`]).
- **path\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. [`".csv"`]). If `None`, will try to read all files. (default)
- **path\_ignore\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. [`"_SUCCESS"`]). If `None`, will try to read all files. (default)
- **ignore\_empty** (*bool*) – Ignore files with 0 bytes.
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **last\_modified\_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last\_modified\_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.
- **s3\_additional\_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **dataset** (*bool*) – If `True` read a CSV dataset instead of simple file(s) loading all the related partitions as columns.
- **partition\_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (`Dict[str, str]`) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False to ignore it. Ignored if `dataset=False`. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False` <https://github.com/awslabs/aws-data-wrangler/blob/master/tutorials/023%20-%20Flexible%20Partitions%20Filter.ipynb>
- **pandas\_kwargs** – KEYWORD arguments forwarded to `pandas.read_csv()`. You can NOT pass `pandas_kwargs` explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. `wr.s3.read_csv('s3://bucket/prefix/', sep='|', na_values=['null', 'none'], skip_blank_lines=True)` [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read\\_csv.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html)

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

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

## Examples

Reading all CSV files under a prefix

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

Reading all CSV files under a prefix and using pandas\_kwargs

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

Reading all CSV files from a list

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

Reading in chunks of 100 lines

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

Reading CSV Dataset with PUSH-DOWN filter over partitions

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

## awswrangler.s3.read\_fwf

`awswrangler.s3.read_fwf` (path: Union[str, List[str]], path\_suffix: Optional[Union[str, List[str]]] = None, path\_ignore\_suffix: Optional[Union[str, List[str]]] = None, ignore\_empty: bool = True, use\_threads: bool = True, last\_modified\_begin: Optional[datetime.datetime] = None, last\_modified\_end: Optional[datetime.datetime] = None, boto3\_session: Optional[boto3.session.Session] = None, s3\_additional\_kwargs: Optional[Dict[str, Any]] = None, chunksize: Optional[int] = None, dataset: bool = False, partition\_filter: Optional[Callable[[Dict[str, str]], bool]] = None, \*\*pandas\_kwargs: Any) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

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

This function accepts Unix shell-style wildcards in the path argument. \* (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq).

---

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

---

---

**Note:** In case of `use_threads=True` the number of threads that will be spawned will be gotten from

---

---

`os.cpu_count()`.

---



---

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

---

### Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **path\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. `[".txt"]`). If `None`, will try to read all files. (default)
- **path\_ignore\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. `["_SUCCESS"]`). If `None`, will try to read all files. (default)
- **ignore\_empty** (*bool*) – Ignore files with 0 bytes.
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **last\_modified\_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last\_modified\_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.
- **s3\_additional\_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **dataset** (*bool*) – If `True` read a FWF dataset instead of simple file(s) loading all the related partitions as columns.
- **partition\_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (`Dict[str, str]`) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False to ignore it. Ignored if `dataset=False`. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False` <https://github.com/aws-labs/aws-data-wrangler/blob/master/tutorials/023%20-%20Flexible%20Partitions%20Filter.ipynb>
- **pandas\_kwargs** – KEYWORD arguments forwarded to `pandas.read_fwf()`. You can NOT pass `pandas_kwargs` explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. `wr.s3.read_fwf(path='s3://bucket/prefix/', widths=[1, 3], names=["c0", "c1"])` [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read\\_fwf.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_fwf.html)

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

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

## Examples

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

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

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

```
>>> import awswrangler as wr
>>> df = wr.s3.read_fwf(path=['s3://bucket/0.txt', 's3://bucket/1.txt'],
↳ widths=[1, 3], names=['c0', 'c1'])
```

Reading in chunks of 100 lines

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_fwf(
...     path=['s3://bucket/0.txt', 's3://bucket/1.txt'],
...     chunksize=100,
...     widths=[1, 3],
...     names=["c0", "c1"]
... )
>>> for df in dfs:
>>>     print(df) # 100 lines Pandas DataFrame
```

Reading FWF Dataset with PUSH-DOWN filter over partitions

```
>>> import awswrangler as wr
>>> my_filter = lambda x: True if x["city"].startswith("new") else False
>>> df = wr.s3.read_fwf(path, dataset=True, partition_filter=my_filter, widths=[1,
↳ 3], names=["c0", "c1"])
```

## awswrangler.s3.read\_json

`awswrangler.s3.read_json` (*path*: Union[str, List[str]], *path\_suffix*: Optional[Union[str, List[str]]] = None, *path\_ignore\_suffix*: Optional[Union[str, List[str]]] = None, *ignore\_empty*: bool = True, *orient*: str = 'columns', *use\_threads*: bool = True, *last\_modified\_begin*: Optional[datetime.datetime] = None, *last\_modified\_end*: Optional[datetime.datetime] = None, *boto3\_session*: Optional[boto3.session.Session] = None, *s3\_additional\_kwargs*: Optional[Dict[str, Any]] = None, *chunksize*: Optional[int] = None, *dataset*: bool = False, *partition\_filter*: Optional[Callable[[Dict[str, str]], bool]] = None, *\*\*pandas\_kwargs*: Any) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

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

This function accepts Unix shell-style wildcards in the path argument. \* (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq).

---

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

---



---

**Note:** In case of `use_threads=True` the number of threads that will be spawned will be gotten from

---

---

`os.cpu_count()`.

---



---

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

---

### Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **path\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. `[“.json”]`). If `None`, will try to read all files. (default)
- **path\_ignore\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. `[“_SUCCESS”]`). If `None`, will try to read all files. (default)
- **ignore\_empty** (*bool*) – Ignore files with 0 bytes.
- **orient** (*str*) – Same as Pandas: [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read\\_json.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html)
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **last\_modified\_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last\_modified\_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.
- **s3\_additional\_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **dataset** (*bool*) – If `True` read a JSON dataset instead of simple file(s) loading all the related partitions as columns. If `True`, the `lines=True` will be assumed by default.
- **partition\_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (`Dict[str, str]`) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False to ignore it. Ignored if `dataset=False`. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False` <https://github.com/aws-labs/aws-data-wrangler/blob/master/tutorials/023%20-%20Flexible%20Partitions%20Filter.ipynb>
- **pandas\_kwargs** – KEYWORD arguments forwarded to `pandas.read_json()`. You can NOT pass `pandas_kwargs` explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. `wr.s3.read_json('s3://bucket/prefix/', lines=True, keep_default_dates=True)` [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read\\_json.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html)

**Returns** Pandas DataFrame or a Generator in case of *chunksize*  $\neq$  *None*.

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

## Examples

Reading all JSON files under a prefix

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

Reading all CSV files under a prefix and using pandas\_kwargs

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

Reading all JSON files from a list

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

Reading in chunks of 100 lines

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

Reading JSON Dataset with PUSH-DOWN filter over partitions

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

## awswrangler.s3.read\_parquet

`awswrangler.s3.read_parquet` (path: Union[str, List[str]], path\_suffix: Optional[Union[str, List[str]]] = None, path\_ignore\_suffix: Optional[Union[str, List[str]]] = None, ignore\_empty: bool = True, partition\_filter: Optional[Callable[[Dict[str, str]], bool]] = None, columns: Optional[List[str]] = None, validate\_schema: bool = False, chunked: Union[bool, int] = False, dataset: bool = False, categories: Optional[List[str]] = None, safe: bool = True, use\_threads: bool = True, last\_modified\_begin: Optional[datetime.datetime] = None, last\_modified\_end: Optional[datetime.datetime] = None, boto3\_session: Optional[boto3.session.Session] = None, s3\_additional\_kwargs: Optional[Dict[str, Any]] = None) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

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

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



This function accepts Unix shell-style wildcards in the path argument. \* (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq).

---

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

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

There are two batching strategies on Wrangler:

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

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

---



---

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

---



---

**Note:** The filter by *last\_modified* begin *last\_modified* end is applied after list all S3 files

---

### Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **path\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. `[“.gz.parquet”, “.snappy.parquet”]`). If *None*, will try to read all files. (default)
- **path\_ignore\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. `[“.csv”, “_SUCCESS”]`). If *None*, will try to read all files. (default)
- **ignore\_empty** (*bool*) – Ignore files with 0 bytes.
- **partition\_filter** (*Optional[Callable[[Dict[str, str], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (*Dict[str, str]*) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, *True* to read the partition or *False* to ignore it. Ignored if *dataset=False*. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False`
- **columns** (*List[str], optional*) – Names of columns to read from the file(s).
- **validate\_schema** – Check that individual file schemas are all the same / compatible. Schemas within a folder prefix should all be the same. Disable if you have schemas that are different and want to disable this check.
- **chunked** (*Union[int, bool]*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* wrangler will iterate on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed Wrangler will iterate on the data by number of rows equal the received INTEGER.
- **dataset** (*bool*) – If *True* read a parquet dataset instead of simple file(s) loading all the related partitions as columns.

- **categories** (*Optional[List[str]], optional*) – List of columns names that should be returned as `pandas.Categorical`. Recommended for memory restricted environments.
- **safe** (*bool, default True*) – For certain data types, a cast is needed in order to store the data in a pandas DataFrame or Series (e.g. timestamps are always stored as nanoseconds in pandas). This option controls whether it is a safe cast or not.
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **last\_modified\_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last\_modified\_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3\_additional\_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.

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

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

## Examples

Reading all Parquet files under a prefix

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

Reading all Parquet files from a list

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

Reading in chunks (Chunk by file)

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_parquet(path=['s3://bucket/filename0.csv', 's3://bucket/
↳ filename1.csv'], chunked=True)
>>> for df in dfs:
>>>     print(df) # Smaller Pandas DataFrame
```

Reading in chunks (Chunk by IMM rows)

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_parquet(path=['s3://bucket/filename0.csv', 's3://bucket/
↳ filename1.csv'], chunked=1_000_000)
>>> for df in dfs:
>>>     print(df) # IMM Pandas DataFrame
```

Reading Parquet Dataset with PUSH-DOWN filter over partitions

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

## awswrangler.s3.read\_parquet\_metadata

```
awswrangler.s3.read_parquet_metadata(path: Union[str, List[str]], path_suffix: Optional[str]
                                     = None, path_ignore_suffix: Optional[str] =
                                     None, ignore_empty: bool = True, dtype: Op-
                                     tional[Dict[str, str]] = None, sampling: float = 1.0,
                                     dataset: bool = False, use_threads: bool = True,
                                     boto3_session: Optional[boto3.session.Session] =
                                     None, s3_additional_kwargs: Optional[Dict[str, Any]]
                                     = None) → Any
```

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

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

This function accepts Unix shell-style wildcards in the path argument. \* (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq).

---

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

---



---

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

---

Check out the [Global Configurations Tutorial](#) for details.

### Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **path\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. `[“.gz.parquet”, “.snappy.parquet”]`). If None, will try to read all files. (default)
- **path\_ignore\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. `[“.csv”, “_SUCCESS”]`). If None, will try to read all files. (default)
- **ignore\_empty** (*bool*) – Ignore files with 0 bytes.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined data types as partitions columns. (e.g. `{‘col name’: ‘bigint’, ‘col2 name’: ‘int’}`)
- **sampling** (*float*) – Random sample ratio of files that will have the metadata inspected. Must be  $0.0 < sampling \leq 1.0$ . The higher, the more accurate. The lower, the faster.
- **dataset** (*bool*) – If True read a parquet dataset instead of simple file(s) loading all the related partitions as columns.
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.

- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.
- **s3\_additional\_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.

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

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

## Examples

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

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

Reading all Parquet files metadata from a list

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

## awswrangler.s3.read\_parquet\_table

`awswrangler.s3.read_parquet_table` (*table: str, database: str, catalog\_id: Optional[str] = None, partition\_filter: Optional[Callable[[Dict[str, str]], bool]] = None, columns: Optional[List[str]] = None, validate\_schema: bool = True, categories: Optional[List[str]] = None, safe: bool = True, chunked: Union[bool, int] = False, use\_threads: bool = True, boto3\_session: Optional[boto3.session.Session] = None, s3\_additional\_kwargs: Optional[Dict[str, Any]] = None*)  
→ Any

Read Apache Parquet table registered on AWS Glue Catalog.

---

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

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

There are two batching strategies on Wrangler:

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

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

---

---

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

---

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

---

### Parameters

- **table** (*str*) – AWS Glue Catalog table name.
- **database** (*str*) – AWS Glue Catalog database name.
- **catalog\_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **partition\_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (Dict[str, str]) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False to ignore it. Ignored if `dataset=False`. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False` <https://github.com/aws-labs/aws-data-wrangler/blob/master/tutorials/023%20-%20Flexible%20Partitions%20Filter.ipynb>
- **columns** (*List[str]*, *optional*) – Names of columns to read from the file(s).
- **validate\_schema** – Check that individual file schemas are all the same / compatible. Schemas within a folder prefix should all be the same. Disable if you have schemas that are different and want to disable this check.
- **categories** (*Optional[List[str]]*, *optional*) – List of columns names that should be returned as `pandas.Categorical`. Recommended for memory restricted environments.
- **safe** (*bool*, *default True*) – For certain data types, a cast is needed in order to store the data in a `pandas.DataFrame` or `Series` (e.g. timestamps are always stored as nanoseconds in `pandas`). This option controls whether it is a safe cast or not.
- **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** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.

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

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

## Examples

### Reading Parquet Table

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

### Reading Parquet Table encrypted

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

### Reading Parquet Table in chunks (Chunk by file)

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

### Reading Parquet Dataset with PUSH-DOWN filter over partitions

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

## awswrangler.s3.size\_objects

`awswrangler.s3.size_objects` (*path*: Union[str, List[str]], *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.

This function accepts Unix shell-style wildcards in the path argument. \* (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq).

---

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

---

### Parameters

- **path** (Union[str, List[str]]) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **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
```

### awswrangler.s3.store\_parquet\_metadata

`awswrangler.s3.store_parquet_metadata` (*path: str, database: str, table: str, catalog\_id: Optional[str] = None, path\_suffix: Optional[str] = None, path\_ignore\_suffix: Optional[str] = None, ignore\_empty: bool = True, dtype: Optional[Dict[str, str]] = None, sampling: float = 1.0, dataset: bool = False, use\_threads: bool = True, description: Optional[str] = None, parameters: Optional[Dict[str, str]] = None, columns\_comments: Optional[Dict[str, str]] = None, compression: Optional[str] = None, mode: str = 'overwrite', catalog\_versioning: bool = False, regular\_partitions: bool = True, projection\_enabled: bool = False, projection\_types: Optional[Dict[str, str]] = None, projection\_ranges: Optional[Dict[str, str]] = None, projection\_values: Optional[Dict[str, str]] = None, projection\_intervals: Optional[Dict[str, str]] = None, projection\_digits: Optional[Dict[str, str]] = None, s3\_additional\_kwargs: Optional[Dict[str, Any]] = None, boto3\_session: Optional[boto3.session.Session] = None) → Any*

Infer and store parquet metadata on AWS Glue Catalog.

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

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

This function accepts Unix shell-style wildcards in the path argument. \* (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq).

---

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

---



---

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

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

---

### Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).  
database : str Glue/Athena catalog: Database name.
- **table** (*str*) – Glue/Athena catalog: Table name.
- **database** (*str*) – AWS Glue Catalog database name.
- **catalog\_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **path\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for filtering S3 keys.
- **path\_ignore\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.
- **ignore\_empty** (*bool*) – Ignore files with 0 bytes.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined data types as partitions columns. (e.g. `{‘col name’: ‘bigint’, ‘col2 name’: ‘int’}`)
- **sampling** (*float*) – Random sample ratio of files that will have the metadata inspected. Must be  $0.0 < sampling \leq 1.0$ . The higher, the more accurate. The lower, the faster.
- **dataset** (*bool*) – If True read a parquet dataset instead of simple file(s) loading all the related partitions as columns.
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **description** (*str, optional*) – Glue/Athena catalog: Table description
- **parameters** (*Dict[str, str], optional*) – Glue/Athena catalog: Key/value pairs to tag the table.
- **columns\_comments** (*Dict[str, str], optional*) – Glue/Athena catalog: Columns names and the related comments (e.g. `{‘col0’: ‘Column 0.’, ‘col1’: ‘Column 1.’, ‘col2’: ‘Partition.’}`).
- **compression** (*str, optional*) – Compression style (None, snappy, gzip, etc).
- **mode** (*str*) – ‘overwrite’ to recreate any possible existing table or ‘append’ to keep any possible existing table.
- **catalog\_versioning** (*bool*) – If True and *mode=“overwrite”*, creates an archived version of the table catalog before updating it.
- **regular\_partitions** (*bool*) – Create regular partitions (Non projected partitions) on Glue Catalog. Disable when you will work only with Partition Projection. Keep enabled



even when working with projections is useful to keep Redshift Spectrum working with the regular partitions.

- **projection\_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection\_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: “enum”, “integer”, “date”, “injected” <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': 'enum', 'col2\_name': 'integer'})
- **projection\_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '0,10', 'col2\_name': '-1,8675309'})
- **projection\_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': 'A,B,Unknown', 'col2\_name': 'foo,boo,bar'})
- **projection\_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '1', 'col2\_name': '5'})
- **projection\_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '1', 'col2\_name': '2'})
- **s3\_additional\_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests. Valid parameters: “ACL”, “Metadata”, “ServerSideEncryption”, “StorageClass”, “SSECustomerAlgorithm”, “SSECustomerKey”, “SSEKMSKeyId”, “SSEKMSEncryptionContext”, “Tagging”. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

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

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

## Examples

Reading all Parquet files metadata under a prefix

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

## aws wrangler.s3.to\_csv

```
aws wrangler.s3.to_csv(df: 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, Any]] = None, sanitize_columns: bool = False, dataset: bool = False, partition_cols: Optional[List[str]] = None, concurrent_partitioning: bool = False, mode: Optional[str] = None, catalog_versioning: bool = False, database: Optional[str] = None, table: Optional[str] = None, dtype: Optional[Dict[str, str]] = None, description: Optional[str] = None, parameters: Optional[Dict[str, str]] = None, columns_comments: Optional[Dict[str, str]] = None, regular_partitions: bool = True, projection_enabled: bool = False, projection_types: Optional[Dict[str, str]] = None, projection_ranges: Optional[Dict[str, str]] = None, projection_values: Optional[Dict[str, str]] = None, projection_intervals: Optional[Dict[str, str]] = None, projection_digits: Optional[Dict[str, str]] = None, catalog_id: Optional[str] = None, **pandas_kwargs: Any) → Any
```

Write CSV file or dataset on Amazon S3.

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

---

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

---

---

**Note:** If `dataset=True`, `pandas_kwargs` will be ignored due restrictive quoting, `date_format`, `escapechar`, encoding, etc required by Athena/Glue Catalog.

---

---

**Note:** By now Pandas does not support in-memory CSV compression. <https://github.com/pandas-dev/pandas/issues/22555> So the `compression` will not be supported on Wrangler too.

---

---

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

---

---

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

---

---

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

- `catalog_id`
- `concurrent_partitioning`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

---

## Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **path** (*str*) – 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** (*Optional[Dict[str, Any]]*) – Forward to botocore requests. Valid parameters: “ACL”, “Metadata”, “ServerSideEncryption”, “StorageClass”, “SSECustomerAlgorithm”, “SSECustomerKey”, “SSEKMSKeyId”, “SSEKMSEncryption-Context”, “Tagging”. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`
- **sanitize\_columns** (*bool*) – True to sanitize columns names or False to keep it as is. True value is forced if `dataset=True`.
- **dataset** (*bool*) – If True store a parquet dataset instead of a ordinary file(s) If True, enable all follow arguments: `partition_cols`, `mode`, `database`, `table`, `description`, `parameters`, `columns_comments`, `concurrent_partitioning`, `catalog_versioning`, `projection_enabled`, `projection_types`, `projection_ranges`, `projection_values`, `projection_intervals`, `projection_digits`, `catalog_id`, `schema_evolution`.
- **partition\_cols** (*List[str], optional*) – List of column names that will be used to create partitions. Only takes effect if `dataset=True`.
- **concurrent\_partitioning** (*bool*) – If True will increase the parallelism level during the partitions writing. It will decrease the writing time and increase the memory usage. <https://github.com/awslabs/aws-data-wrangler/blob/master/tutorials/022%20-%20Writing%20Partitions%20Concurrently.ipynb>
- **mode** (*str, optional*) – `append` (Default), `overwrite`, `overwrite_partitions`. Only takes effect if `dataset=True`. For details check the related tutorial: [https://aws-data-wrangler.readthedocs.io/en/stable/stubs/awswrangler.s3.to\\_parquet.html#awswrangler.s3.to\\_parquet](https://aws-data-wrangler.readthedocs.io/en/stable/stubs/awswrangler.s3.to_parquet.html#awswrangler.s3.to_parquet)
- **catalog\_versioning** (*bool*) – If True and `mode="overwrite"`, creates an archived version of the table catalog before updating it.
- **database** (*str, optional*) – Glue/Athena catalog: Database name.
- **table** (*str, optional*) – Glue/Athena catalog: Table name.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. `{'col name': 'bigint', 'col2 name': 'int'}`)
- **description** (*str, optional*) – Glue/Athena catalog: Table description
- **parameters** (*Dict[str, str], optional*) – Glue/Athena catalog: Key/value pairs to tag the table.

