pyspark.sql module

Module Context

Important classes of Spark SQL and DataFrames:

- **pyspark.sql.SparkSession** Main entry point for **DataFrame** and SQL functionality.
- **pyspark.sql.DataFrame** A distributed collection of data grouped into named columns.
- **pyspark.sql.Column** A column expression in a **DataFrame**.
- **pyspark.sql.Row** A row of data in a **DataFrame**.
- **pyspark.sql.GroupedData** Aggregation methods, returned by **DataFrame.groupBy()**.
- **pyspark.sql.DataFrameNaFunctions** Methods for handling missing data (null values).
- **pyspark.sql.DataFrameStatFunctions** Methods for statistics functionality.
- **pyspark.sql.functions** List of built-in functions available for **DataFrame**.
- **pyspark.sql.types** List of data types available.
- **pyspark.sql.Window** For working with window functions.

class pyspark.sql.**SparkSession**(*sparkContext*, *jsparkSession=None*) [source]

The entry point to programming Spark with the Dataset and DataFrame API.

A SparkSession can be used create **DataFrame**, register **DataFrame** as tables, execute SQL over tables, cache tables, and read parquet files. To create a SparkSession, use the following builder pattern:

```
>>> spark = SparkSession.builder \
...     .master("local") \
...     .appName("Word Count") \
...     .config("spark.some.config.option", "some-value") \
...     .getOrCreate()
```

builder

A class attribute having a **Builder** to construct **SparkSession** instances.

class **Builder** [source]

Builder for **SparkSession**.

appName(*name*) [source]

Sets a name for the application, which will be shown in the Spark web UI.

If no application name is set, a randomly generated name will be used.

Parameters:
name – an application name

New in version 2.0.

config(*key=None*, *value=None*, *conf=None*) [source]

Sets a config option. Options set using this method are automatically propagated to both **SparkConf** and **SparkSession**'s own configuration.

For an existing SparkConf, use *conf* parameter.

```
>>> from pyspark.conf import SparkConf
>>> SparkSession.builder.config(conf=SparkConf())
```

```
<pyspark.sql.session...
```

For a (key, value) pair, you can omit parameter names.

```
>>> SparkSession.builder.config("spark.some.config.option",
<pyspark.sql.session...
```

Parameters:

- **key** – a key name string for configuration property
- **value** – a value for configuration property
- **conf** – an instance of **`SparkConf`**

New in version 2.0.

`enableHiveSupport`() [source]

Enables Hive support, including connectivity to a persistent Hive metastore, support for Hive SerDes, and Hive user-defined functions.

New in version 2.0.

`getOrCreate`() [source]

Gets an existing **`SparkSession`** or, if there is no existing one, creates a new one based on the options set in this builder.

This method first checks whether there is a valid global default SparkSession, and if yes, return that one. If no valid global default SparkSession exists, the method creates a new SparkSession and assigns the newly created SparkSession as the global default.

```
>>> s1 = SparkSession.builder.config("k1", "v1").getOrCreat
>>> s1.conf.get("k1") == "v1"
True
```

In case an existing SparkSession is returned, the config options specified in this builder will be applied to the existing SparkSession.

```
>>> s2 = SparkSession.builder.config("k2", "v2").getOrCreat
>>> s1.conf.get("k1") == s2.conf.get("k1")
True
>>> s1.conf.get("k2") == s2.conf.get("k2")
True
```

New in version 2.0.

`master`(*master*) [source]

Sets the Spark master URL to connect to, such as “local” to run locally, “local[4]” to run locally with 4 cores, or “spark://master:7077” to run on a Spark standalone cluster.

Parameters:
master – a url for spark master

New in version 2.0.

property **`catalog`**

Interface through which the user may create, drop, alter or query underlying databases, tables, functions, etc.

Returns:
`Catalog`

New in version 2.0.

property **conf**

Runtime configuration interface for Spark.

This is the interface through which the user can get and set all Spark and Hadoop configurations that are relevant to Spark SQL. When getting the value of a config, this defaults to the value set in the underlying **SparkContext**, if any.

New in version 2.0.

createDataFrame(*data*, *schema=None*, *samplingRatio=None*, *verifySchema=True*) [source]

Creates a **DataFrame** from an **RDD**, a list or a **pandas.DataFrame**.

When `schema` is a list of column names, the type of each column will be inferred from `data`.

When `schema` is `None`, it will try to infer the schema (column names and types) from `data`, which should be an RDD of either **Row**, **namedtuple**, or **dict**.

When `schema` is **pyspark.sql.types.DataType** or a datatype string, it must match the real data, or an exception will be thrown at runtime. If the given schema is not **pyspark.sql.types.StructType**, it will be wrapped into a **pyspark.sql.types.StructType** as its only field, and the field name will be "value". Each record will also be wrapped into a tuple, which can be converted to row later.

If schema inference is needed, `samplingRatio` is used to determined the ratio of rows used for schema inference. The first row will be used if `samplingRatio` is `None`.

Parameters:

- **data** – an RDD of any kind of SQL data representation (e.g. row, tuple, int, boolean, etc.), **list**, or **pandas.DataFrame**.
- **schema** – a **pyspark.sql.types.DataType** or a datatype string or a list of column names, default is `None`. The data type string format equals to **pyspark.sql.types.DataType.simpleString**, except that top level struct type can omit the `struct<>` and atomic types use `typeName()` as their format, e.g. use `byte` instead of `tinyint` for **pyspark.sql.types.ByteType**. We can also use `int` as a short name for `IntegerType`.
- **samplingRatio** – the sample ratio of rows used for inferring
- **verifySchema** – verify data types of every row against schema.

Returns:

DataFrame

Changed in version 2.1: Added verifySchema.

> **Note:** Usage with spark.sql.execution.arrow.pyspark.enabled=True is experimental.

> **Note:** When Arrow optimization is enabled, strings inside Pandas DataFrame in Python 2 are converted into bytes as they are bytes in Python 2 whereas regular strings are left as strings. When using strings in Python 2, use unicode *u""* as Python standard practice.

```
>>> l = [('Alice', 1)]
>>> spark.createDataFrame(l).collect()
[Row(_1='Alice', _2=1)]
>>> spark.createDataFrame(l, ['name', 'age']).collect()
[Row(name='Alice', age=1)]
```

```
>>> d = [{'name': 'Alice', 'age': 1}]
>>> spark.createDataFrame(d).collect()
[Row(age=1, name='Alice')]
```

```
>>> rdd = sc.parallelize(l)
>>> spark.createDataFrame(rdd).collect()
[Row(_1='Alice', _2=1)]
>>> df = spark.createDataFrame(rdd, ['name', 'age'])
>>> df.collect()
[Row(name='Alice', age=1)]
```

```
>>> from pyspark.sql import Row
>>> Person = Row('name', 'age')
>>> person = rdd.map(lambda r: Person(*r))
>>> df2 = spark.createDataFrame(person)
>>> df2.collect()
[Row(name='Alice', age=1)]
```

```
>>> from pyspark.sql.types import *
>>> schema = StructType([
...    StructField("name", StringType(), True),
...    StructField("age", IntegerType(), True)])
>>> df3 = spark.createDataFrame(rdd, schema)
>>> df3.collect()
[Row(name='Alice', age=1)]
```

```
>>> spark.createDataFrame(df.toPandas()).collect()
[Row(name='Alice', age=1)]
>>> spark.createDataFrame(pandas.DataFrame([[1, 2]])).collect()
[Row(0=1, 1=2)]
```

```
>>> spark.createDataFrame(rdd, "a: string, b: int").collect()
[Row(a='Alice', b=1)]
>>> rdd = rdd.map(lambda row: row[1])
>>> spark.createDataFrame(rdd, "int").collect()
[Row(value=1)]
>>> spark.createDataFrame(rdd, "boolean").collect()
Traceback (most recent call last):
    ...
Py4JJavaError: ...
```

New in version 2.0.

classmethod **getActiveSession**() [source]

Returns the active SparkSession for the current thread, returned by the builder. >>> s = SparkSession.getActiveSession() >>> l = [('Alice', 1)] >>> rdd = s.sparkContext.parallelize(l) >>> df = s.createDataFrame(rdd, ['name', 'age']) >>> df.select("age").collect() [Row(age=1)]

New in version 3.0.

newSession() [source]

Returns a new SparkSession as new session, that has separate SQLConf, registered temporary views and UDFs, but shared SparkContext and table cache.

New in version 2.0.

range(*start*, *end=None*, *step=1*, *numPartitions=None*) [source]

Create a **DataFrame** with single **pyspark.sql.types.LongType** column named `id`, containing elements in a range from `start` to `end` (exclusive) with step value `step`.

Parameters:

- **start** – the start value
- **end** – the end value (exclusive)
- **step** – the incremental step (default: 1)

- **numPartitions** – the number of partitions of the DataFrame

Returns:

DataFrame

```
>>> spark.range(1, 7, 2).collect()
[Row(id=1), Row(id=3), Row(id=5)]
```

If only one argument is specified, it will be used as the end value.

```
>>> spark.range(3).collect()
[Row(id=0), Row(id=1), Row(id=2)]
```

New in version 2.0.

property **read**

Returns a **DataFrameReader** that can be used to read data in as a **DataFrame**.

Returns:

DataFrameReader

New in version 2.0.

property **readStream**

Returns a **DataStreamReader** that can be used to read data streams as a streaming **DataFrame**.

Note: Evolving.

Returns:

DataStreamReader

New in version 2.0.

property **sparkContext**

Returns the underlying **SparkContext**.

New in version 2.0.

sql(*sqlQuery*) [source]

Returns a **DataFrame** representing the result of the given query.

Returns:

DataFrame

```
>>> df.createOrReplaceTempView("table1")
>>> df2 = spark.sql("SELECT field1 AS f1, field2 as f2 from tab
>>> df2.collect()
[Row(f1=1, f2='row1'), Row(f1=2, f2='row2'), Row(f1=3, f2='row3
```

New in version 2.0.

stop() [source]

Stop the underlying **SparkContext**.

New in version 2.0.

property **streams**

Returns a **StreamingQueryManager** that allows managing all the **StreamingQuery** instances active on *this* context.

Note: Evolving.

Returns:
StreamingQueryManager

New in version 2.0.

table(*tableName*) [source]

Returns the specified table as a **DataFrame**.

Returns:
DataFrame

```
>>> df.createOrReplaceTempView("table1")
>>> df2 = spark.table("table1")
>>> sorted(df.collect()) == sorted(df2.collect())
True
```

New in version 2.0.

property **udf**

Returns a **UDFRegistration** for UDF registration.

Returns:
UDFRegistration

New in version 2.0.

property **version**

The version of Spark on which this application is running.

New in version 2.0.

class pyspark.sql.**SQLContext**(*sparkContext*, *sparkSession=None*, *jsqlContext=None*) [source]

The entry point for working with structured data (rows and columns) in Spark, in Spark 1.x.

As of Spark 2.0, this is replaced by **SparkSession**. However, we are keeping the class here for backward compatibility.

A SQLContext can be used create **DataFrame**, register **DataFrame** as tables, execute SQL over tables, cache tables, and read parquet files.

Parameters:

- **sparkContext** – The **SparkContext** backing this SQLContext.
- **sparkSession** – The **SparkSession** around which this SQLContext wraps.
- **jsqlContext** – An optional JVM Scala SQLContext. If set, we do not instantiate a new SQLContext in the JVM, instead we make all calls to this object.

cacheTable(*tableName*) [source]

Caches the specified table in-memory.

New in version 1.0.

clearCache() [source]

Removes all cached tables from the in-memory cache.

New in version 1.3.

createDataFrame(*data*, *schema=None*, *samplingRatio=None*, *verifySchema=True*) [source]

Creates a **DataFrame** from an **RDD**, a list or a **pandas.DataFrame**.

When `schema` is a list of column names, the type of each column will be inferred from `data`.

When `schema` is `None`, it will try to infer the schema (column names and types)

from `data`, which should be an RDD of **`Row`**, or **`namedtuple`**, or **`dict`**.

When `schema` is **`pyspark.sql.types.DataType`** or a datatype string it must match the real data, or an exception will be thrown at runtime. If the given schema is not **`pyspark.sql.types.StructType`**, it will be wrapped into a **`pyspark.sql.types.StructType`** as its only field, and the field name will be "value", each record will also be wrapped into a tuple, which can be converted to row later.

If schema inference is needed, `samplingRatio` is used to determined the ratio of rows used for schema inference. The first row will be used if `samplingRatio` is `None`.

Parameters:

- **data** – an RDD of any kind of SQL data representation(e.g. **`Row`**, **`tuple`**, `int`, `boolean`, etc.), or **`list`**, or **`pandas.DataFrame`**.
- **schema** – a **`pyspark.sql.types.DataType`** or a datatype string or a list of column names, default is None. The data type string format equals to **`pyspark.sql.types.DataType.simpleString`**, except that top level struct type can omit the `struct<>` and atomic types use `typeName()` as their format, e.g. use `byte` instead of `tinyint` for **`pyspark.sql.types.ByteType`**. We can also use `int` as a short name for **`pyspark.sql.types.IntegerType`**.
- **samplingRatio** – the sample ratio of rows used for inferring
- **verifySchema** – verify data types of every row against schema.

Returns:

`DataFrame`

Changed in version 2.0: The `schema` parameter can be a **`pyspark.sql.types.DataType`** or a datatype string after 2.0. If it's not a **`pyspark.sql.types.StructType`**, it will be wrapped into a **`pyspark.sql.types.StructType`** and each record will also be wrapped into a tuple.

Changed in version 2.1: Added verifySchema.

```
>>> l = [('Alice', 1)]
>>> sqlContext.createDataFrame(l).collect()
[Row(_1='Alice', _2=1)]
>>> sqlContext.createDataFrame(l, ['name', 'age']).collect()
[Row(name='Alice', age=1)]
```

```
>>> d = [{'name': 'Alice', 'age': 1}]
>>> sqlContext.createDataFrame(d).collect()
[Row(age=1, name='Alice')]
```

```
>>> rdd = sc.parallelize(l)
>>> sqlContext.createDataFrame(rdd).collect()
[Row(_1='Alice', _2=1)]
>>> df = sqlContext.createDataFrame(rdd, ['name', 'age'])
>>> df.collect()
[Row(name='Alice', age=1)]
```

```
>>> from pyspark.sql import Row
>>> Person = Row('name', 'age')
>>> person = rdd.map(lambda r: Person(*r))
>>> df2 = sqlContext.createDataFrame(person)
>>> df2.collect()
[Row(name='Alice', age=1)]
```

```
>>> from pyspark.sql.types import *
>>> schema = StructType([
...    StructField("name", StringType(), True),
...    StructField("age", IntegerType(), True)])
>>> df3 = sqlContext.createDataFrame(rdd, schema)
>>> df3.collect()
[Row(name='Alice', age=1)]
```

```
>>> sqlContext.createDataFrame(df.toPandas()).collect()
[Row(name='Alice', age=1)]
>>> sqlContext.createDataFrame(pandas.DataFrame([[1, 2]])).coll
[Row(0=1, 1=2)]
```

```
>>> sqlContext.createDataFrame(rdd, "a: string, b: int").collec
[Row(a='Alice', b=1)]
>>> rdd = rdd.map(lambda row: row[1])
>>> sqlContext.createDataFrame(rdd, "int").collect()
[Row(value=1)]
>>> sqlContext.createDataFrame(rdd, "boolean").collect()
Traceback (most recent call last):
    ...
Py4JJavaError: ...
```

New in version 1.3.

dropTempTable(*tableName*) [source]

Remove the temporary table from catalog.

```
>>> sqlContext.registerDataFrameAsTable(df, "table1")
>>> sqlContext.dropTempTable("table1")
```

New in version 1.6.

getConf(*key*, *defaultValue=<no value>*) [source]

Returns the value of Spark SQL configuration property for the given key.

If the key is not set and defaultValue is set, return defaultValue. If the key is not set and defaultValue is not set, return the system default value.

```
>>> sqlContext.getConf("spark.sql.shuffle.partitions")
'200'
>>> sqlContext.getConf("spark.sql.shuffle.partitions", u"10")
'10'
>>> sqlContext.setConf("spark.sql.shuffle.partitions", u"50")
>>> sqlContext.getConf("spark.sql.shuffle.partitions", u"10")
'50'
```

New in version 1.3.

classmethod **getOrCreate**(*sc*) [source]

Get the existing SQLContext or create a new one with given SparkContext.

Parameters:
sc – SparkContext

New in version 1.6.

newSession() [source]

Returns a new SQLContext as new session, that has separate SQLConf, registered temporary views and UDFs, but shared SparkContext and table cache.

New in version 1.6.

range(*start*, *end=None*, *step=1*, *numPartitions=None*) [source]

Create a **DataFrame** with single **pyspark.sql.types.LongType** column named

`id`, containing elements in a range from `start` to `end` (exclusive) with step value `step`.

Parameters:

- **start** – the start value
- **end** – the end value (exclusive)
- **step** – the incremental step (default: 1)
- **numPartitions** – the number of partitions of the DataFrame

Returns:

DataFrame

```
>>> sqlContext.range(1, 7, 2).collect()
[Row(id=1), Row(id=3), Row(id=5)]
```

If only one argument is specified, it will be used as the end value.

```
>>> sqlContext.range(3).collect()
[Row(id=0), Row(id=1), Row(id=2)]
```

New in version 1.4.

property **read**

Returns a **DataFrameReader** that can be used to read data in as a **DataFrame**.

Returns:

DataFrameReader

New in version 1.4.

property **readStream**

Returns a **DataStreamReader** that can be used to read data streams as a streaming **DataFrame**.

Note: Evolving.

Returns:

DataStreamReader

```
>>> text_sdf = sqlContext.readStream.text(tempfile.mkdtemp())
>>> text_sdf.isStreaming
True
```

New in version 2.0.

registerDataFrameAsTable(*df*, *tableName*) [source]

Registers the given **DataFrame** as a temporary table in the catalog.

Temporary tables exist only during the lifetime of this instance of **SQLContext**.

```
>>> sqlContext.registerDataFrameAsTable(df, "table1")
```

New in version 1.3.

registerFunction(*name*, *f*, *returnType=None*) [source]

An alias for **spark.udf.register()**. See **pyspark.sql.UDFRegistration.register()**.

Note: Deprecated in 2.3.0. Use **spark.udf.register()** instead.

New in version 1.2.

registerJavaFunction(*name*, *javaClassName*, *returnType=None*) [source]

An alias for **spark.udf.registerJavaFunction()**. See **pyspark.sql.UDFRegistration.registerJavaFunction()**.

> **Note:** Deprecated in 2.3.0. Use **spark.udf.registerJavaFunction()** instead.

New in version 2.1.

setConf(*key*, *value*) [source]

Sets the given Spark SQL configuration property.

New in version 1.3.

sql(*sqlQuery*) [source]

Returns a **DataFrame** representing the result of the given query.

Returns:
DataFrame

```
>>> sqlContext.registerDataFrameAsTable(df, "table1")
>>> df2 = sqlContext.sql("SELECT field1 AS f1, field2 as f2 fro
>>> df2.collect()
[Row(f1=1, f2='row1'), Row(f1=2, f2='row2'), Row(f1=3, f2='row3
```

New in version 1.0.

property **streams**

Returns a **StreamingQueryManager** that allows managing all the **StreamingQuery** StreamingQueries active on *this* context.

> **Note:** Evolving.

New in version 2.0.

table(*tableName*) [source]

Returns the specified table or view as a **DataFrame**.

Returns:
DataFrame

```
>>> sqlContext.registerDataFrameAsTable(df, "table1")
>>> df2 = sqlContext.table("table1")
>>> sorted(df.collect()) == sorted(df2.collect())
True
```

New in version 1.0.

tableNames(*dbName=None*) [source]

Returns a list of names of tables in the database `dbName`.

Parameters:
dbName – string, name of the database to use. Default to the current database.
Returns:
list of table names, in string

```
>>> sqlContext.registerDataFrameAsTable(df, "table1")
>>> "table1" in sqlContext.tableNames()
True
>>> "table1" in sqlContext.tableNames("default")
True
```

New in version 1.3.

tables(*dbName=None*) [source]

Returns a **DataFrame** containing names of tables in the given database.

If `dbName` is not specified, the current database will be used.

The returned DataFrame has two columns: `tableName` and `isTemporary` (a column with **BooleanType** indicating if a table is a temporary one or not).

Parameters:
dbName – string, name of the database to use.
Returns:
DataFrame

```
>>> sqlContext.registerDataFrameAsTable(df, "table1")
>>> df2 = sqlContext.tables()
>>> df2.filter("tableName = 'table1'").first()
Row(database='', tableName='table1', isTemporary=True)
```

New in version 1.3.

property **udf**

Returns a **UDFRegistration** for UDF registration.

Returns:
UDFRegistration

New in version 1.3.1.

uncacheTable(*tableName*) [source]

Removes the specified table from the in-memory cache.

New in version 1.0.

class pyspark.sql.**UDFRegistration**(*sparkSession*) [source]

Wrapper for user-defined function registration. This instance can be accessed by **spark.udf** or **sqlContext.udf**.

New in version 1.3.1.

register(*name*, *f*, *returnType=None*) [source]

Register a Python function (including lambda function) or a user-defined function as a SQL function.

Parameters:
- **name** – name of the user-defined function in SQL statements.
- **f** – a Python function, or a user-defined function. The user-defined function can be either row-at-a-time or vectorized. See **pyspark.sql.functions.udf()** and **pyspark.sql.functions.pandas_udf()**.
- **returnType** – the return type of the registered user-defined function. The value can be either a **pyspark.sql.types.DataType** object or a DDL-formatted type string.

Returns:
a user-defined function.

To register a nondeterministic Python function, users need to first build a nondeterministic user-defined function for the Python function and then register it as a SQL function.

returnType can be optionally specified when *f* is a Python function but not when *f* is a user-defined function. Please see below.

1. When *f* is a Python function:

   returnType defaults to string type and can be optionally specified. The produced object must match the specified type. In this case, this API works as if *register(name, f,*

returnType=StringType().

```
>>> strlen = spark.udf.register("stringLengthSt
>>> spark.sql("SELECT stringLengthString('test'
[Row(stringLengthString(test)='4')]
```

```
>>> spark.sql("SELECT 'foo' AS text").select(st
[Row(stringLengthString(text)='3')]
```

```
>>> from pyspark.sql.types import IntegerType
>>> _ = spark.udf.register("stringLengthInt", l
>>> spark.sql("SELECT stringLengthInt('test')")
[Row(stringLengthInt(test)=4)]
```

```
>>> from pyspark.sql.types import IntegerType
>>> _ = spark.udf.register("stringLengthInt", l
>>> spark.sql("SELECT stringLengthInt('test')")
[Row(stringLengthInt(test)=4)]
```

2. When *f* is a user-defined function:

   Spark uses the return type of the given user-defined function as the return type of the registered user-defined function. *returnType* should not be specified. In this case, this API works as if *register(name, f)*.

```
>>> from pyspark.sql.types import IntegerType
>>> from pyspark.sql.functions import udf
>>> slen = udf(lambda s: len(s), IntegerType())
>>> _ = spark.udf.register("slen", slen)
>>> spark.sql("SELECT slen('test')").collect()
[Row(slen(test)=4)]
```

```
>>> import random
>>> from pyspark.sql.functions import udf
>>> from pyspark.sql.types import IntegerType
>>> random_udf = udf(lambda: random.randint(0,
>>> new_random_udf = spark.udf.register("random_
>>> spark.sql("SELECT random_udf()").collect()
[Row(random_udf()=82)]
```

```
>>> from pyspark.sql.functions import pandas_ud
>>> @pandas_udf("integer", PandasUDFType.SCALAR
... def add_one(x):
...     return x + 1
...
>>> _ = spark.udf.register("add_one", add_one)
>>> spark.sql("SELECT add_one(id) FROM range(3)"
[Row(add_one(id)=1), Row(add_one(id)=2), Row(ad
```

```
>>> @pandas_udf("integer", PandasUDFType.GROUPE
... def sum_udf(v):
...     return v.sum()
...
>>> _ = spark.udf.register("sum_udf", sum_udf)
>>> q = "SELECT sum_udf(v1) FROM VALUES (3, 0),
>>> spark.sql(q).collect()
[Row(sum_udf(v1)=1), Row(sum_udf(v1)=5)]
```

Note: Registration for a user-defined function (case 2.) was added from Spark 2.3.0.

New in version 1.3.1.

registerJavaFunction(*name*, *javaClassName*, *returnType=None*) [source]

Register a Java user-defined function as a SQL function.

In addition to a name and the function itself, the return type can be optionally specified. When the return type is not specified we would infer it via reflection.

Parameters:

- **name** – name of the user-defined function
- **javaClassName** – fully qualified name of java class
- **returnType** – the return type of the registered Java function. The value can be either a **pyspark.sql.types.DataType** object or a DDL-formatted type string.

```
>>> from pyspark.sql.types import IntegerType
>>> spark.udf.registerJavaFunction(
...     "javaStringLength", "test.org.apache.spark.sql.JavaStri
>>> spark.sql("SELECT javaStringLength('test')").collect()
[Row(javaStringLength(test)=4)]
```

```
>>> spark.udf.registerJavaFunction(
...     "javaStringLength2", "test.org.apache.spark.sql.JavaStr
>>> spark.sql("SELECT javaStringLength2('test')").collect()
[Row(javaStringLength2(test)=4)]
```

```
>>> spark.udf.registerJavaFunction(
...     "javaStringLength3", "test.org.apache.spark.sql.JavaStr
>>> spark.sql("SELECT javaStringLength3('test')").collect()
[Row(javaStringLength3(test)=4)]
```

New in version 2.3.

registerJavaUDAF(*name*, *javaClassName*) [source]

Register a Java user-defined aggregate function as a SQL function.

Parameters:

- **name** – name of the user-defined aggregate function
- **javaClassName** – fully qualified name of java class

```
>>> spark.udf.registerJavaUDAF("javaUDAF", "test.org.apache.spa
>>> df = spark.createDataFrame([(1, "a"),(2, "b"), (3, "a")],["
>>> df.createOrReplaceTempView("df")
>>> spark.sql("SELECT name, javaUDAF(id) as avg from df group b
[Row(name='b', avg=102.0), Row(name='a', avg=102.0)]
```

New in version 2.3.

class pyspark.sql.**DataFrame**(*jdf*, *sql_ctx*) [source]

A distributed collection of data grouped into named columns.

A **DataFrame** is equivalent to a relational table in Spark SQL, and can be created using various functions in **SparkSession**:

```
people = spark.read.parquet("...")
```

Once created, it can be manipulated using the various domain-specific-language (DSL) functions defined in: **DataFrame**, **Column**.

To select a column from the **DataFrame**, use the apply method:

```
ageCol = people.age
```

A more concrete example:

```
# To create DataFrame using SparkSession
people = spark.read.parquet("...")
department = spark.read.parquet("...")

people.filter(people.age > 30).join(department, people.deptId == d
  .groupBy(department.name, "gender").agg({"salary": "avg", "age":
```

New in version 1.3.

agg(**exprs*) [source]

Aggregate on the entire **DataFrame** without groups (shorthand for `df.groupBy.agg()`).

```
>>> df.agg({"age": "max"}).collect()
[Row(max(age)=5)]
>>> from pyspark.sql import functions as F
>>> df.agg(F.min(df.age)).collect()
[Row(min(age)=2)]
```

New in version 1.3.

alias(*alias*) [source]

Returns a new **DataFrame** with an alias set.

Parameters:
alias – string, an alias name to be set for the **DataFrame**.

```
>>> from pyspark.sql.functions import *
>>> df_as1 = df.alias("df_as1")
>>> df_as2 = df.alias("df_as2")
>>> joined_df = df_as1.join(df_as2, col("df_as1.name") == col("
>>> joined_df.select("df_as1.name", "df_as2.name", "df_as2.age"
[Row(name='Bob', name='Bob', age=5), Row(name='Alice', name='Al
```

New in version 1.3.

approxQuantile(*col*, *probabilities*, *relativeError*) [source]

Calculates the approximate quantiles of numerical columns of a **DataFrame**.

The result of this algorithm has the following deterministic bound: If the **DataFrame** has N elements and if we request the quantile at probability *p* up to error *err*, then the algorithm will return a sample *x* from the **DataFrame** so that the *exact* rank of *x* is close to (p * N). More precisely,

floor((p - err) * N) <= rank(x) <= ceil((p + err) * N).

This method implements a variation of the Greenwald-Khanna algorithm (with some speed optimizations). The algorithm was first present in [[://doi.org/10.1145/375663.375670 Space-efficient Online Computation of Quantile Summaries]] by Greenwald and Khanna.

Note that null values will be ignored in numerical columns before calculation. For columns only containing null values, an empty list is returned.

Parameters:

- **col** – str, list. Can be a single column name, or a list of names for multiple columns.
- **probabilities** – a list of quantile probabilities Each number must belong to [0, 1]. For example 0 is the minimum, 0.5 is the median, 1 is the maximum.
- **relativeError** – The relative target precision to achieve (>= 0). If set to zero,

the exact quantiles are computed, which could be very expensive. Note that values greater than 1 are accepted but give the same result as 1.

Returns:
the approximate quantiles at the given probabilities. If the input *col* is a string, the output is a list of floats. If the input *col* is a list or tuple of strings, the output is also a list, but each element in it is a list of floats, i.e., the output is a list of list of floats.

Changed in version 2.2: Added support for multiple columns.

New in version 2.0.

`cache`() [source]

Persists the **`DataFrame`** with the default storage level (*MEMORY_AND_DISK*).

> **Note:** The default storage level has changed to *MEMORY_AND_DISK* to match Scala in 2.0.

New in version 1.3.

`checkpoint`(*eager=True*) [source]

Returns a checkpointed version of this Dataset. Checkpointing can be used to truncate the logical plan of this **`DataFrame`**, which is especially useful in iterative algorithms where the plan may grow exponentially. It will be saved to files inside the checkpoint directory set with **`SparkContext.setCheckpointDir()`**.

Parameters:
eager – Whether to checkpoint this **`DataFrame`** immediately

> **Note:** Experimental

New in version 2.1.

`coalesce`(*numPartitions*) [source]

Returns a new **`DataFrame`** that has exactly *numPartitions* partitions.

Parameters:
numPartitions – int, to specify the target number of partitions

Similar to coalesce defined on an **`RDD`**, this operation results in a narrow dependency, e.g. if you go from 1000 partitions to 100 partitions, there will not be a shuffle, instead each of the 100 new partitions will claim 10 of the current partitions. If a larger number of partitions is requested, it will stay at the current number of partitions.

However, if you're doing a drastic coalesce, e.g. to numPartitions = 1, this may result in your computation taking place on fewer nodes than you like (e.g. one node in the case of numPartitions = 1). To avoid this, you can call repartition(). This will add a shuffle step, but means the current upstream partitions will be executed in parallel (per whatever the current partitioning is).

```
>>> df.coalesce(1).rdd.getNumPartitions()
1
```

New in version 1.4.

`colRegex`(*colName*) [source]

Selects column based on the column name specified as a regex and returns it as **`Column`**.

Parameters:
colName – string, column name specified as a regex.

```
>>> df = spark.createDataFrame([("a", 1), ("b", 2), ("c",  3)],
>>> df.select(df.colRegex("`(Col1)?+.+`")).show()
+----+
|Col2|
+----+
|   1|
|   2|
|   3|
+----+
```

New in version 2.3.

collect() [source]

Returns all the records as a list of **Row**.

```
>>> df.collect()
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
```

New in version 1.3.

property **columns**

Returns all column names as a list.

```
>>> df.columns
['age', 'name']
```

New in version 1.3.

corr(*col1*, *col2*, *method=None*) [source]

Calculates the correlation of two columns of a **DataFrame** as a double value. Currently only supports the Pearson Correlation Coefficient. **DataFrame.corr()** and **DataFrameStatFunctions.corr()** are aliases of each other.

Parameters:

- **col1** – The name of the first column
- **col2** – The name of the second column
- **method** – The correlation method. Currently only supports “pearson”

New in version 1.4.

count() [source]

Returns the number of rows in this **DataFrame**.

```
>>> df.count()
2
```

New in version 1.3.

cov(*col1*, *col2*) [source]

Calculate the sample covariance for the given columns, specified by their names, as a double value. **DataFrame.cov()** and **DataFrameStatFunctions.cov()** are aliases.

Parameters:

- **col1** – The name of the first column
- **col2** – The name of the second column

New in version 1.4.

createGlobalTempView(*name*) [source]

Creates a global temporary view with this **DataFrame**.

The lifetime of this temporary view is tied to this Spark application. throws **TempTableAlreadyExistsException**, if the view name already exists in the

catalog.

```
>>> df.createGlobalTempView("people")
>>> df2 = spark.sql("select * from global_temp.people")
>>> sorted(df.collect()) == sorted(df2.collect())
True
>>> df.createGlobalTempView("people")
Traceback (most recent call last):
...
AnalysisException: u"Temporary table 'people' already exists;"
>>> spark.catalog.dropGlobalTempView("people")
```

New in version 2.1.

createOrReplaceGlobalTempView(*name*) [source]

Creates or replaces a global temporary view using the given name.

The lifetime of this temporary view is tied to this Spark application.

```
>>> df.createOrReplaceGlobalTempView("people")
>>> df2 = df.filter(df.age > 3)
>>> df2.createOrReplaceGlobalTempView("people")
>>> df3 = spark.sql("select * from global_temp.people")
>>> sorted(df3.collect()) == sorted(df2.collect())
True
>>> spark.catalog.dropGlobalTempView("people")
```

New in version 2.2.

createOrReplaceTempView(*name*) [source]

Creates or replaces a local temporary view with this **DataFrame**.

The lifetime of this temporary table is tied to the **SparkSession** that was used to create this **DataFrame**.

```
>>> df.createOrReplaceTempView("people")
>>> df2 = df.filter(df.age > 3)
>>> df2.createOrReplaceTempView("people")
>>> df3 = spark.sql("select * from people")
>>> sorted(df3.collect()) == sorted(df2.collect())
True
>>> spark.catalog.dropTempView("people")
```

New in version 2.0.

createTempView(*name*) [source]

Creates a local temporary view with this **DataFrame**.

The lifetime of this temporary table is tied to the **SparkSession** that was used to create this **DataFrame**. throws **TempTableAlreadyExistsException**, if the view name already exists in the catalog.

```
>>> df.createTempView("people")
>>> df2 = spark.sql("select * from people")
>>> sorted(df.collect()) == sorted(df2.collect())
True
>>> df.createTempView("people")
Traceback (most recent call last):
...
AnalysisException: u"Temporary table 'people' already exists;"
>>> spark.catalog.dropTempView("people")
```

New in version 2.0.

crossJoin(*other*) [source]

Returns the cartesian product with another **DataFrame**.

Parameters:
other – Right side of the cartesian product.

```
>>> df.select("age", "name").collect()
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
>>> df2.select("name", "height").collect()
[Row(name='Tom', height=80), Row(name='Bob', height=85)]
>>> df.crossJoin(df2.select("height")).select("age", "name", "h
[Row(age=2, name='Alice', height=80), Row(age=2, name='Alice',
 Row(age=5, name='Bob', height=80), Row(age=5, name='Bob', heig
```

New in version 2.1.

crosstab(*col1*, *col2*) [source]

Computes a pair-wise frequency table of the given columns. Also known as a contingency table. The number of distinct values for each column should be less than 1e4. At most 1e6 non-zero pair frequencies will be returned. The first column of each row will be the distinct values of *col1* and the column names will be the distinct values of *col2*. The name of the first column will be *$col1_$col2*. Pairs that have no occurrences will have zero as their counts. **DataFrame.crosstab()** and **DataFrameStatFunctions.crosstab()** are aliases.

Parameters:
- **col1** – The name of the first column. Distinct items will make the first item of each row.
- **col2** – The name of the second column. Distinct items will make the column names of the **DataFrame**.

New in version 1.4.

cube(**cols*) [source]

Create a multi-dimensional cube for the current **DataFrame** using the specified columns, so we can run aggregations on them.

```
>>> df.cube("name", df.age).count().orderBy("name", "age").sho
+-----+----+-----+
| name| age|count|
+-----+----+-----+
| null|null|    2|
| null|   2|    1|
| null|   5|    1|
|Alice|null|    1|
|Alice|   2|    1|
|  Bob|null|    1|
|  Bob|   5|    1|
+-----+----+-----+
```

New in version 1.4.

describe(**cols*) [source]

Computes basic statistics for numeric and string columns.

This include count, mean, stddev, min, and max. If no columns are given, this function computes statistics for all numerical or string columns.

Note: This function is meant for exploratory data analysis, as we make no guarantee about the backward compatibility of the schema of the resulting **DataFrame**.

```
>>> df.describe(['age']).show()
+-------+------------------+
|summary|               age|
+-------+------------------+
|  count|                 2|
|   mean|               3.5|
| stddev|2.1213203435596424|
|    min|                 2|
|    max|                 5|
+-------+------------------+
>>> df.describe().show()
+-------+------------------+-----+
|summary|               age| name|
+-------+------------------+-----+
|  count|                 2|    2|
|   mean|               3.5| null|
| stddev|2.1213203435596424| null|
|    min|                 2|Alice|
|    max|                 5|  Bob|
+-------+------------------+-----+
```

Use summary for expanded statistics and control over which statistics to compute.

New in version 1.3.1.

distinct() [source]

Returns a new **DataFrame** containing the distinct rows in this **DataFrame**.

```
>>> df.distinct().count()
2
```

New in version 1.3.

drop(**cols*) [source]

Returns a new **DataFrame** that drops the specified column. This is a no-op if schema doesn't contain the given column name(s).

Parameters:
cols – a string name of the column to drop, or a **Column** to drop, or a list of string name of the columns to drop.

```
>>> df.drop('age').collect()
[Row(name='Alice'), Row(name='Bob')]
```

```
>>> df.drop(df.age).collect()
[Row(name='Alice'), Row(name='Bob')]
```

```
>>> df.join(df2, df.name == df2.name, 'inner').drop(df.name).co
[Row(age=5, height=85, name='Bob')]
```

```
>>> df.join(df2, df.name == df2.name, 'inner').drop(df2.name).c
[Row(age=5, name='Bob', height=85)]
```

```
>>> df.join(df2, 'name', 'inner').drop('age', 'height').collect
[Row(name='Bob')]
```

New in version 1.4.

dropDuplicates(*subset=None*) [source]

Return a new **DataFrame** with duplicate rows removed, optionally only considering certain columns.

For a static batch **DataFrame**, it just drops duplicate rows. For a streaming **DataFrame**, it will keep all data across triggers as intermediate state to drop duplicates rows. You can use **withWatermark()** to limit how late the duplicate data can be and system will accordingly limit the state. In addition, too late data older than watermark will be dropped to avoid any possibility of duplicates.

drop_duplicates() is an alias for **dropDuplicates()**.

```
>>> from pyspark.sql import Row
>>> df = sc.parallelize([ \
...     Row(name='Alice', age=5, height=80), \
...     Row(name='Alice', age=5, height=80), \
...     Row(name='Alice', age=10, height=80)]).toDF()
>>> df.dropDuplicates().show()
+---+------+-----+
|age|height| name|
+---+------+-----+
|  5|    80|Alice|
| 10|    80|Alice|
+---+------+-----+
```

```
>>> df.dropDuplicates(['name', 'height']).show()
+---+------+-----+
|age|height| name|
+---+------+-----+
|  5|    80|Alice|
+---+------+-----+
```

New in version 1.4.

drop_duplicates(*subset=None*)

drop_duplicates() is an alias for **dropDuplicates()**.

New in version 1.4.

dropna(*how='any'*, *thresh=None*, *subset=None*) [source]

Returns a new **DataFrame** omitting rows with null values. **DataFrame.dropna()** and **DataFrameNaFunctions.drop()** are aliases of each other.

Parameters:

- **how** – 'any' or 'all'. If 'any', drop a row if it contains any nulls. If 'all', drop a row only if all its values are null.
- **thresh** – int, default None If specified, drop rows that have less than *thresh* non-null values. This overwrites the *how* parameter.
- **subset** – optional list of column names to consider.

```
>>> df4.na.drop().show()
+---+------+-----+
|age|height| name|
+---+------+-----+
| 10|    80|Alice|
+---+------+-----+
```

New in version 1.3.1.

property **dtypes**

Returns all column names and their data types as a list.

```
>>> df.dtypes
[('age', 'int'), ('name', 'string')]
```

New in version 1.3.

exceptAll(*other*) [source]

Return a new **DataFrame** containing rows in this **DataFrame** but not in another **DataFrame** while preserving duplicates.

This is equivalent to *EXCEPT ALL* in SQL.

```
>>> df1 = spark.createDataFrame(
...         [("a", 1), ("a", 1), ("a", 1), ("a", 2), ("b",  3),
>>> df2 = spark.createDataFrame([("a", 1), ("b", 3)], ["C1", "C
```

```
>>> df1.exceptAll(df2).show()
+---+---+
| C1| C2|
+---+---+
|  a|  1|
|  a|  1|
|  a|  2|
|  c|  4|
+---+---+
```

Also as standard in SQL, this function resolves columns by position (not by name).

New in version 2.4.

explain(*extended=None*, *mode=None*) [source]

Prints the (logical and physical) plans to the console for debugging purpose.

Parameters:

- **extended** – boolean, default `False`. If `False`, prints only the physical plan.
- **mode** –
  specifies the expected output format of plans.
  - `simple`: Print only a physical plan.
  - `extended`: Print both logical and physical plans.
  - `codegen`: Print a physical plan and generated codes if they are available.
  - `cost`: Print a logical plan and statistics if they are available.
  - `formatted`: Split explain output into two sections: a physical plan outline and node details.

```
>>> df.explain()
== Physical Plan ==
*(1) Scan ExistingRDD[age#0,name#1]
```

```
>>> df.explain(True)
== Parsed Logical Plan ==
...
== Analyzed Logical Plan ==
...
== Optimized Logical Plan ==
...
== Physical Plan ==
...
```

```
>>> df.explain(mode="formatted")
== Physical Plan ==
* Scan ExistingRDD (1)
(1) Scan ExistingRDD [codegen id : 1]
Output: [age#0, name#1]
```

Changed in version 3.0.0: Added optional argument *mode* to specify the expected output format of plans.

New in version 1.3.

fillna(*value*, *subset=None*) [source]

Replace null values, alias for `na.fill()`. **`DataFrame.fillna()`** and **`DataFrameNaFunctions.fill()`** are aliases of each other.

Parameters:

- **value** – int, long, float, string, bool or dict. Value to replace null values with. If

the value is a dict, then *subset* is ignored and *value* must be a mapping from column name (string) to replacement value. The replacement value must be an int, long, float, boolean, or string.

- **subset** – optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if *value* is a string, and subset contains a non-string column, then the non-string column is simply ignored.

```
>>> df4.na.fill(50).show()
+---+------+-----+
|age|height| name|
+---+------+-----+
| 10|    80|Alice|
|  5|    50|  Bob|
| 50|    50|  Tom|
| 50|    50| null|
+---+------+-----+
```

```
>>> df5.na.fill(False).show()
+----+-------+-----+
| age|   name|  spy|
+----+-------+-----+
|  10|  Alice|false|
|   5|    Bob|false|
|null|Mallory| true|
+----+-------+-----+
```

```
>>> df4.na.fill({'age': 50, 'name': 'unknown'}).show()
+---+------+-------+
|age|height|   name|
+---+------+-------+
| 10|    80|  Alice|
|  5|  null|    Bob|
| 50|  null|    Tom|
| 50|  null|unknown|
+---+------+-------+
```

New in version 1.3.1.

filter(*condition*) [source]

Filters rows using the given condition.

where() is an alias for **filter()**.

Parameters:
condition – a **Column** of **types.BooleanType** or a string of SQL expression.

```
>>> df.filter(df.age > 3).collect()
[Row(age=5, name='Bob')]
>>> df.where(df.age == 2).collect()
[Row(age=2, name='Alice')]
```

```
>>> df.filter("age > 3").collect()
[Row(age=5, name='Bob')]
>>> df.where("age = 2").collect()
[Row(age=2, name='Alice')]
```

New in version 1.3.

first() [source]

Returns the first row as a **Row**.

```
>>> df.first()
Row(age=2, name='Alice')
```

New in version 1.3.

foreach(*f*) [source]

Applies the `f` function to all **`Row`** of this **`DataFrame`**.

This is a shorthand for `df.rdd.foreach()`.

```
>>> def f(person):
...     print(person.name)
>>> df.foreach(f)
```

New in version 1.3.

`foreachPartition`(*f*) [source]

Applies the `f` function to each partition of this **`DataFrame`**.

This a shorthand for `df.rdd.foreachPartition()`.

```
>>> def f(people):
...     for person in people:
...         print(person.name)
>>> df.foreachPartition(f)
```

New in version 1.3.

`freqItems`(*cols*, *support=None*) [source]

Finding frequent items for columns, possibly with false positives. Using the frequent element count algorithm described in ":://doi.org/10.1145/762471.762473, proposed by Karp, Schenker, and Papadimitriou". **`DataFrame.freqItems()`** and **`DataFrameStatFunctions.freqItems()`** are aliases.

> **Note:** This function is meant for exploratory data analysis, as we make no guarantee about the backward compatibility of the schema of the resulting **`DataFrame`**.

Parameters:

- **cols** – Names of the columns to calculate frequent items for as a list or tuple of strings.
- **support** – The frequency with which to consider an item 'frequent'. Default is 1%. The support must be greater than 1e-4.

New in version 1.4.

`groupBy`(**cols*) [source]

Groups the **`DataFrame`** using the specified columns, so we can run aggregation on them. See **`GroupedData`** for all the available aggregate functions.

`groupby()` is an alias for **`groupBy()`**.

Parameters:
cols – list of columns to group by. Each element should be a column name (string) or an expression (**`Column`**).

```
>>> df.groupBy().avg().collect()
[Row(avg(age)=3.5)]
>>> sorted(df.groupBy('name').agg({'age': 'mean'}).collect())
[Row(name='Alice', avg(age)=2.0), Row(name='Bob', avg(age)=5.0)
>>> sorted(df.groupBy(df.name).avg().collect())
[Row(name='Alice', avg(age)=2.0), Row(name='Bob', avg(age)=5.0)
>>> sorted(df.groupBy(['name', df.age]).count().collect())
[Row(name='Alice', age=2, count=1), Row(name='Bob', age=5, coun
```

New in version 1.3.

`groupby`(**cols*)

`groupby()` is an alias for **`groupBy()`**.

New in version 1.4.

head(*n=None*) [source]

Returns the first `n` rows.

> **Note:** This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

Parameters:
n – int, default 1. Number of rows to return.
Returns:
If n is greater than 1, return a list of **Row**. If n is 1, return a single Row.

```
>>> df.head()
Row(age=2, name='Alice')
>>> df.head(1)
[Row(age=2, name='Alice')]
```

New in version 1.3.

hint(*name*, **parameters*) [source]

Specifies some hint on the current **DataFrame**.

Parameters:
- **name** – A name of the hint.
- **parameters** – Optional parameters.

Returns:
DataFrame

```
>>> df.join(df2.hint("broadcast"), "name").show()
+----+---+------+
|name|age|height|
+----+---+------+
| Bob|  5|    85|
+----+---+------+
```

New in version 2.2.

intersect(*other*) [source]

Return a new **DataFrame** containing rows only in both this **DataFrame** and another **DataFrame**.

This is equivalent to *INTERSECT* in SQL.

New in version 1.3.

intersectAll(*other*) [source]

Return a new **DataFrame** containing rows in both this **DataFrame** and another **DataFrame** while preserving duplicates.

This is equivalent to *INTERSECT ALL* in SQL. >>> df1 = spark.createDataFrame([("a", 1), ("a", 1), ("b", 3), ("c", 4)], ["C1", "C2"]) >>> df2 = spark.createDataFrame([("a", 1), ("a", 1), ("b", 3)], ["C1", "C2"])

```
>>> df1.intersectAll(df2).sort("C1", "C2").show()
+---+---+
| C1| C2|
+---+---+
|  a|  1|
|  a|  1|
|  b|  3|
+---+---+
```

Also as standard in SQL, this function resolves columns by position (not by name).

New in version 2.4.

isLocal() [source]

Returns `True` if the **collect()** and **take()** methods can be run locally (without any Spark executors).

New in version 1.3.

property **isStreaming**

Returns `True` if this **Dataset** contains one or more sources that continuously return data as it arrives. A **Dataset** that reads data from a streaming source must be executed as a **StreamingQuery** using the **start()** method in **DataStreamWriter**. Methods that return a single answer, (e.g., **count()** or **collect()**) will throw an **AnalysisException** when there is a streaming source present.

> **Note:** Evolving

New in version 2.0.

join(*other*, *on=None*, *how=None*) [source]

Joins with another **DataFrame**, using the given join expression.

Parameters:

- **other** – Right side of the join
- **on** – a string for the join column name, a list of column names, a join expression (Column), or a list of Columns. If *on* is a string or a list of strings indicating the name of the join column(s), the column(s) must exist on both sides, and this performs an equi-join.
- **how** – str, default `inner`. Must be one of: `inner`, `cross`, `outer`, `full`, `fullouter`, `full_outer`, `left`, `leftouter`, `left_outer`, `right`, `rightouter`, `right_outer`, `semi`, `leftsemi`, `left_semi`, `anti`, `leftanti` and `left_anti`.

The following performs a full outer join between `df1` and `df2`.

```
>>> df.join(df2, df.name == df2.name, 'outer').select(df.name,
[Row(name=None, height=80), Row(name='Bob', height=85), Row(nam
```

```
>>> df.join(df2, 'name', 'outer').select('name', 'height').coll
[Row(name='Tom', height=80), Row(name='Bob', height=85), Row(na
```

```
>>> cond = [df.name == df3.name, df.age == df3.age]
>>> df.join(df3, cond, 'outer').select(df.name, df3.age).collec
[Row(name='Alice', age=2), Row(name='Bob', age=5)]
```

```
>>> df.join(df2, 'name').select(df.name, df2.height).collect()
[Row(name='Bob', height=85)]
```

```
>>> df.join(df4, ['name', 'age']).select(df.name, df.age).colle
[Row(name='Bob', age=5)]
```

New in version 1.3.

limit(*num*) [source]

Limits the result count to the number specified.

```
>>> df.limit(1).collect()
[Row(age=2, name='Alice')]
>>> df.limit(0).collect()
[]
```

New in version 1.3.

localCheckpoint(*eager=True*) [source]

Returns a locally checkpointed version of this Dataset. Checkpointing can be used to truncate the logical plan of this **DataFrame**, which is especially useful in iterative algorithms where the plan may grow exponentially. Local checkpoints are stored in the executors using the caching subsystem and therefore they are not reliable.

Parameters:
eager – Whether to checkpoint this **DataFrame** immediately

Note: Experimental

New in version 2.3.

mapInPandas(*udf*) [source]

Maps an iterator of batches in the current **DataFrame** using a Pandas user-defined function and returns the result as a **DataFrame**.

The user-defined function should take an iterator of *pandas.DataFrame*s and return another iterator of *pandas.DataFrame*s. All columns are passed together as an iterator of *pandas.DataFrame*s to the user-defined function and the returned iterator of *pandas.DataFrame*s are combined as a **DataFrame**. Each *pandas.DataFrame* size can be controlled by *spark.sql.execution.arrow.maxRecordsPerBatch*. Its schema must match the returnType of the Pandas user-defined function.

Parameters:
udf – A function object returned by **pyspark.sql.functions.pandas_udf()**

```
>>> from pyspark.sql.functions import pandas_udf, PandasUDFType
>>> df = spark.createDataFrame([(1, 21), (2, 30)],
...                            ("id", "age"))
>>> @pandas_udf(df.schema, PandasUDFType.MAP_ITER)
... def filter_func(batch_iter):
...     for pdf in batch_iter:
...         yield pdf[pdf.id == 1]
>>> df.mapInPandas(filter_func).show()
+---+---+
| id|age|
+---+---+
|  1| 21|
+---+---+
```

See also: **pyspark.sql.functions.pandas_udf()**

property **na**

Returns a **DataFrameNaFunctions** for handling missing values.

New in version 1.3.1.

orderBy(**cols*, ***kwargs*)

Returns a new **DataFrame** sorted by the specified column(s).

Parameters:
- **cols** – list of **Column** or column names to sort by.
- **ascending** – boolean or list of boolean (default `True`). Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, length of the list must equal length of the *cols*.

```
>>> df.sort(df.age.desc()).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.sort("age", ascending=False).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.orderBy(df.age.desc()).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> from pyspark.sql.functions import *
>>> df.sort(asc("age")).collect()
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
>>> df.orderBy(desc("age"), "name").collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.orderBy(["age", "name"], ascending=[0, 1]).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
```

New in version 1.3.

persist(*storageLevel=StorageLevel(True, True, False, False, 1)*) [source]

Sets the storage level to persist the contents of the **DataFrame** across operations after the first time it is computed. This can only be used to assign a new storage level if the **DataFrame** does not have a storage level set yet. If no storage level is specified defaults to (*MEMORY_AND_DISK*).

> **Note:** The default storage level has changed to *MEMORY_AND_DISK* to match Scala in 2.0.

New in version 1.3.

printSchema() [source]

Prints out the schema in the tree format.

```
>>> df.printSchema()
root
 |-- age: integer (nullable = true)
 |-- name: string (nullable = true)
```

New in version 1.3.

randomSplit(*weights*, *seed=None*) [source]

Randomly splits this **DataFrame** with the provided weights.

Parameters:

- **weights** – list of doubles as weights with which to split the **DataFrame**. Weights will be normalized if they don't sum up to 1.0.
- **seed** – The seed for sampling.

```
>>> splits = df4.randomSplit([1.0, 2.0], 24)
>>> splits[0].count()
2
```

```
>>> splits[1].count()
2
```

New in version 1.4.

property **rdd**

Returns the content as an **pyspark.RDD** of **Row**.

New in version 1.3.

repartition(*numPartitions*, **cols*) [source]

Returns a new **DataFrame** partitioned by the given partitioning expressions. The resulting **DataFrame** is hash partitioned.

Parameters:

numPartitions – can be an int to specify the target number of partitions or a

Column. If it is a Column, it will be used as the first partitioning column. If not specified, the default number of partitions is used.

Changed in version 1.6: Added optional arguments to specify the partitioning columns. Also made numPartitions optional if partitioning columns are specified.

```
>>> df.repartition(10).rdd.getNumPartitions()
10
>>> data = df.union(df).repartition("age")
>>> data.show()
+---+-----+
|age| name|
+---+-----+
|  5|  Bob|
|  5|  Bob|
|  2|Alice|
|  2|Alice|
+---+-----+
>>> data = data.repartition(7, "age")
>>> data.show()
+---+-----+
|age| name|
+---+-----+
|  2|Alice|
|  5|  Bob|
|  2|Alice|
|  5|  Bob|
+---+-----+
>>> data.rdd.getNumPartitions()
7
>>> data = data.repartition("name", "age")
>>> data.show()
+---+-----+
|age| name|
+---+-----+
|  5|  Bob|
|  5|  Bob|
|  2|Alice|
|  2|Alice|
+---+-----+
```

New in version 1.3.

repartitionByRange(*numPartitions*, **cols*) [source]

Returns a new **DataFrame** partitioned by the given partitioning expressions. The resulting **DataFrame** is range partitioned.

Parameters:
numPartitions – can be an int to specify the target number of partitions or a Column. If it is a Column, it will be used as the first partitioning column. If not specified, the default number of partitions is used.

At least one partition-by expression must be specified. When no explicit sort order is specified, “ascending nulls first” is assumed.

Note that due to performance reasons this method uses sampling to estimate the ranges. Hence, the output may not be consistent, since sampling can return different values. The sample size can be controlled by the config *spark.sql.execution.rangeExchange.sampleSizePerPartition*.

```
>>> df.repartitionByRange(2, "age").rdd.getNumPartitions()
2
>>> df.show()
+---+-----+
|age| name|
+---+-----+
|  2|Alice|
|  5|  Bob|
+---+-----+
>>> df.repartitionByRange(1, "age").rdd.getNumPartitions()
1
>>> data = df.repartitionByRange("age")
>>> df.show()
+---+-----+
|age| name|
+---+-----+
|  2|Alice|
|  5|  Bob|
+---+-----+
```

New in version 2.4.0.

replace(*to_replace*, *value=<no value>*, *subset=None*) [source]

Returns a new **DataFrame** replacing a value with another value. **DataFrame.replace()** and **DataFrameNaFunctions.replace()** are aliases of each other. Values to_replace and value must have the same type and can only be numerics, booleans, or strings. Value can have None. When replacing, the new value will be cast to the type of the existing column. For numeric replacements all values to be replaced should have unique floating point representation. In case of conflicts (for example with *{42: -1, 42.0: 1}*) and arbitrary replacement will be used.

Parameters:

- **to_replace** – bool, int, long, float, string, list or dict. Value to be replaced. If the value is a dict, then *value* is ignored or can be omitted, and *to_replace* must be a mapping between a value and a replacement.
- **value** – bool, int, long, float, string, list or None. The replacement value must be a bool, int, long, float, string or None. If *value* is a list, *value* should be of the same length and type as *to_replace*. If *value* is a scalar and *to_replace* is a sequence, then *value* is used as a replacement for each item in *to_replace*.
- **subset** – optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if *value* is a string, and subset contains a non-string column, then the non-string column is simply ignored.

```
>>> df4.na.replace(10, 20).show()
+----+------+-----+
| age|height| name|
+----+------+-----+
|  20|    80|Alice|
|   5|  null|  Bob|
|null|  null|  Tom|
|null|  null| null|
+----+------+-----+
```

```
>>> df4.na.replace('Alice', None).show()
+----+------+----+
| age|height|name|
+----+------+----+
|  10|    80|null|
|   5|  null| Bob|
|null|  null| Tom|
|null|  null|null|
+----+------+----+
```

```
>>> df4.na.replace({'Alice': None}).show()
+----+------+----+
| age|height|name|
+----+------+----+
|  10|    80|null|
|   5|  null| Bob|
|null|  null| Tom|
|null|  null|null|
+----+------+----+
```

```
>>> df4.na.replace(['Alice', 'Bob'], ['A', 'B'], 'name').show()
+----+------+----+
| age|height|name|
+----+------+----+
|  10|    80|   A|
|   5|  null|   B|
|null|  null| Tom|
|null|  null|null|
+----+------+----+
```

New in version 1.4.

rollup(**cols*) [source]

Create a multi-dimensional rollup for the current **DataFrame** using the specified columns, so we can run aggregation on them.

```
>>> df.rollup("name", df.age).count().orderBy("name", "age").sh
+-----+----+-----+
| name| age|count|
+-----+----+-----+
| null|null|    2|
|Alice|null|    1|
|Alice|   2|    1|
|  Bob|null|    1|
|  Bob|   5|    1|
+-----+----+-----+
```

New in version 1.4.

sample(*withReplacement=None*, *fraction=None*, *seed=None*) [source]

Returns a sampled subset of this **DataFrame**.

Parameters:

- **withReplacement** – Sample with replacement or not (default `False`).
- **fraction** – Fraction of rows to generate, range [0.0, 1.0].
- **seed** – Seed for sampling (default a random seed).

Note: This is not guaranteed to provide exactly the fraction specified of the total count of the given **DataFrame**.

Note: *fraction* is required and, *withReplacement* and *seed* are optional.

```
>>> df = spark.range(10)
>>> df.sample(0.5, 3).count()
7
>>> df.sample(fraction=0.5, seed=3).count()
7
>>> df.sample(withReplacement=True, fraction=0.5, seed=3).count
1
>>> df.sample(1.0).count()
10
>>> df.sample(fraction=1.0).count()
10
>>> df.sample(False, fraction=1.0).count()
10
```

New in version 1.3.

sampleBy(*col*, *fractions*, *seed=None*) [source]

Returns a stratified sample without replacement based on the fraction given on each stratum.

Parameters:

- **col** – column that defines strata
- **fractions** – sampling fraction for each stratum. If a stratum is not specified, we treat its fraction as zero.
- **seed** – random seed

Returns:

a new **DataFrame** that represents the stratified sample

```
>>> from pyspark.sql.functions import col
>>> dataset = sqlContext.range(0, 100).select((col("id") % 3).a
>>> sampled = dataset.sampleBy("key", fractions={0: 0.1, 1: 0.2
>>> sampled.groupBy("key").count().orderBy("key").show()
+---+-----+
|key|count|
+---+-----+
|  0|    3|
|  1|    6|
+---+-----+
>>> dataset.sampleBy(col("key"), fractions={2: 1.0}, seed=0).co
33
```

Changed in version 3.0: Added sampling by a column of **Column**

New in version 1.5.

property **schema**

Returns the schema of this **DataFrame** as a **pyspark.sql.types.StructType**.

```
>>> df.schema
StructType(List(StructField(age,IntegerType,true),StructField(n
```

New in version 1.3.

select(**cols*) [source]

Projects a set of expressions and returns a new **DataFrame**.

Parameters:

cols – list of column names (string) or expressions (**Column**). If one of the column names is '*', that column is expanded to include all columns in the current **DataFrame**.

```
>>> df.select('*').collect()
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
>>> df.select('name', 'age').collect()
[Row(name='Alice', age=2), Row(name='Bob', age=5)]
>>> df.select(df.name, (df.age + 10).alias('age')).collect()
[Row(name='Alice', age=12), Row(name='Bob', age=15)]
```

New in version 1.3.

selectExpr(**expr*) [source]

Projects a set of SQL expressions and returns a new **DataFrame**.

This is a variant of **select()** that accepts SQL expressions.

```
>>> df.selectExpr("age * 2", "abs(age)").collect()
[Row((age * 2)=4, abs(age)=2), Row((age * 2)=10, abs(age)=5)]
```

New in version 1.3.

show(*n=20*, *truncate=True*, *vertical=False*) [source]

Prints the first `n` rows to the console.

Parameters:

- **n** – Number of rows to show.
- **truncate** – If set to `True`, truncate strings longer than 20 chars by default. If set to a number greater than one, truncates long strings to length `truncate` and align cells right.
- **vertical** – If set to `True`, print output rows vertically (one line per column value).

```
>>> df
DataFrame[age: int, name: string]
>>> df.show()
+---+-----+
|age| name|
+---+-----+
|  2|Alice|
|  5|  Bob|
+---+-----+
>>> df.show(truncate=3)
+---+----+
|age|name|
+---+----+
|  2| Ali|
|  5| Bob|
+---+----+
>>> df.show(vertical=True)
-RECORD 0-----
 age  | 2
 name | Alice
-RECORD 1-----
 age  | 5
 name | Bob
```

New in version 1.3.

sort(**cols*, ***kwargs*) [source]

Returns a new **DataFrame** sorted by the specified column(s).

Parameters:

- **cols** – list of **Column** or column names to sort by.
- **ascending** – boolean or list of boolean (default `True`). Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, length of the list must equal length of the *cols*.

```
>>> df.sort(df.age.desc()).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.sort("age", ascending=False).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.orderBy(df.age.desc()).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> from pyspark.sql.functions import *
>>> df.sort(asc("age")).collect()
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
>>> df.orderBy(desc("age"), "name").collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
>>> df.orderBy(["age", "name"], ascending=[0, 1]).collect()
[Row(age=5, name='Bob'), Row(age=2, name='Alice')]
```

New in version 1.3.

sortWithinPartitions(**cols*, ***kwargs*) [source]

Returns a new **DataFrame** with each partition sorted by the specified column(s).

Parameters:

- **cols** – list of **Column** or column names to sort by.
- **ascending** – boolean or list of boolean (default `True`). Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, length of the list must equal length of the *cols*.

```
>>> df.sortWithinPartitions("age", ascending=False).show()
+---+-----+
|age| name|
+---+-----+
|  2|Alice|
|  5|  Bob|
+---+-----+
```

New in version 1.6.

property **stat**

Returns a **DataFrameStatFunctions** for statistic functions.

New in version 1.4.

property **storageLevel**

Get the **DataFrame**'s current storage level.

```
>>> df.storageLevel
StorageLevel(False, False, False, False, 1)
>>> df.cache().storageLevel
StorageLevel(True, True, False, True, 1)
>>> df2.persist(StorageLevel.DISK_ONLY_2).storageLevel
StorageLevel(True, False, False, False, 2)
```

New in version 2.1.

subtract(*other*) [source]

Return a new **DataFrame** containing rows in this **DataFrame** but not in another **DataFrame**.

This is equivalent to *EXCEPT DISTINCT* in SQL.

New in version 1.3.

summary(**statistics*) [source]

Computes specified statistics for numeric and string columns. Available statistics are: - count - mean - stddev - min - max - arbitrary approximate percentiles specified as a percentage (eg, 75%)

If no statistics are given, this function computes count, mean, stddev, min, approximate quartiles (percentiles at 25%, 50%, and 75%), and max.

Note: This function is meant for exploratory data analysis, as we make no guarantee about the backward compatibility of the schema of the resulting **DataFrame**.

```
>>> df.summary().show()
+-------+------------------+-----+
|summary|               age| name|
+-------+------------------+-----+
|  count|                 2|    2|
|   mean|               3.5| null|
| stddev|2.1213203435596424| null|
|    min|                 2|Alice|
|    25%|                 2| null|
|    50%|                 2| null|
|    75%|                 5| null|
|    max|                 5|  Bob|
+-------+------------------+-----+
```

```
>>> df.summary("count", "min", "25%", "75%", "max").show()
+-------+---+-----+
|summary|age| name|
+-------+---+-----+
|  count|  2|    2|
|    min|  2|Alice|
|    25%|  2| null|
|    75%|  5| null|
|    max|  5|  Bob|
+-------+---+-----+
```

To do a summary for specific columns first select them:

```
>>> df.select("age", "name").summary("count").show()
+-------+---+----+
|summary|age|name|
+-------+---+----+
|  count|  2|   2|
+-------+---+----+
```

See also describe for basic statistics.

New in version 2.3.0.

take(*num*) [source]

Returns the first `num` rows as a **list** of **Row**.

```
>>> df.take(2)
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
```

New in version 1.3.

toDF(**cols*) [source]

Returns a new class:*DataFrame* that with new specified column names

Parameters:
cols – list of new column names (string)

```
>>> df.toDF('f1', 'f2').collect()
[Row(f1=2, f2='Alice'), Row(f1=5, f2='Bob')]
```

toJSON(*use_unicode=True*) [source]

Converts a **DataFrame** into a **RDD** of string.

Each row is turned into a JSON document as one element in the returned RDD.

```
>>> df.toJSON().first()
'{"age":2,"name":"Alice"}'
```

New in version 1.3.

toLocalIterator(*prefetchPartitions=False*) [source]

Returns an iterator that contains all of the rows in this **DataFrame**. The iterator will consume as much memory as the largest partition in this **DataFrame**. With prefetch it may consume up to the memory of the 2 largest partitions.

Parameters:
prefetchPartitions – If Spark should pre-fetch the next partition before it is needed.

```
>>> list(df.toLocalIterator())
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
```

New in version 2.0.

toPandas() [source]

Returns the contents of this **DataFrame** as Pandas `pandas.DataFrame`.

This is only available if Pandas is installed and available.

> **Note:** This method should only be used if the resulting Pandas's **DataFrame** is expected to be small, as all the data is loaded into the driver's memory.

> **Note:** Usage with spark.sql.execution.arrow.pyspark.enabled=True is experimental.

```
>>> df.toPandas()
   age   name
0    2  Alice
1    5    Bob
```

New in version 1.3.

transform(*func*) [source]

Returns a new class:*DataFrame*. Concise syntax for chaining custom transformations.

Parameters:
func – a function that takes and returns a class:*DataFrame*.

```
>>> from pyspark.sql.functions import col
>>> df = spark.createDataFrame([(1, 1.0), (2, 2.0)], ["int", "f
>>> def cast_all_to_int(input_df):
...     return input_df.select([col(col_name).cast("int") for c
>>> def sort_columns_asc(input_df):
...     return input_df.select(*sorted(input_df.columns))
>>> df.transform(cast_all_to_int).transform(sort_columns_asc).s
+-----+---+
|float|int|
+-----+---+
|    1|  1|
|    2|  2|
+-----+---+
```

New in version 3.0.

union(*other*) [source]

Return a new **DataFrame** containing union of rows in this and another **DataFrame**.

This is equivalent to *UNION ALL* in SQL. To do a SQL-style set union (that does deduplication of elements), use this function followed by **distinct()**.

Also as standard in SQL, this function resolves columns by position (not by name).

New in version 2.0.

unionAll(*other*) [source]

Return a new **DataFrame** containing union of rows in this and another **DataFrame**.

This is equivalent to *UNION ALL* in SQL. To do a SQL-style set union (that does deduplication of elements), use this function followed by **distinct()**.

Also as standard in SQL, this function resolves columns by position (not by name).

New in version 1.3.

unionByName(*other*) [source]

Returns a new **DataFrame** containing union of rows in this and another **DataFrame**.

This is different from both *UNION ALL* and *UNION DISTINCT* in SQL. To do a SQL-style set union (that does deduplication of elements), use this function followed by **distinct()**.

The difference between this function and **union()** is that this function resolves columns by name (not by position):

```
>>> df1 = spark.createDataFrame([[1, 2, 3]], ["col0", "col1", "
>>> df2 = spark.createDataFrame([[4, 5, 6]], ["col1", "col2", "
>>> df1.unionByName(df2).show()
+----+----+----+
|col0|col1|col2|
+----+----+----+
|   1|   2|   3|
|   6|   4|   5|
+----+----+----+
```

New in version 2.3.

unpersist(*blocking=False*) [source]

Marks the **DataFrame** as non-persistent, and remove all blocks for it from memory and disk.

> **Note:** *blocking* default has changed to `False` to match Scala in 2.0.

New in version 1.3.

where(*condition*)

where() is an alias for **filter()**.

New in version 1.3.

withColumn(*colName*, *col*) [source]

Returns a new **DataFrame** by adding a column or replacing the existing column that has the same name.

The column expression must be an expression over this **DataFrame**; attempting to add a column from some other **DataFrame** will raise an error.

Parameters:

- **colName** – string, name of the new column.
- **col** – a **Column** expression for the new column.