- **columns\_comments** (*Dict[str, str], optional*) – Glue/Athena catalog: Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **regular\_partitions** (*bool*) – Create regular partitions (Non projected partitions) on Glue Catalog. Disable when you will work only with Partition Projection. Keep enabled even when working with projections is useful to keep Redshift Spectrum working with the regular partitions.
- **projection\_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection\_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: “enum”, “integer”, “date”, “injected” <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': 'enum', 'col2\_name': 'integer'})
- **projection\_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '0,10', 'col2\_name': '-1,8675309'})
- **projection\_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': 'A,B,Unknown', 'col2\_name': 'foo,boo,bar'})
- **projection\_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '1', 'col2\_name': '5'})
- **projection\_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '1', 'col2\_name': '2'})
- **catalog\_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **pandas\_kwargs** – KEYWORD arguments forwarded to `pandas.DataFrame.to_csv()`. You can NOT pass `pandas_kwargs` explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. `wr.s3.to_csv(df, path, sep='|', na_rep='NULL', decimal=',')` [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to\\_csv.html](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html)

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

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

## Examples

### Writing single file

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

### Writing single file with pandas\_kwargs

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

### Writing single file encrypted with a KMS key

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

### Writing partitioned dataset

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

(continues on next page)

(continued from previous page)

```

...     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'],
  '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, Any]] = None, use_threads: bool = True, **pandas_kwargs: Any) → None`

Write JSON file on Amazon S3.

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

### Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **path** (*str*) – Amazon S3 path (e.g. `s3://bucket/filename.csv`).
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3\_session receive None.
- **s3\_additional\_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests. Valid parameters: “ACL”, “Metadata”, “ServerSideEncryption”, “StorageClass”, “SSECustomerAlgorithm”, “SSECustomerKey”, “SSEKMSKeyId”, “SSEKMSEncryptionContext”, “Tagging”. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **pandas\_kwargs** – KEYWORD arguments forwarded to `pandas.DataFrame.to_json()`. You can NOT pass `pandas_kwargs` explicit, just add valid Pandas arguments in the function call and Wrangler will accept it. e.g. `wr.s3.to_json(df, path, lines=True, date_format='iso')` [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to\\_json.html](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 JSON file using pandas\_kwargs

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

(continues on next page)

(continued from previous page)

```
...     path='s3://bucket/filename.json',
...     lines=True,
...     date_format='iso'
... )
```

Writing CSV file encrypted with a KMS key

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

## awswrangler.s3.to\_parquet

`awswrangler.s3.to_parquet` (*df*: `pandas.core.frame.DataFrame`, *path*: `str`, *index*: `bool` = `False`, *compression*: `Optional[str]` = `'snappy'`, *max\_rows\_by\_file*: `Optional[int]` = `None`, *use\_threads*: `bool` = `True`, *boto3\_session*: `Optional[boto3.session.Session]` = `None`, *s3\_additional\_kwargs*: `Optional[Dict[str, Any]]` = `None`, *sanitize\_columns*: `bool` = `False`, *dataset*: `bool` = `False`, *partition\_cols*: `Optional[List[str]]` = `None`, *concurrent\_partitioning*: `bool` = `False`, *mode*: `Optional[str]` = `None`, *catalog\_versioning*: `bool` = `False`, *schema\_evolution*: `bool` = `True`, *database*: `Optional[str]` = `None`, *table*: `Optional[str]` = `None`, *dtype*: `Optional[Dict[str, str]]` = `None`, *description*: `Optional[str]` = `None`, *parameters*: `Optional[Dict[str, str]]` = `None`, *columns\_comments*: `Optional[Dict[str, str]]` = `None`, *regular\_partitions*: `bool` = `True`, *projection\_enabled*: `bool = False, projection_types: Optional[Dict[str, str]] = None, projection_ranges: Optional[Dict[str, str]] = None, projection_values: Optional[Dict[str, str]] = None, projection_intervals: Optional[Dict[str, str]] = None, projection_digits: Optional[Dict[str, str]] = None, catalog_id: Optional[str] = None) → Any`

Write Parquet file or dataset on Amazon S3.

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

---

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

---



---

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

---



---

**Note:** In case of `use_threads=True` the number of threads that will be spawned will be gotten from

---



---

`os.cpu_count()`.

---

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

- `catalog_id`
- `concurrent_partitioning`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

---

### Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **path** (*str*) – 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).
- **max\_rows\_by\_file** (*int*) – Max number of rows in each file. Default is None i.e. dont split the files. (e.g. 33554432, 268435456)
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3\_additional\_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests. Valid parameters: “ACL”, “Metadata”, “ServerSideEncryption”, “StorageClass”, “SSECustomerAlgorithm”, “SSECustomerKey”, “SSEKMSKeyId”, “SSEKMSEncryption-Context”, “Tagging”. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`
- **sanitize\_columns** (*bool*) – True to sanitize columns names (using `wr.catalog.sanitize_table_name` and `wr.catalog.sanitize_column_name`) or False to keep it as is. True value behaviour is enforced if `database` and `table` arguments are passed.
- **dataset** (*bool*) – If True store a parquet dataset instead of a ordinary file(s) If True, enable all follow arguments: `partition_cols`, `mode`, `database`, `table`, `description`, `parameters`, `columns_comments`, `concurrent_partitioning`, `catalog_versioning`, `projection_enabled`, `projection_types`, `projection_ranges`, `projection_values`, `projection_intervals`, `projection_digits`, `catalog_id`, `schema_evolution`.
- **partition\_cols** (*List[str], optional*) – List of column names that will be used to create partitions. Only takes effect if `dataset=True`.
- **concurrent\_partitioning** (*bool*) – If True will increase the parallelism level during the partitions writing. It will decrease the writing time and increase the memory usage. <https://github.com/aws-labs/aws-data-wrangler/blob/master/tutorials/022%20-%20Writing%20Partitions%20Concurrently.ipynb>
- **mode** (*str, optional*) – `append` (Default), `overwrite`, `overwrite_partitions`. Only takes effect if `dataset=True`. For details check

the related tutorial: [https://aws-data-wrangler.readthedocs.io/en/stable/stubs/awswrangler.s3.to\\_parquet.html#awswrangler.s3.to\\_parquet](https://aws-data-wrangler.readthedocs.io/en/stable/stubs/awswrangler.s3.to_parquet.html#awswrangler.s3.to_parquet)

- **catalog\_versioning** (*bool*) – If True and *mode*=“*overwrite*”, creates an archived version of the table catalog before updating it.
- **schema\_evolution** (*bool*) – If True allows schema evolution (new or missing columns), otherwise a exception will be raised. (Only considered if *dataset*=True and *mode* in (“*append*”, “*overwrite\_partitions*”)) Related tutorial: <https://github.com/awslabs/aws-data-wrangler/blob/master/tutorials/014%20-%20Schema%20Evolution.ipynb>
- **database** (*str*, *optional*) – Glue/Athena catalog: Database name.
- **table** (*str*, *optional*) – Glue/Athena catalog: Table name.
- **dtype** (*Dict[str, str]*, *optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {‘col name’: ‘bigint’, ‘col2 name’: ‘int’})
- **description** (*str*, *optional*) – Glue/Athena catalog: Table description
- **parameters** (*Dict[str, str]*, *optional*) – Glue/Athena catalog: Key/value pairs to tag the table.
- **columns\_comments** (*Dict[str, str]*, *optional*) – Glue/Athena catalog: Columns names and the related comments (e.g. {‘col0’: ‘Column 0.’, ‘col1’: ‘Column 1.’, ‘col2’: ‘Partition.’}).
- **regular\_partitions** (*bool*) – Create regular partitions (Non projected partitions) on Glue Catalog. Disable when you will work only with Partition Projection. Keep enabled even when working with projections is useful to keep Redshift Spectrum working with the regular partitions.
- **projection\_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection\_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: “enum”, “integer”, “date”, “injected” <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {‘col\_name’: ‘enum’, ‘col2\_name’: ‘integer’})
- **projection\_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {‘col\_name’: ‘0,10’, ‘col2\_name’: ‘-1,8675309’})
- **projection\_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {‘col\_name’: ‘A,B,Unknown’, ‘col2\_name’: ‘foo,boo,bar’})
- **projection\_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {‘col\_name’: ‘1’, ‘col2\_name’: ‘5’})
- **projection\_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {‘col\_name’: ‘1’, ‘col2\_name’: ‘2’})
- **catalog\_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

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

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

## Examples

Writing single file

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

Writing single file encrypted with a KMS key

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

Writing partitioned dataset

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

Writing 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*: *Optional[float]* = *None*, *max\_attempts*: *Optional[int]* = *None*, *use\_threads*: *bool* = *True*, *boto3\_session*: *Optional[boto3.session.Session]* = *None*) → *None*

Wait Amazon S3 objects exist.

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

**Note:** In case of `use_threads=True` the number of threads that will be spawned will be gotten from

`os.cpu_count()`.

#### Parameters

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

**Returns** None.

**Return type** None

#### Examples

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

#### awswrangler.s3.wait\_objects\_not\_exist

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

Wait Amazon S3 objects not exist.

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

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

#### Parameters

- **paths** (*List[str]*) – List of S3 objects paths (e.g. [s3://bucket/key0, s3://bucket/key1]).
- **delay** (*Union[int, float]*, *optional*) – The amount of time in seconds to wait between attempts. Default: 5
- **max\_attempts** (*int*, *optional*) – The maximum number of attempts to be made. Default: 20
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.

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

**Returns** None.

**Return type** None

### Examples

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

## 1.3.2 AWS Glue Catalog

<code>add_column(database, table, column_name[, ...])</code>	Delete a column in a AWS Glue Catalog table.
<code>add_csv_partitions(database, table, ...[, ...])</code>	Add partitions (metadata) to a CSV Table in the AWS Glue Catalog.
<code>add_parquet_partitions(database, table, ...)</code>	Add partitions (metadata) to a Parquet Table in the AWS Glue Catalog.
<code>create_csv_table(database, table, path, ...)</code>	Create a CSV Table (Metadata Only) in the AWS Glue Catalog.
<code>create_database(name[, description, ...])</code>	Create a database in AWS Glue Catalog.
<code>create_parquet_table(database, table, path, ...)</code>	Create a Parquet Table (Metadata Only) in the AWS Glue Catalog.
<code>databases([limit, catalog_id, boto3_session])</code>	Get a Pandas DataFrame with all listed databases.
<code>delete_column(database, table, column_name)</code>	Delete a column in a AWS Glue Catalog table.
<code>delete_database(name[, catalog_id, ...])</code>	Create a database in AWS Glue Catalog.
<code>delete_partitions(table, database, ...[, ...])</code>	Delete specified partitions in a AWS Glue Catalog table.
<code>delete_all_partitions(table, database[, ...])</code>	Delete all partitions in a AWS Glue Catalog table.
<code>delete_table_if_exists(database, table[, ...])</code>	Delete Glue table if exists.
<code>does_table_exist(database, table[, ...])</code>	Check if the table exists.
<code>drop_duplicated_columns(df)</code>	Drop all repeated columns (duplicated names).
<code>extract_athena_types(df[, index, ...])</code>	Extract columns and partitions types (Amazon Athena) from Pandas DataFrame.
<code>get_columns_comments(database, table[, ...])</code>	Get all columns comments.
<code>get_csv_partitions(database, table[, ...])</code>	Get all partitions from a Table in the AWS Glue Catalog.
<code>get_databases([catalog_id, boto3_session])</code>	Get an iterator of databases.
<code>get_parquet_partitions(database, table[, ...])</code>	Get all partitions from a Table in the AWS Glue Catalog.
<code>get_partitions(database, table[, ...])</code>	Get all partitions from a Table in the AWS Glue Catalog.
<code>get_table_description(database, table[, ...])</code>	Get table description.
<code>get_table_location(database, table[, ...])</code>	Get table's location on Glue catalog.
<code>get_table_number_of_versions(database, table)</code>	Get total number of versions.
<code>get_table_parameters(database, table[, ...])</code>	Get all parameters.
<code>get_table_types(database, table[, boto3_session])</code>	Get all columns and types from a table.
<code>get_table_versions(database, table[, ...])</code>	Get all versions.
<code>get_tables([catalog_id, database, ...])</code>	Get an iterator of tables.

continues on next page

Table 2 – continued from previous page

<code>overwrite_table_parameters(parameters, ...)</code>	Overwrite all existing parameters.
<code>sanitize_column_name(column)</code>	Convert the column name to be compatible with Amazon Athena.
<code>sanitize_dataframe_columns_names(df)</code>	Normalize all columns names to be compatible with Amazon Athena.
<code>sanitize_table_name(table)</code>	Convert the table name to be compatible with Amazon Athena.
<code>search_tables(text[, catalog_id, boto3_session])</code>	Get Pandas DataFrame of tables filtered by a search string.
<code>table(database, table[, catalog_id, ...])</code>	Get table details as Pandas DataFrame.
<code>tables([limit, catalog_id, database, ...])</code>	Get a DataFrame with tables filtered by a search term, prefix, suffix.
<code>upsert_table_parameters(parameters, ..., ...)</code>	Insert or Update the received parameters.

**aws wrangler.catalog.add\_column**

`aws wrangler.catalog.add_column` (*database: str, table: str, column\_name: str, column\_type: str = 'string', column\_comment: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None, catalog\_id: Optional[str] = None*) → Any

Delete a column in a AWS Glue Catalog table.

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

**Parameters**

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **column\_name** (*str*) – Column name
- **column\_type** (*str*) – Column type.
- **column\_comment** (*str*) – Column Comment
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.
- **catalog\_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

**Returns** None

**Return type** None

## Examples

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

### awswrangler.catalog.add\_csv\_partitions

`awswrangler.catalog.add_csv_partitions` (*database: str, table: str, partitions\_values: Dict[str, List[str]], catalog\_id: Optional[str] = None, compression: Optional[str] = None, sep: str = ',', boto3\_session: Optional[boto3.session.Session] = None, columns\_types: Optional[Dict[str, str]] = None*) → Any

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

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

---

#### Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **partitions\_values** (*Dict[str, List[str]]*) – Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. {'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).
- **catalog\_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **compression** (*str, optional*) – Compression style (None, gzip, etc).
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.
- **columns\_types** (*Optional[Dict[str, str]]*) – Only required for Hive compatibility. Dictionary with keys as column names and values as data types (e.g. {'col0': 'bigint', 'col1': 'double'}). P.S. Only materialized columns please, not partition columns.

**Returns** None.

**Return type** None



## Examples

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

### awswrangler.catalog.add\_parquet\_partitions

`awswrangler.catalog.add_parquet_partitions` (*database: str, table: str, partitions\_values: Dict[str, List[str]], catalog\_id: Optional[str] = None, compression: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None, columns\_types: Optional[Dict[str, str]] = None*) → Any

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

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

#### Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **partitions\_values** (*Dict[str, List[str]]*) – Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. {'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).
- **catalog\_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **compression** (*str, optional*) – Compression style (None, snappy, gzip, etc).
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.
- **columns\_types** (*Optional[Dict[str, str]]*) – Only required for Hive compatibility. Dictionary with keys as column names and values as data types (e.g. {'col0': 'bigint', 'col1': 'double'}). P.S. Only materialized columns please, not partition columns.

**Returns** None.

**Return type** None

## Examples

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

## awswrangler.catalog.create\_csv\_table

`awswrangler.catalog.create_csv_table` (*database: str, table: str, path: str, columns\_types: Dict[str, str], partitions\_types: Optional[Dict[str, str]] = None, compression: Optional[str] = None, description: Optional[str] = None, parameters: Optional[Dict[str, str]] = None, columns\_comments: Optional[Dict[str, str]] = None, mode: str = 'overwrite', catalog\_versioning: bool = False, sep: str = ',', skip\_header\_line\_count: Optional[int] = None, boto3\_session: Optional[boto3.session.Session] = None, projection\_enabled: bool = False, projection\_types: Optional[Dict[str, str]] = None, projection\_ranges: Optional[Dict[str, str]] = None, projection\_values: Optional[Dict[str, str]] = None, projection\_intervals: Optional[Dict[str, str]] = None, projection\_digits: Optional[Dict[str, str]] = None, catalog\_id: Optional[str] = None*) → Any

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

[‘https://docs.aws.amazon.com/athena/latest/ug/data-types.html’](https://docs.aws.amazon.com/athena/latest/ug/data-types.html)

---

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

---

### Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **path** (*str*) – Amazon S3 path (e.g. `s3://bucket/prefix/`).
- **columns\_types** (*Dict[str, str]*) – Dictionary with keys as column names and values as data types (e.g. `{‘col0’: ‘bigint’, ‘col1’: ‘double’}`).

- **partitions\_types** (*Dict[str, str], optional*) – Dictionary with keys as partition names and values as data types (e.g. {'col2': 'date'}).
- **compression** (*str, optional*) – Compression style (None, gzip, etc).
- **description** (*str, optional*) – Table description
- **parameters** (*Dict[str, str], optional*) – Key/value pairs to tag the table.
- **columns\_comments** (*Dict[str, str], optional*) – Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **mode** (*str*) – 'overwrite' to recreate any possible axisting table or 'append' to keep any possible axisting table.
- **catalog\_versioning** (*bool*) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **skip\_header\_line\_count** (*Optional[int]*) – Number of Lines to skip regarding to the header.
- **projection\_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection\_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: "enum", "integer", "date", "injected" <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': 'enum', 'col2\_name': 'integer'})
- **projection\_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '0,10', 'col2\_name': '-1,8675309'})
- **projection\_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': 'A,B,Unknown', 'col2\_name': 'foo,boo,bar'})
- **projection\_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '1', 'col2\_name': '5'})
- **projection\_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '1', 'col2\_name': '2'})
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.
- **catalog\_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

**Returns** None.

**Return type** None

## Examples

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

### awswrangler.catalog.create\_database

`awswrangler.catalog.create_database` (*name: str, description: Optional[str] = None, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any

Create a database in AWS Glue Catalog.

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`

Check out the [Global Configurations Tutorial](#) for details.

---

#### Parameters

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

**Returns** None.

**Return type** None

## Examples

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

### awswrangler.catalog.create\_parquet\_table

```
awswrangler.catalog.create_parquet_table(database: str, table: str, path: str,
                                         columns_types: Dict[str, str], partitions_types:
                                         Optional[Dict[str, str]] = None, catalog_id:
                                         Optional[str] = None, compression: Op-
                                         tional[str] = None, description: Optional[str]
                                         = None, parameters: Optional[Dict[str,
                                         str]] = None, columns_comments: Op-
                                         tional[Dict[str, str]] = None, mode: str =
                                         'overwrite', catalog_versioning: bool = False,
                                         projection_enabled: bool = False, projec-
                                         tion_types: Optional[Dict[str, str]] = None,
                                         projection_ranges: Optional[Dict[str, str]] =
                                         None, projection_values: Optional[Dict[str,
                                         str]] = None, projection_intervals: Op-
                                         tional[Dict[str, str]] = None, projection_digits:
                                         Optional[Dict[str, str]] = None, boto3_session:
                                         Optional[boto3.session.Session] = None) →
                                         Any
```

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

[‘https://docs.aws.amazon.com/athena/latest/ug/data-types.html’](https://docs.aws.amazon.com/athena/latest/ug/data-types.html)

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

#### Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **path** (*str*) – Amazon S3 path (e.g. `s3://bucket/prefix/`).
- **columns\_types** (*Dict[str, str]*) – Dictionary with keys as column names and values as data types (e.g. `{‘col0’: ‘bigint’, ‘col1’: ‘double’}`).
- **partitions\_types** (*Dict[str, str], optional*) – Dictionary with keys as partition names and values as data types (e.g. `{‘col2’: ‘date’}`).
- **catalog\_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

- **compression** (*str*, *optional*) – Compression style (None, snappy, gzip, etc).
- **description** (*str*, *optional*) – Table description
- **parameters** (*Dict[str, str]*, *optional*) – Key/value pairs to tag the table.
- **columns\_comments** (*Dict[str, str]*, *optional*) – Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **mode** (*str*) – 'overwrite' to recreate any possible existing table or 'append' to keep any possible existing table.
- **catalog\_versioning** (*bool*) – If True and *mode*="overwrite", creates an archived version of the table catalog before updating it.
- **projection\_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection\_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: "enum", "integer", "date", "injected" <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': 'enum', 'col2\_name': 'integer'})
- **projection\_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '0,10', 'col2\_name': '-1,8675309'})
- **projection\_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': 'A,B,Unknown', 'col2\_name': 'foo,boo,bar'})
- **projection\_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '1', 'col2\_name': '5'})
- **projection\_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col\_name': '1', 'col2\_name': '2'})
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** None.

**Return type** None

## Examples

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

(continues on next page)

(continued from previous page)

```

...     columns_comments={'col0': 'Column 0.', 'col1': 'Column 1.', 'col2':
↪ 'Partition.'}
... )

```

### awswrangler.catalog.databases

`awswrangler.catalog.databases` (*limit: int = 100, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any

Get a Pandas DataFrame with all listed databases.

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

- `catalog_id`

Check out the [Global Configurations Tutorial](#) for details.

#### Parameters

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

**Returns** Pandas DataFrame filled by formatted infos.

**Return type** `pandas.DataFrame`

#### Examples