> **Note:** This method introduces a projection internally. Therefore, calling it multiple times, for instance, via loops in order to add multiple columns can generate big plans which can cause performance issues and even *StackOverflowException*. To avoid this, use **select()** with the multiple columns at once.

```
>>> df.withColumn('age2', df.age + 2).collect()
[Row(age=2, name='Alice', age2=4), Row(age=5, name='Bob', age2=
```

New in version 1.3.

withColumnRenamed(*existing*, *new*) [source]

Returns a new **DataFrame** by renaming an existing column. This is a no-op if schema doesn't contain the given column name.

Parameters:

- **existing** – string, name of the existing column to rename.

- **new** – string, new name of the column.

```
>>> df.withColumnRenamed('age', 'age2').collect()
[Row(age2=2, name='Alice'), Row(age2=5, name='Bob')]
```

New in version 1.3.

withWatermark(*eventTime*, *delayThreshold*) [source]

Defines an event time watermark for this **DataFrame**. A watermark tracks a point in time before which we assume no more late data is going to arrive.

Spark will use this watermark for several purposes:

- To know when a given time window aggregation can be finalized and thus can be emitted when using output modes that do not allow updates.
- To minimize the amount of state that we need to keep for on-going aggregations.

The current watermark is computed by looking at the *MAX(eventTime)* seen across all of the partitions in the query minus a user specified *delayThreshold*. Due to the cost of coordinating this value across partitions, the actual watermark used is only guaranteed to be at least *delayThreshold* behind the actual event time. In some cases we may still process records that arrive more than *delayThreshold* late.

Parameters:
- **eventTime** – the name of the column that contains the event time of the row.
- **delayThreshold** – the minimum delay to wait to data to arrive late, relative to the latest record that has been processed in the form of an interval (e.g. “1 minute” or “5 hours”).

Note: Evolving

```
>>> sdf.select('name', sdf.time.cast('timestamp')).withWatermar
DataFrame[name: string, time: timestamp]
```

New in version 2.1.

property **write**

Interface for saving the content of the non-streaming **DataFrame** out into external storage.

Returns:
DataFrameWriter

New in version 1.4.

property **writeStream**

Interface for saving the content of the streaming **DataFrame** out into external storage.

Note: Evolving.

Returns:
DataStreamWriter

New in version 2.0.

class pyspark.sql.**GroupedData**(*jgd*, *df*) [source]

A set of methods for aggregations on a **DataFrame**, created by **DataFrame.groupBy()**.

New in version 1.3.

agg(**exprs*) [source]

Compute aggregates and returns the result as a **DataFrame**.

The available aggregate functions can be:

1. built-in aggregation functions, such as *avg*, *max*, *min*, *sum*, *count*
2. group aggregate pandas UDFs, created with **pyspark.sql.functions.pandas_udf()**

> **Note:** There is no partial aggregation with group aggregate UDFs, i.e., a full shuffle is required. Also, all the data of a group will be loaded into memory, so the user should be aware of the potential OOM risk if data is skewed and certain groups are too large to fit in memory.

> **See also:** **pyspark.sql.functions.pandas_udf()**

If `exprs` is a single **dict** mapping from string to string, then the key is the column to perform aggregation on, and the value is the aggregate function.

Alternatively, `exprs` can also be a list of aggregate **Column** expressions.

> **Note:** Built-in aggregation functions and group aggregate pandas UDFs cannot be mixed in a single call to this function.

Parameters:
exprs – a dict mapping from column name (string) to aggregate functions (string), or a list of **Column**.

```
>>> gdf = df.groupBy(df.name)
>>> sorted(gdf.agg({"*": "count"}).collect())
[Row(name='Alice', count(1)=1), Row(name='Bob', count(1)=1)]
```

```
>>> from pyspark.sql import functions as F
>>> sorted(gdf.agg(F.min(df.age)).collect())
[Row(name='Alice', min(age)=2), Row(name='Bob', min(age)=5)]
```

```
>>> from pyspark.sql.functions import pandas_udf, PandasUDFType
>>> @pandas_udf('int', PandasUDFType.GROUPED_AGG)
... def min_udf(v):
...     return v.min()
>>> sorted(gdf.agg(min_udf(df.age)).collect())
[Row(name='Alice', min_udf(age)=2), Row(name='Bob', min_udf(age
```

New in version 1.3.

apply(*udf*) [source]

Maps each group of the current **DataFrame** using a pandas udf and returns the result as a *DataFrame*.

The user-defined function should take a *pandas.DataFrame* and return another *pandas.DataFrame*. For each group, all columns are passed together as a *pandas.DataFrame* to the user-function and the returned *pandas.DataFrame* are combined as a **DataFrame**.

The returned *pandas.DataFrame* can be of arbitrary length and its schema must match the returnType of the pandas udf.

> **Note:** This function requires a full shuffle. All the data of a group will be loaded into memory, so the user should be aware of the potential OOM risk if data is skewed and certain groups are too large to fit in memory.

Parameters:
udf – a grouped map user-defined function returned by **pyspark.sql.functions.pandas_udf()**.

```
>>> from pyspark.sql.functions import pandas_udf, PandasUDFType
>>> df = spark.createDataFrame(
...     [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)],
...     ("id", "v"))
>>> @pandas_udf("id long, v double", PandasUDFType.GROUPED_MAP)
... def normalize(pdf):
...     v = pdf.v
...     return pdf.assign(v=(v - v.mean()) / v.std())
>>> df.groupby("id").apply(normalize).show()
+---+-------------------+
| id|                  v|
+---+-------------------+
|  1|-0.7071067811865475|
|  1| 0.7071067811865475|
|  2|-0.8320502943378437|
|  2|-0.2773500981126146|
|  2| 1.1094003924504583|
+---+-------------------+
```

See also: **pyspark.sql.functions.pandas_udf()**

New in version 2.3.

avg(**cols*) [source]

Computes average values for each numeric columns for each group.

mean() is an alias for **avg()**.

Parameters:
cols – list of column names (string). Non-numeric columns are ignored.

```
>>> df.groupBy().avg('age').collect()
[Row(avg(age)=3.5)]
>>> df3.groupBy().avg('age', 'height').collect()
[Row(avg(age)=3.5, avg(height)=82.5)]
```

New in version 1.3.

cogroup(*other*) [source]

Cogroups this group with another group so that we can run cogrouped operations.

See **CoGroupedData** for the operations that can be run.

New in version 3.0.

count() [source]

Counts the number of records for each group.

```
>>> sorted(df.groupBy(df.age).count().collect())
[Row(age=2, count=1), Row(age=5, count=1)]
```

New in version 1.3.

max(**cols*) [source]

Computes the max value for each numeric columns for each group.

```
>>> df.groupBy().max('age').collect()
[Row(max(age)=5)]
>>> df3.groupBy().max('age', 'height').collect()
[Row(max(age)=5, max(height)=85)]
```

New in version 1.3.

mean(**cols*) [source]

Computes average values for each numeric columns for each group.

mean() is an alias for **avg()**.

Parameters:
cols – list of column names (string). Non-numeric columns are ignored.

```
>>> df.groupBy().mean('age').collect()
[Row(avg(age)=3.5)]
>>> df3.groupBy().mean('age', 'height').collect()
[Row(avg(age)=3.5, avg(height)=82.5)]
```

New in version 1.3.

min(**cols*) [source]

Computes the min value for each numeric column for each group.

Parameters:
cols – list of column names (string). Non-numeric columns are ignored.

```
>>> df.groupBy().min('age').collect()
[Row(min(age)=2)]
>>> df3.groupBy().min('age', 'height').collect()
[Row(min(age)=2, min(height)=80)]
```

New in version 1.3.

pivot(*pivot_col*, *values=None*) [source]

Pivots a column of the current **DataFrame** and perform the specified aggregation. There are two versions of pivot function: one that requires the caller to specify the list of distinct values to pivot on, and one that does not. The latter is more concise but less efficient, because Spark needs to first compute the list of distinct values internally.

Parameters:
- **pivot_col** – Name of the column to pivot.
- **values** – List of values that will be translated to columns in the output DataFrame.

Compute the sum of earnings for each year by course with each course as a separate column

```
>>> df4.groupBy("year").pivot("course", ["dotNET", "Java"]).sum
[Row(year=2012, dotNET=15000, Java=20000), Row(year=2013, dotNE
```

Or without specifying column values (less efficient)

```
>>> df4.groupBy("year").pivot("course").sum("earnings").collect
[Row(year=2012, Java=20000, dotNET=15000), Row(year=2013, Java=
>>> df5.groupBy("sales.year").pivot("sales.course").sum("sales.
[Row(year=2012, Java=20000, dotNET=15000), Row(year=2013, Java=
```

New in version 1.6.

sum(**cols*) [source]

Compute the sum for each numeric columns for each group.

Parameters:
cols – list of column names (string). Non-numeric columns are ignored.

```
>>> df.groupBy().sum('age').collect()
[Row(sum(age)=7)]
>>> df3.groupBy().sum('age', 'height').collect()
[Row(sum(age)=7, sum(height)=165)]
```

New in version 1.3.

class pyspark.sql.**Column**(*jc*) [source]

A column in a DataFrame.

Column instances can be created by:

```
# 1. Select a column out of a DataFrame

df.colName
df["colName"]

# 2. Create from an expression
df.colName + 1
1 / df.colName
```

New in version 1.3.

alias(**alias*, ***kwargs*) [source]

Returns this column aliased with a new name or names (in the case of expressions that return more than one column, such as explode).

Parameters:

- **alias** – strings of desired column names (collects all positional arguments passed)
- **metadata** – a dict of information to be stored in `metadata` attribute of the corresponding :class: *StructField* (optional, keyword only argument)

Changed in version 2.2: Added optional `metadata` argument.

```
>>> df.select(df.age.alias("age2")).collect()
[Row(age2=2), Row(age2=5)]
>>> df.select(df.age.alias("age3", metadata={'max': 99})).schem
99
```

New in version 1.3.

asc()

Returns a sort expression based on ascending order of the column.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([('Tom', 80), ('Alice', None)],
>>> df.select(df.name).orderBy(df.name.asc()).collect()
[Row(name='Alice'), Row(name='Tom')]
```

asc_nulls_first()

Returns a sort expression based on ascending order of the column, and null values return before non-null values.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([('Tom', 80), (None, 60), ('Alic
>>> df.select(df.name).orderBy(df.name.asc_nulls_first()).colle
[Row(name=None), Row(name='Alice'), Row(name='Tom')]
```

New in version 2.4.

asc_nulls_last()

Returns a sort expression based on ascending order of the column, and null

values appear after non-null values.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([('Tom', 80), (None, 60), ('Alic
>>> df.select(df.name).orderBy(df.name.asc_nulls_last()).collec
[Row(name='Alice'), Row(name='Tom'), Row(name=None)]
```

New in version 2.4.

astype(*dataType*)

astype() is an alias for **cast()**.

New in version 1.4.

between(*lowerBound*, *upperBound*) [source]

A boolean expression that is evaluated to true if the value of this expression is between the given columns.

```
>>> df.select(df.name, df.age.between(2, 4)).show()
+-----+---------------------------+
| name|((age >= 2) AND (age <= 4))|
+-----+---------------------------+
|Alice|                       true|
|  Bob|                      false|
+-----+---------------------------+
```

New in version 1.3.

bitwiseAND(*other*)

Compute bitwise AND of this expression with another expression.

Parameters:
other – a value or **Column** to calculate bitwise and(&) against this **Column**.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([Row(a=170, b=75)])
>>> df.select(df.a.bitwiseAND(df.b)).collect()
[Row((a & b)=10)]
```

bitwiseOR(*other*)

Compute bitwise OR of this expression with another expression.

Parameters:
other – a value or **Column** to calculate bitwise or(|) against this **Column**.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([Row(a=170, b=75)])
>>> df.select(df.a.bitwiseOR(df.b)).collect()
[Row((a | b)=235)]
```

bitwiseXOR(*other*)

Compute bitwise XOR of this expression with another expression.

Parameters:
other – a value or **Column** to calculate bitwise xor(^) against this **Column**.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([Row(a=170, b=75)])
>>> df.select(df.a.bitwiseXOR(df.b)).collect()
[Row((a ^ b)=225)]
```

cast(*dataType*) [source]

Convert the column into type `dataType`.

```
>>> df.select(df.age.cast("string").alias('ages')).collect()
[Row(ages='2'), Row(ages='5')]
>>> df.select(df.age.cast(StringType()).alias('ages')).collect(
[Row(ages='2'), Row(ages='5')]
```

New in version 1.3.

contains(*other*)

Contains the other element. Returns a boolean **Column** based on a string match.

Parameters:
other – string in line

```
>>> df.filter(df.name.contains('o')).collect()
[Row(age=5, name='Bob')]
```

desc()

Returns a sort expression based on the descending order of the column.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([('Tom', 80), ('Alice', None)],
>>> df.select(df.name).orderBy(df.name.desc()).collect()
[Row(name='Tom'), Row(name='Alice')]
```

desc_nulls_first()

Returns a sort expression based on the descending order of the column, and null values appear before non-null values.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([('Tom', 80), (None, 60), ('Alic
>>> df.select(df.name).orderBy(df.name.desc_nulls_first()).coll
[Row(name=None), Row(name='Tom'), Row(name='Alice')]
```

New in version 2.4.

desc_nulls_last()

Returns a sort expression based on the descending order of the column, and null values appear after non-null values.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([('Tom', 80), (None, 60), ('Alic
>>> df.select(df.name).orderBy(df.name.desc_nulls_last()).colle
[Row(name='Tom'), Row(name='Alice'), Row(name=None)]
```

New in version 2.4.

endswith(*other*)

String ends with. Returns a boolean **Column** based on a string match.

Parameters:
other – string at end of line (do not use a regex *$*)

```
>>> df.filter(df.name.endswith('ice')).collect()
[Row(age=2, name='Alice')]
>>> df.filter(df.name.endswith('ice$')).collect()
[]
```

eqNullSafe(*other*)

Equality test that is safe for null values.

Parameters:
other – a value or `Column`

```
>>> from pyspark.sql import Row
>>> df1 = spark.createDataFrame([
...     Row(id=1, value='foo'),
...     Row(id=2, value=None)
... ])
>>> df1.select(
...     df1['value'] == 'foo',
...     df1['value'].eqNullSafe('foo'),
...     df1['value'].eqNullSafe(None)
... ).show()
+-------------+---------------+----------------+
|(value = foo)|(value <=> foo)|(value <=> NULL)|
+-------------+---------------+----------------+
|         true|           true|           false|
|         null|          false|            true|
+-------------+---------------+----------------+
>>> df2 = spark.createDataFrame([
...     Row(value = 'bar'),
...     Row(value = None)
... ])
>>> df1.join(df2, df1["value"] == df2["value"]).count()
0
>>> df1.join(df2, df1["value"].eqNullSafe(df2["value"])).count(
1
>>> df2 = spark.createDataFrame([
...     Row(id=1, value=float('NaN')),
...     Row(id=2, value=42.0),
...     Row(id=3, value=None)
... ])
>>> df2.select(
...     df2['value'].eqNullSafe(None),
...     df2['value'].eqNullSafe(float('NaN')),
...     df2['value'].eqNullSafe(42.0)
... ).show()
+----------------+---------------+----------------+
|(value <=> NULL)|(value <=> NaN)|(value <=> 42.0)|
+----------------+---------------+----------------+
|           false|           true|           false|
|           false|          false|            true|
|            true|          false|           false|
+----------------+---------------+----------------+
```

Note: Unlike Pandas, PySpark doesn't consider NaN values to be NULL. See the NaN Semantics for details.

New in version 2.3.0.

getField(*name*) [source]

An expression that gets a field by name in a StructField.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([Row(r=Row(a=1, b="b"))])
>>> df.select(df.r.getField("b")).show()
+---+
|r.b|
+---+
|  b|
+---+
>>> df.select(df.r.a).show()
+---+
|r.a|
+---+
|  1|
+---+
```

New in version 1.3.

getItem(*key*) [source]

An expression that gets an item at position `ordinal` out of a list, or gets an item by key out of a dict.

```
>>> df = spark.createDataFrame([([1, 2], {"key": "value"})], ['
>>> df.select(df.l.getItem(0), df.d.getItem("key")).show()
+----+------+
|l[0]|d[key]|
+----+------+
|   1| value|
+----+------+
```

Changed in version 3.0: If *key* is a *Column* object, the indexing operator should be used instead. For example, *map_col.getItem(col('id'))* should be replaced with *map_col[col('id')]*.

New in version 1.3.

isNotNull()

True if the current expression is NOT null.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([Row(name='Tom', height=80), Row
>>> df.filter(df.height.isNotNull()).collect()
[Row(height=80, name='Tom')]
```

isNull()

True if the current expression is null.

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([Row(name='Tom', height=80), Row
>>> df.filter(df.height.isNull()).collect()
[Row(height=None, name='Alice')]
```

isin(**cols*) [source]

A boolean expression that is evaluated to true if the value of this expression is contained by the evaluated values of the arguments.

```
>>> df[df.name.isin("Bob", "Mike")].collect()
[Row(age=5, name='Bob')]
>>> df[df.age.isin([1, 2, 3])].collect()
[Row(age=2, name='Alice')]
```

New in version 1.5.

like(*other*)

SQL like expression. Returns a boolean **Column** based on a SQL LIKE match.

Parameters:
other – a SQL LIKE pattern

See **rlike()** for a regex version

```
>>> df.filter(df.name.like('Al%')).collect()
[Row(age=2, name='Alice')]
```

name(**alias*, ***kwargs*)

name() is an alias for **alias()**.

New in version 2.0.

otherwise(*value*) [source]

Evaluates a list of conditions and returns one of multiple possible result expressions. If **Column.otherwise()** is not invoked, None is returned for unmatched conditions.

See **pyspark.sql.functions.when()** for example usage.

Parameters:
value – a literal value, or a **Column** expression.

```
>>> from pyspark.sql import functions as F
>>> df.select(df.name, F.when(df.age > 3, 1).otherwise(0)).sho
+-----+--------------------------------------+
| name|CASE WHEN (age > 3) THEN 1 ELSE 0 END|
+-----+--------------------------------------+
|Alice|                                     0|
|  Bob|                                     1|
+-----+--------------------------------------+
```

New in version 1.4.

over(*window*) [source]

Define a windowing column.

Parameters:
window – a **WindowSpec**
Returns:
a Column

```
>>> from pyspark.sql import Window
>>> window = Window.partitionBy("name").orderBy("age")
>>> from pyspark.sql.functions import rank, min
>>> df.withColumn("rank", rank().over(window))
+---+-----+----+---+
|age| name|rank|min|
+---+-----+----+---+
|  5|  Bob|   1|  5|
|  2|Alice|   1|  2|
+---+-----+----+---+
```

New in version 1.4.

rlike(*other*)

SQL RLIKE expression (LIKE with Regex). Returns a boolean **Column** based on a regex match.

Parameters:
other – an extended regex expression

```
>>> df.filter(df.name.rlike('ice$')).collect()
[Row(age=2, name='Alice')]
```

startswith(*other*)

String starts with. Returns a boolean **Column** based on a string match.

Parameters:
other – string at start of line (do not use a regex ^)

```
>>> df.filter(df.name.startswith('Al')).collect()
[Row(age=2, name='Alice')]
>>> df.filter(df.name.startswith('^Al')).collect()
[]
```

substr(*startPos*, *length*) [source]

Return a **Column** which is a substring of the column.

Parameters:

- **startPos** – start position (int or Column)
- **length** – length of the substring (int or Column)

```
>>> df.select(df.name.substr(1, 3).alias("col")).collect()
[Row(col='Ali'), Row(col='Bob')]
```

New in version 1.3.

when(*condition*, *value*) [source]

Evaluates a list of conditions and returns one of multiple possible result expressions. If **Column.otherwise()** is not invoked, None is returned for unmatched conditions.

See **pyspark.sql.functions.when()** for example usage.

Parameters:

- **condition** – a boolean **Column** expression.
- **value** – a literal value, or a **Column** expression.

```
>>> from pyspark.sql import functions as F
>>> df.select(df.name, F.when(df.age > 4, 1).when(df.age < 3,
+-----+-------------------------------------------------------
| name|CASE WHEN (age > 4) THEN 1 WHEN (age < 3) THEN -1 ELSE 0
+-----+-------------------------------------------------------
|Alice|
|  Bob|
+-----+-------------------------------------------------------
```

New in version 1.4.

class pyspark.sql.**Catalog**(*sparkSession*) [source]

User-facing catalog API, accessible through *SparkSession.catalog*.

This is a thin wrapper around its Scala implementation org.apache.spark.sql.catalog.Catalog.

cacheTable(*tableName*) [source]

Caches the specified table in-memory.

New in version 2.0.

clearCache() [source]

Removes all cached tables from the in-memory cache.

New in version 2.0.

createTable(*tableName*, *path=None*, *source=None*, *schema=None*, ***options*) [source]

Creates a table based on the dataset in a data source.

It returns the DataFrame associated with the table.

The data source is specified by the `source` and a set of `options`. If `source` is not specified, the default data source configured by `spark.sql.sources.default` will be used. When `path` is specified, an external table is created from the data at the given path. Otherwise a managed table is created.

Optionally, a schema can be provided as the schema of the returned **DataFrame** and created table.

Returns:

DataFrame

New in version 2.2.

currentDatabase() [source]

Returns the current default database in this session.

New in version 2.0.

dropGlobalTempView(*viewName*) [source]

Drops the global temporary view with the given view name in the catalog. If the view has been cached before, then it will also be uncached. Returns true if this view is dropped successfully, false otherwise.

```
>>> spark.createDataFrame([(1, 1)]).createGlobalTempView("my_ta
>>> spark.table("global_temp.my_table").collect()
[Row(_1=1, _2=1)]
>>> spark.catalog.dropGlobalTempView("my_table")
>>> spark.table("global_temp.my_table")
Traceback (most recent call last):
    ...
AnalysisException: ...
```

New in version 2.1.

dropTempView(*viewName*) [source]

Drops the local temporary view with the given view name in the catalog. If the view has been cached before, then it will also be uncached. Returns true if this view is dropped successfully, false otherwise.

Note that, the return type of this method was None in Spark 2.0, but changed to Boolean in Spark 2.1.

```
>>> spark.createDataFrame([(1, 1)]).createTempView("my_table")
>>> spark.table("my_table").collect()
[Row(_1=1, _2=1)]
>>> spark.catalog.dropTempView("my_table")
>>> spark.table("my_table")
Traceback (most recent call last):
    ...
AnalysisException: ...
```

New in version 2.0.

isCached(*tableName*) [source]

Returns true if the table is currently cached in-memory.

New in version 2.0.

listColumns(*tableName*, *dbName=None*) [source]

Returns a list of columns for the given table/view in the specified database.

If no database is specified, the current database is used.

Note: the order of arguments here is different from that of its JVM counterpart because Python does not support method overloading.

New in version 2.0.

listDatabases() [source]

Returns a list of databases available across all sessions.

New in version 2.0.

listFunctions(*dbName=None*) [source]

Returns a list of functions registered in the specified database.

If no database is specified, the current database is used. This includes all temporary functions.

New in version 2.0.

listTables(*dbName=None*) [source]

Returns a list of tables/views in the specified database.

If no database is specified, the current database is used. This includes all temporary views.

New in version 2.0.

recoverPartitions(*tableName*) [source]

Recovers all the partitions of the given table and update the catalog.

Only works with a partitioned table, and not a view.

New in version 2.1.1.

refreshByPath(*path*) [source]

Invalidates and refreshes all the cached data (and the associated metadata) for any DataFrame that contains the given data source path.

New in version 2.2.0.

refreshTable(*tableName*) [source]

Invalidates and refreshes all the cached data and metadata of the given table.

New in version 2.0.

registerFunction(*name*, *f*, *returnType=None*) [source]

An alias for **spark.udf.register()**. See **pyspark.sql.UDFRegistration.register()**.

> **Note:** Deprecated in 2.3.0. Use **spark.udf.register()** instead.

New in version 2.0.

setCurrentDatabase(*dbName*) [source]

Sets the current default database in this session.

New in version 2.0.

uncacheTable(*tableName*) [source]

Removes the specified table from the in-memory cache.

New in version 2.0.

class pyspark.sql.**Row** [source]

A row in **DataFrame**. The fields in it can be accessed:

- like attributes (`row.key`)
- like dictionary values (`row[key]`)

`key in row` will search through row keys.

Row can be used to create a row object by using named arguments, the fields will be sorted by names. It is not allowed to omit a named argument to represent the value is None or missing. This should be explicitly set to None in this case.

```
>>> row = Row(name="Alice", age=11)
>>> row
Row(age=11, name='Alice')
>>> row['name'], row['age']
('Alice', 11)
>>> row.name, row.age
('Alice', 11)
>>> 'name' in row
True
>>> 'wrong_key' in row
False
```

Row also can be used to create another Row like class, then it could be used to create Row objects, such as

```
>>> Person = Row("name", "age")
>>> Person
<Row('name', 'age')>
>>> 'name' in Person
True
>>> 'wrong_key' in Person
False
>>> Person("Alice", 11)
Row(name='Alice', age=11)
```

This form can also be used to create rows as tuple values, i.e. with unnamed fields. Beware that such Row objects have different equality semantics:

```
>>> row1 = Row("Alice", 11)
>>> row2 = Row(name="Alice", age=11)
>>> row1 == row2
False
>>> row3 = Row(a="Alice", b=11)
>>> row1 == row3
True
```

asDict(*recursive=False*) [source]

Return as an dict

Parameters:
recursive – turns the nested Row as dict (default: False).

```
>>> Row(name="Alice", age=11).asDict() == {'name': 'Alice', 'a
True
>>> row = Row(key=1, value=Row(name='a', age=2))
>>> row.asDict() == {'key': 1, 'value': Row(age=2, name='a')}
True
>>> row.asDict(True) == {'key': 1, 'value': {'name': 'a', 'age
True
```

class pyspark.sql.**DataFrameNaFunctions**(*df*) [source]

Functionality for working with missing data in **DataFrame**.

New in version 1.4.

drop(*how='any'*, *thresh=None*, *subset=None*) [source]

Returns a new **DataFrame** omitting rows with null values. **DataFrame.dropna()** and **DataFrameNaFunctions.drop()** are aliases of each other.

Parameters:
- **how** – 'any' or 'all'. If 'any', drop a row if it contains any nulls. If 'all', drop a row only if all its values are null.
- **thresh** – int, default None If specified, drop rows that have less than *thresh* non-null values. This overwrites the *how* parameter.
- **subset** – optional list of column names to consider.

```
>>> df4.na.drop().show()
+---+------+-----+
|age|height| name|
+---+------+-----+
| 10|    80|Alice|
+---+------+-----+
```

New in version 1.3.1.

fill(*value*, *subset=None*) [source]

Replace null values, alias for `na.fill()`. **DataFrame.fillna()** and **DataFrameNaFunctions.fill()** are aliases of each other.

Parameters:
- **value** – int, long, float, string, bool or dict. Value to replace null values with. If the value is a dict, then *subset* is ignored and *value* must be a mapping from

column name (string) to replacement value. The replacement value must be an int, long, float, boolean, or string.

- **subset** – optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if *value* is a string, and subset contains a non-string column, then the non-string column is simply ignored.

```
>>> df4.na.fill(50).show()
+---+------+-----+
|age|height| name|
+---+------+-----+
| 10|    80|Alice|
|  5|    50|  Bob|
| 50|    50|  Tom|
| 50|    50| null|
+---+------+-----+
```

```
>>> df5.na.fill(False).show()
+----+-------+-----+
| age|   name|  spy|
+----+-------+-----+
|  10|  Alice|false|
|   5|    Bob|false|
|null|Mallory| true|
+----+-------+-----+
```

```
>>> df4.na.fill({'age': 50, 'name': 'unknown'}).show()
+---+------+-------+
|age|height|   name|
+---+------+-------+
| 10|    80|  Alice|
|  5|  null|    Bob|
| 50|  null|    Tom|
| 50|  null|unknown|
+---+------+-------+
```

New in version 1.3.1.

replace(*to_replace*, *value=<no value>*, *subset=None*) [source]

Returns a new **DataFrame** replacing a value with another value. **DataFrame.replace()** and **DataFrameNaFunctions.replace()** are aliases of each other. Values to_replace and value must have the same type and can only be numerics, booleans, or strings. Value can have None. When replacing, the new value will be cast to the type of the existing column. For numeric replacements all values to be replaced should have unique floating point representation. In case of conflicts (for example with *{42: -1, 42.0: 1}*) and arbitrary replacement will be used.

Parameters:

- **to_replace** – bool, int, long, float, string, list or dict. Value to be replaced. If the value is a dict, then *value* is ignored or can be omitted, and *to_replace* must be a mapping between a value and a replacement.
- **value** – bool, int, long, float, string, list or None. The replacement value must be a bool, int, long, float, string or None. If *value* is a list, *value* should be of the same length and type as *to_replace*. If *value* is a scalar and *to_replace* is a sequence, then *value* is used as a replacement for each item in *to_replace*.
- **subset** – optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if *value* is a string, and subset contains a non-string column, then the non-string column is simply ignored.

```
>>> df4.na.replace(10, 20).show()
+----+------+-----+
| age|height| name|
+----+------+-----+
|  20|    80|Alice|
|   5|  null|  Bob|
|null|  null|  Tom|
|null|  null| null|
+----+------+-----+
```

```
>>> df4.na.replace('Alice', None).show()
+----+------+----+
| age|height|name|
+----+------+----+
|  10|    80|null|
|   5|  null| Bob|
|null|  null| Tom|
|null|  null|null|
+----+------+----+
```

```
>>> df4.na.replace({'Alice': None}).show()
+----+------+----+
| age|height|name|
+----+------+----+
|  10|    80|null|
|   5|  null| Bob|
|null|  null| Tom|
|null|  null|null|
+----+------+----+
```

```
>>> df4.na.replace(['Alice', 'Bob'], ['A', 'B'], 'name').show()
+----+------+----+
| age|height|name|
+----+------+----+
|  10|    80|   A|
|   5|  null|   B|
|null|  null| Tom|
|null|  null|null|
+----+------+----+
```

New in version 1.4.

class pyspark.sql.**DataFrameStatFunctions**(*df*) [source]

Functionality for statistic functions with **DataFrame**.

New in version 1.4.

approxQuantile(*col*, *probabilities*, *relativeError*) [source]

Calculates the approximate quantiles of numerical columns of a **DataFrame**.

The result of this algorithm has the following deterministic bound: If the **DataFrame** has N elements and if we request the quantile at probability *p* up to error *err*, then the algorithm will return a sample *x* from the **DataFrame** so that the *exact* rank of *x* is close to (p * N). More precisely,

floor((p - err) * N) <= rank(x) <= ceil((p + err) * N).

This method implements a variation of the Greenwald-Khanna algorithm (with some speed optimizations). The algorithm was first present in [[://doi.org/10.1145/375663.375670 Space-efficient Online Computation of Quantile Summaries]] by Greenwald and Khanna.

Note that null values will be ignored in numerical columns before calculation. For columns only containing null values, an empty list is returned.

Parameters:

- **col** – str, list. Can be a single column name, or a list of names for multiple columns.
- **probabilities** – a list of quantile probabilities Each number must belong to [0, 1]. For example 0 is the minimum, 0.5 is the median, 1 is the maximum.

- **relativeError** – The relative target precision to achieve (>= 0). If set to zero, the exact quantiles are computed, which could be very expensive. Note that values greater than 1 are accepted but give the same result as 1.

Returns:
the approximate quantiles at the given probabilities. If the input *col* is a string, the output is a list of floats. If the input *col* is a list or tuple of strings, the output is also a list, but each element in it is a list of floats, i.e., the output is a list of list of floats.

Changed in version 2.2: Added support for multiple columns.

New in version 2.0.

corr(*col1*, *col2*, *method=None*) [source]

Calculates the correlation of two columns of a **DataFrame** as a double value. Currently only supports the Pearson Correlation Coefficient. **DataFrame.corr()** and **DataFrameStatFunctions.corr()** are aliases of each other.

Parameters:
- **col1** – The name of the first column
- **col2** – The name of the second column
- **method** – The correlation method. Currently only supports “pearson”

New in version 1.4.

cov(*col1*, *col2*) [source]

Calculate the sample covariance for the given columns, specified by their names, as a double value. **DataFrame.cov()** and **DataFrameStatFunctions.cov()** are aliases.

Parameters:
- **col1** – The name of the first column
- **col2** – The name of the second column

New in version 1.4.

crosstab(*col1*, *col2*) [source]

Computes a pair-wise frequency table of the given columns. Also known as a contingency table. The number of distinct values for each column should be less than 1e4. At most 1e6 non-zero pair frequencies will be returned. The first column of each row will be the distinct values of *col1* and the column names will be the distinct values of *col2*. The name of the first column will be *$col1_$col2*. Pairs that have no occurrences will have zero as their counts. **DataFrame.crosstab()** and **DataFrameStatFunctions.crosstab()** are aliases.

Parameters:
- **col1** – The name of the first column. Distinct items will make the first item of each row.
- **col2** – The name of the second column. Distinct items will make the column names of the **DataFrame**.

New in version 1.4.

freqItems(*cols*, *support=None*) [source]

Finding frequent items for columns, possibly with false positives. Using the frequent element count algorithm described in “://doi.org/10.1145/762471.762473, proposed by Karp, Schenker, and Papadimitriou”. **DataFrame.freqItems()** and **DataFrameStatFunctions.freqItems()** are aliases.

> **Note:** This function is meant for exploratory data analysis, as we make no guarantee about the backward compatibility of the schema of the resulting **DataFrame**.

Parameters:

- **cols** – Names of the columns to calculate frequent items for as a list or tuple of strings.
- **support** – The frequency with which to consider an item 'frequent'. Default is 1%. The support must be greater than 1e-4.

New in version 1.4.

sampleBy(*col*, *fractions*, *seed=None*) [source]

Returns a stratified sample without replacement based on the fraction given on each stratum.

Parameters:

- **col** – column that defines strata
- **fractions** – sampling fraction for each stratum. If a stratum is not specified, we treat its fraction as zero.
- **seed** – random seed

Returns:

a new **DataFrame** that represents the stratified sample

```
>>> from pyspark.sql.functions import col
>>> dataset = sqlContext.range(0, 100).select((col("id") % 3).a
>>> sampled = dataset.sampleBy("key", fractions={0: 0.1, 1: 0.2
>>> sampled.groupBy("key").count().orderBy("key").show()
+---+-----+
|key|count|
+---+-----+
|  0|    3|
|  1|    6|
+---+-----+
>>> dataset.sampleBy(col("key"), fractions={2: 1.0}, seed=0).co
33
```

Changed in version 3.0: Added sampling by a column of **Column**

New in version 1.5.

class pyspark.sql.**Window** [source]

Utility functions for defining window in DataFrames.

For example:

```
>>> # ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT RO
>>> window = Window.orderBy("date").rowsBetween(Window.unboundedPre
```

```
>>> # PARTITION BY country ORDER BY date RANGE BETWEEN 3 PRECEDING
>>> window = Window.orderBy("date").partitionBy("country").rangeBe
```

Note: When ordering is not defined, an unbounded window frame (rowFrame, unboundedPreceding, unboundedFollowing) is used by default. When ordering is defined, a growing window frame (rangeFrame, unboundedPreceding, currentRow) is used by default.

New in version 1.4.

currentRow = *0*

static **orderBy**(**cols*) [source]

Creates a **WindowSpec** with the ordering defined.