```

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

```

### awswrangler.catalog.delete\_column

`awswrangler.catalog.delete_column` (*database: str, table: str, column\_name: str, boto3\_session: Optional[boto3.session.Session] = None, catalog\_id: Optional[str] = None*) → Any

Delete a column in a AWS Glue Catalog table.

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

**Parameters**

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

**Returns** None**Return type** None**Examples**

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

**awswrangler.catalog.delete\_database**

`awswrangler.catalog.delete_database` (*name: str, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any

Create a database in AWS Glue Catalog.

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`

Check out the [Global Configurations Tutorial](#) for details.

---

**Parameters**

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

**Returns** None.**Return type** None



## Examples

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

## awswrangler.catalog.delete\_partitions

`awswrangler.catalog.delete_partitions` (*table: str, database: str, partitions\_values: List[List[str]], catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any

Delete specified partitions in a AWS Glue Catalog table.

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

### Parameters

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

**Returns** None.

**Return type** None

## Examples

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

## awswrangler.catalog.delete\_all\_partitions

`awswrangler.catalog.delete_all_partitions` (*table: str, database: str, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any

Delete all partitions in a AWS Glue Catalog table.

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

---

### Parameters

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

**Returns** Partitions values.

**Return type** List[List[str]]

### Examples

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

## awswrangler.catalog.delete\_table\_if\_exists

`awswrangler.catalog.delete_table_if_exists` (*database: str, table: str, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any

Delete Glue table if exists.

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

#### Parameters

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

**Returns** True if deleted, otherwise False.

**Return type** bool

#### Examples

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

#### awswrangler.catalog.does\_table\_exist

**awswrangler.catalog.does\_table\_exist** (*database: str, table: str, boto3\_session: Optional[boto3.session.Session] = None*) → Any

Check if the table exists.

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- database

Check out the [Global Configurations Tutorial](#) for details.

#### Parameters

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

**Returns** True if exists, otherwise False.

**Return type** bool

## Examples

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

## awswrangler.catalog.drop\_duplicated\_columns

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

Drop all repeated columns (duplicated names).

---

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

---



---

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

---

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

**Returns** Pandas `DataFrame` without duplicated columns.

**Return type** `pandas.DataFrame`

## Examples

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

## awswrangler.catalog.extract\_athena\_types

`awswrangler.catalog.extract_athena_types` (*df*: `pandas.core.frame.DataFrame`, *index*: `bool` = `False`, *partition\_cols*: `Optional[List[str]]` = `None`, *dtype*: `Optional[Dict[str, str]]` = `None`, *file\_format*: `str` = `'parquet'`) → `Tuple[Dict[str, str], Dict[str, str]]`

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

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

### Parameters

- **df** (`pandas.DataFrame`) – Pandas `DataFrame`.
- **index** (`bool`) – Should consider the `DataFrame` index as a column?
- **partition\_cols** (`List[str]`, *optional*) – List of partitions names.

- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'bigint', 'col2 name': 'int'})
- **file\_format** (*str, optional*) – File format to be considered to place the index column: "parquet" | "csv".

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

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

## Examples

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

## awswrangler.catalog.get\_columns\_comments

`awswrangler.catalog.get_columns_comments` (*database: str, table: str, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any

Get all columns comments.

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- catalog\_id
- database

Check out the [Global Configurations Tutorial](#) for details.

### Parameters

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

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

**Return type** Dict[str, str]

## Examples

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

## awswrangler.catalog.get\_csv\_partitions

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

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

Expression argument instructions: [https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get\\_partitions](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_partitions)

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

### Parameters

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

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

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

## Examples

Fetch all partitions

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

(continues on next page)

(continued from previous page)

```

's3://bucket/prefix/y=2020/m=11/': ['2020', '11'],
's3://bucket/prefix/y=2020/m=12/': ['2020', '12']
}

```

### Filtering partitions

```

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

```

## awswrangler.catalog.get\_databases

`awswrangler.catalog.get_databases` (*catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → `Iterator[Dict[str, Any]]`

Get an iterator of databases.

### Parameters

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

**Returns** Iterator of Databases.

**Return type** `Iterator[Dict[str, Any]]`

### Examples

```

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

```

## awswrangler.catalog.get\_parquet\_partitions

`awswrangler.catalog.get_parquet_partitions` (*database: str, table: str, expression: Optional[str] = None, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → `Any`

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

Expression argument instructions: [https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get\\_partitions](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_partitions)

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

---

### Parameters

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

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

**Return type** `Dict[str, List[str]]`

### Examples

Fetch all partitions

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

Filtering partitions

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



**awswrangler.catalog.get\_partitions**

`awswrangler.catalog.get_partitions` (*database: str, table: str, expression: Optional[str] = None, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any

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

Expression argument instructions: [https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get\\_partitions](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_partitions)

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

**Parameters**

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

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

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

**Examples**

Fetch all partitions

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

Filtering partitions

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

### awswrangler.catalog.get\_table\_description

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

Get table description.

#### Parameters

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

**Returns** Description if exists.

**Return type** Optional[str]

### Examples

```
>>> import awswrangler as wr
>>> desc = wr.catalog.get_table_description(database="...", table="...")
```

### awswrangler.catalog.get\_table\_location

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

Get table's location on Glue catalog.

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- database

Check out the [Global Configurations Tutorial](#) for details.

---

#### Parameters

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

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

**Returns** Table's location.

**Return type** `str`

### Examples

```
>>> import awswrangler as wr
>>> wr.catalog.get_table_location(database='default', table='my_table')
's3://bucket/prefix/'
```

### awswrangler.catalog.get\_table\_number\_of\_versions

`awswrangler.catalog.get_table_number_of_versions` (*database: str, table: str, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → *Any*

Get total number of versions.

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

#### Parameters

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

**Returns** Total number of versions.

**Return type** `int`

## Examples

```
>>> import awswrangler as wr
>>> num = wr.catalog.get_table_number_of_versions(database="...", table="...")
```

## awswrangler.catalog.get\_table\_parameters

`awswrangler.catalog.get_table_parameters` (*database: str, table: str, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Dict[str, str]

Get all parameters.

### Parameters

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

**Returns** Dictionary of parameters.

**Return type** Dict[str, str]

## Examples

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

## awswrangler.catalog.get\_table\_types

`awswrangler.catalog.get_table_types` (*database: str, table: str, boto3\_session: Optional[boto3.session.Session] = None*) → Any

Get all columns and types from a table.

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- database

Check out the [Global Configurations Tutorial](#) for details.

---

### Parameters

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

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

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

### Examples

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

### awswrangler.catalog.get\_table\_versions

`awswrangler.catalog.get_table_versions` (*database: str, table: str, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any

Get all versions.

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

#### Parameters

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

**Returns** List of table inputs: [https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get\\_table\\_versions](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_table_versions)

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

### Examples

```
>>> import awswrangler as wr
>>> tables_versions = wr.catalog.get_table_versions(database="...", table="...")
```

## aws wrangler.catalog.get\_tables

```
aws wrangler.catalog.get_tables (catalog_id: Optional[str] = None, database: Optional[str]
                                = None, name_contains: Optional[str] = None, name_prefix:
                                Optional[str] = None, name_suffix: Optional[str] = None,
                                boto3_session: Optional[boto3.session.Session] = None) →
                                Any
```

Get an iterator of tables.

---

**Note:** Please, does not filter using name\_contains and name\_prefix/name\_suffix at the same time. Only name\_prefix and name\_suffix can be combined together.

---

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- catalog\_id
- database

Check out the [Global Configurations Tutorial](#) for details.

---

### Parameters

- **catalog\_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **database** (*str*, *optional*) – Database name.
- **name\_contains** (*str*, *optional*) – Select by a specific string on table name
- **name\_prefix** (*str*, *optional*) – Select by a specific prefix on table name
- **name\_suffix** (*str*, *optional*) – Select by a specific suffix on table name
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Iterator of tables.

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

### Examples

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

## awswrangler.catalog.override\_table\_parameters

awswrangler.catalog.override\_table\_parameters (parameters: Dict[str, str], database: str, table: str, catalog\_versioning: bool = False, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None) → Any

Overwrite all existing parameters.

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- catalog\_id
- database

Check out the [Global Configurations Tutorial](#) for details.

### Parameters

- **parameters** (Dict[str, str]) – e.g. {"source": "mysql", "destination": "datalake"}
- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog\_versioning** (bool) – If True and *mode*="overwrite", creates an archived version of the table catalog before updating it.
- **catalog\_id** (str, optional) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3\_session** (boto3.Session(), optional) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

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

**Return type** Dict[str, str]

### Examples

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

### awswrangler.catalog.sanitize\_column\_name

awswrangler.catalog.**sanitize\_column\_name** (*column: str*) → str

Convert the column name to be compatible with Amazon Athena.

<https://docs.aws.amazon.com/athena/latest/ug/tables-databases-columns-names.html>

Possible transformations: - Strip accents - Remove non alphanumeric characters - Convert CamelCase to snake\_case

**Parameters** *column* (*str*) – Column name.

**Returns** Normalized column name.

**Return type** str

#### Examples

```
>>> import awswrangler as wr
>>> wr.catalog.sanitize_column_name('MyNewColumn')
'my_new_column'
```

### awswrangler.catalog.sanitize\_dataframe\_columns\_names

awswrangler.catalog.**sanitize\_dataframe\_columns\_names** (*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Normalize all columns names to be compatible with Amazon Athena.

<https://docs.aws.amazon.com/athena/latest/ug/tables-databases-columns-names.html>

Possible transformations: - Strip accents - Remove non alphanumeric characters - Convert CamelCase to snake\_case

---

**Note:** After transformation, some column names might not be unique anymore. Example: the columns ["A", "a"] will be sanitized to ["a", "a"]

---

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

**Returns** Original Pandas DataFrame with columns names normalized.

**Return type** pandas.DataFrame

#### Examples

```
>>> import awswrangler as wr
>>> df_normalized = wr.catalog.sanitize_dataframe_columns_names(df=pd.DataFrame({
↪ 'A': [1, 2]}))
```



**awswrangler.catalog.sanitize\_table\_name**

`awswrangler.catalog.sanitize_table_name(table: str) → str`

Convert the table name to be compatible with Amazon Athena.

<https://docs.aws.amazon.com/athena/latest/ug/tables-databases-columns-names.html>

Possible transformations: - Strip accents - Remove non alphanumeric characters - Convert CamelCase to snake\_case

**Parameters** `table` (*str*) – Table name.

**Returns** Normalized table name.

**Return type** `str`

**Examples**

```
>>> import awswrangler as wr
>>> wr.catalog.sanitize_table_name('MyNewTable')
'my_new_table'
```

**awswrangler.catalog.search\_tables**

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

Get Pandas DataFrame of tables filtered by a search string.

**Parameters**

- **text** (*str, optional*) – Select only tables with the given string in table's properties.
- **catalog\_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Iterator of tables.

**Return type** `Iterator[Dict[str, Any]]`

**Examples**

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

### aws wrangler.catalog.table

`aws wrangler.catalog.table` (*database: str, table: str, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any  
Get table details as Pandas DataFrame.

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

---

#### Parameters

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

**Returns** Pandas DataFrame filled by formatted infos.

**Return type** `pandas.DataFrame`

#### Examples

```
>>> import aws wrangler as wr
>>> df_table = wr.catalog.table(database='default', table='my_table')
```

### aws wrangler.catalog.tables

`aws wrangler.catalog.tables` (*limit: int = 100, catalog\_id: Optional[str] = None, database: Optional[str] = None, search\_text: Optional[str] = None, name\_contains: Optional[str] = None, name\_prefix: Optional[str] = None, name\_suffix: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any  
Get a DataFrame with tables filtered by a search term, prefix, suffix.

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

---

#### Parameters

- **limit** (*int*, *optional*) – Max number of tables to be returned.
- **catalog\_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **database** (*str*, *optional*) – Database name.
- **search\_text** (*str*, *optional*) – Select only tables with the given string in table's properties.
- **name\_contains** (*str*, *optional*) – Select by a specific string on table name
- **name\_prefix** (*str*, *optional*) – Select by a specific prefix on table name
- **name\_suffix** (*str*, *optional*) – Select by a specific suffix on table name
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Pandas Dataframe filled by formatted infos.

**Return type** pandas.DataFrame

### Examples

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

### awswrangler.catalog.upsert\_table\_parameters

```
awswrangler.catalog.upsert_table_parameters (parameters: Dict[str, str], database: str, table: str, catalog_versioning: bool = False, catalog_id: Optional[str] = None, boto3_session: Optional[boto3.session.Session] = None) → Any
```

Insert or Update the received parameters.

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- catalog\_id
- database

Check out the [Global Configurations Tutorial](#) for details.

### Parameters

- **parameters** (*Dict[str, str]*) – e.g. {"source": "mysql", "destination": "data-lake"}
- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **catalog\_versioning** (*bool*) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.

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

**Returns** All parameters after the upsert.

**Return type** Dict[str, str]

### Examples

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

## 1.3.3 Amazon Athena

<code>create_athena_bucket([boto3_session])</code>	Create the default Athena bucket if it doesn't exist.
<code>get_query_columns_types(query_execution_id)</code>	Get the data type of all columns queried.
<code>get_query_execution(query_execution_id[, ...])</code>	Fetch query execution details.
<code>get_work_group(workgroup[, boto3_session])</code>	Return information about the workgroup with the specified name.
<code>read_sql_query(sql, database[, ...])</code>	Execute any SQL query on AWS Athena and return the results as a Pandas DataFrame.
<code>read_sql_table(table, database[, ...])</code>	Extract the full table AWS Athena and return the results as a Pandas DataFrame.
<code>repair_table(table[, database, s3_output, ...])</code>	Run the Hive's metastore consistency check: 'MSCK REPAIR TABLE table;'. MSCK REPAIR TABLE table;
<code>start_query_execution(sql[, database, ...])</code>	Start a SQL Query against AWS Athena.
<code>stop_query_execution(query_execution_id[, ...])</code>	Stop a query execution.
<code>wait_query(query_execution_id[, boto3_session])</code>	Wait for the query end.

### awswrangler.athena.create\_athena\_bucket

`awswrangler.athena.create_athena_bucket` (*boto3\_session: Optional[boto3.session.Session] = None*) → str

Create the default Athena bucket if it doesn't exist.

**Parameters** **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

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

**Return type** str

## Examples

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

### awswrangler.athena.get\_query\_columns\_types

`awswrangler.athena.get_query_columns_types(query_execution_id: str, boto3_session: Optional[boto3.session.Session] = None) → Dict[str, str]`

Get the data type of all columns queried.

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

#### Parameters

- **query\_execution\_id** (*str*) – Athena query execution ID.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Dictionary with all data types.

**Return type** Dict[str, str]

## Examples

```
>>> import awswrangler as wr
>>> wr.athena.get_query_columns_types('query-execution-id')
{'col0': 'int', 'col1': 'double'}
```

### awswrangler.athena.get\_query\_execution

`awswrangler.athena.get_query_execution(query_execution_id: str, boto3_session: Optional[boto3.session.Session] = None) → Dict[str, Any]`

Fetch query execution details.

[https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/athena.html#Athena.Client.get\\_query\\_execution](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/athena.html#Athena.Client.get_query_execution)

#### Parameters

- **query\_execution\_id** (*str*) – Athena query execution ID.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Dictionary with the get\_query\_execution response.

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

## Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.get_query_execution(query_execution_id='query-execution-id')
```

## awswrangler.athena.get\_work\_group

`awswrangler.athena.get_work_group` (*workgroup: str, boto3\_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Return information about the workgroup with the specified name.

### Parameters

- **workgroup** (*str*) – Work Group name.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** [https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/athena.html#Athena.Client.get\\_work\\_group](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/athena.html#Athena.Client.get_work_group)

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

## Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.get_work_group(workgroup='workgroup_name')
```

## awswrangler.athena.read\_sql\_query

`awswrangler.athena.read_sql_query` (*sql: str, database: str, ctas\_approach: bool = True, categories: Optional[List[str]] = None, chunksize: Optional[Union[int, bool]] = None, s3\_output: Optional[str] = None, workgroup: Optional[str] = None, encryption: Optional[str] = None, kms\_key: Optional[str] = None, keep\_files: bool = True, ctas\_temp\_table\_name: Optional[str] = None, use\_threads: bool = True, boto3\_session: Optional[boto3.session.Session] = None, max\_cache\_seconds: int = 0, max\_cache\_query\_inspections: int = 50, data\_source: Optional[str] = None*) → Any

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

### Related tutorial:

- [Amazon Athena](#)
- [Athena Cache](#)
- [Global Configurations](#)

**There are two approaches to be defined through `ctas_approach` parameter:**

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

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

**PROS:**

- Faster for mid and big result sizes.
- Can handle some level of nested types.

**CONS:**

- Requires create/delete table permissions on Glue.
- Does not support timestamp with time zone
- Does not support columns with repeated names.
- Does not support columns with undefined data types.
- A temporary table will be created and then deleted immediately.
- Does not support custom data\_source/catalog\_id.

**2 - ctas\_approach=False:**

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

**PROS:**

- Faster for small result sizes (less latency).
- Does not require create/delete table permissions on Glue
- Supports timestamp with time zone.
- Support custom data\_source/catalog\_id.

**CONS:**

- Slower for big results (But stills faster than other libraries that uses the regular Athena's API)
- Does not handle nested types at all.

---

**Note:** The resulting DataFrame (or every DataFrame in the returned Iterator for chunked queries) have a *query\_metadata* attribute, which brings the query result metadata returned by [Boto3/Athena](#) .

For a practical example check out the [related tutorial](#)!

---



---

**Note:** Valid encryption modes: [None, 'SSE\_S3', 'SSE\_KMS'].

*P.S. 'CSE\_KMS' is not supported.*

---



---

**Note:** Create the default Athena bucket if it doesn't exist and s3\_output is None.

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

---



---

**Note:** *chunksize* argument (Memory Friendly) (i.e batching):

Return an Iterable of DataFrames instead of a regular DataFrame.

There are two batching strategies:

- If **chunksize=True**, a new DataFrame will be returned for each file in the query result.
- If **chunksize=INTEGER**, Wrangler will iterate on the data by number of rows igual the received INTEGER.

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

*P.P.S.* If `ctas_approach=False` and `chunksize=True`, you will always receive an iterator with a single DataFrame because regular Athena queries only produces a single output file.

---

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

---

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

- `ctas_approach`
- `database`
- `max_cache_query_inspections`
- `max_cache_seconds`
- `workgroup`

Check out the [Global Configurations Tutorial](#) for details.

---

### Parameters

- **sql** (*str*) – SQL query.
- **database** (*str*) – AWS Glue/Athena database name - It is only the origin database from where the query will be launched. You can still using and mixing several databases writing the full table name within the sql (e.g. `database.table`).
- **ctas\_approach** (*bool*) – Wraps the query using a CTAS, and read the resulted parquet data on S3. If false, read the regular CSV on S3.
- **categories** (*List[str], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **chunksize** (*Union[int, bool], optional*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* wrangler will iterate on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed Wrangler will iterate on the data by number of rows igual the received INTEGER.
- **s3\_output** (*str, optional*) – Amazon S3 path.
- **workgroup** (*str, optional*) – Athena workgroup.
- **encryption** (*str, optional*) – Valid values: [None, 'SSE\_S3', 'SSE\_KMS']. Notice: 'CSE\_KMS' is not supported.
- **kms\_key** (*str, optional*) – For SSE-KMS, this is the KMS key ARN or ID.
- **keep\_files** (*bool*) – Should Wrangler delete or keep the staging files produced by Athena?
- **ctas\_temp\_table\_name** (*str, optional*) – The name of the temporary table and also the directory name on S3 where the CTAS result is stored. If None, it will use the follow random pattern: `f"temp_table_{uuid.uuid4().hex()}"`. On S3 this directory will be under under the pattern: `f"{s3_output}/{ctas_temp_table_name}/"`.



- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **max\_cache\_seconds** (*int*) – Wrangler can look up in Athena's history if this query has been run before. If so, and its completion time is less than *max\_cache\_seconds* before now, wrangler skips query execution and just returns the same results as last time. If cached results are valid, wrangler ignores the *ctas\_approach*, *s3\_output*, *encryption*, *kms\_key*, *keep\_files* and *ctas\_temp\_table\_name* params. If reading cached data fails for any reason, execution falls back to the usual query run path.
- **max\_cache\_query\_inspections** (*int*) – Max number of queries that will be inspected from the history to try to find some result to reuse. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if *max\_cache\_seconds* > 0.
- **data\_source** (*str*, *optional*) – Data Source / Catalog name. If None, 'AwsDataCatalog' will be used by default.

**Returns** Pandas DataFrame or Generator of Pandas DataFrames if chunksize is passed.

**Return type** Union[pd.DataFrame, Iterator[pd.DataFrame]]

## Examples

```
>>> import awswrangler as wr
>>> df = wr.athena.read_sql_query(sql="...", database="...")
>>> scanned_bytes = df.query_metadata["Statistics"]["DataScannedInBytes"]
```

## awswrangler.athena.read\_sql\_table

```
awswrangler.athena.read_sql_table(table: str, database: str, ctas_approach: bool = True, categories: Optional[List[str]] = None, chunksize: Optional[Union[int, bool]] = None, s3_output: Optional[str] = None, workgroup: Optional[str] = None, encryption: Optional[str] = None, kms_key: Optional[str] = None, keep_files: bool = True, ctas_temp_table_name: Optional[str] = None, use_threads: bool = True, boto3_session: Optional[boto3.session.Session] = None, max_cache_seconds: int = 0, max_cache_query_inspections: int = 50, data_source: Optional[str] = None) → Any
```

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

### Related tutorial:

- [Amazon Athena](#)
- [Athena Cache](#)
- [Global Configurations](#)

There are two approaches to be defined through *ctas\_approach* parameter:

1 - *ctas\_approach*=True (Default):

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

PROS:

- Faster for mid and big result sizes.
- Can handle some level of nested types.

CONS:

- Requires create/delete table permissions on Glue.
- Does not support timestamp with time zone
- Does not support columns with repeated names.
- Does not support columns with undefined data types.
- A temporary table will be created and then deleted immediately.

**2** - `ctas_approach=False`:

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

PROS:

- Faster for small result sizes (less latency).
- Does not require create/delete table permissions on Glue
- Supports timestamp with time zone.

CONS:

- Slower for big results (But stills faster than other libraries that uses the regular Athena's API)
- Does not handle nested types at all.

---

**Note:** The resulting DataFrame (or every DataFrame in the returned Iterator for chunked queries) have a `query_metadata` attribute, which brings the query result metadata returned by [Boto3/Athena](#) .

For a practical example check out the [related tutorial](#)!

---

---

**Note:** Valid encryption modes: [None, 'SSE\_S3', 'SSE\_KMS'].

*P.S. 'CSE\_KMS' is not supported.*

---

---

**Note:** Create the default Athena bucket if it doesn't exist and `s3_output` is None.

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

---

---

**Note:** `chunksize` argument (Memory Friendly) (i.e batching):

Return an Iterable of DataFrames instead of a regular DataFrame.

There are two batching strategies:

- If **`chunksize=True`**, a new DataFrame will be returned for each file in the query result.
- If **`chunksize=INTEGER`**, Wrangler will iterate on the data by number of rows igual the received INTEGER.

*P.S. `chunksize=True` is faster and uses less memory while `chunksize=INTEGER` is more precise in number of rows for each DataFrame.*

*P.P.S. If `ctas_approach=False` and `chunksize=True`, you will always receive an iterator with a single DataFrame because regular Athena queries only produces a single output file.*

---

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

---



---

**Note:** This functions has arguments that can has default values configured globally through `wr.config` or environment variables:

- `ctas_approach`
- `database`
- `max_cache_query_inspections`
- `max_cache_seconds`
- `workgroup`

Check out the [Global Configurations Tutorial](#) for details.

---

### Parameters

- **table** (*str*) – Table name.
- **database** (*str*) – AWS Glue/Athena database name.
- **ctas\_approach** (*bool*) – Wraps the query using a CTAS, and read the resulted parquet data on S3. If false, read the regular CSV on S3.
- **categories** (*List[str], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **chunksize** (*Union[int, bool], optional*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* wrangler will iterate on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed Wrangler will iterate on the data by number of rows igual the received INTEGER.
- **s3\_output** (*str, optional*) – AWS S3 path.
- **workgroup** (*str, optional*) – Athena workgroup.
- **encryption** (*str, optional*) – Valid values: [None, 'SSE\_S3', 'SSE\_KMS']. Notice: 'CSE\_KMS' is not supported.
- **kms\_key** (*str, optional*) – For SSE-KMS, this is the KMS key ARN or ID.
- **keep\_files** (*bool*) – Should Wrangler delete or keep the staging files produced by Athena?
- **ctas\_temp\_table\_name** (*str, optional*) – The name of the temporary table and also the directory name on S3 where the CTAS result is stored. If None, it will use the follow random pattern: `f"temp_table_{uuid.uuid4().hex}"`. On S3 this directory will be under under the pattern: `f"{s3_output}/{ctas_temp_table_name}/"`.

- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **max\_cache\_seconds** (*int*) – Wrangler can look up in Athena's history if this table has been read before. If so, and its completion time is less than *max\_cache\_seconds* before now, wrangler skips query execution and just returns the same results as last time. If cached results are valid, wrangler ignores the *ctas\_approach*, *s3\_output*, *encryption*, *kms\_key*, *keep\_files* and *ctas\_temp\_table\_name* params. If reading cached data fails for any reason, execution falls back to the usual query run path.
- **max\_cache\_query\_inspections** (*int*) – Max number of queries that will be inspected from the history to try to find some result to reuse. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if `max_cache_seconds > 0`.
- **data\_source** (*str*, *optional*) – Data Source / Catalog name. If None, 'AwsDataCatalog' will be used by default.

**Returns** Pandas DataFrame or Generator of Pandas DataFrames if chunksize is passed.

**Return type** Union[pd.DataFrame, Iterator[pd.DataFrame]]

## Examples

```
>>> import awswrangler as wr
>>> df = wr.athena.read_sql_table(table="...", database="...")
>>> scanned_bytes = df.query_metadata["Statistics"]["DataScannedInBytes"]
```

## awswrangler.athena.repair\_table

`awswrangler.athena.repair_table` (*table: str, database: Optional[str] = None, s3\_output: Optional[str] = None, workgroup: Optional[str] = None, encryption: Optional[str] = None, kms\_key: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Any

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

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

---

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

---

---

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- database
- workgroup

Check out the [Global Configurations Tutorial](#) for details.

#### Parameters

- **table** (*str*) – Table name.
- **database** (*str*, *optional*) – AWS Glue/Athena database name.
- **s3\_output** (*str*, *optional*) – AWS S3 path.
- **workgroup** (*str*, *optional*) – Athena workgroup.
- **encryption** (*str*, *optional*) – None, 'SSE\_S3', 'SSE\_KMS', 'CSE\_KMS'.
- **kms\_key** (*str*, *optional*) – For SSE-KMS and CSE-KMS, this is the KMS key ARN or ID.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Query final state ('SUCCEEDED', 'FAILED', 'CANCELLED').

**Return type** str

#### Examples

```
>>> import awswrangler as wr
>>> query_final_state = wr.athena.repair_table(table='...', database='...')
```

#### awswrangler.athena.start\_query\_execution

```
awswrangler.athena.start_query_execution(sql: str, database: Optional[str] = None,
                                         s3_output: Optional[str] = None, workgroup:
                                         Optional[str] = None, encryption: Optional[str]
                                         = None, kms_key: Optional[str] = None,
                                         boto3_session: Optional[boto3.session.Session]
                                         = None, data_source: Optional[str] = None) →
                                         Any
```

Start a SQL Query against AWS Athena.

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

**Note:** This functions has arguments that can has default values configured globally through *wr.config* or environment variables:

- database
- workgroup

Check out the [Global Configurations Tutorial](#) for details.

#### Parameters

- **sql** (*str*) – SQL query.

- **database** (*str*, *optional*) – AWS Glue/Athena database name.
- **s3\_output** (*str*, *optional*) – AWS S3 path.
- **workgroup** (*str*, *optional*) – Athena workgroup.
- **encryption** (*str*, *optional*) – None, 'SSE\_S3', 'SSE\_KMS', 'CSE\_KMS'.
- **kms\_key** (*str*, *optional*) – For SSE-KMS and CSE-KMS, this is the KMS key ARN or ID.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.
- **data\_source** (*str*, *optional*) – Data Source / Catalog name. If None, 'AwsDataCatalog' will be used by default.

**Returns** Query execution ID

**Return type** str

## Examples

Querying into the default data source (Amazon s3 - 'AwsDataCatalog')

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

Querying into another data source (PostgreSQL, Redshift, etc)

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

## awswrangler.athena.stop\_query\_execution

**awswrangler.athena.stop\_query\_execution** (*query\_execution\_id*: str, *boto3\_session*: *Optional*[*boto3.session.Session*] = None) → None

Stop a query execution.

Requires you to have access to the workgroup in which the query ran.

### Parameters

- **query\_execution\_id** (*str*) – Athena query execution ID.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** None.

**Return type** None

## Examples

```
>>> import awswrangler as wr
>>> wr.athena.stop_query_execution(query_execution_id='query-execution-id')
```

### awswrangler.athena.wait\_query

`awswrangler.athena.wait_query(query_execution_id: str, boto3_session: Optional[boto3.session.Session] = None) → Dict[str, Any]`

Wait for the query end.

#### Parameters

- **query\_execution\_id** (*str*) – Athena query execution ID.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Dictionary with the get\_query\_execution response.

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

## Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.wait_query(query_execution_id='query-execution-id')
```

### 1.3.4 Amazon Redshift

<code>connect(connection[, catalog_id, ...])</code>	Return a redshift_connector connection from a Glue Catalog Connection.
<code>connect_temp(cluster_identifier, user[, ...])</code>	Return a redshift_connector temporary connection (No password required).
<code>copy(df, path, con, table, schema, iam_role)</code>	Load Pandas DataFrame as a Table on Amazon Redshift using parquet files on S3 as stage.
<code>copy_from_files(path, con, table, schema, ...)</code>	Load Parquet files from S3 to a Table on Amazon Redshift (Through COPY command).
<code>read_sql_query(sql, con[, index_col, ...])</code>	Return a DataFrame corresponding to the result set of the query string.
<code>read_sql_table(table, con[, schema, ...])</code>	Return a DataFrame corresponding the table.
<code>to_sql(df, con, table, schema[, mode, ...])</code>	Write records stored in a DataFrame into Redshift.
<code>unload(sql, path, con, iam_role[, region, ...])</code>	Load Pandas DataFrame from a Amazon Redshift query result using Parquet files on s3 as stage.
<code>unload_to_files(sql, path, con, iam_role[, ...])</code>	Unload Parquet files on s3 from a Redshift query result (Through the UNLOAD command).

### awswrangler.redshift.connect

`awswrangler.redshift.connect` (*connection: str, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None, ssl: bool = True, timeout: Optional[int] = None, max\_prepared\_statements: int = 1000, tcp\_keepalive: bool = True*) → `redshift_connector.core.Connection`

Return a `redshift_connector` connection from a Glue Catalog Connection.

<https://github.com/aws/amazon-redshift-python-driver>

#### Parameters

- **connection** (*str*) – Glue Catalog Connection name.
- **catalog\_id** (*str, optional*) – The ID of the Data Catalog. 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`.
- **ssl** (*bool*) – This governs SSL encryption for TCP/IP sockets. This parameter is forward to `redshift_connector`. <https://github.com/aws/amazon-redshift-python-driver>
- **timeout** (*Optional[int]*) – This is the time in seconds before the connection to the server will time out. The default is `None` which means no timeout. This parameter is forward to `redshift_connector`. <https://github.com/aws/amazon-redshift-python-driver>
- **max\_prepared\_statements** (*int*) – This parameter is forward to `redshift_connector`. <https://github.com/aws/amazon-redshift-python-driver>
- **tcp\_keepalive** (*bool*) – If `True` then use TCP keepalive. The default is `True`. This parameter is forward to `redshift_connector`. <https://github.com/aws/amazon-redshift-python-driver>

**Returns** `redshift_connector` connection.

**Return type** `redshift_connector.Connection`

#### Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

### awswrangler.redshift.connect\_temp

`awswrangler.redshift.connect_temp` (*cluster\_identifier: str, user: str, database: Optional[str] = None, duration: int = 900, auto\_create: bool = True, db\_groups: Optional[List[str]] = None, boto3\_session: Optional[boto3.session.Session] = None, ssl: bool = True, timeout: Optional[int] = None, max\_prepared\_statements: int = 1000, tcp\_keepalive: bool = True*) → `redshift_connector.core.Connection`

Return a `redshift_connector` temporary connection (No password required).

<https://github.com/aws/amazon-redshift-python-driver>



### Parameters

- **cluster\_identifier** (*str*) – The unique identifier of a cluster. This parameter is case sensitive.
- **user** (*str*, *optional*) – The name of a database user.
- **database** (*str*, *optional*) – Database name. If None, the default Database is used.
- **duration** (*int*, *optional*) – The number of seconds until the returned temporary password expires. Constraint: minimum 900, maximum 3600. Default: 900
- **auto\_create** (*bool*) – Create a database user with the name specified for the user named in user if one does not exist.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.
- **ssl** (*bool*) – This governs SSL encryption for TCP/IP sockets. This parameter is forward to redshift\_connector. <https://github.com/aws/amazon-redshift-python-driver>
- **timeout** (*Optional[int]*) – This is the time in seconds before the connection to the server will time out. The default is None which means no timeout. This parameter is forward to redshift\_connector. <https://github.com/aws/amazon-redshift-python-driver>
- **max\_prepared\_statements** (*int*) – This parameter is forward to redshift\_connector. <https://github.com/aws/amazon-redshift-python-driver>
- **tcp\_keepalive** (*bool*) – If True then use TCP keepalive. The default is True. This parameter is forward to redshift\_connector. <https://github.com/aws/amazon-redshift-python-driver>

**Returns** redshift\_connector connection.

**Return type** redshift\_connector.Connection

### Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

### awswrangler.redshift.copy

`awswrangler.redshift.copy` (*df*: *pandas.core.frame.DataFrame*, *path*: *str*, *con*: *redshift\_connector.core.Connection*, *table*: *str*, *schema*: *str*, *iam\_role*: *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[List[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*, *max\_rows\_by\_file*: *Optional[int]* = *10000000*) → *None*

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

This is a **HIGH** latency and **HIGH** throughput alternative to `wr.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](https://docs.aws.amazon.com/redshift/latest/dg/r_COPY.html)

---

**Note:** If the table does not exist yet, it will be automatically created for you using the Parquet metadata to infer the columns data types.

---

---

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

---

### Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame.
- **path** (*str*) – S3 path to write stage files (e.g. `s3://bucket_name/any_name/`)
- **con** (*redshift\_connector.Connection*) – Use `redshift_connector.connect()` to use ” ”credentials directly or `wr.redshift.connect()` to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **iam\_role** (*str*) – 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](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](https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html)
- **sortkey** (*List[str], optional*) – List of columns to be sorted.
- **primary\_keys** (*List[str], optional*) – Primary keys.
- **varchar\_lengths\_default** (*int*) – The size that will be set for all VARCHAR columns not specified with `varchar_lengths`.
- **varchar\_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. `{“col1”: 10, “col5”: 200}`).
- **keep\_files** (*bool*) – Should keep 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 botocore requests. Valid parameters: “ACL”, “Metadata”, “ServerSideEncryption”, “StorageClass”, “SSECustomerAlgorithm”, “SSECustomerKey”, “SSEKMSKeyId”, “SSEKMSEncryptionContext”, “Tagging”. e.g. s3\_additional\_kwargs={ ‘ServerSideEncryption’: ‘aws:kms’, ‘SSEKMSKeyId’: ‘YOUR\_KMS\_KEY\_ARN’ }
- **max\_rows\_by\_file** (*int*) – Max number of rows in each file. Default is None i.e. dont split the files. (e.g. 33554432, 268435456)

**Returns** None.

**Return type** None

### Examples

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

### awswrangler.redshift.copy\_from\_files

**awswrangler.redshift.copy\_from\_files** (*path: str, con: redshift\_connector.core.Connection, table: str, schema: str, iam\_role: str, parquet\_infer\_sampling: float = 1.0, mode: str = 'append', diststyle: str = 'AUTO', distkey: Optional[str] = None, sortstyle: str = 'COMPOUND', sortkey: Optional[List[str]] = None, primary\_keys: Optional[List[str]] = None, varchar\_lengths\_default: int = 256, varchar\_lengths: Optional[Dict[str, int]] = None, path\_suffix: Optional[str] = None, path\_ignore\_suffix: Optional[str] = None, use\_threads: bool = True, boto3\_session: Optional[boto3.session.Session] = None, s3\_additional\_kwargs: Optional[Dict[str, str]] = None*) → None

Load Parquet files from S3 to a Table on Amazon Redshift (Through COPY command).

[https://docs.aws.amazon.com/redshift/latest/dg/r\\_COPY.html](https://docs.aws.amazon.com/redshift/latest/dg/r_COPY.html)

**Note:** If the table does not exist yet, it will be automatically created for you using the Parquet metadata to infer the columns data types.

---

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

---

### Parameters

- **path** (*str*) – S3 prefix (e.g. `s3://bucket/prefix/`)
- **con** (*redshift\_connector.Connection*) – Use `redshift_connector.connect()` to use ” “credentials directly or `wr.redshift.connect()` to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **iam\_role** (*str*) – AWS IAM role with the related permissions.
- **parquet\_infer\_sampling** (*float*) – Random sample ratio of files that will have the metadata inspected. Must be  $0.0 < sampling \leq 1.0$ . The higher, the more accurate. The lower, the faster.
- **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](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](https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html)
- **sortkey** (*List[str]*, *optional*) – List of columns to be sorted.
- **primary\_keys** (*List[str]*, *optional*) – Primary keys.
- **varchar\_lengths\_default** (*int*) – The size that will be set for all VARCHAR columns not specified with `varchar_lengths`.
- **varchar\_lengths** (*Dict[str, int]*, *optional*) – Dict of VARCHAR length by columns. (e.g. {“col1”: 10, “col5”: 200}).
- **path\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be scanned on s3 for the schema extraction (e.g. [“.gz.parquet”, “.snappy.parquet”]). Only has effect during the table creation. If None, will try to read all files. (default)
- **path\_ignore\_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored during the schema extraction. (e.g. [“.csv”, “\_SUCCESS”]). Only has effect during the table creation. If None, will try to read all files. (default)
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3\_additional\_kwargs** – Forward to botocore requests. Valid parameters: “ACL”, “Metadata”, “ServerSideEncryption”, “StorageClass”, “SSECustomerAlgorithm”, “SSECustomerKey”, “SSEKMSKeyId”, “SSEKMSEncryptionContext”, “Tagging”. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`

**Returns** None.

**Return type** None

### Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> wr.db.copy_from_files(
...     path="s3://bucket/my_parquet_files/",
...     con=con,
...     table="my_table",
...     schema="public"
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
>>> con.close()
```

### awswrangler.redshift.read\_sql\_query

`awswrangler.redshift.read_sql_query` (*sql*: str, *con*: *redshift\_connector.core.Connection*, *index\_col*: Optional[Union[str, List[str]]] = None, *params*: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None, *chunksize*: Optional[int] = None, *dtype*: Optional[Dict[str, pyarrow.lib.DataType]] = None, *safe*: bool = True) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

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

---

**Note:** For large extractions (1K+ rows) consider the function `wr.redshift.unload()`.

---

#### Parameters

- **sql** (str) – SQL query.
- **con** (*redshift\_connector.Connection*) – Use `redshift_connector.connect()` to use "credentials directly or `wr.redshift.connect()` to fetch it from the Glue Catalog.
- **index\_col** (Union[str, List[str]], optional) – Column(s) to set as index(MultiIndex).
- **params** (Union[List, Tuple, Dict], optional) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (int, optional) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (Dict[str, pyarrow.DataType], optional) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (bool) – Check for overflows or other unsafe data type conversions.

**Returns** Result as Pandas DataFrame(s).

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

## Examples

Reading from Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> df = wr.redshift.read_sql_query(
...     sql="SELECT * FROM public.my_table",
...     con=con
... )
>>> con.close()
```

## awswrangler.redshift.read\_sql\_table

`awswrangler.redshift.read_sql_table` (*table: str, con: redshift\_connector.core.Connection, schema: Optional[str] = None, index\_col: Optional[Union[str, List[str]]] = None, params: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None, chunksize: Optional[int] = None, dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None, safe: bool = True*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Return a DataFrame corresponding the table.

---

**Note:** For large extractions (1K+ rows) consider the function `wr.redshift.unload()`.

---

### Parameters

- **table** (*str*) – Table name.
- **con** (*redshift\_connector.Connection*) – Use `redshift_connector.connect()` to use ”credentials directly or `wr.redshift.connect()` to fetch it from the Glue Catalog.
- **schema** (*str, optional*) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).
- **index\_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249’s paramstyle, is supported.
- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

**Returns** Result as Pandas DataFrame(s).

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

## Examples

Reading from Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> df = wr.redshift.read_sql_table(
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

## awswrangler.redshift.to\_sql

`awswrangler.redshift.to_sql` (*df*: *pandas.core.frame.DataFrame*, *con*: *redshift\_connector.core.Connection*, *table*: *str*, *schema*: *str*, *mode*: *str* = 'append', *index*: *bool* = False, *dtype*: *Optional[Dict[str, str]]* = None, *diststyle*: *str* = 'AUTO', *distkey*: *Optional[str]* = None, *sortstyle*: *str* = 'COMPOUND', *sortkey*: *Optional[List[str]]* = None, *primary\_keys*: *Optional[List[str]]* = None, *varchar\_lengths\_default*: *int* = 256, *varchar\_lengths*: *Optional[Dict[str, int]]* = None) → None

Write records stored in a DataFrame into Redshift.

---

**Note:** For large DataFrames (1K+ rows) consider the function `wr.redshift.copy()`.

---

### Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **con** (*redshift\_connector.Connection*) – Use `redshift_connector.connect()` to use “credentials directly or `wr.redshift.connect()` to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **mode** (*str*) – Append, overwrite or upsert.
- **index** (*bool*) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (*Dict[str, str]*, *optional*) – Dictionary of columns names and Redshift types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'VARCHAR(10)', 'col2 name': 'FLOAT'}) *diststyle*: *str* Redshift distribution styles. Must be in ["AUTO", "EVEN", "ALL", "KEY"]. [https://docs.aws.amazon.com/redshift/latest/dg/t\\_Distributing\\_data.html](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](https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html)
- **sortkey** (*List[str]*, *optional*) – List of columns to be sorted.
- **primary\_keys** (*List[str]*, *optional*) – Primary keys.
- **varchar\_lengths\_default** (*int*) – The size that will be set for all VARCHAR columns not specified with `varchar_lengths`.
- **varchar\_lengths** (*Dict[str, int]*, *optional*) – Dict of VARCHAR length by columns. (e.g. {"col1": 10, "col5": 200}).