New in version 1.4.

static **partitionBy**(**cols*) [source]

Creates a **WindowSpec** with the partitioning defined.

New in version 1.4.

static **rangeBetween**(*start*, *end*) [source]

Creates a **WindowSpec** with the frame boundaries defined, from *start* (inclusive) to *end* (inclusive).

Both *start* and *end* are relative from the current row. For example, “0” means “current row”, while “-1” means one off before the current row, and “5” means the five off after the current row.

We recommend users use `Window.unboundedPreceding`, `Window.unboundedFollowing`, and `Window.currentRow` to specify special boundary values, rather than using integral values directly.

A range-based boundary is based on the actual value of the ORDER BY expression(s). An offset is used to alter the value of the ORDER BY expression, for instance if the current ORDER BY expression has a value of 10 and the lower bound offset is -3, the resulting lower bound for the current row will be 10 - 3 = 7. This however puts a number of constraints on the ORDER BY expressions: there can be only one expression and this expression must have a numerical data type. An exception can be made when the offset is unbounded, because no value modification is needed, in this case multiple and non-numeric ORDER BY expression are allowed.

```
>>> from pyspark.sql import Window
>>> from pyspark.sql import functions as func
>>> from pyspark.sql import SQLContext
>>> sc = SparkContext.getOrCreate()
>>> sqlContext = SQLContext(sc)
>>> tup = [(1, "a"), (1, "a"), (2, "a"), (1, "b"), (2, "b"), (
>>> df = sqlContext.createDataFrame(tup, ["id", "category"])
>>> window = Window.partitionBy("category").orderBy("id").range
>>> df.withColumn("sum", func.sum("id").over(window)).show()
+---+--------+---+
| id|category|sum|
+---+--------+---+
|  1|       b|  3|
|  2|       b|  5|
|  3|       b|  3|
|  1|       a|  4|
|  1|       a|  4|
|  2|       a|  2|
+---+--------+---+
```

Parameters:

- **start** – boundary start, inclusive. The frame is unbounded if this is `Window.unboundedPreceding`, or any value less than or equal to max(-sys.maxsize, -9223372036854775808).
- **end** – boundary end, inclusive. The frame is unbounded if this is `Window.unboundedFollowing`, or any value greater than or equal to min(sys.maxsize, 9223372036854775807).

New in version 2.1.

static **rowsBetween**(*start*, *end*) [source]

Creates a **WindowSpec** with the frame boundaries defined, from *start* (inclusive) to *end* (inclusive).

Both *start* and *end* are relative positions from the current row. For example, “0” means “current row”, while “-1” means the row before the current row, and “5” means the fifth row after the current row.

We recommend users use `Window.unboundedPreceding`, `Window.unboundedFollowing`, and `Window.currentRow` to specify special boundary values, rather than using integral values directly.

A row based boundary is based on the position of the row within the partition. An

offset indicates the number of rows above or below the current row, the frame for the current row starts or ends. For instance, given a row based sliding frame with a lower bound offset of -1 and a upper bound offset of +2. The frame for row with index 5 would range from index 4 to index 7.

```
>>> from pyspark.sql import Window
>>> from pyspark.sql import functions as func
>>> from pyspark.sql import SQLContext
>>> sc = SparkContext.getOrCreate()
>>> sqlContext = SQLContext(sc)
>>> tup = [(1, "a"), (1, "a"), (2, "a"), (1, "b"), (2, "b"), (
>>> df = sqlContext.createDataFrame(tup, ["id", "category"])
>>> window = Window.partitionBy("category").orderBy("id").rowsB
>>> df.withColumn("sum", func.sum("id").over(window)).show()
+---+--------+---+
| id|category|sum|
+---+--------+---+
|  1|       b|  3|
|  2|       b|  5|
|  3|       b|  3|
|  1|       a|  2|
|  1|       a|  3|
|  2|       a|  2|
+---+--------+---+
```

Parameters:

- **start** – boundary start, inclusive. The frame is unbounded if this is `Window.unboundedPreceding`, or any value less than or equal to -9223372036854775808.
- **end** – boundary end, inclusive. The frame is unbounded if this is `Window.unboundedFollowing`, or any value greater than or equal to 9223372036854775807.

New in version 2.1.

unboundedFollowing = *9223372036854775807*

unboundedPreceding = *-9223372036854775808*

class pyspark.sql.**WindowSpec**(*jspec*) [source]

A window specification that defines the partitioning, ordering, and frame boundaries.

Use the static methods in **Window** to create a **WindowSpec**.

New in version 1.4.

orderBy(**cols*) [source]

Defines the ordering columns in a **WindowSpec**.

Parameters:
cols – names of columns or expressions

New in version 1.4.

partitionBy(**cols*) [source]

Defines the partitioning columns in a **WindowSpec**.

Parameters:
cols – names of columns or expressions

New in version 1.4.

rangeBetween(*start*, *end*) [source]

Defines the frame boundaries, from *start* (inclusive) to *end* (inclusive).

Both *start* and *end* are relative from the current row. For example, “0” means “current row”, while “-1” means one off before the current row, and “5” means the five off after the current row.

We recommend users use `Window.unboundedPreceding`, `Window.unboundedFollowing`, and `Window.currentRow` to specify special boundary values, rather than using integral values directly.

Parameters:

- **start** – boundary start, inclusive. The frame is unbounded if this is `Window.unboundedPreceding`, or any value less than or equal to max(-sys.maxsize, -9223372036854775808).
- **end** – boundary end, inclusive. The frame is unbounded if this is `Window.unboundedFollowing`, or any value greater than or equal to min(sys.maxsize, 9223372036854775807).

New in version 1.4.

rowsBetween(*start*, *end*) [source]

Defines the frame boundaries, from *start* (inclusive) to *end* (inclusive).

Both *start* and *end* are relative positions from the current row. For example, “0” means “current row”, while “-1” means the row before the current row, and “5” means the fifth row after the current row.

We recommend users use `Window.unboundedPreceding`, `Window.unboundedFollowing`, and `Window.currentRow` to specify special boundary values, rather than using integral values directly.

Parameters:

- **start** – boundary start, inclusive. The frame is unbounded if this is `Window.unboundedPreceding`, or any value less than or equal to max(-sys.maxsize, -9223372036854775808).
- **end** – boundary end, inclusive. The frame is unbounded if this is `Window.unboundedFollowing`, or any value greater than or equal to min(sys.maxsize, 9223372036854775807).

New in version 1.4.

class pyspark.sql.**DataFrameReader**(*spark*) [source]

Interface used to load a **DataFrame** from external storage systems (e.g. file systems, key-value stores, etc). Use **spark.read()** to access this.

New in version 1.4.

csv(*path*, *schema=None*, *sep=None*, *encoding=None*, *quote=None*, *escape=None*, *comment=None*, *header=None*, *inferSchema=None*, *ignoreLeadingWhiteSpace=None*, *ignoreTrailingWhiteSpace=None*, *nullValue=None*, *nanValue=None*, *positiveInf=None*, *negativeInf=None*, *dateFormat=None*, *timestampFormat=None*, *maxColumns=None*, *maxCharsPerColumn=None*, *maxMalformedLogPerPartition=None*, *mode=None*, *columnNameOfCorruptRecord=None*, *multiLine=None*, *charToEscapeQuoteEscaping=None*, *samplingRatio=None*, *enforceSchema=None*, *emptyValue=None*, *locale=None*, *lineSep=None*, *recursiveFileLookup=None*) [source]

Loads a CSV file and returns the result as a **DataFrame**.

This function will go through the input once to determine the input schema if `inferSchema` is enabled. To avoid going through the entire data once, disable `inferSchema` option or specify the schema explicitly using `schema`.

Parameters:

- **path** – string, or list of strings, for input path(s), or RDD of Strings storing CSV rows.
- **schema** – an optional **pyspark.sql.types.StructType** for the input schema or a DDL-formatted string (For example `col0 INT, col1 DOUBLE`).
- **sep** – sets a separator (one or more characters) for each field and value. If None is set, it uses the default value, `,`.

- **encoding** – decodes the CSV files by the given encoding type. If None is set, it uses the default value, `UTF-8`.
- **quote** – sets a single character used for escaping quoted values where the separator can be part of the value. If None is set, it uses the default value, `"`. If you would like to turn off quotations, you need to set an empty string.
- **escape** – sets a single character used for escaping quotes inside an already quoted value. If None is set, it uses the default value, `\`.
- **comment** – sets a single character used for skipping lines beginning with this character. By default (None), it is disabled.
- **header** – uses the first line as names of columns. If None is set, it uses the default value, `false`.
- **inferSchema** – infers the input schema automatically from data. It requires one extra pass over the data. If None is set, it uses the default value, `false`.
- **enforceSchema** – If it is set to `true`, the specified or inferred schema will be forcibly applied to datasource files, and headers in CSV files will be ignored. If the option is set to `false`, the schema will be validated against all headers in CSV files or the first header in RDD if the `header` option is set to `true`. Field names in the schema and column names in CSV headers are checked by their positions taking into account `spark.sql.caseSensitive`. If None is set, `true` is used by default. Though the default value is `true`, it is recommended to disable the `enforceSchema` option to avoid incorrect results.
- **ignoreLeadingWhiteSpace** – A flag indicating whether or not leading whitespaces from values being read should be skipped. If None is set, it uses the default value, `false`.
- **ignoreTrailingWhiteSpace** – A flag indicating whether or not trailing whitespaces from values being read should be skipped. If None is set, it uses the default value, `false`.
- **nullValue** – sets the string representation of a null value. If None is set, it uses the default value, empty string. Since 2.0.1, this `nullValue` param applies to all supported types including the string type.
- **nanValue** – sets the string representation of a non-number value. If None is set, it uses the default value, `NaN`.
- **positiveInf** – sets the string representation of a positive infinity value. If None is set, it uses the default value, `Inf`.
- **negativeInf** – sets the string representation of a negative infinity value. If None is set, it uses the default value, `Inf`.
- **dateFormat** – sets the string that indicates a date format. Custom date formats follow the formats at `java.time.format.DateTimeFormatter`. This applies to date type. If None is set, it uses the default value, `uuuu-MM-dd`.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at `java.time.format.DateTimeFormatter`. This applies to timestamp type. If None is set, it uses the default value, `uuuu-MM-dd'T'HH:mm:ss.SSSXXX`.
- **maxColumns** – defines a hard limit of how many columns a record can have. If None is set, it uses the default value, `20480`.
- **maxCharsPerColumn** – defines the maximum number of characters allowed for any given value being read. If None is set, it uses the default value, `-1` meaning unlimited length.
- **maxMalformedLogPerPartition** – this parameter is no longer used since Spark 2.2.0. If specified, it is ignored.
- **mode** –

  allows a mode for dealing with corrupt records during parsing. If None is set, it uses the default value, `PERMISSIVE`. Note that Spark tries to parse only required columns in CSV under column pruning. Therefore, corrupt records can be different based on required set of fields. This behavior can be controlled by `spark.sql.csv.parser.columnPruning.enabled` (enabled by default).

- `PERMISSIVE` : when it meets a corrupted record, puts the malformed string into a field configured by `columnNameOfCorruptRecord`, and sets malformed fields to `null`. To keep corrupt records, an user can set a string type field named `columnNameOfCorruptRecord` in an user-defined schema. If a schema does not have the field, it drops corrupt records during parsing. A record with less/more tokens than schema is not a corrupted record to CSV. When it meets a record having fewer tokens than the length of the schema, sets `null` to extra fields. When the record has more tokens than the length of the schema, it drops extra tokens.
- `DROPMALFORMED` : ignores the whole corrupted records.
- `FAILFAST` : throws an exception when it meets corrupted records.

- **columnNameOfCorruptRecord** – allows renaming the new field having malformed string created by `PERMISSIVE` mode. This overrides `spark.sql.columnNameOfCorruptRecord`. If None is set, it uses the value specified in `spark.sql.columnNameOfCorruptRecord`.
- **multiLine** – parse records, which may span multiple lines. If None is set, it uses the default value, `false`.
- **charToEscapeQuoteEscaping** – sets a single character used for escaping the escape for the quote character. If None is set, the default value is escape character when escape and quote characters are different, `\0` otherwise.
- **samplingRatio** – defines fraction of rows used for schema inferring. If None is set, it uses the default value, `1.0`.
- **emptyValue** – sets the string representation of an empty value. If None is set, it uses the default value, empty string.
- **locale** – sets a locale as language tag in IETF BCP 47 format. If None is set, it uses the default value, `en-US`. For instance, `locale` is used while parsing dates and timestamps.
- **lineSep** – defines the line separator that should be used for parsing. If None is set, it covers all `\\r`, `\\r\\n` and `\\n`. Maximum length is 1 character.
- **recursiveFileLookup** – recursively scan a directory for files. Using this option disables partition discovery.

```
>>> df = spark.read.csv('python/test_support/sql/ages.csv')
>>> df.dtypes
[('_c0', 'string'), ('_c1', 'string')]
>>> rdd = sc.textFile('python/test_support/sql/ages.csv')
>>> df2 = spark.read.csv(rdd)
>>> df2.dtypes
[('_c0', 'string'), ('_c1', 'string')]
```

New in version 2.0.

format(*source*) [source]

Specifies the input data source format.

Parameters:
source – string, name of the data source, e.g. ‘json’, ‘parquet’.

```
>>> df = spark.read.format('json').load('python/test_support/sq
>>> df.dtypes
[('age', 'bigint'), ('name', 'string')]
```

New in version 1.4.

jdbc(*url*, *table*, *column=None*, *lowerBound=None*, *upperBound=None*, *numPartitions=None*, *predicates=None*, *properties=None*) [source]

Construct a **DataFrame** representing the database table named `table` accessible via JDBC URL `url` and connection `properties`.

Partitions of the table will be retrieved in parallel if either `column` or `predicates` is specified. `lowerBound`, ``upperBound` and `numPartitions` is needed when `column` is specified.

If both `column` and `predicates` are specified, `column` will be used.

> **Note:** Don't create too many partitions in parallel on a large cluster; otherwise Spark might crash your external database systems.

Parameters:

- **url** – a JDBC URL of the form `jdbc:subprotocol:subname`
- **table** – the name of the table
- **column** – the name of a column of numeric, date, or timestamp type that will be used for partitioning; if this parameter is specified, then `numPartitions`, `lowerBound` (inclusive), and `upperBound` (exclusive) will form partition strides for generated WHERE clause expressions used to split the column `column` evenly
- **lowerBound** – the minimum value of `column` used to decide partition stride
- **upperBound** – the maximum value of `column` used to decide partition stride
- **numPartitions** – the number of partitions
- **predicates** – a list of expressions suitable for inclusion in WHERE clauses; each one defines one partition of the **`DataFrame`**
- **properties** – a dictionary of JDBC database connection arguments. Normally at least properties "user" and "password" with their corresponding values. For example { 'user' : 'SYSTEM', 'password' : 'mypassword' }

Returns:
a DataFrame

New in version 1.4.

`json`(*path*, *schema=None*, *primitivesAsString=None*, *prefersDecimal=None*, *allowComments=None*, *allowUnquotedFieldNames=None*, *allowSingleQuotes=None*, *allowNumericLeadingZero=None*, *allowBackslashEscapingAnyCharacter=None*, *mode=None*, *columnNameOfCorruptRecord=None*, *dateFormat=None*, *timestampFormat=None*, *multiLine=None*, *allowUnquotedControlChars=None*, *lineSep=None*, *samplingRatio=None*, *dropFieldIfAllNull=None*, *encoding=None*, *locale=None*, *recursiveFileLookup=None*) [source]

Loads JSON files and returns the results as a **`DataFrame`**.

JSON Lines (newline-delimited JSON) is supported by default. For JSON (one record per file), set the `multiLine` parameter to `true`.

If the `schema` parameter is not specified, this function goes through the input once to determine the input schema.

Parameters:

- **path** – string represents path to the JSON dataset, or a list of paths, or RDD of Strings storing JSON objects.
- **schema** – an optional **`pyspark.sql.types.StructType`** for the input schema or a DDL-formatted string (For example `col0 INT, col1 DOUBLE`).
- **primitivesAsString** – infers all primitive values as a string type. If None is set, it uses the default value, `false`.
- **prefersDecimal** – infers all floating-point values as a decimal type. If the values do not fit in decimal, then it infers them as doubles. If None is set, it uses the default value, `false`.
- **allowComments** – ignores Java/C++ style comment in JSON records. If None is set, it uses the default value, `false`.
- **allowUnquotedFieldNames** – allows unquoted JSON field names. If None is set, it uses the default value, `false`.
- **allowSingleQuotes** – allows single quotes in addition to double quotes. If None is set, it uses the default value, `true`.
- **allowNumericLeadingZero** – allows leading zeros in numbers (e.g. 00012). If None is set, it uses the default value, `false`.
- **allowBackslashEscapingAnyCharacter** – allows accepting quoting of all character using backslash quoting mechanism. If None is set, it uses the default value, `false`.

- **mode** –

  allows a mode for dealing with corrupt records during parsing. If None is set, it uses the default value, `PERMISSIVE`.

  - `PERMISSIVE` : when it meets a corrupted record, puts the malformed string into a field configured by `columnNameOfCorruptRecord`, and sets malformed fields to `null`. To keep corrupt records, an user can set a string type field named `columnNameOfCorruptRecord` in an user-defined schema. If a schema does not have the field, it drops corrupt records during parsing. When inferring a schema, it implicitly adds a `columnNameOfCorruptRecord` field in an output schema.
  - `DROPMALFORMED` : ignores the whole corrupted records.
  - `FAILFAST` : throws an exception when it meets corrupted records.
- **columnNameOfCorruptRecord** – allows renaming the new field having malformed string created by `PERMISSIVE` mode. This overrides `spark.sql.columnNameOfCorruptRecord`. If None is set, it uses the value specified in `spark.sql.columnNameOfCorruptRecord`.
- **dateFormat** – sets the string that indicates a date format. Custom date formats follow the formats at `java.time.format.DateTimeFormatter`. This applies to date type. If None is set, it uses the default value, `uuuu-MM-dd`.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at `java.time.format.DateTimeFormatter`. This applies to timestamp type. If None is set, it uses the default value, `uuuu-MM-dd'T'HH:mm:ss.SSSXXX`.
- **multiLine** – parse one record, which may span multiple lines, per file. If None is set, it uses the default value, `false`.
- **allowUnquotedControlChars** – allows JSON Strings to contain unquoted control characters (ASCII characters with value less than 32, including tab and line feed characters) or not.
- **encoding** – allows to forcibly set one of standard basic or extended encoding for the JSON files. For example UTF-16BE, UTF-32LE. If None is set, the encoding of input JSON will be detected automatically when the multiLine option is set to `true`.
- **lineSep** – defines the line separator that should be used for parsing. If None is set, it covers all `\r`, `\r\n` and `\n`.
- **samplingRatio** – defines fraction of input JSON objects used for schema inferring. If None is set, it uses the default value, `1.0`.
- **dropFieldIfAllNull** – whether to ignore column of all null values or empty array/struct during schema inference. If None is set, it uses the default value, `false`.
- **locale** – sets a locale as language tag in IETF BCP 47 format. If None is set, it uses the default value, `en-US`. For instance, `locale` is used while parsing dates and timestamps.
- **recursiveFileLookup** – recursively scan a directory for files. Using this option disables partition discovery.

```
>>> df1 = spark.read.json('python/test_support/sql/people.json'
>>> df1.dtypes
[('age', 'bigint'), ('name', 'string')]
>>> rdd = sc.textFile('python/test_support/sql/people.json')
>>> df2 = spark.read.json(rdd)
>>> df2.dtypes
[('age', 'bigint'), ('name', 'string')]
```

New in version 1.4.

load(*path=None*, *format=None*, *schema=None*, ***options*) [source]

Loads data from a data source and returns it as a :class`DataFrame`.

Parameters:

- **path** – optional string or a list of string for file-system backed data sources.

- **format** – optional string for format of the data source. Default to ‘parquet’.
- **schema** – optional **pyspark.sql.types.StructType** for the input schema or a DDL-formatted string (For example `col0 INT, col1 DOUBLE`).
- **options** – all other string options

```
>>> df = spark.read.format("parquet").load('python/test_support
...     opt1=True, opt2=1, opt3='str')
>>> df.dtypes
[('name', 'string'), ('year', 'int'), ('month', 'int'), ('day',
```

```
>>> df = spark.read.format('json').load(['python/test_support/s
...     'python/test_support/sql/people1.json'])
>>> df.dtypes
[('age', 'bigint'), ('aka', 'string'), ('name', 'string')]
```

New in version 1.4.

option(*key*, *value*) [source]

Adds an input option for the underlying data source.

You can set the following option(s) for reading files:

- `timeZone`: sets the string that indicates a timezone to be used to parse timestamps
  in the JSON/CSV datasources or partition values. If it isn’t set, it uses the default value, session local timezone.
- `pathGlobFilter`: an optional glob pattern to only include files with paths matching
  the pattern. The syntax follows org.apache.hadoop.fs.GlobFilter. It does not change the behavior of partition discovery.

New in version 1.5.

options(***options*) [source]

Adds input options for the underlying data source.

You can set the following option(s) for reading files:

- `timeZone`: sets the string that indicates a timezone to be used to parse timestamps
  in the JSON/CSV datasources or partition values. If it isn’t set, it uses the default value, session local timezone.
- `pathGlobFilter`: an optional glob pattern to only include files with paths matching
  the pattern. The syntax follows org.apache.hadoop.fs.GlobFilter. It does not change the behavior of partition discovery.

New in version 1.4.

orc(*path*, *mergeSchema=None*, *recursiveFileLookup=None*) [source]

Loads ORC files, returning the result as a **DataFrame**.

Parameters:

- **mergeSchema** – sets whether we should merge schemas collected from all ORC part-files. This will override `spark.sql.orc.mergeSchema`. The default value is specified in `spark.sql.orc.mergeSchema`.
- **recursiveFileLookup** – recursively scan a directory for files. Using this option disables partition discovery.

```
>>> df = spark.read.orc('python/test_support/sql/orc_partitione
>>> df.dtypes
[('a', 'bigint'), ('b', 'int'), ('c', 'int')]
```

New in version 1.5.

parquet(**paths*, ***options*) [source]

Loads Parquet files, returning the result as a **DataFrame**.

Parameters:

- **mergeSchema** – sets whether we should merge schemas collected from all Parquet part-files. This will override `spark.sql.parquet.mergeSchema`. The default value is specified in `spark.sql.parquet.mergeSchema`.
- **recursiveFileLookup** – recursively scan a directory for files. Using this option disables partition discovery.

```
>>> df = spark.read.parquet('python/test_support/sql/parquet_pa
>>> df.dtypes
[('name', 'string'), ('year', 'int'), ('month', 'int'), ('day',
```

New in version 1.4.

schema(*schema*) [source]

Specifies the input schema.

Some data sources (e.g. JSON) can infer the input schema automatically from data. By specifying the schema here, the underlying data source can skip the schema inference step, and thus speed up data loading.

Parameters:

schema – a **pyspark.sql.types.StructType** object or a DDL-formatted string (For example `col0 INT, col1 DOUBLE`).

```
>>> s = spark.read.schema("col0 INT, col1 DOUBLE")
```

New in version 1.4.

table(*tableName*) [source]

Returns the specified table as a **DataFrame**.

Parameters:

tableName – string, name of the table.

```
>>> df = spark.read.parquet('python/test_support/sql/parquet_pa
>>> df.createOrReplaceTempView('tmpTable')
>>> spark.read.table('tmpTable').dtypes
[('name', 'string'), ('year', 'int'), ('month', 'int'), ('day',
```

New in version 1.4.

text(*paths*, *wholetext=False*, *lineSep=None*, *recursiveFileLookup=None*) [source]

Loads text files and returns a **DataFrame** whose schema starts with a string column named “value”, and followed by partitioned columns if there are any. The text files must be encoded as UTF-8.

By default, each line in the text file is a new row in the resulting DataFrame.

Parameters:

- **paths** – string, or list of strings, for input path(s).
- **wholetext** – if true, read each file from input path(s) as a single row.
- **lineSep** – defines the line separator that should be used for parsing. If None

is set, it covers all `\r`, `\r\n` and `\n`.

- **recursiveFileLookup** – recursively scan a directory for files. Using this option disables partition discovery.

```
>>> df = spark.read.text('python/test_support/sql/text-test.txt
>>> df.collect()
[Row(value='hello'), Row(value='this')]
>>> df = spark.read.text('python/test_support/sql/text-test.txt
>>> df.collect()
[Row(value='hello\nthis')]
```

New in version 1.6.

class pyspark.sql.**DataFrameWriter**(*df*) [source]

Interface used to write a **DataFrame** to external storage systems (e.g. file systems, key-value stores, etc). Use **DataFrame.write()** to access this.

New in version 1.4.

bucketBy(*numBuckets*, *col*, **cols*) [source]

Buckets the output by the given columns.If specified, the output is laid out on the file system similar to Hive's bucketing scheme.

Parameters:

- **numBuckets** – the number of buckets to save
- **col** – a name of a column, or a list of names.
- **cols** – additional names (optional). If *col* is a list it should be empty.

Note: Applicable for file-based data sources in combination with **DataFrameWriter.saveAsTable()**.

```
>>> (df.write.format('parquet')
...     .bucketBy(100, 'year', 'month')
...     .mode("overwrite")
...     .saveAsTable('bucketed_table'))
```

New in version 2.3.

csv(*path*, *mode=None*, *compression=None*, *sep=None*, *quote=None*, *escape=None*, *header=None*, *nullValue=None*, *escapeQuotes=None*, *quoteAll=None*, *dateFormat=None*, *timestampFormat=None*, *ignoreLeadingWhiteSpace=None*, *ignoreTrailingWhiteSpace=None*, *charToEscapeQuoteEscaping=None*, *encoding=None*, *emptyValue=None*, *lineSep=None*) [source]

Saves the content of the **DataFrame** in CSV format at the specified path.

Parameters:

- **path** – the path in any Hadoop supported file system
- **mode** –
  specifies the behavior of the save operation when data already exists.
  - `append`: Append contents of this **DataFrame** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` or `errorifexists` (default case): Throw an exception if data already exists.
- **compression** – compression codec to use when saving to file. This can be one of the known case-insensitive shorten names (none, bzip2, gzip, lz4, snappy and deflate).
- **sep** – sets a separator (one or more characters) for each field and value. If None is set, it uses the default value, `,`.
- **quote** – sets a single character used for escaping quoted values where the

separator can be part of the value. If None is set, it uses the default value, `"`. If an empty string is set, it uses `u0000` (null character).

- **escape** – sets a single character used for escaping quotes inside an already quoted value. If None is set, it uses the default value, `\`
- **escapeQuotes** – a flag indicating whether values containing quotes should always be enclosed in quotes. If None is set, it uses the default value `true`, escaping all values containing a quote character.
- **quoteAll** – a flag indicating whether all values should always be enclosed in quotes. If None is set, it uses the default value `false`, only escaping values containing a quote character.
- **header** – writes the names of columns as the first line. If None is set, it uses the default value, `false`.
- **nullValue** – sets the string representation of a null value. If None is set, it uses the default value, empty string.
- **dateFormat** – sets the string that indicates a date format. Custom date formats follow the formats at `java.time.format.DateTimeFormatter`. This applies to date type. If None is set, it uses the default value, `uuuu-MM-dd`.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at `java.time.format.DateTimeFormatter`. This applies to timestamp type. If None is set, it uses the default value, `uuuu-MM-dd'T'HH:mm:ss.SSSXXX`.
- **ignoreLeadingWhiteSpace** – a flag indicating whether or not leading whitespaces from values being written should be skipped. If None is set, it uses the default value, `true`.
- **ignoreTrailingWhiteSpace** – a flag indicating whether or not trailing whitespaces from values being written should be skipped. If None is set, it uses the default value, `true`.
- **charToEscapeQuoteEscaping** – sets a single character used for escaping the escape for the quote character. If None is set, the default value is escape character when escape and quote characters are different, `\0` otherwise..
- **encoding** – sets the encoding (charset) of saved csv files. If None is set, the default UTF-8 charset will be used.
- **emptyValue** – sets the string representation of an empty value. If None is set, it uses the default value, `""`.
- **lineSep** – defines the line separator that should be used for writing. If None is set, it uses the default value, `\\n`. Maximum length is 1 character.

```
>>> df.write.csv(os.path.join(tempfile.mkdtemp(), 'data'))
```

New in version 2.0.

format(*source*) [source]

Specifies the underlying output data source.

Parameters:
source – string, name of the data source, e.g. ‘json’, ‘parquet’.

```
>>> df.write.format('json').save(os.path.join(tempfile.mkdtemp(
```

New in version 1.4.

insertInto(*tableName*, *overwrite=None*) [source]

Inserts the content of the **DataFrame** to the specified table.

It requires that the schema of the class:*DataFrame* is the same as the schema of the table.

Optionally overwriting any existing data.

New in version 1.4.

jdbc(*url*, *table*, *mode=None*, *properties=None*) [source]

Saves the content of the **DataFrame** to an external database table via JDBC.

> **Note:** Don't create too many partitions in parallel on a large cluster; otherwise Spark might crash your external database systems.

Parameters:

- **url** – a JDBC URL of the form `jdbc:subprotocol:subname`
- **table** – Name of the table in the external database.
- **mode** –
  specifies the behavior of the save operation when data already exists.
  - `append`: Append contents of this **DataFrame** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` or `errorifexists` (default case): Throw an exception if data already exists.
- **properties** – a dictionary of JDBC database connection arguments. Normally at least properties "user" and "password" with their corresponding values. For example { 'user' : 'SYSTEM', 'password' : 'mypassword' }

New in version 1.4.

json(*path*, *mode=None*, *compression=None*, *dateFormat=None*, *timestampFormat=None*, *lineSep=None*, *encoding=None*, *ignoreNullFields=None*) [source]

Saves the content of the **DataFrame** in JSON format (JSON Lines text format or newline-delimited JSON) at the specified path.

Parameters:

- **path** – the path in any Hadoop supported file system
- **mode** –
  specifies the behavior of the save operation when data already exists.
  - `append`: Append contents of this **DataFrame** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` or `errorifexists` (default case): Throw an exception if data already exists.
- **compression** – compression codec to use when saving to file. This can be one of the known case-insensitive shorten names (none, bzip2, gzip, lz4, snappy and deflate).
- **dateFormat** – sets the string that indicates a date format. Custom date formats follow the formats at `java.time.format.DateTimeFormatter`. This applies to date type. If None is set, it uses the default value, `uuuu-MM-dd`.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at `java.time.format.DateTimeFormatter`. This applies to timestamp type. If None is set, it uses the default value, `uuuu-MM-dd'T'HH:mm:ss.SSSXXX`.
- **encoding** – specifies encoding (charset) of saved json files. If None is set, the default UTF-8 charset will be used.
- **lineSep** – defines the line separator that should be used for writing. If None is set, it uses the default value, `\n`.
- **ignoreNullFields** – Whether to ignore null fields when generating JSON objects. If None is set, it uses the default value, `true`.

```
>>> df.write.json(os.path.join(tempfile.mkdtemp(), 'data'))
```

New in version 1.4.

mode(*saveMode*) [source]

Specifies the behavior when data or table already exists.

Options include:

- *append*: Append contents of this **DataFrame** to existing data.
- *overwrite*: Overwrite existing data.
- *error* or *errorifexists*: Throw an exception if data already exists.
- *ignore*: Silently ignore this operation if data already exists.

```
>>> df.write.mode('append').parquet(os.path.join(tempfile.mkdte
```

New in version 1.4.

option(*key*, *value*) [source]

Adds an output option for the underlying data source.

You can set the following option(s) for writing files:

- `timeZone`: sets the string that indicates a timezone to be used to format
  timestamps in the JSON/CSV datasources or partition values. If it isn't set, it uses the default value, session local timezone.

New in version 1.5.

options(***options*) [source]

Adds output options for the underlying data source.

You can set the following option(s) for writing files:

- `timeZone`: sets the string that indicates a timezone to be used to format
  timestamps in the JSON/CSV datasources or partition values. If it isn't set, it uses the default value, session local timezone.

New in version 1.4.

orc(*path*, *mode=None*, *partitionBy=None*, *compression=None*) [source]

Saves the content of the **DataFrame** in ORC format at the specified path.

Parameters:

- **path** – the path in any Hadoop supported file system
- **mode** –
  specifies the behavior of the save operation when data already exists.
  - `append`: Append contents of this **DataFrame** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` or `errorifexists` (default case): Throw an exception if data already exists.
- **partitionBy** – names of partitioning columns
- **compression** – compression codec to use when saving to file. This can be one of the known case-insensitive shorten names (none, snappy, zlib, and lzo). This will override `orc.compress` and `spark.sql.orc.compression.codec`. If None is set, it uses the value specified in `spark.sql.orc.compression.codec`.

```
>>> orc_df = spark.read.orc('python/test_support/sql/orc_partit
>>> orc_df.write.orc(os.path.join(tempfile.mkdtemp(), 'data'))
```

New in version 1.5.

parquet(*path*, *mode=None*, *partitionBy=None*, *compression=None*) [source]

Saves the content of the **DataFrame** in Parquet format at the specified path.

Parameters:

- **path** – the path in any Hadoop supported file system

- **mode** –
  specifies the behavior of the save operation when data already exists.
  - `append`: Append contents of this **`DataFrame`** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` or `errorifexists` (default case): Throw an exception if data already exists.
- **partitionBy** – names of partitioning columns
- **compression** – compression codec to use when saving to file. This can be one of the known case-insensitive shorten names (none, uncompressed, snappy, gzip, lzo, brotli, lz4, and zstd). This will override `spark.sql.parquet.compression.codec`. If None is set, it uses the value specified in `spark.sql.parquet.compression.codec`.

```
>>> df.write.parquet(os.path.join(tempfile.mkdtemp(), 'data'))
```

New in version 1.4.

`partitionBy`(**cols*) [source]

Partitions the output by the given columns on the file system.

If specified, the output is laid out on the file system similar to Hive's partitioning scheme.

Parameters:
cols – name of columns

```
>>> df.write.partitionBy('year', 'month').parquet(os.path.join(
```

New in version 1.4.

`save`(*path=None*, *format=None*, *mode=None*, *partitionBy=None*, ***options*) [source]

Saves the contents of the **`DataFrame`** to a data source.

The data source is specified by the `format` and a set of `options`. If `format` is not specified, the default data source configured by `spark.sql.sources.default` will be used.

Parameters:
- **path** – the path in a Hadoop supported file system
- **format** – the format used to save
- **mode** –
  specifies the behavior of the save operation when data already exists.
  - `append`: Append contents of this **`DataFrame`** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` or `errorifexists` (default case): Throw an exception if data already exists.
- **partitionBy** – names of partitioning columns
- **options** – all other string options

```
>>> df.write.mode("append").save(os.path.join(tempfile.mkdtemp(
```

New in version 1.4.

`saveAsTable`(*name*, *format=None*, *mode=None*, *partitionBy=None*, ***options*)

Saves the content of the **`DataFrame`** as the specified table. [source]

In the case the table already exists, behavior of this function depends on the save mode, specified by the *mode* function (default to throwing an exception). When *mode* is *Overwrite*, the schema of the **DataFrame** does not need to be the same as that of the existing table.

- *append*: Append contents of this **DataFrame** to existing data.
- *overwrite*: Overwrite existing data.
- *error* or *errorifexists*: Throw an exception if data already exists.
- *ignore*: Silently ignore this operation if data already exists.

Parameters:

- **name** – the table name
- **format** – the format used to save
- **mode** – one of *append*, *overwrite*, *error*, *errorifexists*, *ignore* (default: error)
- **partitionBy** – names of partitioning columns
- **options** – all other string options

New in version 1.4.

sortBy(*col*, **cols*) [source]

Sorts the output in each bucket by the given columns on the file system.

Parameters:

- **col** – a name of a column, or a list of names.
- **cols** – additional names (optional). If *col* is a list it should be empty.

```
>>> (df.write.format('parquet')
...     .bucketBy(100, 'year', 'month')
...     .sortBy('day')
...     .mode("overwrite")
...     .saveAsTable('sorted_bucketed_table'))
```

New in version 2.3.

text(*path*, *compression=None*, *lineSep=None*) [source]

Saves the content of the DataFrame in a text file at the specified path. The text files will be encoded as UTF-8.

Parameters:

- **path** – the path in any Hadoop supported file system
- **compression** – compression codec to use when saving to file. This can be one of the known case-insensitive shorten names (none, bzip2, gzip, lz4, snappy and deflate).
- **lineSep** – defines the line separator that should be used for writing. If None is set, it uses the default value, `\n`.

The DataFrame must have only one column that is of string type. Each row becomes a new line in the output file.

New in version 1.6.

class pyspark.sql.**CoGroupedData**(*gd1*, *gd2*) [source]

A logical grouping of two **GroupedData**, created by **GroupedData.cogroup()**.

Note: Experimental

New in version 3.0.

apply(*udf*) [source]

Applies a function to each cogroup using a pandas udf and returns the result as a *DataFrame*.

The user-defined function should take two *pandas.DataFrame* and return another *pandas.DataFrame*. For each side of the cogroup, all columns are passed together as a *pandas.DataFrame* to the user-function and the returned

pandas.DataFrame are combined as a **DataFrame**.

The returned *pandas.DataFrame* can be of arbitrary length and its schema must match the returnType of the pandas udf.

> **Note:** This function requires a full shuffle. All the data of a cogroup will be loaded into memory, so the user should be aware of the potential OOM risk if data is skewed and certain groups are too large to fit in memory.

> **Note:** Experimental

Parameters:
udf – a cogrouped map user-defined function returned by **pyspark.sql.functions.pandas_udf()**.

```
>>> from pyspark.sql.functions import pandas_udf, PandasUDFType
>>> df1 = spark.createDataFrame(
...     [(20000101, 1, 1.0), (20000101, 2, 2.0), (20000102, 1,
...     ("time", "id", "v1"))
>>> df2 = spark.createDataFrame(
...     [(20000101, 1, "x"), (20000101, 2, "y")],
...     ("time", "id", "v2"))
>>> @pandas_udf("time int, id int, v1 double, v2 string",
...             PandasUDFType.COGROUPED_MAP)
... def asof_join(l, r):
...     return pd.merge_asof(l, r, on="time", by="id")
>>> df1.groupby("id").cogroup(df2.groupby("id")).apply(asof_joi
+--------+---+---+---+
|    time| id| v1| v2|
+--------+---+---+---+
|20000101|  1|1.0|  x|
|20000102|  1|3.0|  x|
|20000101|  2|2.0|  y|
|20000102|  2|4.0|  y|
+--------+---+---+---+
```

Alternatively, the user can define a function that takes three arguments. In this case, the grouping key(s) will be passed as the first argument and the data will be passed as the second and third arguments. The grouping key(s) will be passed as a tuple of numpy data types, e.g., *numpy.int32* and *numpy.float64*. The data will still be passed in as two *pandas.DataFrame* containing all columns from the original Spark DataFrames.

```
>>> @pandas_udf("time int, id int, v1 double, v2 string",
...             PandasUDFType.COGROUPED_MAP)
... def asof_join(k, l, r):
...     if k == (1,):
...         return pd.merge_asof(l, r, on="time", by="id")
...     else:
...         return pd.DataFrame(columns=['time', 'id', 'v1', 'v
>>> df1.groupby("id").cogroup(df2.groupby("id")).apply(asof_joi
+--------+---+---+---+
|    time| id| v1| v2|
+--------+---+---+---+
|20000101|  1|1.0|  x|
|20000102|  1|3.0|  x|
+--------+---+---+---+
```

> **See also:** **pyspark.sql.functions.pandas_udf()**

New in version 3.0.

pyspark.sql.types module

class pyspark.sql.types.**DataType** [source]

Base class for data types.

fromInternal(*obj*) [source]

Converts an internal SQL object into a native Python object.

json() [source]

jsonValue() [source]

needConversion() [source]

Does this type need to conversion between Python object and internal SQL object.

This is used to avoid the unnecessary conversion for ArrayType/MapType/StructType.

simpleString() [source]

toInternal(*obj*) [source]

Converts a Python object into an internal SQL object.

classmethod **typeName**() [source]

class pyspark.sql.types.**NullType** [source]

Null type.

The data type representing None, used for the types that cannot be inferred.

class pyspark.sql.types.**StringType** [source]

String data type.

class pyspark.sql.types.**BinaryType** [source]

Binary (byte array) data type.

class pyspark.sql.types.**BooleanType** [source]

Boolean data type.

class pyspark.sql.types.**DateType** [source]

Date (datetime.date) data type.

EPOCH_ORDINAL = *719163*

fromInternal(*v*) [source]

Converts an internal SQL object into a native Python object.

needConversion() [source]

Does this type need to conversion between Python object and internal SQL object.

This is used to avoid the unnecessary conversion for ArrayType/MapType/StructType.

toInternal(*d*) [source]

Converts a Python object into an internal SQL object.

class pyspark.sql.types.**TimestampType** [source]

Timestamp (datetime.datetime) data type.

fromInternal(*ts*) [source]

Converts an internal SQL object into a native Python object.

needConversion() [source]

Does this type need to conversion between Python object and internal SQL object.

This is used to avoid the unnecessary conversion for ArrayType/MapType/StructType.

toInternal(*dt*) [source]

Converts a Python object into an internal SQL object.

class pyspark.sql.types.**DecimalType**(*precision=10*, *scale=0*) [source]

Decimal (decimal.Decimal) data type.

The DecimalType must have fixed precision (the maximum total number of digits) and scale (the number of digits on the right of dot). For example, (5, 2) can support the value from [-999.99 to 999.99].

The precision can be up to 38, the scale must be less or equal to precision.

When create a DecimalType, the default precision and scale is (10, 0). When infer schema from decimal.Decimal objects, it will be DecimalType(38, 18).

Parameters:

- **precision** – the maximum total number of digits (default: 10)
- **scale** – the number of digits on right side of dot. (default: 0)

jsonValue() [source]

simpleString() [source]

class pyspark.sql.types.**DoubleType** [source]

Double data type, representing double precision floats.

class pyspark.sql.types.**FloatType** [source]

Float data type, representing single precision floats.

class pyspark.sql.types.**ByteType** [source]

Byte data type, i.e. a signed integer in a single byte.

simpleString() [source]

class pyspark.sql.types.**IntegerType** [source]

Int data type, i.e. a signed 32-bit integer.

simpleString() [source]

class pyspark.sql.types.**LongType** [source]

Long data type, i.e. a signed 64-bit integer.

If the values are beyond the range of [-9223372036854775808, 9223372036854775807], please use **DecimalType**.

simpleString() [source]

class pyspark.sql.types.**ShortType** [source]

Short data type, i.e. a signed 16-bit integer.

simpleString() [source]

class pyspark.sql.types.**ArrayType**(*elementType*, *containsNull=True*) [source]

Array data type.

Parameters:

- **elementType** – **DataType** of each element in the array.
- **containsNull** – boolean, whether the array can contain null (None) values.

fromInternal(*obj*) [source]

Converts an internal SQL object into a native Python object.

classmethod **fromJson**(*json*) [source]

jsonValue() [source]

needConversion() [source]

Does this type need to conversion between Python object and internal SQL object.

This is used to avoid the unnecessary conversion for

ArrayType/MapType/StructType.

simpleString() [source]

toInternal(*obj*) [source]

Converts a Python object into an internal SQL object.

class pyspark.sql.types.**MapType**(*keyType*, *valueType*, *valueContainsNull=True*) [source]

Map data type.

Parameters:

- **keyType** – **DataType** of the keys in the map.
- **valueType** – **DataType** of the values in the map.
- **valueContainsNull** – indicates whether values can contain null (None) values.

Keys in a map data type are not allowed to be null (None).

fromInternal(*obj*) [source]

Converts an internal SQL object into a native Python object.

classmethod **fromJson**(*json*) [source]

jsonValue() [source]

needConversion() [source]

Does this type need to conversion between Python object and internal SQL object.

This is used to avoid the unnecessary conversion for ArrayType/MapType/StructType.

simpleString() [source]

toInternal(*obj*) [source]

Converts a Python object into an internal SQL object.

class pyspark.sql.types.**StructField**(*name*, *dataType*, *nullable=True*, *metadata=None*) [source]

A field in **StructType**.

Parameters:

- **name** – string, name of the field.
- **dataType** – **DataType** of the field.
- **nullable** – boolean, whether the field can be null (None) or not.
- **metadata** – a dict from string to simple type that can be toInternald to JSON automatically

fromInternal(*obj*) [source]

Converts an internal SQL object into a native Python object.

classmethod **fromJson**(*json*) [source]

jsonValue() [source]

needConversion() [source]

Does this type need to conversion between Python object and internal SQL object.

This is used to avoid the unnecessary conversion for ArrayType/MapType/StructType.

simpleString() [source]

toInternal(*obj*) [source]

Converts a Python object into an internal SQL object.

typeName() [source]

class pyspark.sql.types.**StructType**(*fields=None*) [source]

Struct type, consisting of a list of **StructField**.

This is the data type representing a **Row**.

Iterating a **StructType** will iterate its **StructField**s. A contained **StructField** can be accessed by name or position.

```
>>> struct1 = StructType([StructField("f1", StringType(), True)])
>>> struct1["f1"]
StructField(f1,StringType,true)
>>> struct1[0]
StructField(f1,StringType,true)
```

add(*field*, *data_type=None*, *nullable=True*, *metadata=None*) [source]

Construct a StructType by adding new elements to it to define the schema. The method accepts either:

a. A single parameter which is a StructField object.
b. Between 2 and 4 parameters as (name, data_type, nullable (optional), metadata(optional). The data_type parameter may be either a String or a DataType object.

```
>>> struct1 = StructType().add("f1", StringType(), True).add("f
>>> struct2 = StructType([StructField("f1", StringType(), True)
...     StructField("f2", StringType(), True, None)])
>>> struct1 == struct2
True
>>> struct1 = StructType().add(StructField("f1", StringType(),
>>> struct2 = StructType([StructField("f1", StringType(), True)
>>> struct1 == struct2
True
>>> struct1 = StructType().add("f1", "string", True)
>>> struct2 = StructType([StructField("f1", StringType(), True)
>>> struct1 == struct2
True
```

Parameters:

- **field** – Either the name of the field or a StructField object
- **data_type** – If present, the DataType of the StructField to create
- **nullable** – Whether the field to add should be nullable (default True)
- **metadata** – Any additional metadata (default None)

Returns:

a new updated StructType

fieldNames() [source]

Returns all field names in a list.

```
>>> struct = StructType([StructField("f1", StringType(), True)]
>>> struct.fieldNames()
['f1']
```

fromInternal(*obj*) [source]

Converts an internal SQL object into a native Python object.

classmethod **fromJson**(*json*) [source]

jsonValue() [source]

needConversion() [source]

Does this type need to conversion between Python object and internal SQL object.

This is used to avoid the unnecessary conversion for ArrayType/MapType/StructType.

simpleString() [source]

toInternal(*obj*) [source]

Converts a Python object into an internal SQL object.

pyspark.sql.functions module

A collections of builtin functions

class pyspark.sql.functions.**PandasUDFType** [source]

Pandas UDF Types. See **pyspark.sql.functions.pandas_udf()**.

COGROUPED_MAP = *206*

GROUPED_AGG = *202*

GROUPED_MAP = *201*

MAP_ITER = *205*

SCALAR = *200*

SCALAR_ITER = *204*

pyspark.sql.functions.**abs**(*col*)

Computes the absolute value.

New in version 1.3.

pyspark.sql.functions.**acos**(*col*)

Returns:
inverse cosine of *col*, as if computed by *java.lang.Math.acos()*

New in version 1.4.

pyspark.sql.functions.**add_months**(*start*, *months*) [source]

Returns the date that is *months* months after *start*

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> df.select(add_months(df.dt, 1).alias('next_month')).collect()
[Row(next_month=datetime.date(2015, 5, 8))]
```

New in version 1.5.

pyspark.sql.functions.**approx_count_distinct**(*col*, *rsd=None*) [source]

Aggregate function: returns a new **Column** for approximate distinct count of column *col*.

Parameters:
rsd – maximum estimation error allowed (default = 0.05). For rsd < 0.01, it is more efficient to use **countDistinct()**

```
>>> df.agg(approx_count_distinct(df.age).alias('distinct_ages')).co
[Row(distinct_ages=2)]
```

New in version 2.1.

pyspark.sql.functions.**array**(**cols*) [source]

Creates a new array column.

Parameters:

cols – list of column names (string) or list of `Column` expressions that have the same data type.

```
>>> df.select(array('age', 'age').alias("arr")).collect()
[Row(arr=[2, 2]), Row(arr=[5, 5])]
>>> df.select(array([df.age, df.age]).alias("arr")).collect()
[Row(arr=[2, 2]), Row(arr=[5, 5])]
```

New in version 1.4.

`pyspark.sql.functions.`**`array_contains`**(*col*, *value*) [source]

Collection function: returns null if the array is null, true if the array contains the given value, and false otherwise.

Parameters:

- **col** – name of column containing array
- **value** – value or column to check for in array

```
>>> df = spark.createDataFrame([(["a", "b", "c"],), ([],)], ['data'
>>> df.select(array_contains(df.data, "a")).collect()
[Row(array_contains(data, a)=True), Row(array_contains(data, a)=Fal
>>> df.select(array_contains(df.data, lit("a"))).collect()
[Row(array_contains(data, a)=True), Row(array_contains(data, a)=Fal
```

New in version 1.5.

`pyspark.sql.functions.`**`array_distinct`**(*col*) [source]

Collection function: removes duplicate values from the array. :param col: name of column or expression

```
>>> df = spark.createDataFrame([([1, 2, 3, 2],), ([4, 5, 5, 4],)],
>>> df.select(array_distinct(df.data)).collect()
[Row(array_distinct(data)=[1, 2, 3]), Row(array_distinct(data)=[4,
```

New in version 2.4.

`pyspark.sql.functions.`**`array_except`**(*col1*, *col2*) [source]

Collection function: returns an array of the elements in col1 but not in col2, without duplicates.

Parameters:

- **col1** – name of column containing array
- **col2** – name of column containing array

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([Row(c1=["b", "a", "c"], c2=["c", "
>>> df.select(array_except(df.c1, df.c2)).collect()
[Row(array_except(c1, c2)=['b'])]
```

New in version 2.4.

`pyspark.sql.functions.`**`array_intersect`**(*col1*, *col2*) [source]

Collection function: returns an array of the elements in the intersection of col1 and col2, without duplicates.

Parameters:

- **col1** – name of column containing array
- **col2** – name of column containing array

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([Row(c1=["b", "a", "c"], c2=["c", "
>>> df.select(array_intersect(df.c1, df.c2)).collect()
[Row(array_intersect(c1, c2)=['a', 'c'])]
```

New in version 2.4.

`pyspark.sql.functions.`**`array_join`**(*col*, *delimiter*, *null_replacement=None*) [source]

Concatenates the elements of *column* using the *delimiter*. Null values are replaced with *null_replacement* if set, otherwise they are ignored.

```
>>> df = spark.createDataFrame([(["a", "b", "c"],), (["a", None],)]
>>> df.select(array_join(df.data, ",").alias("joined")).collect()
[Row(joined='a,b,c'), Row(joined='a')]
>>> df.select(array_join(df.data, ",", "NULL").alias("joined")).col
[Row(joined='a,b,c'), Row(joined='a,NULL')]
```

New in version 2.4.

`pyspark.sql.functions.`**`array_max`**(*col*) [source]

Collection function: returns the maximum value of the array.

Parameters:

col – name of column or expression

```
>>> df = spark.createDataFrame([([2, 1, 3],), ([None, 10, -1],)],
>>> df.select(array_max(df.data).alias('max')).collect()
[Row(max=3), Row(max=10)]
```

New in version 2.4.

`pyspark.sql.functions.`**`array_min`**(*col*) [source]

Collection function: returns the minimum value of the array.

Parameters:

col – name of column or expression

```
>>> df = spark.createDataFrame([([2, 1, 3],), ([None, 10, -1],)],
>>> df.select(array_min(df.data).alias('min')).collect()
[Row(min=1), Row(min=-1)]
```

New in version 2.4.

`pyspark.sql.functions.`**`array_position`**(*col*, *value*) [source]

Collection function: Locates the position of the first occurrence of the given value in the given array. Returns null if either of the arguments are null.

Note: The position is not zero based, but 1 based index. Returns 0 if the given value could not be found in the array.

```
>>> df = spark.createDataFrame([(["c", "b", "a"],), ([],)], ['data'
>>> df.select(array_position(df.data, "a")).collect()
[Row(array_position(data, a)=3), Row(array_position(data, a)=0)]
```

New in version 2.4.

`pyspark.sql.functions.`**`array_remove`**(*col*, *element*) [source]

Collection function: Remove all elements that equal to element from the given array.

Parameters:

- **col** – name of column containing array
- **element** – element to be removed from the array

```
>>> df = spark.createDataFrame([([1, 2, 3, 1, 1],), ([],)], ['data
>>> df.select(array_remove(df.data, 1)).collect()
[Row(array_remove(data, 1)=[2, 3]), Row(array_remove(data, 1)=[])]
```

New in version 2.4.

pyspark.sql.functions.**array_repeat**(*col*, *count*) [source]

Collection function: creates an array containing a column repeated count times.

```
>>> df = spark.createDataFrame([('ab',)], ['data'])
>>> df.select(array_repeat(df.data, 3).alias('r')).collect()
[Row(r=['ab', 'ab', 'ab'])]
```

New in version 2.4.

pyspark.sql.functions.**array_sort**(*col*) [source]

Collection function: sorts the input array in ascending order. The elements of the input array must be orderable. Null elements will be placed at the end of the returned array.

Parameters:

col – name of column or expression

```
>>> df = spark.createDataFrame([([2, 1, None, 3],),([1],),([],)],
>>> df.select(array_sort(df.data).alias('r')).collect()
[Row(r=[1, 2, 3, None]), Row(r=[1]), Row(r=[])]
```

New in version 2.4.

pyspark.sql.functions.**array_union**(*col1*, *col2*) [source]

Collection function: returns an array of the elements in the union of col1 and col2, without duplicates.

Parameters:

- **col1** – name of column containing array
- **col2** – name of column containing array

```
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([Row(c1=["b", "a", "c"], c2=["c", "
>>> df.select(array_union(df.c1, df.c2)).collect()
[Row(array_union(c1, c2)=['b', 'a', 'c', 'd', 'f'])]
```

New in version 2.4.

pyspark.sql.functions.**arrays_overlap**(*a1*, *a2*) [source]

Collection function: returns true if the arrays contain any common non-null element; if not, returns null if both the arrays are non-empty and any of them contains a null element; returns false otherwise.

```
>>> df = spark.createDataFrame([(["a", "b"], ["b", "c"]), (["a"], [
>>> df.select(arrays_overlap(df.x, df.y).alias("overlap")).collect
[Row(overlap=True), Row(overlap=False)]
```

New in version 2.4.

pyspark.sql.functions.**arrays_zip**(**cols*) [source]

Collection function: Returns a merged array of structs in which the N-th struct contains all N-th values of input arrays.

Parameters:
cols – columns of arrays to be merged.

```
>>> from pyspark.sql.functions import arrays_zip
>>> df = spark.createDataFrame([([1, 2, 3], [2, 3, 4]))], ['vals1
>>> df.select(arrays_zip(df.vals1, df.vals2).alias('zipped')).colle
[Row(zipped=[Row(vals1=1, vals2=2), Row(vals1=2, vals2=3), Row(vals
```

New in version 2.4.

`pyspark.sql.functions.`**`asc`**(*col*)
Returns a sort expression based on the ascending order of the given column name.

New in version 1.3.

`pyspark.sql.functions.`**`asc_nulls_first`**(*col*)
Returns a sort expression based on the ascending order of the given column name, and null values return before non-null values.

New in version 2.4.

`pyspark.sql.functions.`**`asc_nulls_last`**(*col*)
Returns a sort expression based on the ascending order of the given column name, and null values appear after non-null values.

New in version 2.4.

`pyspark.sql.functions.`**`ascii`**(*col*)
Computes the numeric value of the first character of the string column.

New in version 1.5.

`pyspark.sql.functions.`**`asin`**(*col*)

Returns:
inverse sine of *col*, as if computed by *java.lang.Math.asin()*

New in version 1.4.

`pyspark.sql.functions.`**`atan`**(*col*)

Returns:
inverse tangent of *col*, as if computed by *java.lang.Math.atan()*

New in version 1.4.

`pyspark.sql.functions.`**`atan2`**(*col1*, *col2*)

Parameters:
- **col1** – coordinate on y-axis
- **col2** – coordinate on x-axis

Returns:
the *theta* component of the point (*r*, *theta*) in polar coordinates that corresponds to the point (*x*, *y*) in Cartesian coordinates, as if computed by *java.lang.Math.atan2()*

New in version 1.4.

`pyspark.sql.functions.`**`avg`**(*col*)
Aggregate function: returns the average of the values in a group.

New in version 1.3.

`pyspark.sql.functions.`**`base64`**(*col*)
Computes the BASE64 encoding of a binary column and returns it as a string column.

New in version 1.5.

`pyspark.sql.functions.`**`basestring`**

alias of **builtins.str**

pyspark.sql.functions.**bin**(*col*) [source]

Returns the string representation of the binary value of the given column.

```
>>> df.select(bin(df.age).alias('c')).collect()
[Row(c='10'), Row(c='101')]
```

New in version 1.5.

pyspark.sql.functions.**bitwiseNOT**(*col*)

Computes bitwise not.

New in version 1.4.

pyspark.sql.functions.**broadcast**(*df*) [source]

Marks a DataFrame as small enough for use in broadcast joins.

New in version 1.6.

pyspark.sql.functions.**bround**(*col*, *scale=0*) [source]

Round the given value to *scale* decimal places using HALF_EVEN rounding mode if *scale* >= 0 or at integral part when *scale* < 0.

```
>>> spark.createDataFrame([(2.5,)], ['a']).select(bround('a', 0).al
[Row(r=2.0)]
```

New in version 2.0.

pyspark.sql.functions.**cbrt**(*col*)

Computes the cube-root of the given value.

New in version 1.4.

pyspark.sql.functions.**ceil**(*col*)

Computes the ceiling of the given value.

New in version 1.4.

pyspark.sql.functions.**coalesce**(**cols*) [source]

Returns the first column that is not null.

```
>>> cDf = spark.createDataFrame([(None, None), (1, None), (None, 2)
>>> cDf.show()
+----+----+
|   a|   b|
+----+----+
|null|null|
|   1|null|
|null|   2|
+----+----+
```

```
>>> cDf.select(coalesce(cDf["a"], cDf["b"])).show()
+--------------+
|coalesce(a, b)|
+--------------+
|          null|
|             1|
|             2|
+--------------+
```

```
>>> cDf.select('*', coalesce(cDf["a"], lit(0.0))).show()
+----+----+----------------+
|   a|   b|coalesce(a, 0.0)|
+----+----+----------------+
|null|null|             0.0|
|   1|null|             1.0|
|null|   2|             0.0|
+----+----+----------------+
```

New in version 1.4.

pyspark.sql.functions.**col**(*col*)

Returns a **Column** based on the given column name.

New in version 1.3.

pyspark.sql.functions.**collect_list**(*col*)

Aggregate function: returns a list of objects with duplicates.

Note: The function is non-deterministic because the order of collected results depends on order of rows which may be non-deterministic after a shuffle.

```
>>> df2 = spark.createDataFrame([(2,), (5,), (5,)], ('age',))
>>> df2.agg(collect_list('age')).collect()
[Row(collect_list(age)=[2, 5, 5])]
```

New in version 1.6.

pyspark.sql.functions.**collect_set**(*col*)

Aggregate function: returns a set of objects with duplicate elements eliminated.

Note: The function is non-deterministic because the order of collected results depends on order of rows which may be non-deterministic after a shuffle.

```
>>> df2 = spark.createDataFrame([(2,), (5,), (5,)], ('age',))
>>> df2.agg(collect_set('age')).collect()
[Row(collect_set(age)=[5, 2])]
```

New in version 1.6.

pyspark.sql.functions.**column**(*col*)

Returns a **Column** based on the given column name.

New in version 1.3.

pyspark.sql.functions.**concat**(**cols*) [source]

Concatenates multiple input columns together into a single column. The function works with strings, binary and compatible array columns.

```
>>> df = spark.createDataFrame([('abcd','123')], ['s', 'd'])
>>> df.select(concat(df.s, df.d).alias('s')).collect()
[Row(s='abcd123')]
```

```
>>> df = spark.createDataFrame([([1, 2], [3, 4], [5]), ([1, 2], No
>>> df.select(concat(df.a, df.b, df.c).alias("arr")).collect()
[Row(arr=[1, 2, 3, 4, 5]), Row(arr=None)]
```

New in version 1.5.

pyspark.sql.functions.**concat_ws**(*sep*, **cols*) [source]

Concatenates multiple input string columns together into a single string column, using the given separator.

```
>>> df = spark.createDataFrame([('abcd','123')], ['s', 'd'])
>>> df.select(concat_ws('-', df.s, df.d).alias('s')).collect()
[Row(s='abcd-123')]
```

New in version 1.5.

pyspark.sql.functions.**conv**(*col*, *fromBase*, *toBase*) [source]

Convert a number in a string column from one base to another.

```
>>> df = spark.createDataFrame([("010101",)], ['n'])
>>> df.select(conv(df.n, 2, 16).alias('hex')).collect()
[Row(hex='15')]
```

New in version 1.5.

pyspark.sql.functions.**corr**(*col1*, *col2*) [source]

Returns a new **Column** for the Pearson Correlation Coefficient for `col1` and `col2`.

```
>>> a = range(20)
>>> b = [2 * x for x in range(20)]
>>> df = spark.createDataFrame(zip(a, b), ["a", "b"])
>>> df.agg(corr("a", "b").alias('c')).collect()
[Row(c=1.0)]
```

New in version 1.6.

pyspark.sql.functions.**cos**(*col*)

Parameters:
col – angle in radians
Returns:
cosine of the angle, as if computed by *java.lang.Math.cos()*.

New in version 1.4.

pyspark.sql.functions.**cosh**(*col*)

Parameters:
col – hyperbolic angle
Returns:
hyperbolic cosine of the angle, as if computed by *java.lang.Math.cosh()*

New in version 1.4.

pyspark.sql.functions.**count**(*col*)

Aggregate function: returns the number of items in a group.

New in version 1.3.

pyspark.sql.functions.**countDistinct**(*col*, **cols*) [source]

Returns a new **Column** for distinct count of `col` or `cols`.

```
>>> df.agg(countDistinct(df.age, df.name).alias('c')).collect()
[Row(c=2)]
```

```
>>> df.agg(countDistinct("age", "name").alias('c')).collect()
[Row(c=2)]
```

New in version 1.3.

pyspark.sql.functions.**covar_pop**(*col1*, *col2*) [source]

Returns a new **Column** for the population covariance of `col1` and `col2`.

```
>>> a = [1] * 10
>>> b = [1] * 10
>>> df = spark.createDataFrame(zip(a, b), ["a", "b"])
>>> df.agg(covar_pop("a", "b").alias('c')).collect()
[Row(c=0.0)]
```

New in version 2.0.

pyspark.sql.functions.**covar_samp**(*col1*, *col2*) [source]

Returns a new **Column** for the sample covariance of `col1` and `col2`.

```
>>> a = [1] * 10
>>> b = [1] * 10
>>> df = spark.createDataFrame(zip(a, b), ["a", "b"])
>>> df.agg(covar_samp("a", "b").alias('c')).collect()
[Row(c=0.0)]
```

New in version 2.0.

pyspark.sql.functions.**crc32**(*col*) [source]

Calculates the cyclic redundancy check value (CRC32) of a binary column and returns the value as a bigint.

```
>>> spark.createDataFrame([('ABC',)], ['a']).select(crc32('a').alia
[Row(crc32=2743272264)]
```

New in version 1.5.

pyspark.sql.functions.**create_map**(**cols*) [source]

Creates a new map column.

Parameters:
cols – list of column names (string) or list of **Column** expressions that are grouped as key-value pairs, e.g. (key1, value1, key2, value2, …).

```
>>> df.select(create_map('name', 'age').alias("map")).collect()
[Row(map={'Alice': 2}), Row(map={'Bob': 5})]
>>> df.select(create_map([df.name, df.age]).alias("map")).collect(
[Row(map={'Alice': 2}), Row(map={'Bob': 5})]
```

New in version 2.0.

pyspark.sql.functions.**cume_dist**()

Window function: returns the cumulative distribution of values within a window partition, i.e. the fraction of rows that are below the current row.

New in version 1.6.

pyspark.sql.functions.**current_date**() [source]

Returns the current date as a **DateType** column.

New in version 1.5.

pyspark.sql.functions.**current_timestamp**() [source]

Returns the current timestamp as a **TimestampType** column.

pyspark.sql.functions.**date_add**(*start*, *days*) [source]

Returns the date that is *days* days after *start*

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> df.select(date_add(df.dt, 1).alias('next_date')).collect()
[Row(next_date=datetime.date(2015, 4, 9))]
```

New in version 1.5.

pyspark.sql.functions.**date_format**(*date*, *format*) [source]

Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument.

A pattern could be for instance *dd.MM.yyyy* and could return a string like '18.03.1993'. All pattern letters of the Java class *java.time.format.DateTimeFormatter* can be used.

> **Note:** Use when ever possible specialized functions like *year*. These benefit from a specialized implementation.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> df.select(date_format('dt', 'MM/dd/yyy').alias('date')).collec
[Row(date='04/08/2015')]
```

New in version 1.5.

pyspark.sql.functions.**date_sub**(*start*, *days*) [source]

Returns the date that is *days* days before *start*

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> df.select(date_sub(df.dt, 1).alias('prev_date')).collect()
[Row(prev_date=datetime.date(2015, 4, 7))]
```

New in version 1.5.

pyspark.sql.functions.**date_trunc**(*format*, *timestamp*) [source]

Returns timestamp truncated to the unit specified by the format.

Parameters:
format – 'year', 'yyyy', 'yy', 'month', 'mon', 'mm', 'day', 'dd', 'hour', 'minute', 'second', 'week', 'quarter'

```
>>> df = spark.createDataFrame([('1997-02-28 05:02:11',)], ['t'])
>>> df.select(date_trunc('year', df.t).alias('year')).collect()
[Row(year=datetime.datetime(1997, 1, 1, 0, 0))]
>>> df.select(date_trunc('mon', df.t).alias('month')).collect()
[Row(month=datetime.datetime(1997, 2, 1, 0, 0))]
```

New in version 2.3.

pyspark.sql.functions.**datediff**(*end*, *start*) [source]

Returns the number of days from *start* to *end*.

```
>>> df = spark.createDataFrame([('2015-04-08','2015-05-10')], ['d1'
>>> df.select(datediff(df.d2, df.d1).alias('diff')).collect()
[Row(diff=32)]
```

New in version 1.5.

pyspark.sql.functions.**dayofmonth**(*col*) [source]

Extract the day of the month of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> df.select(dayofmonth('dt').alias('day')).collect()
[Row(day=8)]
```

New in version 1.5.

pyspark.sql.functions.**dayofweek**(*col*) [source]

Extract the day of the week of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> df.select(dayofweek('dt').alias('day')).collect()
[Row(day=4)]
```

New in version 2.3.

pyspark.sql.functions.**dayofyear**(*col*) [source]

Extract the day of the year of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> df.select(dayofyear('dt').alias('day')).collect()
[Row(day=98)]
```

New in version 1.5.

pyspark.sql.functions.**decode**(*col*, *charset*) [source]

Computes the first argument into a string from a binary using the provided character set (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16').

New in version 1.5.

pyspark.sql.functions.**degrees**(*col*)

Converts an angle measured in radians to an approximately equivalent angle measured in degrees. :param col: angle in radians :return: angle in degrees, as if computed by *java.lang.Math.toDegrees()*

New in version 2.1.

pyspark.sql.functions.**dense_rank**()

Window function: returns the rank of rows within a window partition, without any gaps.

The difference between rank and dense_rank is that dense_rank leaves no gaps in ranking sequence when there are ties. That is, if you were ranking a competition using dense_rank and had three people tie for second place, you would say that all three were in second place and that the next person came in third. Rank would give me sequential numbers, making the person that came in third place (after the ties) would register as coming in fifth.

This is equivalent to the DENSE_RANK function in SQL.

New in version 1.6.

pyspark.sql.functions.**desc**(*col*)

Returns a sort expression based on the descending order of the given column name.

New in version 1.3.

pyspark.sql.functions.**desc_nulls_first**(*col*)

Returns a sort expression based on the descending order of the given column name, and null values appear before non-null values.

New in version 2.4.

pyspark.sql.functions.**desc_nulls_last**(*col*)

Returns a sort expression based on the descending order of the given column name, and null values appear after non-null values

New in version 2.4.

pyspark.sql.functions.**element_at**(*col*, *extraction*) [source]

Collection function: Returns element of array at given index in extraction if col is array. Returns value for the given key in extraction if col is map.

Parameters:

- **col** – name of column containing array or map
- **extraction** – index to check for in array or key to check for in map

Note: The position is not zero based, but 1 based index.

```
>>> df = spark.createDataFrame([(["a", "b", "c"],), ([],)], ['data'
>>> df.select(element_at(df.data, 1)).collect()
[Row(element_at(data, 1)='a'), Row(element_at(data, 1)=None)]
```

```
>>> df = spark.createDataFrame([({"a": 1.0, "b": 2.0},), ({},)], ['
>>> df.select(element_at(df.data, lit("a"))).collect()
[Row(element_at(data, a)=1.0), Row(element_at(data, a)=None)]
```

New in version 2.4.

`pyspark.sql.functions.`**`encode`**(*col*, *charset*) [source]

Computes the first argument into a binary from a string using the provided character set (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16').

New in version 1.5.

`pyspark.sql.functions.`**`exp`**(*col*)

Computes the exponential of the given value.

New in version 1.4.

`pyspark.sql.functions.`**`explode`**(*col*) [source]

Returns a new row for each element in the given array or map. Uses the default column name *col* for elements in the array and *key* and *value* for elements in the map unless specified otherwise.