**Returns** None.

**Return type** None

## Examples

Writing to Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> wr.redshift.to_sql(
...     df=df
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

## awswrangler.redshift.unload

`awswrangler.redshift.unload` (*sql: str*, *path: str*, *con: redshift\_connector.core.Connection*, *iam\_role: str*, *region: Optional[str] = None*, *max\_file\_size: Optional[float] = None*, *kms\_key\_id: Optional[str] = None*, *categories: Optional[List[str]] = None*, *chunked: Union[bool, int] = False*, *keep\_files: bool = False*, *use\_threads: bool = True*, *boto3\_session: Optional[boto3.session.Session] = None*, *s3\_additional\_kwargs: Optional[Dict[str, str]] = None*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Load Pandas DataFrame from a Amazon Redshift query result using Parquet files on s3 as stage.

This is a **HIGH** latency and **HIGH** throughput alternative to `wr.redshift.read_sql_query()/wr.redshift.read_sql_table()` to extract large Amazon Redshift data into a Pandas DataFrames through the **UNLOAD command**.

This strategy has more overhead and requires more IAM privileges than the regular `wr.redshift.read_sql_query()/wr.redshift.read_sql_table()` function, so it is only recommended to fetch 1k+ rows at once.

[https://docs.aws.amazon.com/redshift/latest/dg/r\\_UNLOAD.html](https://docs.aws.amazon.com/redshift/latest/dg/r_UNLOAD.html)

---

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

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

There are two batching strategies on Wrangler:



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

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

---

**Note:** In case of *use\_threads=True* the number of threads that will be spawned will be gotten from *os.cpu\_count()*.

---

### Parameters

- **sql** (*str*) – SQL query.
- **path** (*Union[str, List[str]]*) – S3 path to write stage files (e.g. *s3://bucket\_name/any\_name/*)
- **con** (*redshift\_connector.Connection*) – Use *redshift\_connector.connect()* to use “credentials directly or *wr.redshift.connect()* to fetch it from the Glue Catalog.
- **iam\_role** (*str*) – AWS IAM role with the related permissions.
- **region** (*str, optional*) – Specifies the AWS Region where the target Amazon S3 bucket is located. REGION is required for UNLOAD to an Amazon S3 bucket that isn’t in the same AWS Region as the Amazon Redshift cluster. By default, UNLOAD assumes that the target Amazon S3 bucket is located in the same AWS Region as the Amazon Redshift cluster.
- **max\_file\_size** (*float, optional*) – Specifies the maximum size (MB) of files that UNLOAD creates in Amazon S3. Specify a decimal value between 5.0 MB and 6200.0 MB. If None, the default maximum file size is 6200.0 MB.
- **kms\_key\_id** (*str, optional*) – Specifies the key ID for an AWS Key Management Service (AWS KMS) key to be used to encrypt data files on Amazon S3.
- **categories** (*List[str], optional*) – List of columns names that should be returned as *pandas.Categorical*. Recommended for memory restricted environments.
- **keep\_files** (*bool*) – Should keep the stage files?
- **chunked** (*Union[int, bool]*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* wrangler will iterate on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed Wrangler will iterate on the data by number of rows equal the received INTEGER.
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled *os.cpu\_count()* will be used as the max number of threads.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if *boto3\_session* receive None.
- **s3\_additional\_kwargs** – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.

**Returns** Result as Pandas DataFrame(s).

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

## Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> df = wr.db.unload(
...     sql="SELECT * FROM public.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     con=con,
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
>>> con.close()
```

## awswrangler.redshift.unload\_to\_files

`awswrangler.redshift.unload_to_files` (*sql*: *str*, *path*: *str*, *con*: *redshift\_connector.core.Connection*, *iam\_role*: *str*, *region*: *Optional[str]* = *None*, *max\_file\_size*: *Optional[float]* = *None*, *kms\_key\_id*: *Optional[str]* = *None*, *manifest*: *bool* = *False*, *use\_threads*: *bool* = *True*, *partition\_cols*: *Optional[List[str]]* = *None*, *boto3\_session*: *Optional[boto3.session.Session]* = *None*) → *None*

Unload Parquet files on s3 from a Redshift query result (Through the UNLOAD command).

[https://docs.aws.amazon.com/redshift/latest/dg/r\\_UNLOAD.html](https://docs.aws.amazon.com/redshift/latest/dg/r_UNLOAD.html)

---

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

---

### Parameters

- **sql** (*str*) – SQL query.
- **path** (*Union[str, List[str]]*) – S3 path to write stage files (e.g. `s3://bucket_name/any_name/`)
- **con** (*redshift\_connector.Connection*) – Use `redshift_connector.connect()` to use “credentials directly or `wr.redshift.connect()` to fetch it from the Glue Catalog.
- **iam\_role** (*str*) – AWS IAM role with the related permissions.
- **region** (*str*, *optional*) – Specifies the AWS Region where the target Amazon S3 bucket is located. REGION is required for UNLOAD to an Amazon S3 bucket that isn’t in the same AWS Region as the Amazon Redshift cluster. By default, UNLOAD assumes that the target Amazon S3 bucket is located in the same AWS Region as the Amazon Redshift cluster.
- **max\_file\_size** (*float*, *optional*) – Specifies the maximum size (MB) of files that UNLOAD creates in Amazon S3. Specify a decimal value between 5.0 MB and 6200.0 MB. If *None*, the default maximum file size is 6200.0 MB.
- **kms\_key\_id** (*str*, *optional*) – Specifies the key ID for an AWS Key Management Service (AWS KMS) key to be used to encrypt data files on Amazon S3.
- **use\_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads.

- **manifest** (*bool*) – Unload a manifest file on S3.
- **partition\_cols** (*List[str], optional*) – Specifies the partition keys for the unload operation.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns****Return type** None**Examples**

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> wr.redshift.unload_to_files(
...     sql="SELECT * FROM public.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     con=con,
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
>>> con.close()
```

**1.3.5 PostgreSQL**

<code>connect(connection[, catalog_id, ...])</code>	Return a pg8000 connection from a Glue Catalog Connection.
<code>read_sql_query(sql, con[, index_col, ...])</code>	Return a DataFrame corresponding to the result set of the query string.
<code>read_sql_table(table, con[, schema, ...])</code>	Return a DataFrame corresponding the table.
<code>to_sql(df, con, table, schema[, mode, ...])</code>	Write records stored in a DataFrame into PostgreSQL.

**awswrangler.postgresql.connect**

`awswrangler.postgresql.connect` (*connection: str, catalog\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None, ssl\_context: Optional[Dict[Any, Any]] = None, timeout: Optional[int] = None, tcp\_keepalive: bool = True*) → `pg8000.core.Connection`

Return a pg8000 connection from a Glue Catalog Connection.

<https://github.com/tlocke/pg8000>

**Parameters**

- **connection** (*str*) – Glue Catalog Connection name.
- **catalog\_id** (*str, optional*) – The ID of the Data Catalog. 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.

- **ssl\_context** (*Optional[Dict]*) – This governs SSL encryption for TCP/IP sockets. This parameter is forward to pg8000. <https://github.com/tlocke/pg8000#functions>
- **timeout** (*Optional[int]*) – This is the time in seconds before the connection to the server will time out. The default is None which means no timeout. This parameter is forward to pg8000. <https://github.com/tlocke/pg8000#functions>
- **tcp\_keepalive** (*bool*) – If True then use TCP keepalive. The default is True. This parameter is forward to pg8000. <https://github.com/tlocke/pg8000#functions>

**Returns** pg8000 connection.

**Return type** pg8000.Connection

## Examples

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

## awswrangler.postgresql.read\_sql\_query

`awswrangler.postgresql.read_sql_query` (*sql: str, con: pg8000.core.Connection, index\_col: Optional[Union[str, List[str]]] = None, params: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None, chunksize: Optional[int] = None, dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None, safe: bool = True*) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

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

### Parameters

- **sql** (*str*) – SQL query.
- **con** (*pg8000.Connection*) – Use `pg8000.connect()` to use ” “credentials directly or `wr.postgresql.connect()` to fetch it from the Glue Catalog.
- **index\_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249’s paramstyle, is supported.
- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

**Returns** Result as Pandas DataFrame(s).

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

## Examples

Reading from PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> df = wr.postgresql.read_sql_query(
...     sql="SELECT * FROM public.my_table",
...     con=con
... )
>>> con.close()
```

## awswrangler.postgresql.read\_sql\_table

`awswrangler.postgresql.read_sql_table` (*table*: str, *con*: `pg8000.core.Connection`, *schema*: Optional[str] = None, *index\_col*: Optional[Union[str, List[str]]] = None, *params*: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None, *chunksize*: Optional[int] = None, *dtype*: Optional[Dict[str, `pyarrow.lib.DataType`]] = None, *safe*: bool = True) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]

Return a DataFrame corresponding the table.

### Parameters

- **table** (str) – Table name.
- **con** (`pg8000.Connection`) – Use `pg8000.connect()` to use " " "credentials directly or `wr.postgresql.connect()` to fetch it from the Glue Catalog.
- **schema** (str, optional) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).
- **index\_col** (Union[str, List[str]], optional) – Column(s) to set as index(MultiIndex).
- **params** (Union[List, Tuple, Dict], optional) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (int, optional) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (Dict[str, `pyarrow.DataType`], optional) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (bool) – Check for overflows or other unsafe data type conversions.

**Returns** Result as Pandas DataFrame(s).

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

## Examples

Reading from PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> df = wr.postgresql.read_sql_table(
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

## awswrangler.postgresql.to\_sql

`awswrangler.postgresql.to_sql` (*df*: `pandas.core.frame.DataFrame`, *con*: `pg8000.core.Connection`, *table*: `str`, *schema*: `str`, *mode*: `str` = 'append', *index*: `bool` = `False`, *dtype*: `Optional[Dict[str, str]]` = `None`, *varchar\_lengths*: `Optional[Dict[str, int]]` = `None`) → `None`

Write records stored in a `DataFrame` into PostgreSQL.

### Parameters

- **df** (`pandas.DataFrame`) – Pandas `DataFrame` <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **con** (`pg8000.Connection`) – Use `pg8000.connect()` to use " "credentials directly or `wr.postgresql.connect()` to fetch it from the Glue Catalog.
- **table** (`str`) – Table name
- **schema** (`str`) – Schema name
- **mode** (`str`) – Append or overwrite.
- **index** (`bool`) – True to store the `DataFrame` index as a column in the table, otherwise False to ignore it.
- **dtype** (`Dict[str, str]`, *optional*) – Dictionary of columns names and PostgreSQL types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'TEXT', 'col2 name': 'FLOAT'})
- **varchar\_lengths** (`Dict[str, int]`, *optional*) – Dict of VARCHAR length by columns. (e.g. {'col1': 10, 'col5': 200}).

**Returns** `None`.

**Return type** `None`

## Examples

### Writing to PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> wr.postgresql.to_sql(
...     df=df
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

## 1.3.6 MySQL

<code>connect(connection[, catalog_id, ...])</code>	Return a pymysql connection from a Glue Catalog Connection.
<code>read_sql_query(sql, con[, index_col, ...])</code>	Return a DataFrame corresponding to the result set of the query string.
<code>read_sql_table(table, con[, schema, ...])</code>	Return a DataFrame corresponding the table.
<code>to_sql(df, con, table, schema[, mode, ...])</code>	Write records stored in a DataFrame into MySQL.

### awswrangler.mysql.connect

`awswrangler.mysql.connect` (*connection*: *str*, *catalog\_id*: *Optional[str] = None*, *boto3\_session*: *Optional[boto3.session.Session] = None*, *read\_timeout*: *Optional[int] = None*, *write\_timeout*: *Optional[int] = None*, *connect\_timeout*: *int = 10*)  
→ `pymysql.connections.Connection`

Return a pymysql connection from a Glue Catalog Connection.

<https://pymysql.readthedocs.io>

#### Parameters

- **connection** (*str*) – Glue Catalog Connection name.
- **catalog\_id** (*str*, *optional*) – The ID of the Data Catalog. 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`.
- **read\_timeout** (*Optional[int]*) – The timeout for reading from the connection in seconds (default: `None` - no timeout). This parameter is forward to pymysql. <https://pymysql.readthedocs.io/en/latest/modules/connections.html>
- **write\_timeout** (*Optional[int]*) – The timeout for writing to the connection in seconds (default: `None` - no timeout) This parameter is forward to pymysql. <https://pymysql.readthedocs.io/en/latest/modules/connections.html>
- **connect\_timeout** (*int*) – Timeout before throwing an exception when connecting. (default: 10, min: 1, max: 31536000) This parameter is forward to pymysql. <https://pymysql.readthedocs.io/en/latest/modules/connections.html>

**Returns** pymysql connection.

**Return type** pymysql.connections.Connection

### Examples

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

### awswrangler.mysql.read\_sql\_query

`awswrangler.mysql.read_sql_query` (*sql: str, con: pymysql.connections.Connection, index\_col: Optional[Union[str, List[str]]] = None, params: Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None, chunksize: Optional[int] = None, dtype: Optional[Dict[str, pyarrow.lib.DataType]] = None, safe: bool = True) → Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]*

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

#### Parameters

- **sql** (*str*) – SQL query.
- **con** (*pymysql.connections.Connection*) – Use `pymysql.connect()` to use “credentials directly or `wr.mysql.connect()` to fetch it from the Glue Catalog.
- **index\_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249’s paramstyle, is supported.
- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

**Returns** Result as Pandas DataFrame(s).

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



## Examples

### Reading from MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> df = wr.mysql.read_sql_query(
...     sql="SELECT * FROM test.my_table",
...     con=con
... )
>>> con.close()
```

### awswrangler.mysql.read\_sql\_table

`awswrangler.mysql.read_sql_table` (*table*: *str*, *con*: *pymysql.connections.Connection*, *schema*: *Optional[str] = None*, *index\_col*: *Optional[Union[str, List[str]]] = None*, *params*: *Optional[Union[List[Any], Tuple[Any, ...], Dict[Any, Any]]] = None*, *chunksize*: *Optional[int] = None*, *dtype*: *Optional[Dict[str, pyarrow.lib.DataType]] = None*, *safe*: *bool = True*) → *Union[pandas.core.frame.DataFrame, Iterator[pandas.core.frame.DataFrame]]*

Return a DataFrame corresponding the table.

#### Parameters

- **table** (*str*) – Table name.
- **con** (*pymysql.connections.Connection*) – Use `pymysql.connect()` to use “credentials directly or `wr.mysql.connect()` to fetch it from the Glue Catalog.
- **schema** (*str*, *optional*) – Name of SQL schema in database to query. Uses default schema if None.
- **index\_col** (*Union[str, List[str]]*, *optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict]*, *optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249’s `paramstyle`, is supported.
- **chunksize** (*int*, *optional*) – If specified, return an iterator where `chunksize` is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType]*, *optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

**Returns** Result as Pandas DataFrame(s).

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

## Examples

Reading from MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> df = wr.mysql.read_sql_table(
...     table="my_table",
...     schema="test",
...     con=con
... )
>>> con.close()
```

## awswrangler.mysql.to\_sql

`awswrangler.mysql.to_sql(df: pandas.core.frame.DataFrame, con: pymysql.connections.Connection, table: str, schema: str, mode: str = 'append', index: bool = False, dtype: Optional[Dict[str, str]] = None, varchar_lengths: Optional[Dict[str, int]] = None) → None`

Write records stored in a DataFrame into MySQL.

### Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **con** (*pymysql.connections.Connection*) – Use `pymysql.connect()` to use “credentials directly or `wr.mysql.connect()` to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **mode** (*str*) – Append or overwrite.
- **index** (*bool*) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (*Dict[str, str]*, *optional*) – Dictionary of columns names and MySQL types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'TEXT', 'col2 name': 'FLOAT'})
- **varchar\_lengths** (*Dict[str, int]*, *optional*) – Dict of VARCHAR length by columns. (e.g. {"col1": 10, "col5": 200}).

**Returns** None.

**Return type** None

## Examples

Writing to MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> wr.mysql.to_sql(
...     df=df
...     table="my_table",
...     schema="test",
...     con=con
... )
>>> con.close()
```

### 1.3.7 Amazon Timestream

<code>create_database(database[, kms_key_id, ...])</code>	Create a new Timestream database.
<code>create_table(database, table, ...[, tags, ...])</code>	Create a new Timestream database.
<code>delete_database(database[, boto3_session])</code>	Delete a given Timestream database.
<code>delete_table(database, table[, boto3_session])</code>	Delete a given Timestream table.
<code>query(sql[, boto3_session])</code>	Run a query and retrieve the result as a Pandas DataFrame.
<code>write(df, database, table, time_col, ...[, ...])</code>	Store a Pandas DataFrame into a Amazon Timestream table.

#### awswrangler.timestream.create\_database

`awswrangler.timestream.create_database` (*database: str, kms\_key\_id: Optional[str] = None, tags: Optional[Dict[str, str]] = None, boto3\_session: Optional[boto3.session.Session] = None*) → str

Create a new Timestream database.

**Note:** If the KMS key is not specified, the database will be encrypted with a Timestream managed KMS key located in your account.

#### Parameters

- **database** (*str*) – Database name.
- **kms\_key\_id** (*Optional[str]*) – The KMS key for the database. If the KMS key is not specified, the database will be encrypted with a Timestream managed KMS key located in your account.
- **tags** (*Optional[Dict[str, str]]*) – Key/Value dict to put on the database. Tags enable you to categorize databases and/or tables, for example, by purpose, owner, or environment. e.g. {"foo": "boo", "bar": "xoo"}
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3\_session receive None.

**Returns** The Amazon Resource Name that uniquely identifies this database. (ARN)

**Return type** str

## Examples

Creating a database.

```
>>> import awswrangler as wr
>>> arn = wr.timestream.create_database("MyDatabase")
```

## awswrangler.timestream.create\_table

`awswrangler.timestream.create_table`(*database: str, table: str, memory\_retention\_hours: int, magnetic\_retention\_days: int, tags: Optional[Dict[str, str]] = None, boto3\_session: Optional[boto3.session.Session] = None*) → str

Create a new Timestream database.

---

**Note:** If the KMS key is not specified, the database will be encrypted with a Timestream managed KMS key located in your account.

---

### Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **memory\_retention\_hours** (*int*) – The duration for which data must be stored in the memory store.
- **magnetic\_retention\_days** (*int*) – The duration for which data must be stored in the magnetic store.
- **tags** (*Optional[Dict[str, str]]*) – Key/Value dict to put on the table. Tags enable you to categorize databases and/or tables, for example, by purpose, owner, or environment. e.g. {"foo": "boo", "bar": "xoo"}
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3\_session receive None.

**Returns** The Amazon Resource Name that uniquely identifies this database. (ARN)

**Return type** str

## Examples

Creating a table.

```
>>> import awswrangler as wr
>>> arn = wr.timestream.create_table(
...     database="MyDatabase",
...     table="MyTable",
...     memory_retention_hours=3,
...     magnetic_retention_days=7
... )
```

**awswrangler.timestream.delete\_database**

`awswrangler.timestream.delete_database(database: str, boto3_session: Optional[boto3.session.Session] = None) → None`

Delete a given Timestream database. This is an irreversible operation.

After a database is deleted, the time series data from its tables cannot be recovered.

All tables in the database must be deleted first, or a `ValidationException` error will be thrown.

Due to the nature of distributed retries, the operation can return either success or a `ResourceNotFoundException`. Clients should consider them equivalent.

**Parameters**

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

**Returns** `None`.

**Return type** `None`

**Examples**

Deleting a database

```
>>> import awswrangler as wr
>>> arn = wr.timestream.delete_database("MyDatabase")
```

**awswrangler.timestream.delete\_table**

`awswrangler.timestream.delete_table(database: str, table: str, boto3_session: Optional[boto3.session.Session] = None) → None`

Delete a given Timestream table.

This is an irreversible operation.

After a Timestream database table is deleted, the time series data stored in the table cannot be recovered.

Due to the nature of distributed retries, the operation can return either success or a `ResourceNotFoundException`. Clients should consider them equivalent.

**Parameters**

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

**Returns** `None`.

**Return type** `None`

## Examples

Deleting a table

```
>>> import awswrangler as wr
>>> arn = wr.timestream.delete_table("MyDatabase", "MyTable")
```

## awswrangler.timestream.query

`awswrangler.timestream.query(sql: str, boto3_session: Optional[boto3.session.Session] = None)`  
→ `pandas.core.frame.DataFrame`

Run a query and retrieve the result as a Pandas DataFrame.

### Parameters

- **sql** (*str*) – SQL query.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3\_session receive None.

**Returns** Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>

**Return type** `pd.DataFrame`

## Examples

Running a query and storing the result as a Pandas DataFrame

```
>>> import awswrangler as wr
>>> df = wr.timestream.query('SELECT * FROM "sampleDB"."sampleTable" ORDER BY ↵
↵time DESC LIMIT 10')
```

## awswrangler.timestream.write

`awswrangler.timestream.write(df: pandas.core.frame.DataFrame, database: str, table: str, time_col: str, measure_col: str, dimensions_cols: List[str], num_threads: int = 32, boto3_session: Optional[boto3.session.Session] = None) → List[Dict[str, str]]`

Store a Pandas DataFrame into a Amazon Timestream table.

### Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **database** (*str*) – Amazon Timestream database name.
- **table** (*str*) – Amazon Timestream table name.
- **time\_col** (*str*) – DataFrame column name to be used as time. MUST be a timestamp column.
- **measure\_col** (*str*) – DataFrame column name to be used as measure.
- **dimensions\_cols** (*List[str]*) – List of DataFrame column names to be used as dimensions.
- **num\_threads** (*str*) – Number of thread to be used for concurrent writing.

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

**Returns** Rejected records.

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

## Examples

Store a Pandas DataFrame into a Amazon Timestream table.

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> df = pd.DataFrame(
>>>     {
>>>         "time": [datetime.now(), datetime.now(), datetime.now()],
>>>         "dim0": ["foo", "boo", "bar"],
>>>         "dim1": [1, 2, 3],
>>>         "measure": [1.0, 1.1, 1.2],
>>>     }
>>> )
>>> rejected_records = wr.timestream.write(
>>>     df=df,
>>>     database="sampleDB",
>>>     table="sampleTable",
>>>     time_col="time",
>>>     measure_col="measure",
>>>     dimensions_cols=["dim0", "dim1"],
>>> )
>>> assert len(rejected_records) == 0
```

## 1.3.8 Amazon EMR

<i>build_spark_step</i> (path[, deploy_mode, ...])	Build the Step structure (dictionary).
<i>build_step</i> (command[, name, ...])	Build the Step structure (dictionary).
<i>create_cluster</i> (subnet_id[, cluster_name, ...])	Create a EMR cluster with instance fleets configuration.
<i>get_cluster_state</i> (cluster_id[, boto3_session])	Get the EMR cluster state.
<i>get_step_state</i> (cluster_id, step_id[, ...])	Get EMR step state.
<i>submit_ecr_credentials_refresh</i> (cluster_id, path)	Update internal ECR credentials.
<i>submit_spark_step</i> (cluster_id, path[, ...])	Submit Spark Step.
<i>submit_step</i> (cluster_id, command[, name, ...])	Submit new job in the EMR Cluster.
<i>submit_steps</i> (cluster_id, steps[, boto3_session])	Submit a list of steps.
<i>terminate_cluster</i> (cluster_id[, boto3_session])	Terminate EMR cluster.

**awswrangler.emr.build\_spark\_step**

`awswrangler.emr.build_spark_step` (*path*: *str*, *deploy\_mode*: *str* = 'cluster', *docker\_image*: *Optional[str]* = *None*, *name*: *str* = 'my-step', *action\_on\_failure*: *str* = 'CONTINUE', *region*: *Optional[str]* = *None*, *boto3\_session*: *Optional[boto3.session.Session]* = *None*) → *Dict[str, Any]*

Build the Step structure (dictionary).

**Parameters**

- **path** (*str*) – Script path. (e.g. `s3://bucket/app.py`)
- **deploy\_mode** (*str*) – “cluster” | “client”
- **docker\_image** (*str*, *optional*) – e.g. “{ACCOUNT\_ID}.dkr.ecr.{REGION}.amazonaws.com/{IMAGE\_NAME}:{TAG}”
- **name** (*str*, *optional*) – Step name.
- **action\_on\_failure** (*str*) – ‘TERMINATE\_JOB\_FLOW’, ‘TERMINATE\_CLUSTER’, ‘CANCEL\_AND\_WAIT’, ‘CONTINUE’
- **region** (*str*, *optional*) – Region name to not get it from `boto3.Session`. (e.g. *us-east-1*)
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive *None*.

**Returns** Step structure.

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

**Examples**

```
>>> import awswrangler as wr
>>> step_id = wr.emr.submit_steps(
>>>     cluster_id="cluster-id",
>>>     steps=[
>>>         wr.emr.build_spark_step(path="s3://bucket/app.py")
>>>     ]
>>> )
```

**awswrangler.emr.build\_step**

`awswrangler.emr.build_step` (*command*: *str*, *name*: *str* = 'my-step', *action\_on\_failure*: *str* = 'CONTINUE', *script*: *bool* = *False*, *region*: *Optional[str]* = *None*, *boto3\_session*: *Optional[boto3.session.Session]* = *None*) → *Dict[str, Any]*

Build the Step structure (dictionary).

**Parameters**

- **command** (*str*) – e.g. ‘echo “Hello!”’ e.g. for script ‘`s3://.../script.sh arg1 arg2`’
- **name** (*str*, *optional*) – Step name.
- **action\_on\_failure** (*str*) – ‘TERMINATE\_JOB\_FLOW’, ‘TERMINATE\_CLUSTER’, ‘CANCEL\_AND\_WAIT’, ‘CONTINUE’



- **script** (*bool*) – False for raw command or True for script runner. <https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-commandrunner.html>
- **region** (*str*, *optional*) – Region name to not get it from boto3.Session. (e.g. *us-east-1*)
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Step structure.

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

### Examples

```
>>> import awswrangler as wr
>>> steps = []
>>> for cmd in ['echo "Hello"', 'ls -la']:
...     steps.append(wr.emr.build_step(name=cmd, command=cmd))
>>> wr.emr.submit_steps(cluster_id="cluster-id", steps=steps)
```

**aws wrangler.emr.create\_cluster**

```
aws wrangler.emr.create_cluster(subnet_id: str, cluster_name: str = 'my-emr-
cluster', logging_s3_path: Optional[str] = None,
emr_release: str = 'emr-6.0.0', emr_ec2_role:
str = 'EMR_EC2_DefaultRole', emr_role: str =
'EMR_DefaultRole', instance_type_master: str = 'r5.xlarge',
instance_type_core: str = 'r5.xlarge', instance_type_task:
str = 'r5.xlarge', instance_ebs_size_master: int = 64, in-
stance_ebs_size_core: int = 64, instance_ebs_size_task:
int = 64, instance_num_on_demand_master: int
= 1, instance_num_on_demand_core: int = 0,
instance_num_on_demand_task: int = 0, in-
stance_num_spot_master: int = 0, instance_num_spot_core:
int = 0, instance_num_spot_task: int = 0,
spot_bid_percentage_of_on_demand_master: int =
100, spot_bid_percentage_of_on_demand_core: int
= 100, spot_bid_percentage_of_on_demand_task:
int = 100, spot_provisioning_timeout_master:
int = 5, spot_provisioning_timeout_core: int
= 5, spot_provisioning_timeout_task: int = 5,
spot_timeout_to_on_demand_master: bool = True,
spot_timeout_to_on_demand_core: bool = True,
spot_timeout_to_on_demand_task: bool = True, python3: bool
= True, spark_glue_catalog: bool = True, hive_glue_catalog:
bool = True, presto_glue_catalog: bool = True, consistent_view:
bool = False, consistent_view_retry_seconds: int = 10, consis-
tent_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, se-
curity_groups_slave_additional: Optional[List[str]] =
None, security_group_service_access: Optional[str] =
None, docker: bool = False, extra_public_registries: Op-
tional[List[str]] = None, spark_log_level: str = 'WARN',
spark_jars_path: Optional[List[str]] = None, spark_defaults:
Optional[Dict[str, str]] = None, spark_pyarrow: bool
= False, custom_classifications: Optional[List[Dict[str,
Any]]] = None, maximize_resource_allocation: bool =
False, steps: Optional[List[Dict[str, Any]]] = None,
keep_cluster_alive_when_no_steps: bool = True, termina-
tion_protected: bool = False, tags: Optional[Dict[str, str]] =
None, boto3_session: Optional[boto3.session.Session] = None)
→ str
```

Create a EMR cluster with instance fleets configuration.

<https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-instance-fleet.html>

**Parameters**

- **subnet\_id** (*str*) – VPC subnet ID.
- **cluster\_name** (*str*) – Cluster name.

- **logging\_s3\_path** (*str*, *optional*) – Logging s3 path (e.g. `s3://BUCKET_NAME/DIRECTORY_NAME/`). If None, the default is `s3://aws-logs-{AccountId}-{RegionId}/elasticmapreduce/`
- **emr\_release** (*str*) – EMR release (e.g. `emr-5.28.0`).
- **emr\_ec2\_role** (*str*) – IAM role name.
- **emr\_role** (*str*) – IAM role name.
- **instance\_type\_master** (*str*) – EC2 instance type.
- **instance\_type\_core** (*str*) – EC2 instance type.
- **instance\_type\_task** (*str*) – EC2 instance type.
- **instance\_ebs\_size\_master** (*int*) – Size of EBS in GB.
- **instance\_ebs\_size\_core** (*int*) – Size of EBS in GB.
- **instance\_ebs\_size\_task** (*int*) – Size of EBS in GB.
- **instance\_num\_on\_demand\_master** (*int*) – Number of on demand instances.
- **instance\_num\_on\_demand\_core** (*int*) – Number of on demand instances.
- **instance\_num\_on\_demand\_task** (*int*) – Number of on demand instances.
- **instance\_num\_spot\_master** (*int*) – Number of spot instances.
- **instance\_num\_spot\_core** (*int*) – Number of spot instances.
- **instance\_num\_spot\_task** (*int*) – Number of spot instances.
- **spot\_bid\_percentage\_of\_on\_demand\_master** (*int*) – The bid price, as a percentage of On-Demand price.
- **spot\_bid\_percentage\_of\_on\_demand\_core** (*int*) – The bid price, as a percentage of On-Demand price.
- **spot\_bid\_percentage\_of\_on\_demand\_task** (*int*) – The bid price, as a percentage of On-Demand price.
- **spot\_provisioning\_timeout\_master** (*int*) – The spot provisioning timeout period in minutes. If Spot instances are not provisioned within this time period, the `TimeOutAction` is taken. Minimum value is 5 and maximum value is 1440. The timeout applies only during initial provisioning, when the cluster is first created.
- **spot\_provisioning\_timeout\_core** (*int*) – The spot provisioning timeout period in minutes. If Spot instances are not provisioned within this time period, the `TimeOutAction` is taken. Minimum value is 5 and maximum value is 1440. The timeout applies only during initial provisioning, when the cluster is first created.
- **spot\_provisioning\_timeout\_task** (*int*) – The spot provisioning timeout period in minutes. If Spot instances are not provisioned within this time period, the `TimeOutAction` is taken. Minimum value is 5 and maximum value is 1440. The timeout applies only during initial provisioning, when the cluster is first created.
- **spot\_timeout\_to\_on\_demand\_master** (*bool*) – After a provisioning timeout should the cluster switch to on demand or shutdown?
- **spot\_timeout\_to\_on\_demand\_core** (*bool*) – After a provisioning timeout should the cluster switch to on demand or shutdown?
- **spot\_timeout\_to\_on\_demand\_task** (*bool*) – After a provisioning timeout should the cluster switch to on demand or shutdown?

- **python3** (*bool*) – Python 3 Enabled?
- **spark\_glue\_catalog** (*bool*) – Spark integration with Glue Catalog?
- **hive\_glue\_catalog** (*bool*) – Hive integration with Glue Catalog?
- **presto\_glue\_catalog** (*bool*) – Presto integration with Glue Catalog?
- **consistent\_view** (*bool*) – Consistent view allows EMR clusters to check for list and read-after-write consistency for Amazon S3 objects written by or synced with EMRFS. <https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-consistent-view.html>
- **consistent\_view\_retry\_seconds** (*int*) – Delay between the tries (seconds).
- **consistent\_view\_retry\_count** (*int*) – Number of tries.
- **consistent\_view\_table\_name** (*str*) – Name of the DynamoDB table to store the consistent view data.
- **bootstraps\_paths** (*List[str]*, *optional*) – Bootstraps paths (e.g. ["s3://BUCKET\_NAME/script.sh"]).
- **debugging** (*bool*) – Debugging enabled?
- **applications** (*List[str]*, *optional*) – List of applications (e.g. ["Hadoop", "Spark", "Ganglia", "Hive"]). If None, ["Spark"] will be considered.
- **visible\_to\_all\_users** (*bool*) – True or False.
- **key\_pair\_name** (*str*, *optional*) – Key pair name.
- **security\_group\_master** (*str*, *optional*) – The identifier of the Amazon EC2 security group for the master node.
- **security\_groups\_master\_additional** (*str*, *optional*) – A list of additional Amazon EC2 security group IDs for the master node.
- **security\_group\_slave** (*str*, *optional*) – The identifier of the Amazon EC2 security group for the core and task nodes.
- **security\_groups\_slave\_additional** (*str*, *optional*) – A list of additional Amazon EC2 security group IDs for the core and task nodes.
- **security\_group\_service\_access** (*str*, *optional*) – The identifier of the Amazon EC2 security group for the Amazon EMR service to access clusters in VPC private subnets.
- **docker** (*bool*) – Enable Docker Hub and ECR registries access.
- **extra\_public\_registries** (*List[str]*, *optional*) – Additional docker registries.
- **spark\_log\_level** (*str*) – log4j.rootCategory log level (ALL, DEBUG, INFO, WARN, ERROR, FATAL, OFF, TRACE).
- **spark\_jars\_path** (*List[str]*, *optional*) – spark.jars e.g. [s3://.../foo.jar, s3://.../boo.jar] <https://spark.apache.org/docs/latest/configuration.html>
- **spark\_defaults** (*Dict[str, str]*, *optional*) – <https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html#spark-defaults>
- **spark\_pyarrow** (*bool*) – Enable PySpark to use PyArrow behind the scenes. P.S. You must install pyarrow by your self via bootstrap

- **custom\_classifications** (*List[Dict[str, Any]]*, *optional*) – Extra classifications.
- **maximize\_resource\_allocation** (*bool*) – Configure your executors to utilize the maximum resources possible <https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html#emr-spark-maximizeresourceallocation>
- **steps** (*List[Dict[str, Any]]*, *optional*) – Steps definitions (Obs : str Use EMR.build\_step() to build it)
- **keep\_cluster\_alive\_when\_no\_steps** (*bool*) – Specifies whether the cluster should remain available after completing all steps
- **termination\_protected** (*bool*) – Specifies whether the Amazon EC2 instances in the cluster are protected from termination by API calls, user intervention, or in the event of a job-flow error.
- **tags** (*Dict[str, str]*, *optional*) – Key/Value collection to put on the Cluster. e.g. {"foo": "boo", "bar": "xoo"}
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Cluster ID.

**Return type** str

## Examples

### Minimal Example

```
>>> import awswrangler as wr
>>> cluster_id = wr.emr.create_cluster("SUBNET_ID")
```

### Minimal Example With Custom Classification

```
>>> import awswrangler as wr
>>> cluster_id = wr.emr.create_cluster(
>>>     subnet_id="SUBNET_ID",
>>>     custom_classifications=[
>>>         {
>>>             "Classification": "livy-conf",
>>>             "Properties": {
>>>                 "livy.spark.master": "yarn",
>>>                 "livy.spark.deploy-mode": "cluster",
>>>                 "livy.server.session.timeout": "16h",
>>>             },
>>>         },
>>>     ],
>>> )
```

### Full Example

```
>>> import awswrangler as wr
>>> cluster_id = wr.emr.create_cluster(
...     cluster_name="wrangler_cluster",
...     logging_s3_path=f"s3://BUCKET_NAME/emr-logs/",
...     emr_release="emr-5.28.0",
...     subnet_id="SUBNET_ID",
```

(continues on next page)

(continued from previous page)

```

...     emr_ec2_role="EMR_EC2_DefaultRole",
...     emr_role="EMR_DefaultRole",
...     instance_type_master="m5.xlarge",
...     instance_type_core="m5.xlarge",
...     instance_type_task="m5.xlarge",
...     instance_ebs_size_master=50,
...     instance_ebs_size_core=50,
...     instance_ebs_size_task=50,
...     instance_num_on_demand_master=1,
...     instance_num_on_demand_core=1,
...     instance_num_on_demand_task=1,
...     instance_num_spot_master=0,
...     instance_num_spot_core=1,
...     instance_num_spot_task=1,
...     spot_bid_percentage_of_on_demand_master=100,
...     spot_bid_percentage_of_on_demand_core=100,
...     spot_bid_percentage_of_on_demand_task=100,
...     spot_provisioning_timeout_master=5,
...     spot_provisioning_timeout_core=5,
...     spot_provisioning_timeout_task=5,
...     spot_timeout_to_on_demand_master=True,
...     spot_timeout_to_on_demand_core=True,
...     spot_timeout_to_on_demand_task=True,
...     python3=True,
...     spark_glue_catalog=True,
...     hive_glue_catalog=True,
...     presto_glue_catalog=True,
...     bootstraps_paths=None,
...     debugging=True,
...     applications=["Hadoop", "Spark", "Ganglia", "Hive"],
...     visible_to_all_users=True,
...     key_pair_name=None,
...     spark_jars_path=[f"s3://...jar"],
...     maximize_resource_allocation=True,
...     keep_cluster_alive_when_no_steps=True,
...     termination_protected=False,
...     spark_pyarrow=True,
...     tags={
...         "foo": "boo"
...     })

```

### aws wrangler.emr.get\_cluster\_state

`aws wrangler.emr.get_cluster_state` (*cluster\_id*: *str*, *boto3\_session*: *Optional[boto3.session.Session] = None*) → *str*

Get the EMR cluster state.

Possible states: 'STARTING', 'BOOTSTRAPPING', 'RUNNING', 'WAITING', 'TERMINATING', 'TERMINATED', 'TERMINATED\_WITH\_ERRORS'

#### Parameters

- **cluster\_id** (*str*) – Cluster ID.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** State.

**Return type** str

### Examples

```
>>> import awswrangler as wr
>>> state = wr.emr.get_cluster_state("cluster-id")
```

### awswrangler.emr.get\_step\_state

`awswrangler.emr.get_step_state(cluster_id: str, step_id: str, boto3_session: Optional[boto3.session.Session] = None) → str`

Get EMR step state.

Possible states: 'PENDING', 'CANCEL\_PENDING', 'RUNNING', 'COMPLETED', 'CANCELLED', 'FAILED', 'INTERRUPTED'

#### Parameters

- **cluster\_id** (*str*) – Cluster ID.
- **step\_id** (*str*) – Step ID.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** State.

**Return type** str

### Examples

```
>>> import awswrangler as wr
>>> state = wr.emr.get_step_state("cluster-id", "step-id")
```

### awswrangler.emr.submit\_ecr\_credentials\_refresh

`awswrangler.emr.submit_ecr_credentials_refresh(cluster_id: str, path: str, action_on_failure: str = 'CONTINUE', boto3_session: Optional[boto3.session.Session] = None) → str`

Update internal ECR credentials.

#### Parameters

- **cluster\_id** (*str*) – Cluster ID.
- **path** (*str*) – Amazon S3 path where Wrangler will stage the script `ecr_credentials_refresh.py` (e.g. `s3://bucket/emr/`)
- **action\_on\_failure** (*str*) – 'TERMINATE\_JOB\_FLOW', 'TERMINATE\_CLUSTER', 'CANCEL\_AND\_WAIT', 'CONTINUE'
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Step ID.

**Return type** str

## Examples

```
>>> import awswrangler as wr
>>> step_id = wr.emr.submit_ecr_credentials_refresh("cluster_id", "s3://bucket/
↪emr/")
```

## awswrangler.emr.submit\_spark\_step

`awswrangler.emr.submit_spark_step`(*cluster\_id*: str, *path*: str, *deploy\_mode*: str = 'cluster', *docker\_image*: Optional[str] = None, *name*: str = 'my-step', *action\_on\_failure*: str = 'CONTINUE', *region*: Optional[str] = None, *boto3\_session*: Optional[boto3.session.Session] = None) → str

Submit Spark Step.

### Parameters

- **cluster\_id** (str) – Cluster ID.
- **path** (str) – Script path. (e.g. s3://bucket/app.py)
- **deploy\_mode** (str) – “cluster” | “client”
- **docker\_image** (str, optional) – e.g. “{ACCOUNT\_ID}.dkr.ecr.{REGION}.amazonaws.com/{IMAGE\_NAME}:{TAG}”
- **name** (str, optional) – Step name.
- **action\_on\_failure** (str) – ‘TERMINATE\_JOB\_FLOW’, ‘TERMINATE\_CLUSTER’, ‘CANCEL\_AND\_WAIT’, ‘CONTINUE’
- **region** (str, optional) – Region name to not get it from boto3.Session. (e.g. *us-east-1*)
- **boto3\_session** (boto3.Session(), optional) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Step ID.

**Return type** str

## Examples

```
>>> import awswrangler as wr
>>> step_id = wr.emr.submit_spark_step(
>>>     cluster_id="cluster-id",
>>>     path="s3://bucket/emr/app.py"
>>> )
```



### aws wrangler.emr.submit\_step

`aws wrangler.emr.submit_step` (*cluster\_id*: str, *command*: str, *name*: str = 'my-step', *action\_on\_failure*: str = 'CONTINUE', *script*: bool = False, *boto3\_session*: Optional[boto3.session.Session] = None) → str

Submit new job in the EMR Cluster.

#### Parameters

- **cluster\_id** (str) – Cluster ID.
- **command** (str) – e.g. 'echo "Hello!"' e.g. for script 's3://.../script.sh arg1 arg2'
- **name** (str, optional) – Step name.
- **action\_on\_failure** (str) – 'TERMINATE\_JOB\_FLOW', 'TERMINATE\_CLUSTER', 'CANCEL\_AND\_WAIT', 'CONTINUE'
- **script** (bool) – True for raw command or False for script runner. <https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-commandrunner.html>
- **boto3\_session** (boto3.Session(), optional) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Step ID.

**Return type** str

#### Examples