```
>>> from pyspark.sql import Row
>>> eDF = spark.createDataFrame([Row(a=1, intlist=[1,2,3], mapfiel
>>> eDF.select(explode(eDF.intlist).alias("anInt")).collect()
[Row(anInt=1), Row(anInt=2), Row(anInt=3)]
```

```
>>> eDF.select(explode(eDF.mapfield).alias("key", "value")).show()
+---+-----+
|key|value|
+---+-----+
|  a|    b|
+---+-----+
```

New in version 1.4.

`pyspark.sql.functions.`**`explode_outer`**(*col*) [source]

Returns a new row for each element in the given array or map. Unlike explode, if the array/map is null or empty then null is produced. Uses the default column name *col* for elements in the array and *key* and *value* for elements in the map unless specified otherwise.

```
>>> df = spark.createDataFrame(
...     [(1, ["foo", "bar"], {"x": 1.0}), (2, [], {}), (3, None, N
...     ("id", "an_array", "a_map")
... )
>>> df.select("id", "an_array", explode_outer("a_map")).show()
+---+----------+----+-----+
| id|  an_array| key|value|
+---+----------+----+-----+
|  1|[foo, bar]|   x|  1.0|
|  2|        []|null| null|
|  3|      null|null| null|
+---+----------+----+-----+
```

```
>>> df.select("id", "a_map", explode_outer("an_array")).show()
+---+----------+----+
| id|     a_map| col|
+---+----------+----+
|  1|[x -> 1.0]| foo|
|  1|[x -> 1.0]| bar|
|  2|        []|null|
|  3|      null|null|
+---+----------+----+
```

New in version 2.3.

pyspark.sql.functions.**expm1**(*col*)

Computes the exponential of the given value minus one.

New in version 1.4.

pyspark.sql.functions.**expr**(*str*) [source]

Parses the expression string into the column that it represents

```
>>> df.select(expr("length(name)")).collect()
[Row(length(name)=5), Row(length(name)=3)]
```

New in version 1.5.

pyspark.sql.functions.**factorial**(*col*) [source]

Computes the factorial of the given value.

```
>>> df = spark.createDataFrame([(5,)], ['n'])
>>> df.select(factorial(df.n).alias('f')).collect()
[Row(f=120)]
```

New in version 1.5.

pyspark.sql.functions.**first**(*col*, *ignorenulls=False*) [source]

Aggregate function: returns the first value in a group.

The function by default returns the first values it sees. It will return the first non-null value it sees when ignoreNulls is set to true. If all values are null, then null is returned.

Note: The function is non-deterministic because its results depends on order of rows which may be non-deterministic after a shuffle.

New in version 1.3.

pyspark.sql.functions.**flatten**(*col*) [source]

Collection function: creates a single array from an array of arrays. If a structure of nested arrays is deeper than two levels, only one level of nesting is removed.

Parameters:
col – name of column or expression

```
>>> df = spark.createDataFrame([([[1, 2, 3], [4, 5], [6]],), ([Non
>>> df.select(flatten(df.data).alias('r')).collect()
[Row(r=[1, 2, 3, 4, 5, 6]), Row(r=None)]
```

New in version 2.4.

pyspark.sql.functions.**floor**(*col*)

Computes the floor of the given value.

New in version 1.4.

pyspark.sql.functions.**format_number**(*col*, *d*) [source]

Formats the number X to a format like '#,–#,–#.–', rounded to d decimal places with

HALF_EVEN round mode, and returns the result as a string.

Parameters:

- **col** – the column name of the numeric value to be formatted
- **d** – the N decimal places

```
>>> spark.createDataFrame([(5,)], ['a']).select(format_number('a',
[Row(v='5.0000')]
```

New in version 1.5.

`pyspark.sql.functions.`**`format_string`**(*format*, **cols*) [source]

Formats the arguments in printf-style and returns the result as a string column.

Parameters:

- **format** – string that can contain embedded format tags and used as result column's value
- **cols** – list of column names (string) or list of **`Column`** expressions to be used in formatting

```
>>> df = spark.createDataFrame([(5, "hello")], ['a', 'b'])
>>> df.select(format_string('%d %s', df.a, df.b).alias('v')).colle
[Row(v='5 hello')]
```

New in version 1.5.

`pyspark.sql.functions.`**`from_csv`**(*col*, *schema*, *options={}*) [source]

Parses a column containing a CSV string to a row with the specified schema. Returns *null*, in the case of an unparseable string.

Parameters:

- **col** – string column in CSV format
- **schema** – a string with schema in DDL format to use when parsing the CSV column.
- **options** – options to control parsing. accepts the same options as the CSV datasource

```
>>> data = [("1,2,3",)]
>>> df = spark.createDataFrame(data, ("value",))
>>> df.select(from_csv(df.value, "a INT, b INT, c INT").alias("csv"
[Row(csv=Row(a=1, b=2, c=3))]
>>> value = data[0][0]
>>> df.select(from_csv(df.value, schema_of_csv(value)).alias("csv")
[Row(csv=Row(_c0=1, _c1=2, _c2=3))]
>>> data = [("   abc",)]
>>> df = spark.createDataFrame(data, ("value",))
>>> options = {'ignoreLeadingWhiteSpace': True}
>>> df.select(from_csv(df.value, "s string", options).alias("csv")
[Row(csv=Row(s='abc'))]
```

New in version 3.0.

`pyspark.sql.functions.`**`from_json`**(*col*, *schema*, *options={}*) [source]

Parses a column containing a JSON string into a **`MapType`** with **`StringType`** as keys type, **`StructType`** or **`ArrayType`** with the specified schema. Returns *null*, in the case of an unparseable string.

Parameters:

- **col** – string column in json format
- **schema** – a StructType or ArrayType of StructType to use when parsing the json column.
- **options** – options to control parsing. accepts the same options as the json datasource

Note: Since Spark 2.3, the DDL-formatted string or a JSON format string is also supported for `schema`.

```
>>> from pyspark.sql.types import *
>>> data = [(1, '''{"a": 1}''')]
>>> schema = StructType([StructField("a", IntegerType())])
>>> df = spark.createDataFrame(data, ("key", "value"))
>>> df.select(from_json(df.value, schema).alias("json")).collect()
[Row(json=Row(a=1))]
>>> df.select(from_json(df.value, "a INT").alias("json")).collect(
[Row(json=Row(a=1))]
>>> df.select(from_json(df.value, "MAP<STRING,INT>").alias("json")
[Row(json={'a': 1})]
>>> data = [(1, '''[{"a": 1}]''')]
>>> schema = ArrayType(StructType([StructField("a", IntegerType())]
>>> df = spark.createDataFrame(data, ("key", "value"))
>>> df.select(from_json(df.value, schema).alias("json")).collect()
[Row(json=[Row(a=1)])]
>>> schema = schema_of_json(lit('''{"a": 0}'''))
>>> df.select(from_json(df.value, schema).alias("json")).collect()
[Row(json=Row(a=None))]
>>> data = [(1, '''[1, 2, 3]''')]
>>> schema = ArrayType(IntegerType())
>>> df = spark.createDataFrame(data, ("key", "value"))
>>> df.select(from_json(df.value, schema).alias("json")).collect()
[Row(json=[1, 2, 3])]
```

New in version 2.1.

`pyspark.sql.functions.`**`from_unixtime`**(*timestamp*, *format='uuuu-MM-dd HH:mm:ss'*) [source]

Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the given format.

```
>>> spark.conf.set("spark.sql.session.timeZone", "America/Los_Angel
>>> time_df = spark.createDataFrame([(1428476400,)], ['unix_time'])
>>> time_df.select(from_unixtime('unix_time').alias('ts')).collect(
[Row(ts='2015-04-08 00:00:00')]
>>> spark.conf.unset("spark.sql.session.timeZone")
```

New in version 1.5.

`pyspark.sql.functions.`**`from_utc_timestamp`**(*timestamp*, *tz*) [source]

This is a common function for databases supporting TIMESTAMP WITHOUT TIMEZONE. This function takes a timestamp which is timezone-agnostic, and interprets it as a timestamp in UTC, and renders that timestamp as a timestamp in the given time zone.

However, timestamp in Spark represents number of microseconds from the Unix epoch, which is not timezone-agnostic. So in Spark this function just shift the timestamp value from UTC timezone to the given timezone.

This function may return confusing result if the input is a string with timezone, e.g. '2018-03-13T06:18:23+00:00'. The reason is that, Spark firstly cast the string to timestamp according to the timezone in the string, and finally display the result by converting the timestamp to string according to the session local timezone.

Parameters:

- **timestamp** – the column that contains timestamps
- **tz** – a string that has the ID of timezone, e.g. "GMT", "America/Los_Angeles", etc

Changed in version 2.4: tz can take a **`Column`** containing timezone ID strings.

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00', 'JST')], ['
>>> df.select(from_utc_timestamp(df.ts, "PST").alias('local_time')
[Row(local_time=datetime.datetime(1997, 2, 28, 2, 30))]
>>> df.select(from_utc_timestamp(df.ts, df.tz).alias('local_time')
[Row(local_time=datetime.datetime(1997, 2, 28, 19, 30))]
```

Note: Deprecated in 3.0. See SPARK-25496

New in version 1.5.

`pyspark.sql.functions.`**`get_json_object`**`(`*col*, *path*`)` [source]

Extracts json object from a json string based on json path specified, and returns json string of the extracted json object. It will return null if the input json string is invalid.

Parameters:

- **col** – string column in json format
- **path** – path to the json object to extract

```
>>> data = [("1", '''{"f1": "value1", "f2": "value2"}'''), ("2", ''
>>> df = spark.createDataFrame(data, ("key", "jstring"))
>>> df.select(df.key, get_json_object(df.jstring, '$.f1').alias("c
...                   get_json_object(df.jstring, '$.f2').alias("c
[Row(key='1', c0='value1', c1='value2'), Row(key='2', c0='value12',
```

New in version 1.6.

`pyspark.sql.functions.`**`greatest`**`(`**cols*`)` [source]

Returns the greatest value of the list of column names, skipping null values. This function takes at least 2 parameters. It will return null iff all parameters are null.

```
>>> df = spark.createDataFrame([(1, 4, 3)], ['a', 'b', 'c'])
>>> df.select(greatest(df.a, df.b, df.c).alias("greatest")).collec
[Row(greatest=4)]
```

New in version 1.5.

`pyspark.sql.functions.`**`grouping`**`(`*col*`)` [source]

Aggregate function: indicates whether a specified column in a GROUP BY list is aggregated or not, returns 1 for aggregated or 0 for not aggregated in the result set.

```
>>> df.cube("name").agg(grouping("name"), sum("age")).orderBy("name
+-----+--------------+--------+
| name|grouping(name)|sum(age)|
+-----+--------------+--------+
| null|             1|       7|
|Alice|             0|       2|
|  Bob|             0|       5|
+-----+--------------+--------+
```

New in version 2.0.

`pyspark.sql.functions.`**`grouping_id`**`(`**cols*`)` [source]

Aggregate function: returns the level of grouping, equals to

(grouping(c1) << (n-1)) + (grouping(c2) << (n-2)) + … + grouping(cn)

Note: The list of columns should match with grouping columns exactly, or empty (means all the grouping columns).

```
>>> df.cube("name").agg(grouping_id(), sum("age")).orderBy("name").
+-----+-------------+--------+
| name|grouping_id()|sum(age)|
+-----+-------------+--------+
| null|            1|       7|
|Alice|            0|       2|
|  Bob|            0|       5|
+-----+-------------+--------+
```

New in version 2.0.

pyspark.sql.functions.**hash**(**cols*) [source]

Calculates the hash code of given columns, and returns the result as an int column.

```
>>> spark.createDataFrame([('ABC',)], ['a']).select(hash('a').alias
[Row(hash=-757602832)]
```

New in version 2.0.

pyspark.sql.functions.**hex**(*col*) [source]

Computes hex value of the given column, which could be **pyspark.sql.types.StringType**, **pyspark.sql.types.BinaryType**, **pyspark.sql.types.IntegerType** or **pyspark.sql.types.LongType**.

```
>>> spark.createDataFrame([('ABC', 3)], ['a', 'b']).select(hex('a')
[Row(hex(a)='414243', hex(b)='3')]
```

New in version 1.5.

pyspark.sql.functions.**hour**(*col*) [source]

Extract the hours of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08 13:08:15',)], ['ts'])
>>> df.select(hour('ts').alias('hour')).collect()
[Row(hour=13)]
```

New in version 1.5.

pyspark.sql.functions.**hypot**(*col1*, *col2*)

Computes `sqrt(a^2 + b^2)` without intermediate overflow or underflow.

New in version 1.4.

pyspark.sql.functions.**initcap**(*col*) [source]

Translate the first letter of each word to upper case in the sentence.

```
>>> spark.createDataFrame([('ab cd',)], ['a']).select(initcap("a").
[Row(v='Ab Cd')]
```

New in version 1.5.

pyspark.sql.functions.**input_file_name**() [source]

Creates a string column for the file name of the current Spark task.

New in version 1.6.

pyspark.sql.functions.**instr**(*str*, *substr*) [source]

Locate the position of the first occurrence of substr column in the given string. Returns null if either of the arguments are null.

Note: The position is not zero based, but 1 based index. Returns 0 if substr could

not be found in str.

```
>>> df = spark.createDataFrame([('abcd',)], ['s',])
>>> df.select(instr(df.s, 'b').alias('s')).collect()
[Row(s=2)]
```

New in version 1.5.

`pyspark.sql.functions.`**`isnan`**(*col*) [source]

An expression that returns true iff the column is NaN.

```
>>> df = spark.createDataFrame([(1.0, float('nan')), (float('nan'),
>>> df.select(isnan("a").alias("r1"), isnan(df.a).alias("r2")).col
[Row(r1=False, r2=False), Row(r1=True, r2=True)]
```

New in version 1.6.

`pyspark.sql.functions.`**`isnull`**(*col*) [source]

An expression that returns true iff the column is null.

```
>>> df = spark.createDataFrame([(1, None), (None, 2)], ("a", "b"))
>>> df.select(isnull("a").alias("r1"), isnull(df.a).alias("r2")).c
[Row(r1=False, r2=False), Row(r1=True, r2=True)]
```

New in version 1.6.

`pyspark.sql.functions.`**`json_tuple`**(*col*, **fields*) [source]

Creates a new row for a json column according to the given field names.

Parameters:

- **col** – string column in json format
- **fields** – list of fields to extract

```
>>> data = [("1", '''{"f1": "value1", "f2": "value2"}'''), ("2", ''
>>> df = spark.createDataFrame(data, ("key", "jstring"))
>>> df.select(df.key, json_tuple(df.jstring, 'f1', 'f2')).collect(
[Row(key='1', c0='value1', c1='value2'), Row(key='2', c0='value12',
```

New in version 1.6.

`pyspark.sql.functions.`**`kurtosis`**(*col*)

Aggregate function: returns the kurtosis of the values in a group.

New in version 1.6.

`pyspark.sql.functions.`**`lag`**(*col*, *offset=1*, *default=None*) [source]

Window function: returns the value that is *offset* rows before the current row, and *defaultValue* if there is less than *offset* rows before the current row. For example, an *offset* of one will return the previous row at any given point in the window partition.

This is equivalent to the LAG function in SQL.

Parameters:

- **col** – name of column or expression
- **offset** – number of row to extend
- **default** – default value

New in version 1.4.

`pyspark.sql.functions.`**`last`**(*col*, *ignorenulls=False*) [source]

Aggregate function: returns the last value in a group.

The function by default returns the last values it sees. It will return the last non-null

value it sees when ignoreNulls is set to true. If all values are null, then null is returned.

> **Note:** The function is non-deterministic because its results depends on order of rows which may be non-deterministic after a shuffle.

New in version 1.3.

`pyspark.sql.functions.`**`last_day`**(*date*) [source]

Returns the last day of the month which the given date belongs to.

```
>>> df = spark.createDataFrame([('1997-02-10',)], ['d'])
>>> df.select(last_day(df.d).alias('date')).collect()
[Row(date=datetime.date(1997, 2, 28))]
```

New in version 1.5.

`pyspark.sql.functions.`**`lead`**(*col*, *offset=1*, *default=None*) [source]

Window function: returns the value that is *offset* rows after the current row, and *defaultValue* if there is less than *offset* rows after the current row. For example, an *offset* of one will return the next row at any given point in the window partition.

This is equivalent to the LEAD function in SQL.

Parameters:

- **col** – name of column or expression
- **offset** – number of row to extend
- **default** – default value

New in version 1.4.

`pyspark.sql.functions.`**`least`**(**cols*) [source]

Returns the least value of the list of column names, skipping null values. This function takes at least 2 parameters. It will return null iff all parameters are null.

```
>>> df = spark.createDataFrame([(1, 4, 3)], ['a', 'b', 'c'])
>>> df.select(least(df.a, df.b, df.c).alias("least")).collect()
[Row(least=1)]
```

New in version 1.5.

`pyspark.sql.functions.`**`length`**(*col*) [source]

Computes the character length of string data or number of bytes of binary data. The length of character data includes the trailing spaces. The length of binary data includes binary zeros.

```
>>> spark.createDataFrame([('ABC ',)], ['a']).select(length('a').al
[Row(length=4)]
```

New in version 1.5.

`pyspark.sql.functions.`**`levenshtein`**(*left*, *right*) [source]

Computes the Levenshtein distance of the two given strings.

```
>>> df0 = spark.createDataFrame([('kitten', 'sitting',)], ['l', 'r'
>>> df0.select(levenshtein('l', 'r').alias('d')).collect()
[Row(d=3)]
```

New in version 1.5.

`pyspark.sql.functions.`**`lit`**(*col*)

Creates a **`Column`** of literal value.

```
>>> df.select(lit(5).alias('height')).withColumn('spark_user', lit(
[Row(height=5, spark_user=True)]
```

New in version 1.3.

pyspark.sql.functions.**locate**(*substr*, *str*, *pos=1*) [source]

Locate the position of the first occurrence of substr in a string column, after position pos.

Note: The position is not zero based, but 1 based index. Returns 0 if substr could not be found in str.

Parameters:

- **substr** – a string
- **str** – a Column of **pyspark.sql.types.StringType**
- **pos** – start position (zero based)

```
>>> df = spark.createDataFrame([('abcd',)], ['s',])
>>> df.select(locate('b', df.s, 1).alias('s')).collect()
[Row(s=2)]
```

New in version 1.5.

pyspark.sql.functions.**log**(*arg1*, *arg2=None*) [source]

Returns the first argument-based logarithm of the second argument.

If there is only one argument, then this takes the natural logarithm of the argument.

```
>>> df.select(log(10.0, df.age).alias('ten')).rdd.map(lambda l: str
['0.30102', '0.69897']
```

```
>>> df.select(log(df.age).alias('e')).rdd.map(lambda l: str(l.e)[:
['0.69314', '1.60943']
```

New in version 1.5.

pyspark.sql.functions.**log10**(*col*)

Computes the logarithm of the given value in Base 10.

New in version 1.4.

pyspark.sql.functions.**log1p**(*col*)

Computes the natural logarithm of the given value plus one.

New in version 1.4.

pyspark.sql.functions.**log2**(*col*) [source]

Returns the base-2 logarithm of the argument.

```
>>> spark.createDataFrame([(4,)], ['a']).select(log2('a').alias('lo
[Row(log2=2.0)]
```

New in version 1.5.

pyspark.sql.functions.**lower**(*col*)

Converts a string expression to lower case.

New in version 1.5.

pyspark.sql.functions.**lpad**(*col*, *len*, *pad*) [source]

Left-pad the string column to width *len* with *pad*.

```
>>> df = spark.createDataFrame([('abcd',)], ['s',])
>>> df.select(lpad(df.s, 6, '#').alias('s')).collect()
[Row(s='##abcd')]
```

New in version 1.5.

pyspark.sql.functions.**ltrim**(*col*)

Trim the spaces from left end for the specified string value.

New in version 1.5.

pyspark.sql.functions.**map_concat**(**cols*) [source]

Returns the union of all the given maps.

Parameters:

cols – list of column names (string) or list of `Column` expressions

```
>>> from pyspark.sql.functions import map_concat
>>> df = spark.sql("SELECT map(1, 'a', 2, 'b') as map1, map(3, 'c',
>>> df.select(map_concat("map1", "map2").alias("map3")).show(trunc
+------------------------+
|map3                    |
+------------------------+
|[1 -> d, 2 -> b, 3 -> c]|
+------------------------+
```

New in version 2.4.

pyspark.sql.functions.**map_entries**(*col*) [source]

Collection function: Returns an unordered array of all entries in the given map.

Parameters:

col – name of column or expression

```
>>> from pyspark.sql.functions import map_entries
>>> df = spark.sql("SELECT map(1, 'a', 2, 'b') as data")
>>> df.select(map_entries("data").alias("entries")).show()
+----------------+
|         entries|
+----------------+
|[[1, a], [2, b]]|
+----------------+
```

New in version 3.0.

pyspark.sql.functions.**map_from_arrays**(*col1*, *col2*) [source]

Creates a new map from two arrays.

Parameters:

- **col1** – name of column containing a set of keys. All elements should not be null
- **col2** – name of column containing a set of values

```
>>> df = spark.createDataFrame([([2, 5], ['a', 'b'])], ['k', 'v'])
>>> df.select(map_from_arrays(df.k, df.v).alias("map")).show()
+----------------+
|             map|
+----------------+
|[2 -> a, 5 -> b]|
+----------------+
```

New in version 2.4.

pyspark.sql.functions.**map_from_entries**(*col*) [source]

Collection function: Returns a map created from the given array of entries.

Parameters:
col – name of column or expression

```
>>> from pyspark.sql.functions import map_from_entries
>>> df = spark.sql("SELECT array(struct(1, 'a'), struct(2, 'b')) as
>>> df.select(map_from_entries("data").alias("map")).show()
+----------------+
|             map|
+----------------+
|[1 -> a, 2 -> b]|
+----------------+
```

New in version 2.4.

pyspark.sql.functions.**map_keys**(*col*) [source]

Collection function: Returns an unordered array containing the keys of the map.

Parameters:
col – name of column or expression

```
>>> from pyspark.sql.functions import map_keys
>>> df = spark.sql("SELECT map(1, 'a', 2, 'b') as data")
>>> df.select(map_keys("data").alias("keys")).show()
+------+
|  keys|
+------+
|[1, 2]|
+------+
```

New in version 2.3.

pyspark.sql.functions.**map_values**(*col*) [source]

Collection function: Returns an unordered array containing the values of the map.

Parameters:
col – name of column or expression

```
>>> from pyspark.sql.functions import map_values
>>> df = spark.sql("SELECT map(1, 'a', 2, 'b') as data")
>>> df.select(map_values("data").alias("values")).show()
+------+
|values|
+------+
|[a, b]|
+------+
```

New in version 2.3.

pyspark.sql.functions.**max**(*col*)

Aggregate function: returns the maximum value of the expression in a group.

New in version 1.3.

pyspark.sql.functions.**md5**(*col*) [source]

Calculates the MD5 digest and returns the value as a 32 character hex string.

```
>>> spark.createDataFrame([('ABC',)], ['a']).select(md5('a').alias(
[Row(hash='902fbdd2b1df0c4f70b4a5d23525e932')]
```

New in version 1.5.

pyspark.sql.functions.**mean**(*col*)

Aggregate function: returns the average of the values in a group.

New in version 1.3.

pyspark.sql.functions.**min**(*col*)

Aggregate function: returns the minimum value of the expression in a group.

New in version 1.3.

pyspark.sql.functions.**minute**(*col*) [source]

Extract the minutes of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08 13:08:15',)], ['ts'])
>>> df.select(minute('ts').alias('minute')).collect()
[Row(minute=8)]
```

New in version 1.5.

pyspark.sql.functions.**monotonically_increasing_id**() [source]

A column that generates monotonically increasing 64-bit integers.

The generated ID is guaranteed to be monotonically increasing and unique, but not consecutive. The current implementation puts the partition ID in the upper 31 bits, and the record number within each partition in the lower 33 bits. The assumption is that the data frame has less than 1 billion partitions, and each partition has less than 8 billion records.

Note: The function is non-deterministic because its result depends on partition IDs.

As an example, consider a **DataFrame** with two partitions, each with 3 records. This expression would return the following IDs: 0, 1, 2, 8589934592 (1L << 33), 8589934593, 8589934594.

```
>>> df0 = sc.parallelize(range(2), 2).mapPartitions(lambda x: [(1,
>>> df0.select(monotonically_increasing_id().alias('id')).collect()
[Row(id=0), Row(id=1), Row(id=2), Row(id=8589934592), Row(id=858993
```

New in version 1.6.

pyspark.sql.functions.**month**(*col*) [source]

Extract the month of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> df.select(month('dt').alias('month')).collect()
[Row(month=4)]
```

New in version 1.5.

pyspark.sql.functions.**months_between**(*date1*, *date2*, *roundOff=True*) [source]

Returns number of months between dates date1 and date2. If date1 is later than date2, then the result is positive. If date1 and date2 are on the same day of month, or both are the last day of month, returns an integer (time of day will be ignored). The result is rounded off to 8 digits unless *roundOff* is set to *False*.

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00', '1996-10-30
>>> df.select(months_between(df.date1, df.date2).alias('months')).c
[Row(months=3.94959677)]
>>> df.select(months_between(df.date1, df.date2, False).alias('mon
[Row(months=3.9495967741935485)]
```

New in version 1.5.

pyspark.sql.functions.**nanvl**(*col1*, *col2*) [source]

Returns col1 if it is not NaN, or col2 if col1 is NaN.

Both inputs should be floating point columns (**DoubleType** or **FloatType**).

```
>>> df = spark.createDataFrame([(1.0, float('nan')), (float('nan'),
>>> df.select(nanvl("a", "b").alias("r1"), nanvl(df.a, df.b).alias
[Row(r1=1.0, r2=1.0), Row(r1=2.0, r2=2.0)]
```

New in version 1.6.

pyspark.sql.functions.**next_day**(*date*, *dayOfWeek*) [source]

Returns the first date which is later than the value of the date column.

Day of the week parameter is case insensitive, and accepts:

"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun".

```
>>> df = spark.createDataFrame([('2015-07-27',)], ['d'])
>>> df.select(next_day(df.d, 'Sun').alias('date')).collect()
[Row(date=datetime.date(2015, 8, 2))]
```

New in version 1.5.

pyspark.sql.functions.**ntile**(*n*) [source]

Window function: returns the ntile group id (from 1 to *n* inclusive) in an ordered window partition. For example, if *n* is 4, the first quarter of the rows will get value 1, the second quarter will get 2, the third quarter will get 3, and the last quarter will get 4.

This is equivalent to the NTILE function in SQL.

Parameters:
n – an integer

New in version 1.4.

pyspark.sql.functions.**pandas_udf**(*f=None*, *returnType=None*, *functionType=None*) [source]

Creates a vectorized user defined function (UDF).

Parameters:

- **f** – user-defined function. A python function if used as a standalone function
- **returnType** – the return type of the user-defined function. The value can be either a **pyspark.sql.types.DataType** object or a DDL-formatted type string.
- **functionType** – an enum value in **pyspark.sql.functions.PandasUDFType**. Default: SCALAR.

The function type of the UDF can be one of the following:

1. SCALAR

   A scalar UDF defines a transformation: One or more *pandas.Series* -> A *pandas.Series*. The length of the returned *pandas.Series* must be of the same as the input *pandas.Series*. If the return type is **StructType**, the returned value should be a *pandas.DataFrame*.

   MapType, nested **StructType** are currently not supported as output types.

   Scalar UDFs can be used with **pyspark.sql.DataFrame.withColumn()** and **pyspark.sql.DataFrame.select()**.

```
>>> from pyspark.sql.functions import pandas_udf, PandasUDFTyp
>>> from pyspark.sql.types import IntegerType, StringType
>>> slen = pandas_udf(lambda s: s.str.len(), IntegerType())
>>> @pandas_udf(StringType())
... def to_upper(s):
...     return s.str.upper()
...
>>> @pandas_udf("integer", PandasUDFType.SCALAR)
... def add_one(x):
...     return x + 1
...
>>> df = spark.createDataFrame([(1, "John Doe", 21)],
...                            ("id", "name", "age"))
>>> df.select(slen("name").alias("slen(name)"), to_upper("nam
...     .show()
+----------+--------------+------------+
|slen(name)|to_upper(name)|add_one(age)|
+----------+--------------+------------+
|         8|      JOHN DOE|          22|
+----------+--------------+------------+
>>> @pandas_udf("first string, last string")
... def split_expand(n):
...     return n.str.split(expand=True)
>>> df.select(split_expand("name")).show()
+------------------+
|split_expand(name)|
+------------------+
|       [John, Doe]|
+------------------+
```

Note: The length of *pandas.Series* within a scalar UDF is not that of the whole input column, but is the length of an internal batch used for each call to the function. Therefore, this can be used, for example, to ensure the length of each returned *pandas.Series*, and can not be used as the column length.

2. SCALAR_ITER

A scalar iterator UDF is semantically the same as the scalar Pandas UDF above except that the wrapped Python function takes an iterator of batches as input instead of a single batch and, instead of returning a single output batch, it yields output batches or explicitly returns an generator or an iterator of output batches. It is useful when the UDF execution requires initializing some state, e.g., loading a machine learning model file to apply inference to every input batch.

Note: It is not guaranteed that one invocation of a scalar iterator UDF will process all batches from one partition, although it is currently implemented this way. Your code shall not rely on this behavior because it might change in the future for further optimization, e.g., one invocation processes multiple partitions.

Scalar iterator UDFs are used with **`pyspark.sql.DataFrame.withColumn()`** and **`pyspark.sql.DataFrame.select()`**.

```
>>> import pandas as pd
>>> from pyspark.sql.functions import col, pandas_udf, struct,
>>> pdf = pd.DataFrame([1, 2, 3], columns=["x"])
>>> df = spark.createDataFrame(pdf)
```

When the UDF is called with a single column that is not *StructType*, the input to the underlying function is an iterator of *pd.Series*.

```
>>> @pandas_udf("long", PandasUDFType.SCALAR_ITER)
... def plus_one(batch_iter):
...     for x in batch_iter:
...         yield x + 1
...
>>> df.select(plus_one(col("x"))).show()
+-----------+
|plus_one(x)|
+-----------+
|          2|
|          3|
|          4|
+-----------+
```

When the UDF is called with more than one columns, the input to the underlying function is an iterator of *pd.Series* tuple.

```
>>> @pandas_udf("long", PandasUDFType.SCALAR_ITER)
... def multiply_two_cols(batch_iter):
...     for a, b in batch_iter:
...         yield a * b
...
>>> df.select(multiply_two_cols(col("x"), col("x"))).show()
+-----------------------+
|multiply_two_cols(x, x)|
+-----------------------+
|                      1|
|                      4|
|                      9|
+-----------------------+
```

When the UDF is called with a single column that is *StructType*, the input to the underlying function is an iterator of *pd.DataFrame*.

```
>>> @pandas_udf("long", PandasUDFType.SCALAR_ITER)
... def multiply_two_nested_cols(pdf_iter):
...    for pdf in pdf_iter:
...        yield pdf["a"] * pdf["b"]
...
>>> df.select(
...     multiply_two_nested_cols(
...         struct(col("x").alias("a"), col("x").alias("b"))
...     ).alias("y")
... ).show()
+---+
|  y|
+---+
|  1|
|  4|
|  9|
+---+
```

In the UDF, you can initialize some states before processing batches, wrap your code with *try … finally …* or use context managers to ensure the release of resources at the end or in case of early termination.

```
>>> y_bc = spark.sparkContext.broadcast(1)
>>> @pandas_udf("long", PandasUDFType.SCALAR_ITER)
... def plus_y(batch_iter):
...     y = y_bc.value  # initialize some state
...     try:
...         for x in batch_iter:
...             yield x + y
...     finally:
...         pass  # release resources here, if any
...
>>> df.select(plus_y(col("x"))).show()
+---------+
|plus_y(x)|
+---------+
|        2|
|        3|
|        4|
+---------+
```

3. GROUPED_MAP

A grouped map UDF defines transformation: A *pandas.DataFrame* -> A *pandas.DataFrame* The returnType should be a **`StructType`** describing the schema of the returned *pandas.DataFrame*. The column labels of the returned *pandas.DataFrame* must either match the field names in the defined returnType schema if specified as strings, or match the field data types by position if not strings, e.g. integer indices. The length of the returned *pandas.DataFrame* can be arbitrary.

Grouped map UDFs are used with **`pyspark.sql.GroupedData.apply()`**.

```
>>> from pyspark.sql.functions import pandas_udf, PandasUDFTyp
>>> df = spark.createDataFrame(
...     [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)],
...     ("id", "v"))
>>> @pandas_udf("id long, v double", PandasUDFType.GROUPED_MAP
... def normalize(pdf):
...     v = pdf.v
...     return pdf.assign(v=(v - v.mean()) / v.std())
>>> df.groupby("id").apply(normalize).show()
+---+-------------------+
| id|                  v|
+---+-------------------+
|  1|-0.7071067811865475|
|  1| 0.7071067811865475|
|  2|-0.8320502943378437|
|  2|-0.2773500981126146|
|  2| 1.1094003924504583|
+---+-------------------+
```

Alternatively, the user can define a function that takes two arguments. In this case, the grouping key(s) will be passed as the first argument and the data will be passed as the second argument. The grouping key(s) will be passed as a tuple of numpy data types, e.g., *numpy.int32* and *numpy.float64*. The data will still be passed in as a *pandas.DataFrame* containing all columns from the original Spark DataFrame. This is useful when the user does not want to hardcode grouping key(s) in the function.

```
>>> import pandas as pd
>>> from pyspark.sql.functions import pandas_udf, PandasUDFTyp
>>> df = spark.createDataFrame(
...     [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)],
...     ("id", "v"))
>>> @pandas_udf("id long, v double", PandasUDFType.GROUPED_MAP
... def mean_udf(key, pdf):
...     # key is a tuple of one numpy.int64, which is the valu
...     # of 'id' for the current group
...     return pd.DataFrame([key + (pdf.v.mean(),)])
>>> df.groupby('id').apply(mean_udf).show()
+---+---+
| id|  v|
+---+---+
|  1|1.5|
|  2|6.0|
+---+---+
>>> @pandas_udf(
...    "id long, `ceil(v / 2)` long, v double",
...    PandasUDFType.GROUPED_MAP)
>>> def sum_udf(key, pdf):
...     # key is a tuple of two numpy.int64s, which is the val
...     # of 'id' and 'ceil(df.v / 2)' for the current group
...     return pd.DataFrame([key + (pdf.v.sum(),)])
>>> df.groupby(df.id, ceil(df.v / 2)).apply(sum_udf).show()
+---+-----------+----+
| id|ceil(v / 2)|   v|
+---+-----------+----+
|  2|          5|10.0|
|  1|          1| 3.0|
|  2|          3| 5.0|
|  2|          2| 3.0|
+---+-----------+----+
```

Note: If returning a new *pandas.DataFrame* constructed with a dictionary,

it is recommended to explicitly index the columns by name to ensure the positions are correct, or alternatively use an *OrderedDict*. For example, *pd.DataFrame({'id': ids, 'a': data}, columns=['id', 'a'])* or *pd.DataFrame(OrderedDict([('id', ids), ('a', data)]))*.