```
>>> import aws wrangler as wr
>>> step_id = wr.emr.submit_step(
...     cluster_id=cluster_id,
...     name="step_test",
...     command="s3://...script.sh arg1 arg2",
...     script=True)
```

### aws wrangler.emr.submit\_steps

`aws wrangler.emr.submit_steps` (*cluster\_id*: str, *steps*: List[Dict[str, Any]], *boto3\_session*: Optional[boto3.session.Session] = None) → List[str]

Submit a list of steps.

#### Parameters

- **cluster\_id** (str) – Cluster ID.
- **steps** (List[Dict[str, Any]]) – Steps definitions (Obs: Use EMR.build\_step() to build it).
- **boto3\_session** (boto3.Session(), optional) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** List of step IDs.

**Return type** List[str]

## Examples

```
>>> import awswrangler as wr
>>> for cmd in ['echo "Hello"', 'ls -la']:
...     steps.append(wr.emr.build_step(name=cmd, command=cmd))
>>> wr.emr.submit_steps(cluster_id="cluster-id", steps=steps)
```

## awswrangler.emr.terminate\_cluster

`awswrangler.emr.terminate_cluster` (*cluster\_id*: *str*, *boto3\_session*: *Optional[boto3.session.Session]* = *None*) → *None*

Terminate EMR cluster.

### Parameters

- **cluster\_id** (*str*) – Cluster ID.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if *boto3\_session* receive *None*.

**Returns** *None*.

**Return type** *None*

## Examples

```
>>> import awswrangler as wr
>>> wr.emr.terminate_cluster("cluster-id")
```

## 1.3.9 Amazon CloudWatch Logs

<code>read_logs</code> ( <i>query</i> , <i>log_group_names</i> [, ...])	Run a query against AWS CloudWatchLogs Insights and convert the results to Pandas DataFrame.
<code>run_query</code> ( <i>query</i> , <i>log_group_names</i> [, ...])	Run a query against AWS CloudWatchLogs Insights and wait the results.
<code>start_query</code> ( <i>query</i> , <i>log_group_names</i> [, ...])	Run a query against AWS CloudWatchLogs Insights.
<code>wait_query</code> ( <i>query_id</i> [, <i>boto3_session</i> ])	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, 12, 11, 11, 1, 26, 910970)*, *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](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, 12, 11, 11, 1, 26, 910923), 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](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, 12, 11, 11, 1, 26, 910902), 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](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",
... )
```

**aws wrangler.cloudwatch.wait\_query**

`aws wrangler.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](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 aws wrangler as wr
>>> query_id = wr.cloudwatch.start_query(
...     log_group_names=["loggroup"],
...     query="fields @timestamp, @message | sort @timestamp desc | limit 5",
... )
... response = wr.cloudwatch.wait_query(query_id=query_id)
```

**1.3.10 Amazon QuickSight**

<code>cancel_ingestion(ingestion_id[, ...])</code>	Cancel an ongoing ingestion of data into SPICE.
<code>create_athena_data_source(name[, work-group, ...])</code>	Create a QuickSight data source pointing to an Athena/Workgroup.
<code>create_athena_dataset(name[, database, ...])</code>	Create a QuickSight dataset.
<code>create_ingestion([dataset_name, dataset_id, ...])</code>	Create and starts a new SPICE ingestion on a dataset.
<code>delete_all_dashboards([account_id, ...])</code>	Delete all dashboards.
<code>delete_all_data_sources([account_id, ...])</code>	Delete all data sources.
<code>delete_all_datasets([account_id, boto3_session])</code>	Delete all datasets.
<code>delete_all_templates([account_id, boto3_session])</code>	Delete all templates.
<code>delete_dashboard([name, dashboard_id, ...])</code>	Delete a dashboard.
<code>delete_data_source([name, data_source_id, ...])</code>	Delete a data source.
<code>delete_dataset([name, dataset_id, ...])</code>	Delete a dataset.
<code>delete_template([name, template_id, ...])</code>	Delete a tamplate.
<code>describe_dashboard([name, dashboard_id, ...])</code>	Describe a QuickSight dashboard by name or ID.
<code>describe_data_source([name, data_source_id, ...])</code>	Describe a QuickSight data source by name or ID.
<code>describe_data_source_permissions([name, ...])</code>	Describe a QuickSight data source permissions by name or ID.
<code>describe_dataset([name, dataset_id, ...])</code>	Describe a QuickSight dataset by name or ID.

continues on next page

Table 10 – continued from previous page

<code>describe_ingestion(ingestion_id[, ...])</code>	Describe a QuickSight ingestion by ID.
<code>get_dashboard_id(name[, account_id, ...])</code>	Get QuickSight dashboard ID given a name and fails if there is more than 1 ID associated with this name.
<code>get_dashboard_ids(name[, account_id, ...])</code>	Get QuickSight dashboard IDs given a name.
<code>get_data_source_arn(name[, account_id, ...])</code>	Get QuickSight data source ARN given a name and fails if there is more than 1 ARN associated with this name.
<code>get_data_source_arns(name[, account_id, ...])</code>	Get QuickSight Data source ARNs given a name.
<code>get_data_source_id(name[, account_id, ...])</code>	Get QuickSight data source ID given a name and fails if there is more than 1 ID associated with this name.
<code>get_data_source_ids(name[, account_id, ...])</code>	Get QuickSight data source IDs given a name.
<code>get_dataset_id(name[, account_id, boto3_session])</code>	Get QuickSight Dataset ID given a name and fails if there is more than 1 ID associated with this name.
<code>get_dataset_ids(name[, account_id, ...])</code>	Get QuickSight dataset IDs given a name.
<code>get_template_id(name[, account_id, ...])</code>	Get QuickSight template ID given a name and fails if there is more than 1 ID associated with this name.
<code>get_template_ids(name[, account_id, ...])</code>	Get QuickSight template IDs given a name.
<code>list_dashboards([account_id, boto3_session])</code>	List dashboards in an AWS account.
<code>list_data_sources([account_id, boto3_session])</code>	List all QuickSight Data sources summaries.
<code>list_datasets([account_id, boto3_session])</code>	List all QuickSight datasets summaries.
<code>list_groups([namespace, account_id, ...])</code>	List all QuickSight Groups.
<code>list_group_memberships(group_name[, ...])</code>	List all QuickSight Group memberships.
<code>list_iam_policy_assignments([status, ...])</code>	List IAM policy assignments in the current Amazon QuickSight account.
<code>list_iam_policy_assignments_for_user(user[, ...])</code>	List the IAM policy assignments.
<code>list_ingestions([dataset_name, dataset_id, ...])</code>	List the history of SPICE ingestions for a dataset.
<code>list_templates([account_id, boto3_session])</code>	List all QuickSight templates.
<code>list_users([namespace, account_id, ...])</code>	Return a list of all of the Amazon QuickSight users belonging to this account.
<code>list_user_groups(user_name[, namespace, ...])</code>	List the Amazon QuickSight groups that an Amazon QuickSight user is a member of.

## aws wrangler quicksight.cancel\_ingestion

`aws wrangler quicksight.cancel_ingestion(ingestion_id: str, dataset_name: Optional[str] = None, dataset_id: Optional[str] = None, account_id: Optional[str] = None, boto3_session: Optional[boto3.session.Session] = None) → None`

Cancel an ongoing ingestion of data into SPICE.

**Note:** You must pass a not None value for `dataset_name` or `dataset_id` argument.

### Parameters

- **ingestion\_id** (*str*) – Ingestion ID.
- **dataset\_name** (*str, optional*) – Dataset name.
- **dataset\_id** (*str, optional*) – Dataset ID.
- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.

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

**Returns** None.

**Return type** None

### Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.cancel_ingestion(ingestion_id="...", dataset_name="...")
```

### awswrangler.quicksight.create\_athena\_data\_source

```
awswrangler.quicksight.create_athena_data_source(name: str, workgroup: str
                                                  = 'primary', allowed_to_use:
Optional[List[str]] = None,
                                                  allowed_to_manage: Op-
tional[List[str]] = None, tags:
Optional[Dict[str, str]] = None,
                                                  account_id: Optional[str] =
None, boto3_session: Op-
tional[boto3.session.Session] =
None, namespace: str = 'default') →
None
```

Create a QuickSight data source pointing to an Athena/Workgroup.

**Note:** You will not be able to see the the data source in the console if you not pass your user to one of the `allowed_*` arguments.

#### Parameters

- **name** (*str*) – Data source name.
- **workgroup** (*str*) – Athena workgroup.
- **tags** (*Dict[str, str]*, *optional*) – Key/Value collection to put on the Cluster. e.g. {"foo": "boo", "bar": "xoo"}
- **allowed\_to\_use** (*optional*) – List of principals that will be allowed to see and use the data source. e.g. ["John"]
- **allowed\_to\_manage** (*optional*) – List of principals that will be allowed to see, use, update and delete the data source. e.g. ["Mary"]
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.
- **namespace** (*str*) – The namespace. Currently, you should set this to default.

**Returns** None.

**Return type** None

## Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.create_athena_data_source(
...     name="...",
...     allowed_to_manage=["john"]
... )
```

## awswrangler.quicksight.create\_athena\_dataset

```
awswrangler.quicksight.create_athena_dataset(name: str, database: Optional[str] =
None, table: Optional[str] = None, sql:
Optional[str] = None, sql_name: str =
'CustomSQL', data_source_name: Op-
tional[str] = None, data_source_arn:
Optional[str] = None, import_mode: str
= 'DIRECT_QUERY', allowed_to_use:
Optional[List[str]] = None, al-
lowed_to_manage: Optional[List[str]]
= None, logical_table_alias: str
= 'LogicalTable', rename_columns:
Optional[Dict[str, str]] = None,
cast_columns_types: Optional[Dict[str,
str]] = None, tags: Optional[Dict[str,
str]] = None, account_id: Op-
tional[str] = None, boto3_session:
Optional[boto3.session.Session] = None,
namespace: str = 'default') → str
```

Create a QuickSight dataset.

---

**Note:** You will not be able to see the the dataset in the console if you not pass your username to one of the `allowed_*` arguments.

---

---

**Note:** You must pass database/table OR sql argument.

---

---

**Note:** You must pass `data_source_name` OR `data_source_arn` argument.

---

## Parameters

- **name** (*str*) – Dataset name.
- **database** (*str*) – Athena's database name.
- **table** (*str*) – Athena's table name.
- **sql** (*str*) – Use a SQL query to define your table.
- **sql\_name** (*str*) – Query name.
- **data\_source\_name** (*str*, *optional*) – QuickSight data source name.
- **data\_source\_arn** (*str*, *optional*) – QuickSight data source ARN.



- **import\_mode** (*str*) – Indicates whether you want to import the data into SPICE. ‘SPICE’|‘DIRECT\_QUERY’
- **tags** (*Dict[str, str], optional*) – Key/Value collection to put on the Cluster. e.g. {“foo”: “boo”, “bar”: “xoo”}
- **allowed\_to\_use** (*optional*) – List of usernames that will be allowed to see and use the data source. e.g. [“john”, “Mary”]
- **allowed\_to\_manage** (*optional*) – List of usernames that will be allowed to see, use, update and delete the data source. e.g. [“Mary”]
- **logical\_table\_alias** (*str*) – A display name for the logical table.
- **rename\_columns** (*Dict[str, str], optional*) – Dictionary to map column re-names. e.g. {“old\_name”: “new\_name”, “old\_name2”: “new\_name2”}
- **cast\_columns\_types** (*Dict[str, str], optional*) – Dictionary to map column casts. e.g. {“col\_name”: “STRING”, “col\_name2”: “DECIMAL”} Valid types: ‘STRING’|‘INTEGER’|‘DECIMAL’|‘DATETIME’
- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.
- **namespace** (*str*) – The namespace. Currently, you should set this to default.

**Returns** Dataset ID.

**Return type** str

## Examples

```
>>> import awswrangler as wr
>>> dataset_id = wr.quicksight.create_athena_dataset(
...     name="...",
...     database="..."
...     table="..."
...     data_source_name="..."
...     allowed_to_manage=["Mary"]
... )
```

## awswrangler.quicksight.create\_ingestion

`awswrangler.quicksight.create_ingestion` (*dataset\_name: Optional[str] = None, dataset\_id: Optional[str] = None, ingestion\_id: Optional[str] = None, account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → str

Create and starts a new SPICE ingestion on a dataset.

---

**Note:** You must pass `dataset_name` OR `dataset_id` argument.

---

## Parameters

- **dataset\_name**(*str*, *optional*) – Dataset name.
- **dataset\_id**(*str*, *optional*) – Dataset ID.
- **ingestion\_id**(*str*, *optional*) – Ingestion ID.
- **account\_id**(*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session**(*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Ingestion ID

**Return type** *str*

### Examples

```
>>> import awswrangler as wr
>>> status = wr.quicksight.create_ingestion("my_dataset")
```

### awswrangler.quicksight.delete\_all\_dashboards

`awswrangler.quicksight.delete_all_dashboards` (*account\_id*: *Optional[str]* = *None*, *boto3\_session*: *Optional[boto3.session.Session]* = *None*) → *None*

Delete all dashboards.

#### Parameters

- **account\_id**(*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session**(*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** *None*.

**Return type** *None*

### Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_all_dashboards()
```

### awswrangler.quicksight.delete\_all\_data\_sources

`awswrangler.quicksight.delete_all_data_sources` (*account\_id*: *Optional[str]* = *None*, *boto3\_session*: *Optional[boto3.session.Session]* = *None*) → *None*

Delete all data sources.

#### Parameters

- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** None.

**Return type** None

### Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_all_data_sources()
```

### awswrangler.quicksight.delete\_all\_datasets

```
awswrangler.quicksight.delete_all_datasets (account_id: Optional[str] =
None, boto3_session: Optional[boto3.session.Session] = None) →
None
```

Delete all datasets.

#### Parameters

- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** None.

**Return type** None

### Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_all_datasets()
```

### awswrangler.quicksight.delete\_all\_templates

```
awswrangler.quicksight.delete_all_templates (account_id: Optional[str] =
None, boto3_session: Optional[boto3.session.Session] = None)
→ None
```

Delete all templates.

#### Parameters

- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** None.

**Return type** None

### Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_all_templates()
```

### awswrangler.quicksight.delete\_dashboard

`awswrangler.quicksight.delete_dashboard` (*name: Optional[str] = None, dashboard\_id: Optional[str] = None, version\_number: Optional[int] = None, account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → None

Delete a dashboard.

---

**Note:** You must pass a not None name or dashboard\_id argument.

---

#### Parameters

- **name** (*str, optional*) – Dashboard name.
- **dashboard\_id** (*str, optional*) – The ID for the dashboard.
- **version\_number** (*int, optional*) – The version number of the dashboard. If the version number property is provided, only the specified version of the dashboard is deleted.
- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** None.

**Return type** None

### Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_dashboard(name="...")
```

### awswrangler.quicksight.delete\_data\_source

`awswrangler.quicksight.delete_data_source` (*name: Optional[str] = None, data\_source\_id: Optional[str] = None, account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → None

Delete a data source.

---

**Note:** You must pass a not None name or data\_source\_id argument.

---

#### Parameters

- **name** (*str*, *optional*) – Dashboard name.
- **data\_source\_id** (*str*, *optional*) – The ID for the data source.
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** None.

**Return type** None

#### Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_data_source(name="...")
```

#### awswrangler.quicksight.delete\_dataset

`awswrangler.quicksight.delete_dataset` (*name: Optional[str] = None, dataset\_id: Optional[str] = None, account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → None

Delete a dataset.

---

**Note:** You must pass a not None name or dataset\_id argument.

---

#### Parameters

- **name** (*str*, *optional*) – Dashboard name.
- **dataset\_id** (*str*, *optional*) – The ID for the dataset.
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** None.

**Return type** None

## Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_dataset(name="...")
```

### awswrangler.quicksight.delete\_template

`awswrangler.quicksight.delete_template` (*name: Optional[str] = None, template\_id: Optional[str] = None, version\_number: Optional[int] = None, account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → None

Delete a template.

---

**Note:** You must pass a not None name or template\_id argument.

---

#### Parameters

- **name** (*str, optional*) – Dashboard name.
- **template\_id** (*str, optional*) – The ID for the dashboard.
- **version\_number** (*int, optional*) – Specifies the version of the template that you want to delete. If you don't provide a version number, it deletes all versions of the template.
- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** None.

**Return type** None

## Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_template(name="...")
```

### awswrangler.quicksight.describe\_dashboard

`awswrangler.quicksight.describe_dashboard` (*name: Optional[str] = None, dashboard\_id: Optional[str] = None, account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Describe a QuickSight dashboard by name or ID.

---

**Note:** You must pass a not None name or dashboard\_id argument.

---

#### Parameters

- **name** (*str*, *optional*) – Dashboard name.
- **dashboard\_id** (*str*, *optional*) – Dashboard ID.
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Dashboard Description.

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

### Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_dashboard(name="my-dashboard")
```

### awswrangler.quicksight.describe\_data\_source

```
awswrangler.quicksight.describe_data_source (name: Optional[str] = None,
                                              data_source_id: Optional[str] =
                                              None, account_id: Optional[str]
                                              = None, boto3_session: Op-
                                              tional[boto3.session.Session] = None)
                                              → Dict[str, Any]
```

Describe a QuickSight data source by name or ID.

---

**Note:** You must pass a not None name or data\_source\_id argument.

---

#### Parameters

- **name** (*str*, *optional*) – Data source name.
- **data\_source\_id** (*str*, *optional*) – Data source ID.
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Data source Description.

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

## Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_data_source("...")
```

## awswrangler.quicksight.describe\_data\_source\_permissions

`awswrangler.quicksight.describe_data_source_permissions` (*name: Optional[str] = None, data\_source\_id: Optional[str] = None, account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Describe a QuickSight data source permissions by name or ID.

---

**Note:** You must pass a not None name or data\_source\_id argument.

---

### Parameters

- **name** (*str, optional*) – Data source name.
- **data\_source\_id** (*str, optional*) – Data source ID.
- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Data source Permissions Description.

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

## Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_data_source_permissions("my-data-source")
```

## awswrangler.quicksight.describe\_dataset

`awswrangler.quicksight.describe_dataset` (*name: Optional[str] = None, dataset\_id: Optional[str] = None, account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Describe a QuickSight dataset by name or ID.

---

**Note:** You must pass a not None name or dataset\_id argument.

---

### Parameters



- **name**(*str*, *optional*) – Dataset name.
- **dataset\_id**(*str*, *optional*) – Dataset ID.
- **account\_id**(*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session**(*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Dataset Description.

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

### Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_dataset("my-dataset")
```

### awswrangler.quicksight.describe\_ingestion

`awswrangler.quicksight.describe_ingestion`(*ingestion\_id: str*, *dataset\_name: Optional[str] = None*, *dataset\_id: Optional[str] = None*, *account\_id: Optional[str] = None*, *boto3\_session: Optional[boto3.session.Session] = None*) → Dict[str, Any]

Describe a QuickSight ingestion by ID.

---

**Note:** You must pass a not None value for `dataset_name` or `dataset_id` argument.

---

#### Parameters

- **ingestion\_id**(*str*) – Ingestion ID.
- **dataset\_name**(*str*, *optional*) – Dataset name.
- **dataset\_id**(*str*, *optional*) – Dataset ID.
- **account\_id**(*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session**(*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Ingestion Description.

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

## Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_dataset(ingestion_id="...", dataset_name=
↳ "...")
```

### awswrangler.quicksight.get\_dashboard\_id

`awswrangler.quicksight.get_dashboard_id` (*name: str, account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → str

Get QuickSight dashboard ID given a name and fails if there is more than 1 ID associated with this name.

#### Parameters

- **name** (*str*) – Dashboard name.
- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

**Returns** Dashboard ID.

**Return type** str

## Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_dashboard_id(name="...")
```

### awswrangler.quicksight.get\_dashboard\_ids

`awswrangler.quicksight.get_dashboard_ids` (*name: str, account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → List[str]

Get QuickSight dashboard IDs given a name.

---

**Note:** This function returns a list of ID because Quicksight accepts duplicated dashboard names, so you may have more than 1 ID for a given name.

---

#### Parameters

- **name** (*str*) – Dashboard name.
- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

**Returns** Dashboard IDs.

**Return type** List[str]

## Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_dashboard_ids(name="...")
```

### awswrangler.quicksight.get\_data\_source\_arn

```
awswrangler.quicksight.get_data_source_arn(name: str, account_id: Optional[str]
                                           = None, boto3_session: Optional[boto3.session.Session] = None) →
                                           str
```

Get QuickSight data source ARN given a name and fails if there is more than 1 ARN associated with this name.

---

**Note:** This function returns a list of ARNs because QuickSight accepts duplicated data source names, so you may have more than 1 ARN for a given name.

---

#### Parameters

- **name** (*str*) – Data source name.
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Data source ARN.

**Return type** str

## Examples

```
>>> import awswrangler as wr
>>> arn = wr.quicksight.get_data_source_arn("...")
```

### awswrangler.quicksight.get\_data\_source\_arns

```
awswrangler.quicksight.get_data_source_arns(name: str, account_id: Optional[str]
                                              = None, boto3_session: Optional[boto3.session.Session] = None)
                                              → List[str]
```

Get QuickSight Data source ARNs given a name.

---

**Note:** This function returns a list of ARNs because QuickSight accepts duplicated data source names, so you may have more than 1 ARN for a given name.

---

#### Parameters

- **name** (*str*) – Data source name.
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.

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

**Returns** Data source ARNs.

**Return type** List[str]

### Examples

```
>>> import awswrangler as wr
>>> arns = wr.quicksight.get_data_source_arns(name="...")
```

### awswrangler.quicksight.get\_data\_source\_id

```
awswrangler.quicksight.get_data_source_id(name: str, account_id: Optional[str]
                                           = None, boto3_session: Optional[boto3.session.Session] = None) →
                                           str
```

Get QuickSight data source ID given a name and fails if there is more than 1 ID associated with this name.

#### Parameters

- **name** (*str*) – Data source name.
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Dataset ID.

**Return type** str

### Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_data_source_id(name="...")
```

### awswrangler.quicksight.get\_data\_source\_ids

```
awswrangler.quicksight.get_data_source_ids(name: str, account_id: Optional[str]
                                           = None, boto3_session: Optional[boto3.session.Session] = None) →
                                           List[str]
```

Get QuickSight data source IDs given a name.

---

**Note:** This function returns a list of ID because Quicksight accepts duplicated data source names, so you may have more than 1 ID for a given name.

---

#### Parameters

- **name** (*str*) – Data source name.

- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Data source IDs.

**Return type** List[str]

### Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_data_source_ids(name="...")
```

### awswrangler.quicksight.get\_dataset\_id

`awswrangler.quicksight.get_dataset_id(name: str, account_id: Optional[str] = None, boto3_session: Optional[boto3.session.Session] = None) → str`

Get QuickSight Dataset ID given a name and fails if there is more than 1 ID associated with this name.

#### Parameters

- **name** (*str*) – Dataset name.
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Dataset ID.

**Return type** str

### Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_dataset_id(name="...")
```

### awswrangler.quicksight.get\_dataset\_ids

`awswrangler.quicksight.get_dataset_ids(name: str, account_id: Optional[str] = None, boto3_session: Optional[boto3.session.Session] = None) → List[str]`

Get QuickSight dataset IDs given a name.

---

**Note:** This function returns a list of ID because QuickSight accepts duplicated datasets names, so you may have more than 1 ID for a given name.

---

#### Parameters

- **name** (*str*) – Dataset name.

- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Datasets IDs.

**Return type** List[str]

### Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_dataset_ids(name="...")
```

### awswrangler.quicksight.get\_template\_id

`awswrangler.quicksight.get_template_id(name: str, account_id: Optional[str] = None, boto3_session: Optional[boto3.session.Session] = None) → str`

Get QuickSight template ID given a name and fails if there is more than 1 ID associated with this name.

#### Parameters

- **name** (*str*) – Template name.
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Template ID.

**Return type** str

### Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_template_id(name="...")
```

### awswrangler.quicksight.get\_template\_ids

`awswrangler.quicksight.get_template_ids(name: str, account_id: Optional[str] = None, boto3_session: Optional[boto3.session.Session] = None) → List[str]`

Get QuickSight template IDs given a name.

---

**Note:** This function returns a list of ID because Quicksight accepts duplicated templates names, so you may have more than 1 ID for a given name.

---

#### Parameters

- **name** (*str*) – Template name.

- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Template IDs.

**Return type** List[str]

### Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_template_ids(name="...")
```

### awswrangler.quicksight.list\_dashboards

**awswrangler.quicksight.list\_dashboards** (*account\_id: Optional[str] = None*, *boto3\_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List dashboards in an AWS account.

#### Parameters

- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Dashboards.

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

### Examples

```
>>> import awswrangler as wr
>>> dashboards = wr.quicksight.list_dashboards()
```

### awswrangler.quicksight.list\_data\_sources

**awswrangler.quicksight.list\_data\_sources** (*account\_id: Optional[str] = None*, *boto3\_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List all QuickSight Data sources summaries.

#### Parameters

- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Data sources summaries.

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

## Examples

```
>>> import awswrangler as wr
>>> sources = wr.quicksight.list_data_sources()
```

## awswrangler.quicksight.list\_datasets

`awswrangler.quicksight.list_datasets` (*account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List all QuickSight datasets summaries.

### Parameters

- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Datasets summaries.

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

## Examples

```
>>> import awswrangler as wr
>>> datasets = wr.quicksight.list_datasets()
```

## awswrangler.quicksight.list\_groups

`awswrangler.quicksight.list_groups` (*namespace: str = 'default', account\_id: Optional[str] = None, boto3\_session: Optional[boto3.session.Session] = None*) → List[Dict[str, Any]]

List all QuickSight Groups.

### Parameters

- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Groups.

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



## Examples

```
>>> import awswrangler as wr
>>> groups = wr.quicksight.list_groups()
```

### awswrangler.quicksight.list\_group\_memberships

```
awswrangler.quicksight.list_group_memberships (group_name: str, namespace: str = 'default', account_id: Optional[str] = None, boto3_session: Optional[boto3.session.Session] = None)
→ List[Dict[str, Any]]
```

List all QuickSight Group memberships.

#### Parameters

- **group\_name** (*str*) – The name of the group that you want to see a membership list of.
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Group memberships.

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

## Examples

```
>>> import awswrangler as wr
>>> memberships = wr.quicksight.list_group_memberships()
```

### awswrangler.quicksight.list\_iam\_policy\_assignments

```
awswrangler.quicksight.list_iam_policy_assignments (status: Optional[str] = None, namespace: str = 'default', account_id: Optional[str] = None, boto3_session: Optional[boto3.session.Session] = None)
→ List[Dict[str, Any]]
```

List IAM policy assignments in the current Amazon QuickSight account.

#### Parameters

- **status** (*str*, *optional*) – The status of the assignments. 'ENABLED'|'DRAFT'|'DISABLED'
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** IAM policy assignments.

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

### Examples

```
>>> import awswrangler as wr
>>> assigns = wr.quicksight.list_iam_policy_assignments()
```

### awswrangler.quicksight.list\_iam\_policy\_assignments\_for\_user

```
awswrangler.quicksight.list_iam_policy_assignments_for_user(user_name: str,
                                                             namespace: str
                                                             = 'default', ac-
                                                             count_id: Op-
                                                             tional[str] = None,
                                                             boto3_session: Op-
                                                             tional[boto3.session.Session]
                                                             = None) →
                                                             List[Dict[str, Any]]
```

List all the IAM policy assignments.

Including the Amazon Resource Names (ARNs) for the IAM policies assigned to the specified user and group or groups that the user belongs to.

#### Parameters

- **user\_name** (*str*) – The name of the user.
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** IAM policy assignments.

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

### Examples

```
>>> import awswrangler as wr
>>> assigns = wr.quicksight.list_iam_policy_assignments_for_user()
```

**awswrangler.quicksight.list\_ingestions**

```
awswrangler.quicksight.list_ingestions(dataset_name: Optional[str] = None, dataset_id:
                                         Optional[str] = None, account_id: Op-
                                         tional[str] = None, boto3_session: Op-
                                         tional[boto3.session.Session] = None) →
                                         List[Dict[str, Any]]
```

List the history of SPICE ingestions for a dataset.

**Parameters**

- **dataset\_name** (*str, optional*) – Dataset name.
- **dataset\_id** (*str, optional*) – The ID of the dataset used in the ingestion.
- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** IAM policy assignments.

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

**Examples**

```
>>> import awswrangler as wr
>>> ingestions = wr.quicksight.list_ingestions()
```

**awswrangler.quicksight.list\_templates**

```
awswrangler.quicksight.list_templates(account_id: Optional[str] = None, boto3_session:
                                       Optional[boto3.session.Session] = None) →
                                       List[Dict[str, Any]]
```

List all QuickSight templates.

**Parameters**

- **account\_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Templates summaries.

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

## Examples

```
>>> import awswrangler as wr
>>> templates = wr.quicksight.list_templates()
```

### awswrangler.quicksight.list\_users

`awswrangler.quicksight.list_users(namespace: str = 'default', account_id: Optional[str] = None, boto3_session: Optional[boto3.session.Session] = None) → List[Dict[str, Any]]`

Return a list of all of the Amazon QuickSight users belonging to this account.

#### Parameters

- **namespace** (*str*) – The namespace. Currently, you should set this to default.
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Groups.

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

## Examples

```
>>> import awswrangler as wr
>>> users = wr.quicksight.list_users()
```

### awswrangler.quicksight.list\_user\_groups

`awswrangler.quicksight.list_user_groups(user_name: str, namespace: str = 'default', account_id: Optional[str] = None, boto3_session: Optional[boto3.session.Session] = None) → List[Dict[str, Any]]`

List the Amazon QuickSight groups that an Amazon QuickSight user is a member of.

#### Parameters

- **user\_name** (*str:*) – The Amazon QuickSight user name that you want to list group memberships for.
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account\_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3\_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3\_session receive None.

**Returns** Groups.

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

## Examples

```
>>> import awswrangler as wr
>>> groups = wr.quicksight.list_user_groups()
```

### 1.3.11 AWS STS

<code>get_account_id([boto3_session])</code>	Get Account ID.
<code>get_current_identity_arn([boto3_session])</code>	Get current user/role ARN.
<code>get_current_identity_name([boto3_session])</code>	Get current user/role name.

#### awswrangler.sts.get\_account\_id

`awswrangler.sts.get_account_id(boto3_session: Optional[boto3.session.Session] = None) → str`  
Get Account ID.

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

**Returns** Account ID.

**Return type** str

## Examples

```
>>> import awswrangler as wr
>>> account_id = wr.sts.get_account_id()
```

#### awswrangler.sts.get\_current\_identity\_arn

`awswrangler.sts.get_current_identity_arn(boto3_session: Optional[boto3.session.Session] = None) → str`  
Get current user/role ARN.

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

**Returns** User/role ARN.

**Return type** str

## Examples

```
>>> import awswrangler as wr
>>> arn = wr.sts.get_current_identity_arn()
```

**aws wrangler.sts.get\_current\_identity\_name**

`aws wrangler.sts.get_current_identity_name (boto3_session: Optional[boto3.session.Session] = None) → str`

Get current user/role name.

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

**Returns** User/role name.

**Return type** str

**Examples**

```
>>> import aws wrangler as wr
>>> name = wr.sts.get_current_identity_name()
```

### 1.3.12 Global Configurations

<code>reset([item])</code>	Reset one or all (if <code>None</code> is received) configuration values.
<code>to_pandas()</code>	Load all configurations on a Pandas DataFrame.

**aws wrangler.config.reset**

`config.reset (item: Optional[str] = None) → None`

Reset one or all (if `None` is received) configuration values.

**Parameters** `item` (`str`, *optional*) – Configuration item name.

**Returns** `None`.

**Return type** `None`

## Examples

```
>>> import awswrangler as wr
>>> wr.config.reset("database") # Reset one specific configuration
>>> wr.config.reset() # Reset all
```

## awswrangler.config.to\_pandas

`config.to_pandas()` → `pandas.core.frame.DataFrame`

Load all configurations on a Pandas DataFrame.

**Returns** Configuration DataFrame.

**Return type** `pd.DataFrame`

## Examples

```
>>> import awswrangler as wr
>>> wr.config.to_pandas()
```





## A

`add_column()` (in module `awswrangler.catalog`), 43  
`add_csv_partitions()` (in module `awswrangler.catalog`), 44  
`add_parquet_partitions()` (in module `awswrangler.catalog`), 45

## B

`build_spark_step()` (in module `awswrangler.emr`), 108  
`build_step()` (in module `awswrangler.emr`), 108

## C

`cancel_ingestion()` (in module `awswrangler.quicksight`), 122  
`connect()` (in module `awswrangler.mysql`), 99  
`connect()` (in module `awswrangler.postgresql`), 95  
`connect()` (in module `awswrangler.redshift`), 84  
`connect_temp()` (in module `awswrangler.redshift`), 84  
`copy()` (in module `awswrangler.redshift`), 85  
`copy_from_files()` (in module `awswrangler.redshift`), 87  
`copy_objects()` (in module `awswrangler.s3`), 7  
`create_athena_bucket()` (in module `awswrangler.athena`), 72  
`create_athena_data_source()` (in module `awswrangler.quicksight`), 123  
`create_athena_dataset()` (in module `awswrangler.quicksight`), 124  
`create_cluster()` (in module `awswrangler.emr`), 110  
`create_csv_table()` (in module `awswrangler.catalog`), 46  
`create_database()` (in module `awswrangler.catalog`), 48  
`create_database()` (in module `awswrangler.timestream`), 103  
`create_ingestion()` (in module `awswrangler.quicksight`), 125  
`create_parquet_table()` (in module `awswrangler.catalog`), 49

`create_table()` (in module `awswrangler.timestream`), 104

## D

`databases()` (in module `awswrangler.catalog`), 51  
`delete_all_dashboards()` (in module `awswrangler.quicksight`), 126  
`delete_all_data_sources()` (in module `awswrangler.quicksight`), 126  
`delete_all_datasets()` (in module `awswrangler.quicksight`), 127  
`delete_all_partitions()` (in module `awswrangler.catalog`), 54  
`delete_all_templates()` (in module `awswrangler.quicksight`), 127  
`delete_column()` (in module `awswrangler.catalog`), 51  
`delete_dashboard()` (in module `awswrangler.quicksight`), 128  
`delete_data_source()` (in module `awswrangler.quicksight`), 128  
`delete_database()` (in module `awswrangler.catalog`), 52  
`delete_database()` (in module `awswrangler.timestream`), 105  
`delete_dataset()` (in module `awswrangler.quicksight`), 129  
`delete_objects()` (in module `awswrangler.s3`), 8  
`delete_partitions()` (in module `awswrangler.catalog`), 53  
`delete_table()` (in module `awswrangler.timestream`), 105  
`delete_table_if_exists()` (in module `awswrangler.catalog`), 54  
`delete_template()` (in module `awswrangler.quicksight`), 130  
`describe_dashboard()` (in module `awswrangler.quicksight`), 130  
`describe_data_source()` (in module `awswrangler.quicksight`), 131  
`describe_data_source_permissions()` (in module `awswrangler.quicksight`), 132

`describe_dataset()` (in module `aws wrangler.quicksight`), 132  
`describe_ingestion()` (in module `aws wrangler.quicksight`), 133  
`describe_objects()` (in module `aws wrangler.s3`), 9  
`does_object_exist()` (in module `aws wrangler.s3`), 10  
`does_table_exist()` (in module `aws wrangler.catalog`), 55  
`drop_duplicated_columns()` (in module `aws wrangler.catalog`), 56

## E

`extract_athena_types()` (in module `aws wrangler.catalog`), 56

## G

`get_account_id()` (in module `aws wrangler.sts`), 145  
`get_bucket_region()` (in module `aws wrangler.s3`), 11  
`get_cluster_state()` (in module `aws wrangler.emr`), 114  
`get_columns_comments()` (in module `aws wrangler.catalog`), 57  
`get_csv_partitions()` (in module `aws wrangler.catalog`), 58  
`get_current_identity_arn()` (in module `aws wrangler.sts`), 145  
`get_current_identity_name()` (in module `aws wrangler.sts`), 146  
`get_dashboard_id()` (in module `aws wrangler.quicksight`), 134  
`get_dashboard_ids()` (in module `aws wrangler.quicksight`), 134  
`get_data_source_arn()` (in module `aws wrangler.quicksight`), 135  
`get_data_source_arns()` (in module `aws wrangler.quicksight`), 135  
`get_data_source_id()` (in module `aws wrangler.quicksight`), 136  
`get_data_source_ids()` (in module `aws wrangler.quicksight`), 136  
`get_databases()` (in module `aws wrangler.catalog`), 59  
`get_dataset_id()` (in module `aws wrangler.quicksight`), 137  
`get_dataset_ids()` (in module `aws wrangler.quicksight`), 137  
`get_parquet_partitions()` (in module `aws wrangler.catalog`), 59  
`get_partitions()` (in module `aws wrangler.catalog`), 61  
`get_query_columns_types()` (in module `aws wrangler.athena`), 73

`get_query_execution()` (in module `aws wrangler.athena`), 73  
`get_step_state()` (in module `aws wrangler.emr`), 115  
`get_table_description()` (in module `aws wrangler.catalog`), 62  
`get_table_location()` (in module `aws wrangler.catalog`), 62  
`get_table_number_of_versions()` (in module `aws wrangler.catalog`), 63  
`get_table_parameters()` (in module `aws wrangler.catalog`), 64  
`get_table_types()` (in module `aws wrangler.catalog`), 64  
`get_table_versions()` (in module `aws wrangler.catalog`), 65  
`get_tables()` (in module `aws wrangler.catalog`), 66  
`get_template_id()` (in module `aws wrangler.quicksight`), 138  
`get_template_ids()` (in module `aws wrangler.quicksight`), 138  
`get_work_group()` (in module `aws wrangler.athena`), 74

## L

`list_dashboards()` (in module `aws wrangler.quicksight`), 139  
`list_data_sources()` (in module `aws wrangler.quicksight`), 139  
`list_datasets()` (in module `aws wrangler.quicksight`), 140  
`list_directories()` (in module `aws wrangler.s3`), 11  
`list_group_memberships()` (in module `aws wrangler.quicksight`), 141  
`list_groups()` (in module `aws wrangler.quicksight`), 140  
`list_iam_policy_assignments()` (in module `aws wrangler.quicksight`), 141  
`list_iam_policy_assignments_for_user()` (in module `aws wrangler.quicksight`), 142  
`list_ingestions()` (in module `aws wrangler.quicksight`), 143  
`list_objects()` (in module `aws wrangler.s3`), 12  
`list_templates()` (in module `aws wrangler.quicksight`), 143  
`list_user_groups()` (in module `aws wrangler.quicksight`), 144  
`list_users()` (in module `aws wrangler.quicksight`), 144

## M

`merge_datasets()` (in module `aws wrangler.s3`), 13

## O

`overwrite_table_parameters()` (in module `aws wrangler.catalog`), 67

## Q

`query()` (in module `aws wrangler.timestream`), 106

## R

`read_csv()` (in module `aws wrangler.s3`), 14

`read_fwf()` (in module `aws wrangler.s3`), 16

`read_json()` (in module `aws wrangler.s3`), 18

`read_logs()` (in module `aws wrangler.cloudwatch`), 118

`read_parquet()` (in module `aws wrangler.s3`), 20

`read_parquet_metadata()` (in module `aws wrangler.s3`), 23

`read_parquet_table()` (in module `aws wrangler.s3`), 24

`read_sql_query()` (in module `aws wrangler.athena`), 74

`read_sql_query()` (in module `aws wrangler.mysql`), 100

`read_sql_query()` (in module `aws wrangler.postgresql`), 96

`read_sql_query()` (in module `aws wrangler.redshift`), 89

`read_sql_table()` (in module `aws wrangler.athena`), 77

`read_sql_table()` (in module `aws wrangler.mysql`), 101

`read_sql_table()` (in module `aws wrangler.postgresql`), 97

`read_sql_table()` (in module `aws wrangler.redshift`), 90

`repair_table()` (in module `aws wrangler.athena`), 80

`reset()` (`aws wrangler.config` method), 146

`run_query()` (in module `aws wrangler.cloudwatch`), 119

## S

`sanitize_column_name()` (in module `aws wrangler.catalog`), 68

`sanitize_dataframe_columns_names()` (in module `aws wrangler.catalog`), 68

`sanitize_table_name()` (in module `aws wrangler.catalog`), 69

`search_tables()` (in module `aws wrangler.catalog`), 69

`size_objects()` (in module `aws wrangler.s3`), 26

`start_query()` (in module `aws wrangler.cloudwatch`), 120

`start_query_execution()` (in module `aws wrangler.athena`), 81

`stop_query_execution()` (in module `aws wrangler.athena`), 82

`store_parquet_metadata()` (in module `aws wrangler.s3`), 27

`submit_ecr_credentials_refresh()` (in module `aws wrangler.emr`), 115

`submit_spark_step()` (in module `aws wrangler.emr`), 116

`submit_step()` (in module `aws wrangler.emr`), 117

`submit_steps()` (in module `aws wrangler.emr`), 117

## T

`table()` (in module `aws wrangler.catalog`), 70

`tables()` (in module `aws wrangler.catalog`), 70

`terminate_cluster()` (in module `aws wrangler.emr`), 118

`to_csv()` (in module `aws wrangler.s3`), 30

`to_json()` (in module `aws wrangler.s3`), 35

`to_pandas()` (`aws wrangler.config` method), 147

`to_parquet()` (in module `aws wrangler.s3`), 36

`to_sql()` (in module `aws wrangler.mysql`), 102

`to_sql()` (in module `aws wrangler.postgresql`), 98

`to_sql()` (in module `aws wrangler.redshift`), 91

## U

`unload()` (in module `aws wrangler.redshift`), 92

`unload_to_files()` (in module `aws wrangler.redshift`), 94

`upsert_table_parameters()` (in module `aws wrangler.catalog`), 71

## W

`wait_objects_exist()` (in module `aws wrangler.s3`), 40

`wait_objects_not_exist()` (in module `aws wrangler.s3`), 41

`wait_query()` (in module `aws wrangler.athena`), 83

`wait_query()` (in module `aws wrangler.cloudwatch`), 121

`write()` (in module `aws wrangler.timestream`), 106