See also: **`pyspark.sql.GroupedData.apply()`**

4. GROUPED_AGG

   A grouped aggregate UDF defines a transformation: One or more *pandas.Series* -> A scalar The *returnType* should be a primitive data type, e.g., **`DoubleType`**. The returned scalar can be either a python primitive type, e.g., *int* or *float* or a numpy data type, e.g., *numpy.int64* or *numpy.float64*.

   `MapType` and **`StructType`** are currently not supported as output types.

   Group aggregate UDFs are used with **`pyspark.sql.GroupedData.agg()`** and **`pyspark.sql.Window`**

   This example shows using grouped aggregated UDFs with groupby:

```
>>> from pyspark.sql.functions import pandas_udf, PandasUDFTyp
>>> df = spark.createDataFrame(
...     [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)],
...     ("id", "v"))
>>> @pandas_udf("double", PandasUDFType.GROUPED_AGG)
... def mean_udf(v):
...     return v.mean()
>>> df.groupby("id").agg(mean_udf(df['v'])).show()
+---+-----------+
| id|mean_udf(v)|
+---+-----------+
|  1|        1.5|
|  2|        6.0|
+---+-----------+
```

   This example shows using grouped aggregated UDFs as window functions.

```
>>> from pyspark.sql.functions import pandas_udf, PandasUDFTyp
>>> from pyspark.sql import Window
>>> df = spark.createDataFrame(
...     [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)],
...     ("id", "v"))
>>> @pandas_udf("double", PandasUDFType.GROUPED_AGG)
... def mean_udf(v):
...     return v.mean()
>>> w = (Window.partitionBy('id')
...            .orderBy('v')
...            .rowsBetween(-1, 0))
>>> df.withColumn('mean_v', mean_udf(df['v']).over(w)).show()
+---+----+------+
| id|   v|mean_v|
+---+----+------+
|  1| 1.0|   1.0|
|  1| 2.0|   1.5|
|  2| 3.0|   3.0|
|  2| 5.0|   4.0|
|  2|10.0|   7.5|
+---+----+------+
```

   Note: For performance reasons, the input series to window functions are not copied. Therefore, mutating the input series is not allowed and will cause incorrect results. For the same reason, users should also not rely on the index of the input series.

   See also: **`pyspark.sql.GroupedData.agg()`** and **`pyspark.sql.Window`**

5. MAP_ITER

   A map iterator Pandas UDFs are used to transform data with an iterator of

batches. It can be used with **`pyspark.sql.DataFrame.mapInPandas()`**.

It can return the output of arbitrary length in contrast to the scalar Pandas UDF. It maps an iterator of batches in the current **`DataFrame`** using a Pandas user-defined function and returns the result as a **`DataFrame`**.

The user-defined function should take an iterator of *pandas.DataFrame*s and return another iterator of *pandas.DataFrame*s. All columns are passed together as an iterator of *pandas.DataFrame*s to the user-defined function and the returned iterator of *pandas.DataFrame*s are combined as a **`DataFrame`**.

```
>>> df = spark.createDataFrame([(1, 21), (2, 30)],
...                            ("id", "age"))
>>> @pandas_udf(df.schema, PandasUDFType.MAP_ITER)
... def filter_func(batch_iter):
...     for pdf in batch_iter:
...         yield pdf[pdf.id == 1]
>>> df.mapInPandas(filter_func).show()
+---+---+
| id|age|
+---+---+
|  1| 21|
+---+---+
```

6. COGROUPED_MAP

A cogrouped map UDF defines transformation: (*pandas.DataFrame*, *pandas.DataFrame*) -> *pandas.DataFrame*. The *returnType* should be a **`StructType`** describing the schema of the returned *pandas.DataFrame*. The column labels of the returned *pandas.DataFrame* must either match the field names in the defined *returnType* schema if specified as strings, or match the field data types by position if not strings, e.g. integer indices. The length of the returned *pandas.DataFrame* can be arbitrary.

CoGrouped map UDFs are used with **`pyspark.sql.CoGroupedData.apply()`**.

```
>>> from pyspark.sql.functions import pandas_udf, PandasUDFTyp
>>> df1 = spark.createDataFrame(
...     [(20000101, 1, 1.0), (20000101, 2, 2.0), (20000102, 1
...     ("time", "id", "v1"))
>>> df2 = spark.createDataFrame(
...     [(20000101, 1, "x"), (20000101, 2, "y")],
...     ("time", "id", "v2"))
>>> @pandas_udf("time int, id int, v1 double, v2 string",
...             PandasUDFType.COGROUPED_MAP)
... def asof_join(l, r):
...     return pd.merge_asof(l, r, on="time", by="id")
>>> df1.groupby("id").cogroup(df2.groupby("id")).apply(asof_j
+---------+---+---+---+
|     time| id| v1| v2|
+---------+---+---+---+
| 20000101|  1|1.0|  x|
| 20000102|  1|3.0|  x|
| 20000101|  2|2.0|  y|
| 20000102|  2|4.0|  y|
+---------+---+---+---+
```

Alternatively, the user can define a function that takes three arguments. In this case, the grouping key(s) will be passed as the first argument and the data will be passed as the second and third arguments. The grouping key(s) will be passed as a tuple of numpy data types, e.g., *numpy.int32* and *numpy.float64*. The data will still be passed in as two *pandas.DataFrame* containing all columns from the original Spark DataFrames. >>> @pandas_udf(“time int, id int, v1 double, v2 string”, … PandasUDFType.COGROUPED_MAP) # doctest: +SKIP … def asof_join(k, l, r): … if k == (1,): … return pd.merge_asof(l, r, on=”time”, by=”id”) … else: … return pd.DataFrame(columns=[‘time’, ‘id’, ‘v1’, ‘v2’]) >>> df1.groupby(“id”).cogroup(df2.groupby(“id”)).apply(asof_join).show() # doctest: +SKIP +———+—+—+—+ | time| id| v1| v2| +———+—+—+—+ | 20000101| 1|1.0| x| | 20000102| 1|3.0| x| +———+—+—+—+

Note: The user-defined functions are considered deterministic by default. Due to optimization, duplicate invocations may be eliminated or the function may even be invoked more times than it is present in the query. If your function is not deterministic, call *asNondeterministic* on the user defined function. E.g.:

```
>>> @pandas_udf('double', PandasUDFType.SCALAR)
... def random(v):
...     import numpy as np
...     import pandas as pd
...     return pd.Series(np.random.randn(len(v))
>>> random = random.asNondeterministic()
```

Note: The user-defined functions do not support conditional expressions or short circuiting in boolean expressions and it ends up with being executed all internally. If the functions can fail on special rows, the workaround is to incorporate the condition into the functions.

Note: The user-defined functions do not take keyword arguments on the calling side.

Note: The data type of returned *pandas.Series* from the user-defined functions should be matched with defined returnType (see **`types.to_arrow_type()`** and **`types.from_arrow_type()`**). When there is mismatch between them, Spark might do conversion on returned data. The conversion is not guaranteed to be correct and results should be checked for accuracy by users.

New in version 2.3.

`pyspark.sql.functions.`**`percent_rank`**`()`

Window function: returns the relative rank (i.e. percentile) of rows within a window partition.

New in version 1.6.

`pyspark.sql.functions.`**`posexplode`**`(`*col*`)` [source]

Returns a new row for each element with position in the given array or map. Uses the default column name *pos* for position, and *col* for elements in the array and *key* and *value* for elements in the map unless specified otherwise.

```
>>> from pyspark.sql import Row
>>> eDF = spark.createDataFrame([Row(a=1, intlist=[1,2,3], mapfiel
>>> eDF.select(posexplode(eDF.intlist)).collect()
[Row(pos=0, col=1), Row(pos=1, col=2), Row(pos=2, col=3)]
```

```
>>> eDF.select(posexplode(eDF.mapfield)).show()
+---+---+-----+
|pos|key|value|
+---+---+-----+
|  0|  a|    b|
+---+---+-----+
```

New in version 2.1.

`pyspark.sql.functions.`**`posexplode_outer`**`(`*col*`)` [source]

Returns a new row for each element with position in the given array or map. Unlike posexplode, if the array/map is null or empty then the row (null, null) is produced. Uses the default column name *pos* for position, and *col* for elements in the array and *key* and *value* for elements in the map unless specified otherwise.

```
>>> df = spark.createDataFrame(
...     [(1, ["foo", "bar"], {"x": 1.0}), (2, [], {}), (3, None, N
...     ("id", "an_array", "a_map")
... )
>>> df.select("id", "an_array", posexplode_outer("a_map")).show()
+---+----------+----+----+-----+
| id|  an_array| pos| key|value|
+---+----------+----+----+-----+
|  1|[foo, bar]|   0|   x|  1.0|
|  2|        []|null|null| null|
|  3|      null|null|null| null|
+---+----------+----+----+-----+
>>> df.select("id", "a_map", posexplode_outer("an_array")).show()
+---+----------+----+----+
| id|     a_map| pos| col|
+---+----------+----+----+
|  1|[x -> 1.0]|   0| foo|
|  1|[x -> 1.0]|   1| bar|
|  2|        []|null|null|
|  3|      null|null|null|
+---+----------+----+----+
```

New in version 2.3.

pyspark.sql.functions.**pow**(*col1*, *col2*)

Returns the value of the first argument raised to the power of the second argument.

New in version 1.4.

pyspark.sql.functions.**quarter**(*col*) [source]

Extract the quarter of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> df.select(quarter('dt').alias('quarter')).collect()
[Row(quarter=2)]
```

New in version 1.5.

pyspark.sql.functions.**radians**(*col*)

Converts an angle measured in degrees to an approximately equivalent angle measured in radians. :param col: angle in degrees :return: angle in radians, as if computed by *java.lang.Math.toRadians()*

New in version 2.1.

pyspark.sql.functions.**rand**(*seed=None*) [source]

Generates a random column with independent and identically distributed (i.i.d.) samples from U[0.0, 1.0].

Note: The function is non-deterministic in general case.

```
>>> df.withColumn('rand', rand(seed=42) * 3).collect()
[Row(age=2, name='Alice', rand=2.4052597283576684),
 Row(age=5, name='Bob', rand=2.3913904055683974)]
```

New in version 1.4.

pyspark.sql.functions.**randn**(*seed=None*) [source]

Generates a column with independent and identically distributed (i.i.d.) samples from the standard normal distribution.

Note: The function is non-deterministic in general case.

```
>>> df.withColumn('randn', randn(seed=42)).collect()
[Row(age=2, name='Alice', randn=1.1027054481455365),
Row(age=5, name='Bob', randn=0.7400395449950132)]
```

New in version 1.4.

pyspark.sql.functions.**rank**()

Window function: returns the rank of rows within a window partition.

The difference between rank and dense_rank is that dense_rank leaves no gaps in ranking sequence when there are ties. That is, if you were ranking a competition using dense_rank and had three people tie for second place, you would say that all three were in second place and that the next person came in third. Rank would give me sequential numbers, making the person that came in third place (after the ties) would register as coming in fifth.

This is equivalent to the RANK function in SQL.

New in version 1.6.

pyspark.sql.functions.**regexp_extract**(*str*, *pattern*, *idx*) [source]

Extract a specific group matched by a Java regex, from the specified string column. If the regex did not match, or the specified group did not match, an empty string is returned.

```
>>> df = spark.createDataFrame([('100-200',)], ['str'])
>>> df.select(regexp_extract('str', r'(\d+)-(\d+)', 1).alias('d'))
[Row(d='100')]
>>> df = spark.createDataFrame([('foo',)], ['str'])
>>> df.select(regexp_extract('str', r'(\d+)', 1).alias('d')).colle
[Row(d='')]
>>> df = spark.createDataFrame([('aaaac',)], ['str'])
>>> df.select(regexp_extract('str', '(a+)(b)?(c)', 2).alias('d')).
[Row(d='')]
```

New in version 1.5.

pyspark.sql.functions.**regexp_replace**(*str*, *pattern*, *replacement*) [source]

Replace all substrings of the specified string value that match regexp with rep.

```
>>> df = spark.createDataFrame([('100-200',)], ['str'])
>>> df.select(regexp_replace('str', r'(\d+)', '--').alias('d')).co
[Row(d='-----')]
```

New in version 1.5.

pyspark.sql.functions.**repeat**(*col*, *n*) [source]

Repeats a string column n times, and returns it as a new string column.

```
>>> df = spark.createDataFrame([('ab',)], ['s',])
>>> df.select(repeat(df.s, 3).alias('s')).collect()
[Row(s='ababab')]
```

New in version 1.5.

pyspark.sql.functions.**reverse**(*col*) [source]

Collection function: returns a reversed string or an array with reverse order of elements.

Parameters:
col – name of column or expression

```
>>> df = spark.createDataFrame([('Spark SQL',)], ['data'])
>>> df.select(reverse(df.data).alias('s')).collect()
[Row(s='LQS krapS')]
>>> df = spark.createDataFrame([([2, 1, 3],) ,([1],) ,([],)], ['dat
>>> df.select(reverse(df.data).alias('r')).collect()
[Row(r=[3, 1, 2]), Row(r=[1]), Row(r=[])]
```

New in version 1.5.

pyspark.sql.functions.**rint**(*col*)

Returns the double value that is closest in value to the argument and is equal to a mathematical integer.

New in version 1.4.

pyspark.sql.functions.**round**(*col*, *scale=0*) [source]

Round the given value to *scale* decimal places using HALF_UP rounding mode if *scale* >= 0 or at integral part when *scale* < 0.

```
>>> spark.createDataFrame([(2.5,)], ['a']).select(round('a', 0).ali
[Row(r=3.0)]
```

New in version 1.5.

pyspark.sql.functions.**row_number**()

Window function: returns a sequential number starting at 1 within a window partition.

New in version 1.6.

pyspark.sql.functions.**rpad**(*col*, *len*, *pad*) [source]

Right-pad the string column to width *len* with *pad*.

```
>>> df = spark.createDataFrame([('abcd',)], ['s',])
>>> df.select(rpad(df.s, 6, '#').alias('s')).collect()
[Row(s='abcd##')]
```

New in version 1.5.

pyspark.sql.functions.**rtrim**(*col*)

Trim the spaces from right end for the specified string value.

New in version 1.5.

pyspark.sql.functions.**schema_of_csv**(*csv*, *options={}*) [source]

Parses a CSV string and infers its schema in DDL format.

Parameters:

- **col** – a CSV string or a string literal containing a CSV string.
- **options** – options to control parsing. accepts the same options as the CSV datasource

```
>>> df = spark.range(1)
>>> df.select(schema_of_csv(lit('1|a'), {'sep':'|'}).alias("csv"))
[Row(csv='struct<_c0:int,_c1:string>')]
>>> df.select(schema_of_csv('1|a', {'sep':'|'}).alias("csv")).colle
[Row(csv='struct<_c0:int,_c1:string>')]
```

New in version 3.0.

pyspark.sql.functions.**schema_of_json**(*json*, *options={}*) [source]

Parses a JSON string and infers its schema in DDL format.

Parameters:

- **json** – a JSON string or a string literal containing a JSON string.
- **options** – options to control parsing. accepts the same options as the JSON datasource

Changed in version 3.0: It accepts *options* parameter to control schema inferring.

```
>>> df = spark.range(1)
>>> df.select(schema_of_json(lit('{"a": 0}')).alias("json")).collec
[Row(json='struct<a:bigint>')]
>>> schema = schema_of_json('{a: 1}', {'allowUnquotedFieldNames':'t
>>> df.select(schema.alias("json")).collect()
[Row(json='struct<a:bigint>')]
```

New in version 2.4.

`pyspark.sql.functions.`**`second`**(*col*) [source]

Extract the seconds of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08 13:08:15',)], ['ts'])
>>> df.select(second('ts').alias('second')).collect()
[Row(second=15)]
```

New in version 1.5.

`pyspark.sql.functions.`**`sequence`**(*start*, *stop*, *step=None*) [source]

Generate a sequence of integers from *start* to *stop*, incrementing by *step*. If *step* is not set, incrementing by 1 if *start* is less than or equal to *stop*, otherwise -1.

```
>>> df1 = spark.createDataFrame([(-2, 2)], ('C1', 'C2'))
>>> df1.select(sequence('C1', 'C2').alias('r')).collect()
[Row(r=[-2, -1, 0, 1, 2])]
>>> df2 = spark.createDataFrame([(4, -4, -2)], ('C1', 'C2', 'C3'))
>>> df2.select(sequence('C1', 'C2', 'C3').alias('r')).collect()
[Row(r=[4, 2, 0, -2, -4])]
```

New in version 2.4.

`pyspark.sql.functions.`**`sha1`**(*col*) [source]

Returns the hex string result of SHA-1.

```
>>> spark.createDataFrame([('ABC',)], ['a']).select(sha1('a').alias
[Row(hash='3c01bdbb26f358bab27f267924aa2c9a03fcfdb8')]
```

New in version 1.5.

`pyspark.sql.functions.`**`sha2`**(*col*, *numBits*) [source]

Returns the hex string result of SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, and SHA-512). The numBits indicates the desired bit length of the result, which must have a value of 224, 256, 384, 512, or 0 (which is equivalent to 256).

```
>>> digests = df.select(sha2(df.name, 256).alias('s')).collect()
>>> digests[0]
Row(s='3bc51062973c458d5a6f2d8d64a023246354ad7e064b1e4e009ec8a0699a
>>> digests[1]
Row(s='cd9fb1e148ccd8442e5aa74904cc73bf6fb54d1d54d333bd596aa9bb4bb4
```

New in version 1.5.

`pyspark.sql.functions.`**`shiftLeft`**(*col*, *numBits*) [source]

Shift the given value numBits left.

```
>>> spark.createDataFrame([(21,)], ['a']).select(shiftLeft('a', 1).
[Row(r=42)]
```

New in version 1.5.

pyspark.sql.functions.**shiftRight**(*col*, *numBits*) [source]

(Signed) shift the given value numBits right.

```
>>> spark.createDataFrame([(42,)], ['a']).select(shiftRight('a', 1)
[Row(r=21)]
```

New in version 1.5.

pyspark.sql.functions.**shiftRightUnsigned**(*col*, *numBits*) [source]

Unsigned shift the given value numBits right.

```
>>> df = spark.createDataFrame([(-42,)], ['a'])
>>> df.select(shiftRightUnsigned('a', 1).alias('r')).collect()
[Row(r=9223372036854775787)]
```

New in version 1.5.

pyspark.sql.functions.**shuffle**(*col*) [source]

Collection function: Generates a random permutation of the given array.

Note: The function is non-deterministic.

Parameters:
col – name of column or expression

```
>>> df = spark.createDataFrame([([1, 20, 3, 5],), ([1, 20, None, 3
>>> df.select(shuffle(df.data).alias('s')).collect()
[Row(s=[3, 1, 5, 20]), Row(s=[20, None, 3, 1])]
```

New in version 2.4.

pyspark.sql.functions.**signum**(*col*)

Computes the signum of the given value.

New in version 1.4.

pyspark.sql.functions.**sin**(*col*)

Parameters:
col – angle in radians
Returns:
sine of the angle, as if computed by *java.lang.Math.sin()*

New in version 1.4.

pyspark.sql.functions.**sinh**(*col*)

Parameters:
col – hyperbolic angle
Returns:
hyperbolic sine of the given value, as if computed by *java.lang.Math.sinh()*

New in version 1.4.

pyspark.sql.functions.**size**(*col*) [source]

Collection function: returns the length of the array or map stored in the column.

Parameters:
col – name of column or expression

```
>>> df = spark.createDataFrame([([1, 2, 3],),([1],),([],)], ['data
>>> df.select(size(df.data)).collect()
[Row(size(data)=3), Row(size(data)=1), Row(size(data)=0)]
```

New in version 1.5.

pyspark.sql.functions.**skewness**(*col*)

Aggregate function: returns the skewness of the values in a group.

New in version 1.6.

pyspark.sql.functions.**slice**(*x*, *start*, *length*) [source]

Collection function: returns an array containing all the elements in *x* from index *start* (array indices start at 1, or from the end if *start* is negative) with the specified *length*.

Parameters:

- **x** – the array to be sliced
- **start** – the starting index
- **length** – the length of the slice

```
>>> df = spark.createDataFrame([([1, 2, 3],), ([4, 5],)], ['x'])
>>> df.select(slice(df.x, 2, 2).alias("sliced")).collect()
[Row(sliced=[2, 3]), Row(sliced=[5])]
```

New in version 2.4.

pyspark.sql.functions.**sort_array**(*col*, *asc=True*) [source]

Collection function: sorts the input array in ascending or descending order according to the natural ordering of the array elements. Null elements will be placed at the beginning of the returned array in ascending order or at the end of the returned array in descending order.

Parameters:

col – name of column or expression

```
>>> df = spark.createDataFrame([([2, 1, None, 3],),([1],),([],)],
>>> df.select(sort_array(df.data).alias('r')).collect()
[Row(r=[None, 1, 2, 3]), Row(r=[1]), Row(r=[])]
>>> df.select(sort_array(df.data, asc=False).alias('r')).collect()
[Row(r=[3, 2, 1, None]), Row(r=[1]), Row(r=[])]
```

New in version 1.5.

pyspark.sql.functions.**soundex**(*col*) [source]

Returns the SoundEx encoding for a string

```
>>> df = spark.createDataFrame([("Peters",),("Uhrbach",)], ['name']
>>> df.select(soundex(df.name).alias("soundex")).collect()
[Row(soundex='P362'), Row(soundex='U612')]
```

New in version 1.5.

pyspark.sql.functions.**spark_partition_id**() [source]

A column for partition ID.

Note: This is indeterministic because it depends on data partitioning and task scheduling.

```
>>> df.repartition(1).select(spark_partition_id().alias("pid")).co
[Row(pid=0), Row(pid=0)]
```

New in version 1.6.

pyspark.sql.functions.**split**(*str*, *pattern*, *limit=-1*) [source]

Splits str around matches of the given pattern.

Parameters:

- **str** – a string expression to split
- **pattern** – a string representing a regular expression. The regex string should be a Java regular expression.
- **limit** –
  an integer which controls the number of times *pattern* is applied.
  - `limit > 0`: The resulting array's length will not be more than *limit*, and the resulting array's last entry will contain all input beyond the last matched pattern.
  - `limit <= 0`: *pattern* will be applied as many times as possible, and the resulting array can be of any size.

Changed in version 3.0: split now takes an optional *limit* field. If not provided, default limit value is -1.

```
>>> df = spark.createDataFrame([('oneAtwoBthreeC',)], ['s',])
>>> df.select(split(df.s, '[ABC]', 2).alias('s')).collect()
[Row(s=['one', 'twoBthreeC'])]
>>> df.select(split(df.s, '[ABC]', -1).alias('s')).collect()
[Row(s=['one', 'two', 'three', ''])]
```

New in version 1.5.

`pyspark.sql.functions.`**`sqrt`**(*col*)

Computes the square root of the specified float value.

New in version 1.3.

`pyspark.sql.functions.`**`stddev`**(*col*)

Aggregate function: alias for stddev_samp.

New in version 1.6.

`pyspark.sql.functions.`**`stddev_pop`**(*col*)

Aggregate function: returns population standard deviation of the expression in a group.

New in version 1.6.

`pyspark.sql.functions.`**`stddev_samp`**(*col*)

Aggregate function: returns the unbiased sample standard deviation of the expression in a group.

New in version 1.6.

`pyspark.sql.functions.`**`struct`**(**cols*) [source]

Creates a new struct column.

Parameters:
cols – list of column names (string) or list of `Column` expressions

```
>>> df.select(struct('age', 'name').alias("struct")).collect()
[Row(struct=Row(age=2, name='Alice')), Row(struct=Row(age=5, name='
>>> df.select(struct([df.age, df.name]).alias("struct")).collect()
[Row(struct=Row(age=2, name='Alice')), Row(struct=Row(age=5, name='
```

New in version 1.4.

`pyspark.sql.functions.`**`substring`**(*str*, *pos*, *len*) [source]

Substring starts at *pos* and is of length *len* when str is String type or returns the slice of byte array that starts at *pos* in byte and is of length *len* when str is Binary type.

Note: The position is not zero based, but 1 based index.

```
>>> df = spark.createDataFrame([('abcd',)], ['s',])
>>> df.select(substring(df.s, 1, 2).alias('s')).collect()
[Row(s='ab')]
```

New in version 1.5.

`pyspark.sql.functions.`**`substring_index`**(*str*, *delim*, *count*) [source]

Returns the substring from string str before count occurrences of the delimiter delim. If count is positive, everything the left of the final delimiter (counting from left) is returned. If count is negative, every to the right of the final delimiter (counting from the right) is returned. substring_index performs a case-sensitive match when searching for delim.

```
>>> df = spark.createDataFrame([('a.b.c.d',)], ['s'])
>>> df.select(substring_index(df.s, '.', 2).alias('s')).collect()
[Row(s='a.b')]
>>> df.select(substring_index(df.s, '.', -3).alias('s')).collect()
[Row(s='b.c.d')]
```

New in version 1.5.

`pyspark.sql.functions.`**`sum`**(*col*)

Aggregate function: returns the sum of all values in the expression.

New in version 1.3.

`pyspark.sql.functions.`**`sumDistinct`**(*col*)

Aggregate function: returns the sum of distinct values in the expression.

New in version 1.3.

`pyspark.sql.functions.`**`tan`**(*col*)

Parameters:
col – angle in radians
Returns:
tangent of the given value, as if computed by *java.lang.Math.tan()*

New in version 1.4.

`pyspark.sql.functions.`**`tanh`**(*col*)

Parameters:
col – hyperbolic angle
Returns:
hyperbolic tangent of the given value, as if computed by *java.lang.Math.tanh()*

New in version 1.4.

`pyspark.sql.functions.`**`toDegrees`**(*col*)

Note: Deprecated in 2.1, use **`degrees()`** instead.

New in version 1.4.

`pyspark.sql.functions.`**`toRadians`**(*col*)

Note: Deprecated in 2.1, use **`radians()`** instead.

New in version 1.4.

`pyspark.sql.functions.`**`to_csv`**(*col*, *options={}*) [source]

Converts a column containing a **`StructType`** into a CSV string. Throws an exception,

in the case of an unsupported type.

Parameters:

- **col** – name of column containing a struct.
- **options** – options to control converting. accepts the same options as the CSV datasource.

```
>>> from pyspark.sql import Row
>>> data = [(1, Row(name='Alice', age=2))]
>>> df = spark.createDataFrame(data, ("key", "value"))
>>> df.select(to_csv(df.value).alias("csv")).collect()
[Row(csv='2,Alice')]
```

New in version 3.0.

`pyspark.sql.functions.`**`to_date`**(*col*, *format=None*) [source]

Converts a **`Column`** of **`pyspark.sql.types.StringType`** or **`pyspark.sql.types.TimestampType`** into **`pyspark.sql.types.DateType`** using the optionally specified format. Specify formats according to DateTimeFormatter. # noqa By default, it follows casting rules to **`pyspark.sql.types.DateType`** if the format is omitted (equivalent to `col.cast("date")`).

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00',)], ['t'])
>>> df.select(to_date(df.t).alias('date')).collect()
[Row(date=datetime.date(1997, 2, 28))]
```

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00',)], ['t'])
>>> df.select(to_date(df.t, 'yyyy-MM-dd HH:mm:ss').alias('date')).
[Row(date=datetime.date(1997, 2, 28))]
```

New in version 2.2.

`pyspark.sql.functions.`**`to_json`**(*col*, *options={}*) [source]

Converts a column containing a **`StructType`**, **`ArrayType`** or a **`MapType`** into a JSON string. Throws an exception, in the case of an unsupported type.

Parameters:

- **col** – name of column containing a struct, an array or a map.
- **options** – options to control converting. accepts the same options as the JSON datasource. Additionally the function supports the *pretty* option which enables pretty JSON generation.

```
>>> from pyspark.sql import Row
>>> from pyspark.sql.types import *
>>> data = [(1, Row(name='Alice', age=2))]
>>> df = spark.createDataFrame(data, ("key", "value"))
>>> df.select(to_json(df.value).alias("json")).collect()
[Row(json='{"age":2,"name":"Alice"}')]
>>> data = [(1, [Row(name='Alice', age=2), Row(name='Bob', age=3)])]
>>> df = spark.createDataFrame(data, ("key", "value"))
>>> df.select(to_json(df.value).alias("json")).collect()
[Row(json='[{"age":2,"name":"Alice"},{"age":3,"name":"Bob"}]')]
>>> data = [(1, {"name": "Alice"})]
>>> df = spark.createDataFrame(data, ("key", "value"))
>>> df.select(to_json(df.value).alias("json")).collect()
[Row(json='{"name":"Alice"}')]
>>> data = [(1, [{"name": "Alice"}, {"name": "Bob"}])]
>>> df = spark.createDataFrame(data, ("key", "value"))
>>> df.select(to_json(df.value).alias("json")).collect()
[Row(json='[{"name":"Alice"},{"name":"Bob"}]')]
>>> data = [(1, ["Alice", "Bob"])]
>>> df = spark.createDataFrame(data, ("key", "value"))
>>> df.select(to_json(df.value).alias("json")).collect()
[Row(json='["Alice","Bob"]')]
```

New in version 2.1.

`pyspark.sql.functions.`**`to_str`**(*value*) [source]

A wrapper over str(), but converts bool values to lower case strings. If None is given, just returns None, instead of converting it to string “None”.

pyspark.sql.functions.**to_timestamp**(*col*, *format=None*) [source]

Converts a **Column** of **pyspark.sql.types.StringType** or **pyspark.sql.types.TimestampType** into **pyspark.sql.types.DateType** using the optionally specified format. Specify formats according to DateTimeFormatter. # noqa By default, it follows casting rules to **pyspark.sql.types.TimestampType** if the format is omitted (equivalent to `col.cast("timestamp")`).

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00',)], ['t'])
>>> df.select(to_timestamp(df.t).alias('dt')).collect()
[Row(dt=datetime.datetime(1997, 2, 28, 10, 30))]
```

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00',)], ['t'])
>>> df.select(to_timestamp(df.t, 'yyyy-MM-dd HH:mm:ss').alias('dt'
[Row(dt=datetime.datetime(1997, 2, 28, 10, 30))]
```

New in version 2.2.

pyspark.sql.functions.**to_utc_timestamp**(*timestamp*, *tz*) [source]

This is a common function for databases supporting TIMESTAMP WITHOUT TIMEZONE. This function takes a timestamp which is timezone-agnostic, and interprets it as a timestamp in the given timezone, and renders that timestamp as a timestamp in UTC.

However, timestamp in Spark represents number of microseconds from the Unix epoch, which is not timezone-agnostic. So in Spark this function just shift the timestamp value from the given timezone to UTC timezone.

This function may return confusing result if the input is a string with timezone, e.g. ‘2018-03-13T06:18:23+00:00’. The reason is that, Spark firstly cast the string to timestamp according to the timezone in the string, and finally display the result by converting the timestamp to string according to the session local timezone.

Parameters:

- **timestamp** – the column that contains timestamps
- **tz** – a string that has the ID of timezone, e.g. “GMT”, “America/Los_Angeles”, etc

Changed in version 2.4: tz can take a **Column** containing timezone ID strings.

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00', 'JST')], ['
>>> df.select(to_utc_timestamp(df.ts, "PST").alias('utc_time')).co
[Row(utc_time=datetime.datetime(1997, 2, 28, 18, 30))]
>>> df.select(to_utc_timestamp(df.ts, df.tz).alias('utc_time')).co
[Row(utc_time=datetime.datetime(1997, 2, 28, 1, 30))]
```

Note: Deprecated in 3.0. See SPARK-25496

New in version 1.5.

pyspark.sql.functions.**translate**(*srcCol*, *matching*, *replace*) [source]

A function translate any character in the *srcCol* by a character in *matching*. The characters in *replace* is corresponding to the characters in *matching*. The translate will happen when any character in the string matching with the character in the *matching*.

```
>>> spark.createDataFrame([('translate',)], ['a']).select(translate
...     .alias('r')).collect()
[Row(r='1a2s3ae')]
```

New in version 1.5.

pyspark.sql.functions.**trim**(*col*)

Trim the spaces from both ends for the specified string column.

New in version 1.5.

pyspark.sql.functions.**trunc**(*date*, *format*) [source]

Returns date truncated to the unit specified by the format.

Parameters:
format – ‘year’, ‘yyyy’, ‘yy’ or ‘month’, ‘mon’, ‘mm’

```
>>> df = spark.createDataFrame([('1997-02-28',)], ['d'])
>>> df.select(trunc(df.d, 'year').alias('year')).collect()
[Row(year=datetime.date(1997, 1, 1))]
>>> df.select(trunc(df.d, 'mon').alias('month')).collect()
[Row(month=datetime.date(1997, 2, 1))]
```

New in version 1.5.

pyspark.sql.functions.**udf**(*f=None*, *returnType=StringType*) [source]

Creates a user defined function (UDF).

Note: The user-defined functions are considered deterministic by default. Due to optimization, duplicate invocations may be eliminated or the function may even be invoked more times than it is present in the query. If your function is not deterministic, call *asNondeterministic* on the user defined function. E.g.:

```
>>> from pyspark.sql.types import IntegerType
>>> import random
>>> random_udf = udf(lambda: int(random.random() * 100), IntegerTyp
```

Note: The user-defined functions do not support conditional expressions or short circuiting in boolean expressions and it ends up with being executed all internally. If the functions can fail on special rows, the workaround is to incorporate the condition into the functions.

Note: The user-defined functions do not take keyword arguments on the calling side.

Parameters:
- **f** – python function if used as a standalone function
- **returnType** – the return type of the user-defined function. The value can be either a **pyspark.sql.types.DataType** object or a DDL-formatted type string.

```
>>> from pyspark.sql.types import IntegerType
>>> slen = udf(lambda s: len(s), IntegerType())
>>> @udf
... def to_upper(s):
...     if s is not None:
...         return s.upper()
...
>>> @udf(returnType=IntegerType())
... def add_one(x):
...     if x is not None:
...         return x + 1
...
>>> df = spark.createDataFrame([(1, "John Doe", 21)], ("id", "name
>>> df.select(slen("name").alias("slen(name)"), to_upper("name"),
+----------+--------------+------------+
|slen(name)|to_upper(name)|add_one(age)|
+----------+--------------+------------+
|         8|      JOHN DOE|          22|
+----------+--------------+------------+
```

New in version 1.3.

pyspark.sql.functions.**unbase64**(*col*)

Decodes a BASE64 encoded string column and returns it as a binary column.

New in version 1.5.

pyspark.sql.functions.**unhex**(*col*) [source]

Inverse of hex. Interprets each pair of characters as a hexadecimal number and converts to the byte representation of number.

```
>>> spark.createDataFrame([('414243',)], ['a']).select(unhex('a')).
[Row(unhex(a)=bytearray(b'ABC'))]
```

New in version 1.5.

pyspark.sql.functions.**unix_timestamp**(*timestamp=None*, *format='uuuu-MM-dd HH:mm:ss'*) [source]

Convert time string with given pattern ('uuuu-MM-dd HH:mm:ss', by default) to Unix time stamp (in seconds), using the default timezone and the default locale, return null if fail.

if *timestamp* is None, then it returns current timestamp.

```
>>> spark.conf.set("spark.sql.session.timeZone", "America/Los_Angel
>>> time_df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> time_df.select(unix_timestamp('dt', 'yyyy-MM-dd').alias('unix_t
[Row(unix_time=1428476400)]
>>> spark.conf.unset("spark.sql.session.timeZone")
```

New in version 1.5.

pyspark.sql.functions.**upper**(*col*)

Converts a string expression to upper case.

New in version 1.5.

pyspark.sql.functions.**var_pop**(*col*)

Aggregate function: returns the population variance of the values in a group.

New in version 1.6.

pyspark.sql.functions.**var_samp**(*col*)

Aggregate function: returns the unbiased sample variance of the values in a group.

New in version 1.6.

pyspark.sql.functions.**variance**(*col*)

Aggregate function: alias for var_samp.

New in version 1.6.

pyspark.sql.functions.**weekofyear**(*col*) [source]

Extract the week number of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> df.select(weekofyear(df.dt).alias('week')).collect()
[Row(week=15)]
```

New in version 1.5.

pyspark.sql.functions.**when**(*condition*, *value*) [source]

Evaluates a list of conditions and returns one of multiple possible result expressions. If **`Column.otherwise()`** is not invoked, None is returned for unmatched conditions.

Parameters:

- **condition** – a boolean `Column` expression.
- **value** – a literal value, or a `Column` expression.

```
>>> df.select(when(df['age'] == 2, 3).otherwise(4).alias("age")).c
[Row(age=3), Row(age=4)]
```

```
>>> df.select(when(df.age == 2, df.age + 1).alias("age")).collect(
[Row(age=3), Row(age=None)]
```

New in version 1.4.

`pyspark.sql.functions.`**`window`**(*timeColumn*, *windowDuration*, *slideDuration=None*, *startTime=None*) [source]

Bucketize rows into one or more time windows given a timestamp specifying column. Window starts are inclusive but the window ends are exclusive, e.g. 12:05 will be in the window [12:05,12:10) but not in [12:00,12:05). Windows can support microsecond precision. Windows in the order of months are not supported.

The time column must be of **`pyspark.sql.types.TimestampType`**.

Durations are provided as strings, e.g. '1 second', '1 day 12 hours', '2 minutes'. Valid interval strings are 'week', 'day', 'hour', 'minute', 'second', 'millisecond', 'microsecond'. If the `slideDuration` is not provided, the windows will be tumbling windows.

The startTime is the offset with respect to 1970-01-01 00:00:00 UTC with which to start window intervals. For example, in order to have hourly tumbling windows that start 15 minutes past the hour, e.g. 12:15-13:15, 13:15-14:15… provide *startTime* as *15 minutes*.

The output column will be a struct called 'window' by default with the nested columns 'start' and 'end', where 'start' and 'end' will be of **`pyspark.sql.types.TimestampType`**.

```
>>> df = spark.createDataFrame([("2016-03-11 09:00:07", 1)]).toDF('
>>> w = df.groupBy(window("date", "5 seconds")).agg(sum("val").ali
>>> w.select(w.window.start.cast("string").alias("start"),
...           w.window.end.cast("string").alias("end"), "sum").colle
[Row(start='2016-03-11 09:00:05', end='2016-03-11 09:00:10', sum=1)
```

New in version 2.0.

`pyspark.sql.functions.`**`xxhash64`**(**cols*) [source]

Calculates the hash code of given columns using the 64-bit variant of the xxHash algorithm, and returns the result as a long column.

```
>>> spark.createDataFrame([('ABC',)], ['a']).select(xxhash64('a').a
[Row(hash=4105715581806190027)]
```

New in version 3.0.

`pyspark.sql.functions.`**`year`**(*col*) [source]

Extract the year of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> df.select(year('dt').alias('year')).collect()
[Row(year=2015)]
```

New in version 1.5.

pyspark.sql.avro.functions module

A collections of builtin avro functions

pyspark.sql.avro.functions.**from_avro**(*data*, *jsonFormatSchema*, *options={}*) [source]

Converts a binary column of Avro format into its corresponding catalyst value. If a schema is provided via the option actualSchema, a different (but compatible) schema can be used for reading. If no actualSchema option is provided, the specified schema must match the read data, otherwise the behavior is undefined: it may fail or return arbitrary result.

Note: Avro is built-in but external data source module since Spark 2.4. Please deploy the application as per the deployment section of “Apache Avro Data Source Guide”.

Parameters:

- **data** – the binary column.
- **jsonFormatSchema** – the avro schema in JSON string format.
- **options** – options to control how the Avro record is parsed.

```
>>> from pyspark.sql import Row
>>> from pyspark.sql.avro.functions import from_avro, to_avro
>>> data = [(1, Row(name='Alice', age=2))]
>>> df = spark.createDataFrame(data, ("key", "value"))
>>> avroDf = df.select(to_avro(df.value).alias("avro"))
>>> avroDf.collect()
[Row(avro=bytearray(b'\x00\x00\x04\x00\nAlice'))]
>>> jsonFormatSchema = '''{"type":"record","name":"topLevelRecord",
...     [{"name":"avro","type":[{"type":"record","name":"value","na
...     "fields":[{"name":"age","type":["long","null"]},
...     {"name":"name","type":["string","null"]}]},"null"]}]}'''
>>> avroDf.select(from_avro(avroDf.avro, jsonFormatSchema).alias("
[Row(value=Row(avro=Row(age=2, name='Alice')))]
```

New in version 3.0.

pyspark.sql.avro.functions.**to_avro**(*data*, *jsonFormatSchema=''*) [source]

Converts a column into binary of avro format.

Note: Avro is built-in but external data source module since Spark 2.4. Please deploy the application as per the deployment section of “Apache Avro Data Source Guide”.

Parameters:

- **data** – the data column.
- **jsonFormatSchema** – user-specified output avro schema in JSON string format.

```
>>> from pyspark.sql import Row
>>> from pyspark.sql.avro.functions import to_avro
>>> data = ['SPADES']
>>> df = spark.createDataFrame(data, "string")
>>> df.select(to_avro(df.value).alias("suite")).collect()
[Row(suite=bytearray(b'\x00\x0cSPADES'))]
>>> jsonFormatSchema = '''["null", {"type": "enum", "name": "value"
...     "symbols": ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]}]'''
>>> df.select(to_avro(df.value, jsonFormatSchema).alias("suite")).
[Row(suite=bytearray(b'\x02\x00'))]
```

New in version 3.0.

pyspark.sql.streaming module

class pyspark.sql.streaming.**StreamingQuery**(*jsq*) [source]

A handle to a query that is executing continuously in the background as new data arrives. All these methods are thread-safe.

Note: Evolving

New in version 2.0.

awaitTermination(*timeout=None*) [source]

Waits for the termination of *this* query, either by **query.stop()** or by an exception. If the query has terminated with an exception, then the exception will be thrown. If *timeout* is set, it returns whether the query has terminated or not within the *timeout* seconds.

If the query has terminated, then all subsequent calls to this method will either return immediately (if the query was terminated by **stop()**), or throw the exception immediately (if the query has terminated with exception).

throws **StreamingQueryException**, if *this* query has terminated with an exception

New in version 2.0.

exception() [source]

Returns:
the StreamingQueryException if the query was terminated by an exception, or None.

New in version 2.1.

explain(*extended=False*) [source]

Prints the (logical and physical) plans to the console for debugging purpose.

Parameters:
extended – boolean, default `False`. If `False`, prints only the physical plan.

```
>>> sq = sdf.writeStream.format('memory').queryName('query_expl
>>> sq.processAllAvailable() # Wait a bit to generate the runti
>>> sq.explain()
== Physical Plan ==
...
>>> sq.explain(True)
== Parsed Logical Plan ==
...
== Analyzed Logical Plan ==
...
== Optimized Logical Plan ==
...
== Physical Plan ==
...
>>> sq.stop()
```

New in version 2.1.

property **id**

Returns the unique id of this query that persists across restarts from checkpoint data. That is, this id is generated when a query is started for the first time, and will be the same every time it is restarted from checkpoint data. There can only be one query with the same id active in a Spark cluster. Also see, *runId*.

New in version 2.0.

property **isActive**

Whether this streaming query is currently active or not.

New in version 2.0.

property **lastProgress**

Returns the most recent **StreamingQueryProgress** update of this streaming query or None if there were no progress updates :return: a map

New in version 2.1.

property **name**

Returns the user-specified name of the query, or null if not specified. This name can be specified in the *org.apache.spark.sql.streaming.DataStreamWriter* as *dataframe.writeStream.queryName("query").start()*. This name, if set, must be unique across all active queries.

New in version 2.0.

processAllAvailable() [source]

Blocks until all available data in the source has been processed and committed to the sink. This method is intended for testing.

> **Note:** In the case of continually arriving data, this method may block forever. Additionally, this method is only guaranteed to block until data that has been synchronously appended data to a stream source prior to invocation. (i.e. *getOffset* must immediately reflect the addition).

New in version 2.0.

property **recentProgress**

Returns an array of the most recent [[StreamingQueryProgress]] updates for this query. The number of progress updates retained for each stream is configured by Spark session configuration *spark.sql.streaming.numRecentProgressUpdates*.

New in version 2.1.

property **runId**

Returns the unique id of this query that does not persist across restarts. That is, every query that is started (or restarted from checkpoint) will have a different runId.

New in version 2.1.

property **status**

Returns the current status of the query.

New in version 2.1.

stop() [source]

Stop this streaming query.

New in version 2.0.

class pyspark.sql.streaming.**StreamingQueryManager**(*jsqm*) [source]

A class to manage all the **StreamingQuery** StreamingQueries active.

> **Note:** Evolving

New in version 2.0.

property **active**

Returns a list of active queries associated with this SQLContext

```
>>> sq = sdf.writeStream.format('memory').queryName('this_query
>>> sqm = spark.streams
>>> # get the list of active streaming queries
>>> [q.name for q in sqm.active]
['this_query']
>>> sq.stop()
```

New in version 2.0.

awaitAnyTermination(*timeout=None*) [source]

Wait until any of the queries on the associated SQLContext has terminated since the creation of the context, or since **`resetTerminated()`** was called. If any query was terminated with an exception, then the exception will be thrown. If *timeout* is set, it returns whether the query has terminated or not within the *timeout* seconds.

If a query has terminated, then subsequent calls to **`awaitAnyTermination()`** will either return immediately (if the query was terminated by **`query.stop()`**), or throw the exception immediately (if the query was terminated with exception). Use **`resetTerminated()`** to clear past terminations and wait for new terminations.

In the case where multiple queries have terminated since **`resetTermination()`** was called, if any query has terminated with exception, then **`awaitAnyTermination()`** will throw any of the exception. For correctly documenting exceptions across multiple queries, users need to stop all of them after any of them terminates with exception, and then check the *query.exception()* for each query.

throws **`StreamingQueryException`**, if *this* query has terminated with an exception

New in version 2.0.

`get`(*id*) [source]

Returns an active query from this SQLContext or throws exception if an active query with this name doesn't exist.

```
>>> sq = sdf.writeStream.format('memory').queryName('this_query
>>> sq.name
'this_query'
>>> sq = spark.streams.get(sq.id)
>>> sq.isActive
True
>>> sq = sqlContext.streams.get(sq.id)
>>> sq.isActive
True
>>> sq.stop()
```

New in version 2.0.

`resetTerminated`() [source]

Forget about past terminated queries so that **`awaitAnyTermination()`** can be used again to wait for new terminations.

```
>>> spark.streams.resetTerminated()
```

New in version 2.0.

class `pyspark.sql.streaming.`**`DataStreamReader`**(*spark*) [source]

Interface used to load a streaming **`DataFrame`** from external storage systems (e.g. file systems, key-value stores, etc). Use **`spark.readStream()`** to access this.

> **Note:** Evolving.

New in version 2.0.

`csv`(*path*, *schema=None*, *sep=None*, *encoding=None*, *quote=None*, *escape=None*, *comment=None*, *header=None*, *inferSchema=None*, *ignoreLeadingWhiteSpace=None*, *ignoreTrailingWhiteSpace=None*, *nullValue=None*, *nanValue=None*, *positiveInf=None*, *negativeInf=None*, *dateFormat=None*, *timestampFormat=None*, *maxColumns=None*, *maxCharsPerColumn=None*, *maxMalformedLogPerPartition=None*, *mode=None*, *columnNameOfCorruptRecord=None*, *multiLine=None*,

charToEscapeQuoteEscaping=None, *enforceSchema=None*, *emptyValue=None*, *locale=None*, *lineSep=None*, *recursiveFileLookup=None*) [source]

Loads a CSV file stream and returns the result as a **`DataFrame`**.

This function will go through the input once to determine the input schema if `inferSchema` is enabled. To avoid going through the entire data once, disable `inferSchema` option or specify the schema explicitly using `schema`.

> **Note:** Evolving.

Parameters:

- **path** – string, or list of strings, for input path(s).
- **schema** – an optional **`pyspark.sql.types.StructType`** for the input schema or a DDL-formatted string (For example `col0 INT, col1 DOUBLE`).
- **sep** – sets a separator (one or more characters) for each field and value. If None is set, it uses the default value, `,`.
- **encoding** – decodes the CSV files by the given encoding type. If None is set, it uses the default value, `UTF-8`.
- **quote** – sets a single character used for escaping quoted values where the separator can be part of the value. If None is set, it uses the default value, `"`. If you would like to turn off quotations, you need to set an empty string.
- **escape** – sets a single character used for escaping quotes inside an already quoted value. If None is set, it uses the default value, `\`.
- **comment** – sets a single character used for skipping lines beginning with this character. By default (None), it is disabled.
- **header** – uses the first line as names of columns. If None is set, it uses the default value, `false`.
- **inferSchema** – infers the input schema automatically from data. It requires one extra pass over the data. If None is set, it uses the default value, `false`.
- **enforceSchema** – If it is set to `true`, the specified or inferred schema will be forcibly applied to datasource files, and headers in CSV files will be ignored. If the option is set to `false`, the schema will be validated against all headers in CSV files or the first header in RDD if the `header` option is set to `true`. Field names in the schema and column names in CSV headers are checked by their positions taking into account `spark.sql.caseSensitive`. If None is set, `true` is used by default. Though the default value is `true`, it is recommended to disable the `enforceSchema` option to avoid incorrect results.
- **ignoreLeadingWhiteSpace** – a flag indicating whether or not leading whitespaces from values being read should be skipped. If None is set, it uses the default value, `false`.
- **ignoreTrailingWhiteSpace** – a flag indicating whether or not trailing whitespaces from values being read should be skipped. If None is set, it uses the default value, `false`.
- **nullValue** – sets the string representation of a null value. If None is set, it uses the default value, empty string. Since 2.0.1, this `nullValue` param applies to all supported types including the string type.
- **nanValue** – sets the string representation of a non-number value. If None is set, it uses the default value, `NaN`.
- **positiveInf** – sets the string representation of a positive infinity value. If None is set, it uses the default value, `Inf`.
- **negativeInf** – sets the string representation of a negative infinity value. If None is set, it uses the default value, `Inf`.
- **dateFormat** – sets the string that indicates a date format. Custom date formats follow the formats at `java.time.format.DateTimeFormatter`. This applies to date type. If None is set, it uses the default value, `uuuu-MM-dd`.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at `java.time.format.DateTimeFormatter`. This applies to timestamp type. If None is set, it uses the default value, `uuuu-MM-dd'T'HH:mm:ss.SSSXXX`.
- **maxColumns** – defines a hard limit of how many columns a record can

have. If None is set, it uses the default value, `20480`.

- **maxCharsPerColumn** – defines the maximum number of characters allowed for any given value being read. If None is set, it uses the default value, `-1` meaning unlimited length.
- **maxMalformedLogPerPartition** – this parameter is no longer used since Spark 2.2.0. If specified, it is ignored.
- **mode** –

  allows a mode for dealing with corrupt records during parsing. If None is set, it uses the default value, `PERMISSIVE`.

  - `PERMISSIVE` : when it meets a corrupted record, puts the malformed string into a field configured by `columnNameOfCorruptRecord`, and sets malformed fields to `null`. To keep corrupt records, an user can set a string type field named `columnNameOfCorruptRecord` in an user-defined schema. If a schema does not have the field, it drops corrupt records during parsing. A record with less/more tokens than schema is not a corrupted record to CSV. When it meets a record having fewer tokens than the length of the schema, sets `null` to extra fields. When the record has more tokens than the length of the schema, it drops extra tokens.
  - `DROPMALFORMED` : ignores the whole corrupted records.
  - `FAILFAST` : throws an exception when it meets corrupted records.
- **columnNameOfCorruptRecord** – allows renaming the new field having malformed string created by `PERMISSIVE` mode. This overrides `spark.sql.columnNameOfCorruptRecord`. If None is set, it uses the value specified in `spark.sql.columnNameOfCorruptRecord`.
- **multiLine** – parse one record, which may span multiple lines. If None is set, it uses the default value, `false`.
- **charToEscapeQuoteEscaping** – sets a single character used for escaping the escape for the quote character. If None is set, the default value is escape character when escape and quote characters are different, `\0` otherwise..
- **emptyValue** – sets the string representation of an empty value. If None is set, it uses the default value, empty string.
- **locale** – sets a locale as language tag in IETF BCP 47 format. If None is set, it uses the default value, `en-US`. For instance, `locale` is used while parsing dates and timestamps.
- **lineSep** – defines the line separator that should be used for parsing. If None is set, it covers all `\\r`, `\\r\\n` and `\\n`. Maximum length is 1 character.
- **recursiveFileLookup** – recursively scan a directory for files. Using this option disables partition discovery.

```
>>> csv_sdf = spark.readStream.csv(tempfile.mkdtemp(), schema =
>>> csv_sdf.isStreaming
True
>>> csv_sdf.schema == sdf_schema
True
```

New in version 2.0.

format(*source*) [source]

Specifies the input data source format.

> **Note:** Evolving.

Parameters:
source – string, name of the data source, e.g. ‘json’, ‘parquet’.

```
>>> s = spark.readStream.format("text")
```

New in version 2.0.

json(*path*, *schema=None*, *primitivesAsString=None*, *prefersDecimal=None*, *allowComments=None*, *allowUnquotedFieldNames=None*, *allowSingleQuotes=None*, *allowNumericLeadingZero=None*, *allowBackslashEscapingAnyCharacter=None*, *mode=None*, *columnNameOfCorruptRecord=None*, *dateFormat=None*, *timestampFormat=None*, *multiLine=None*, *allowUnquotedControlChars=None*, *lineSep=None*, *locale=None*, *dropFieldIfAllNull=None*, *encoding=None*, *recursiveFileLookup=None*) [source]

Loads a JSON file stream and returns the results as a **DataFrame**.

JSON Lines (newline-delimited JSON) is supported by default. For JSON (one record per file), set the `multiLine` parameter to `true`.

If the `schema` parameter is not specified, this function goes through the input once to determine the input schema.

> **Note:** Evolving.

Parameters:

- **path** – string represents path to the JSON dataset, or RDD of Strings storing JSON objects.
- **schema** – an optional **pyspark.sql.types.StructType** for the input schema or a DDL-formatted string (For example `col0 INT, col1 DOUBLE`).
- **primitivesAsString** – infers all primitive values as a string type. If None is set, it uses the default value, `false`.
- **prefersDecimal** – infers all floating-point values as a decimal type. If the values do not fit in decimal, then it infers them as doubles. If None is set, it uses the default value, `false`.
- **allowComments** – ignores Java/C++ style comment in JSON records. If None is set, it uses the default value, `false`.
- **allowUnquotedFieldNames** – allows unquoted JSON field names. If None is set, it uses the default value, `false`.
- **allowSingleQuotes** – allows single quotes in addition to double quotes. If None is set, it uses the default value, `true`.
- **allowNumericLeadingZero** – allows leading zeros in numbers (e.g. 00012). If None is set, it uses the default value, `false`.
- **allowBackslashEscapingAnyCharacter** – allows accepting quoting of all character using backslash quoting mechanism. If None is set, it uses the default value, `false`.
- **mode** –

  allows a mode for dealing with corrupt records during parsing. If None is set, it uses the default value, `PERMISSIVE`.

  - `PERMISSIVE` : when it meets a corrupted record, puts the malformed string into a field configured by `columnNameOfCorruptRecord`, and sets malformed fields to `null`. To keep corrupt records, an user can set a string type field named `columnNameOfCorruptRecord` in an user-defined schema. If a schema does not have the field, it drops corrupt records during parsing. When inferring a schema, it implicitly adds a `columnNameOfCorruptRecord` field in an output schema.
  - `DROPMALFORMED` : ignores the whole corrupted records.
  - `FAILFAST` : throws an exception when it meets corrupted records.
- **columnNameOfCorruptRecord** – allows renaming the new field having malformed string created by `PERMISSIVE` mode. This overrides `spark.sql.columnNameOfCorruptRecord`. If None is set, it uses the value specified in `spark.sql.columnNameOfCorruptRecord`.
- **dateFormat** – sets the string that indicates a date format. Custom date formats follow the formats at `java.time.format.DateTimeFormatter`. This applies to date type. If None is set, it uses the default value, `uuuu-MM-dd`.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at

`java.time.format.DateTimeFormatter`. This applies to timestamp type. If None is set, it uses the default value, `uuuu-MM-dd'T'HH:mm:ss.SSSXXX`.

- **multiLine** – parse one record, which may span multiple lines, per file. If None is set, it uses the default value, `false`.
- **allowUnquotedControlChars** – allows JSON Strings to contain unquoted control characters (ASCII characters with value less than 32, including tab and line feed characters) or not.
- **lineSep** – defines the line separator that should be used for parsing. If None is set, it covers all `\r`, `\r\n` and `\n`.
- **locale** – sets a locale as language tag in IETF BCP 47 format. If None is set, it uses the default value, `en-US`. For instance, `locale` is used while parsing dates and timestamps.
- **dropFieldIfAllNull** – whether to ignore column of all null values or empty array/struct during schema inference. If None is set, it uses the default value, `false`.
- **encoding** – allows to forcibly set one of standard basic or extended encoding for the JSON files. For example UTF-16BE, UTF-32LE. If None is set, the encoding of input JSON will be detected automatically when the multiLine option is set to `true`.
- **recursiveFileLookup** – recursively scan a directory for files. Using this option disables partition discovery.

```
>>> json_sdf = spark.readStream.json(tempfile.mkdtemp(), schema
>>> json_sdf.isStreaming
True
>>> json_sdf.schema == sdf_schema
True
```

New in version 2.0.

load(*path=None*, *format=None*, *schema=None*, ***options*) [source]

Loads a data stream from a data source and returns it as a :class`DataFrame`.

> **Note:** Evolving.

Parameters:

- **path** – optional string for file-system backed data sources.
- **format** – optional string for format of the data source. Default to ‘parquet’.
- **schema** – optional **`pyspark.sql.types.StructType`** for the input schema or a DDL-formatted string (For example `col0 INT, col1 DOUBLE`).
- **options** – all other string options

```
>>> json_sdf = spark.readStream.format("json") \
...     .schema(sdf_schema) \
...     .load(tempfile.mkdtemp())
>>> json_sdf.isStreaming
True
>>> json_sdf.schema == sdf_schema
True
```

New in version 2.0.

option(*key*, *value*) [source]

Adds an input option for the underlying data source.

You can set the following option(s) for reading files:

- `timeZone`: sets the string that indicates a timezone to be used to parse timestamps
  in the JSON/CSV datasources or partition values. If it isn’t set, it uses the default value, session local timezone.
- `pathGlobFilter`: an optional glob pattern to only include files with

paths matching
the pattern. The syntax follows org.apache.hadoop.fs.GlobFilter. It does not change the behavior of partition discovery.

Note: Evolving.

```
>>> s = spark.readStream.option("x", 1)
```

New in version 2.0.

options(***options*) [source]

Adds input options for the underlying data source.

You can set the following option(s) for reading files:

- `timeZone`: sets the string that indicates a timezone to be used to parse timestamps
  in the JSON/CSV datasources or partition values. If it isn't set, it uses the default value, session local timezone.
- `pathGlobFilter`: an optional glob pattern to only include files with paths matching
  the pattern. The syntax follows org.apache.hadoop.fs.GlobFilter. It does not change the behavior of partition discovery.

Note: Evolving.

```
>>> s = spark.readStream.options(x="1", y=2)
```

New in version 2.0.

orc(*path*, *mergeSchema=None*, *recursiveFileLookup=None*) [source]

Loads a ORC file stream, returning the result as a **DataFrame**.

Note: Evolving.

Parameters:

- **mergeSchema** – sets whether we should merge schemas collected from all ORC part-files. This will override `spark.sql.orc.mergeSchema`. The default value is specified in `spark.sql.orc.mergeSchema`.
- **recursiveFileLookup** – recursively scan a directory for files. Using this option disables partition discovery.

```
>>> orc_sdf = spark.readStream.schema(sdf_schema).orc(tempfile.
>>> orc_sdf.isStreaming
True
>>> orc_sdf.schema == sdf_schema
True
```

New in version 2.3.

parquet(*path*, *mergeSchema=None*, *recursiveFileLookup=None*) [source]

Loads a Parquet file stream, returning the result as a **DataFrame**.

Note: Evolving.

Parameters:

- **mergeSchema** – sets whether we should merge schemas collected from all Parquet part-files. This will override `spark.sql.parquet.mergeSchema`. The default value is specified in `spark.sql.parquet.mergeSchema`.

- **recursiveFileLookup** – recursively scan a directory for files. Using this option disables partition discovery.

```
>>> parquet_sdf = spark.readStream.schema(sdf_schema).parquet(t
>>> parquet_sdf.isStreaming
True
>>> parquet_sdf.schema == sdf_schema
True
```

New in version 2.0.

schema(*schema*) [source]

Specifies the input schema.

Some data sources (e.g. JSON) can infer the input schema automatically from data. By specifying the schema here, the underlying data source can skip the schema inference step, and thus speed up data loading.

Note: Evolving.

Parameters:
schema – a **pyspark.sql.types.StructType** object or a DDL-formatted string (For example `col0 INT, col1 DOUBLE`).

```
>>> s = spark.readStream.schema(sdf_schema)
>>> s = spark.readStream.schema("col0 INT, col1 DOUBLE")
```

New in version 2.0.

text(*path*, *wholetext=False*, *lineSep=None*, *recursiveFileLookup=None*) [source]

Loads a text file stream and returns a **DataFrame** whose schema starts with a string column named “value”, and followed by partitioned columns if there are any. The text files must be encoded as UTF-8.

By default, each line in the text file is a new row in the resulting DataFrame.

Note: Evolving.

Parameters:
- **paths** – string, or list of strings, for input path(s).
- **wholetext** – if true, read each file from input path(s) as a single row.
- **lineSep** – defines the line separator that should be used for parsing. If None is set, it covers all `\r`, `\r\n` and `\n`.
- **recursiveFileLookup** – recursively scan a directory for files. Using this option disables partition discovery.

```
>>> text_sdf = spark.readStream.text(tempfile.mkdtemp())
>>> text_sdf.isStreaming
True
>>> "value" in str(text_sdf.schema)
True
```

New in version 2.0.

class pyspark.sql.streaming.**DataStreamWriter**(*df*) [source]

Interface used to write a streaming **DataFrame** to external storage systems (e.g. file systems, key-value stores, etc). Use **DataFrame.writeStream()** to access this.

Note: Evolving.

New in version 2.0.

foreach(*f*) [source]

Sets the output of the streaming query to be processed using the provided writer `f`. This is often used to write the output of a streaming query to arbitrary storage systems. The processing logic can be specified in two ways.

1. A **function** that takes a row as input.
   This is a simple way to express your processing logic. Note that this does not allow you to deduplicate generated data when failures cause reprocessing of some input data. That would require you to specify the processing logic in the next way.

2. An **object** with a `process` method and optional `open` and `close` methods.
   The object can have the following methods.

   - `open(partition_id, epoch_id)`: *Optional* method that initializes the processing
     (for example, open a connection, start a transaction, etc). Additionally, you can use the *partition_id* and *epoch_id* to deduplicate regenerated data (discussed later).

   - `process(row)`: *Non-optional* method that processes each **`Row`**.

   - `close(error)`: *Optional* method that finalizes and cleans up (for example,
     close connection, commit transaction, etc.) after all rows have been processed.

   The object will be used by Spark in the following way.

   - A single copy of this object is responsible of all the data generated by a
     single task in a query. In other words, one instance is responsible for processing one partition of the data generated in a distributed manner.

   - This object must be serializable because each task will get a fresh
     serialized-deserialized copy of the provided object. Hence, it is strongly recommended that any initialization for writing data (e.g. opening a connection or starting a transaction) is done after the *open(…)* method has been called, which signifies that the task is ready to generate data.

   - The lifecycle of the methods are as follows.

     For each partition with `partition_id`:

     … For each batch/epoch of streaming data with `epoch_id`:

     ……. Method `open(partitionId, epochId)` is called.

     ……. If `open(...)` returns true, for each row in the partition and
     batch/epoch, method `process(row)` is called.

     ……. Method `close(errorOrNull)` is called with error (if any) seen while
     processing rows.

   Important points to note:

   - The *partitionId* and *epochId* can be used to deduplicate generated data when

failures cause reprocessing of some input data. This depends on the execution mode of the query. If the streaming query is being executed in the micro-batch mode, then every partition represented by a unique tuple (partition_id, epoch_id) is guaranteed to have the same data. Hence, (partition_id, epoch_id) can be used to deduplicate and/or transactionally commit data and achieve exactly-once guarantees. However, if the streaming query is being executed in the continuous mode, then this guarantee does not hold and therefore should not be used for deduplication.

- The `close()` method (if exists) will be called if *open()* method exists and

  returns successfully (irrespective of the return value), except if the Python crashes in the middle.

> **Note:** Evolving.

```
>>> # Print every row using a function
>>> def print_row(row):
...     print(row)
...
>>> writer = sdf.writeStream.foreach(print_row)
>>> # Print every row using a object with process() method
>>> class RowPrinter:
...     def open(self, partition_id, epoch_id):
...         print("Opened %d, %d" % (partition_id, epoch_id))
...         return True
...     def process(self, row):
...         print(row)
...     def close(self, error):
...         print("Closed with error: %s" % str(error))
...
>>> writer = sdf.writeStream.foreach(RowPrinter())
```

New in version 2.4.

foreachBatch(*func*) [source]

Sets the output of the streaming query to be processed using the provided function. This is supported only the in the micro-batch execution modes (that is, when the trigger is not continuous). In every micro-batch, the provided function will be called in every micro-batch with (i) the output rows as a DataFrame and (ii) the batch identifier. The batchId can be used deduplicate and transactionally write the output (that is, the provided Dataset) to external systems. The output DataFrame is guaranteed to exactly same for the same batchId (assuming all operations are deterministic in the query).

> **Note:** Evolving.

```
>>> def func(batch_df, batch_id):
...     batch_df.collect()
...
>>> writer = sdf.writeStream.foreachBatch(func)
```

New in version 2.4.

format(*source*) [source]

Specifies the underlying output data source.

> **Note:** Evolving.

Parameters:
source – string, name of the data source, which for now can be ‘parquet’.

```
>>> writer = sdf.writeStream.format('json')
```

New in version 2.0.

option(*key*, *value*) [source]

Adds an output option for the underlying data source.

You can set the following option(s) for writing files:

- `timeZone`: sets the string that indicates a timezone to be used to format
  timestamps in the JSON/CSV datasources or partition values. If it isn't set, it uses the default value, session local timezone.

Note: Evolving.

New in version 2.0.

options(***options*) [source]

Adds output options for the underlying data source.

You can set the following option(s) for writing files:

- `timeZone`: sets the string that indicates a timezone to be used to format
  timestamps in the JSON/CSV datasources or partition values. If it isn't set, it uses the default value, session local timezone.

Note: Evolving.

New in version 2.0.

outputMode(*outputMode*) [source]

Specifies how data of a streaming DataFrame/Dataset is written to a streaming sink.

Options include:

- *append*:Only the new rows in the streaming DataFrame/Dataset will be written to
  the sink
- *complete*:All the rows in the streaming DataFrame/Dataset will be written to the sink
  every time these is some updates
- *update*:only the rows that were updated in the streaming DataFrame/Dataset will be
  written to the sink every time there are some updates. If the query doesn't contain aggregations, it will be equivalent to *append* mode.

Note: Evolving.

```
>>> writer = sdf.writeStream.outputMode('append')
```

New in version 2.0.

partitionBy(**cols*) [source]

Partitions the output by the given columns on the file system.

If specified, the output is laid out on the file system similar to Hive's partitioning scheme.

> **Note:** Evolving.

Parameters:
cols – name of columns

New in version 2.0.

queryName(*queryName*) [source]

Specifies the name of the **StreamingQuery** that can be started with **start()**. This name must be unique among all the currently active queries in the associated SparkSession.

> **Note:** Evolving.

Parameters:
queryName – unique name for the query

```
>>> writer = sdf.writeStream.queryName('streaming_query')
```

New in version 2.0.

start(*path=None*, *format=None*, *outputMode=None*, *partitionBy=None*, *queryName=None*, ***options*) [source]

Streams the contents of the **DataFrame** to a data source.

The data source is specified by the `format` and a set of `options`. If `format` is not specified, the default data source configured by `spark.sql.sources.default` will be used.

> **Note:** Evolving.

Parameters:

- **path** – the path in a Hadoop supported file system
- **format** – the format used to save
- **outputMode** –

  specifies how data of a streaming DataFrame/Dataset is written to a streaming sink.

  - *append*:Only the new rows in the streaming DataFrame/Dataset will be written to the sink
  - *complete*:All the rows in the streaming DataFrame/Dataset will be written to the sink every time these is some updates
  - *update*:only the rows that were updated in the streaming DataFrame/Dataset will be written to the sink every time there are some updates. If the query doesn't contain aggregations, it will be equivalent to *append* mode.
- **partitionBy** – names of partitioning columns
- **queryName** – unique name for the query
- **options** – All other string options. You may want to provide a *checkpointLocation* for most streams, however it is not required for a *memory* stream.

```
>>> sq = sdf.writeStream.format('memory').queryName('this_query
>>> sq.isActive
True
>>> sq.name
'this_query'
>>> sq.stop()
>>> sq.isActive
False
>>> sq = sdf.writeStream.trigger(processingTime='5 seconds').st
...     queryName='that_query', outputMode="append", format='me
>>> sq.name
'that_query'
>>> sq.isActive
True
>>> sq.stop()
```

New in version 2.0.

trigger(*processingTime=None*, *once=None*, *continuous=None*) [source]

Set the trigger for the stream query. If this is not set it will run the query as fast as possible, which is equivalent to setting the trigger to `processingTime='0 seconds'`.

Note: Evolving.

Parameters:

- **processingTime** – a processing time interval as a string, e.g. ‘5 seconds’, ‘1 minute’. Set a trigger that runs a microbatch query periodically based on the processing time. Only one trigger can be set.
- **once** – if set to True, set a trigger that processes only one batch of data in a streaming query then terminates the query. Only one trigger can be set.
- **continuous** – a time interval as a string, e.g. ‘5 seconds’, ‘1 minute’. Set a trigger that runs a continuous query with a given checkpoint interval. Only one trigger can be set.

```
>>> # trigger the query for execution every 5 seconds
>>> writer = sdf.writeStream.trigger(processingTime='5 seconds'
>>> # trigger the query for just once batch of data
>>> writer = sdf.writeStream.trigger(once=True)
>>> # trigger the query for execution every 5 seconds
>>> writer = sdf.writeStream.trigger(continuous='5 seconds')
```

New in version 2.0.