pyspark Documentation

Release master

Author

Jun 02, 2020

CONTENTS

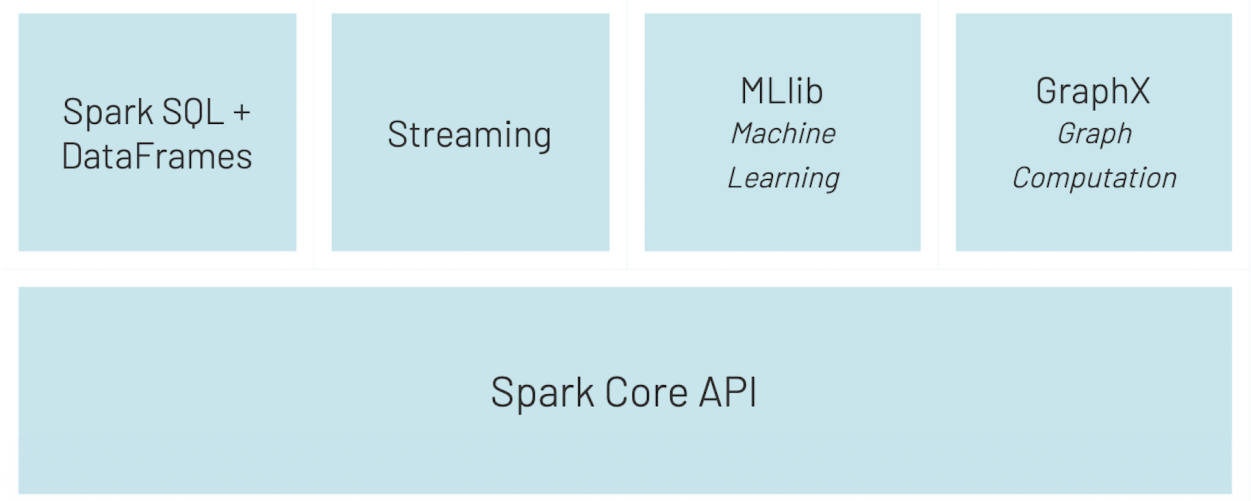
1 Sitemap **3**

1.1 Getting Started . . . 3
1.2 User Guide . . . 6
1.3 API Reference . . . 17
1.4 Development . . . 225
1.5 Release Notes . . . 228

Python Module Index **229**

Index **231**

PySpark is a set of Spark APIs in Python language. It not only offers for you to write an application with Python APIs but also provides PySpark shell so you can interactively analyze your data in a distributed environment. PySpark includes almost all Apache Spark features.



General Execution: Spark Core

Spark Core is the underlying general execution engine for the Spark platform that all other functionality is built on top of. It provides in-memory computing capabilities.

Structured Data: Spark SQL

Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrames and can also act as distributed SQL query engine.

Streaming Analytics: Spark Streaming

Running on top of Spark, Spark Streaming enables powerful interactive and analytical applications across both streaming and historical data, while inheriting Spark's ease of use and fault tolerance characteristics.

Machine Learning: MLlib

Machine learning has quickly emerged as a critical piece in mining Big Data for actionable insights. Built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms (e.g., multiple iterations to increase accuracy) and blazing speed (up to 100x faster than MapReduce).

CHAPTER

ONE

SITEMAP

1.1 Getting Started

1.1.1 Quickstart

Table of contents:

- *Interactive Analysis with the Spark Shell*
- *Self-contained Application*

This tutorial provides a quick introduction to using Spark. We will first introduce the API through Spark's interactive shell (in Python or Scala), then show how to write applications in Java, Scala, and Python.

To follow along with this guide, first, download a packaged release of Spark from the Spark website. Since we won't be using HDFS, you can download a package for any version of Hadoop.

Note that, before Spark 2.0, the main programming interface of Spark was the Resilient Distributed Dataset (RDD). After Spark 2.0, RDDs are replaced by Dataset, which is strongly-typed like an RDD, but with richer optimizations under the hood. The RDD interface is still supported, and you can get a more detailed reference at the RDD programming guide. However, we highly recommend you to switch to use Dataset, which has better performance than RDD. See the SQL programming guide to get more information about Dataset.

Interactive Analysis with the Spark Shell

Basics

Spark's shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively. It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python. Start it by running the following in the Spark directory:

```
./bin/pyspark
```

Or if PySpark is installed with pip in your current environment:

```
pyspark
```

Spark's primary abstraction is a distributed collection of items called a Dataset. Datasets can be created from Hadoop InputFormats (such as HDFS files) or by transforming other Datasets. Due to Python's dynamic nature, we don't need the Dataset to be strongly-typed in Python. As a result, all Datasets in Python are Dataset[Row], and we call it

DataFrame to be consistent with the data frame concept in Pandas and R. Let's make a new DataFrame from the text of the README file in the Spark source directory:

```
>>> textFile = spark.read.text("README.md")
```

You can get values from DataFrame directly, by calling some actions, or transform the DataFrame to get a new one. For more details, please read the _[API doc](api/python/index.html#pyspark.sql.DataFrame)_.

```
>>> textFile.count()  # Number of rows in this DataFrame
126

>>> textFile.first()  # First row in this DataFrame
Row(value=u'# Apache Spark')
```

Now let's transform this DataFrame to a new one. We call *filter* to return a new DataFrame with a subset of the lines in the file.

```
>>> linesWithSpark = textFile.filter(textFile.value.contains("Spark"))
```

We can chain together transformations and actions:

```
>>> textFile.filter(textFile.value.contains("Spark")).count()  # How many lines␣
↪contain "Spark"?
15
```

More on Dataset Operations

Dataset actions and transformations can be used for more complex computations. Let's say we want to find the line with the most words:

```
>>> from pyspark.sql.functions import *
>>> textFile.select(size(split(textFile.value, "\s+")).name("numWords")).agg(max(col(
↪"numWords"))).collect()
[Row(max(numWords)=15)]
```

This first maps a line to an integer value and aliases it as "numWords", creating a new DataFrame. *agg* is called on that DataFrame to find the largest word count. The arguments to *select* and *agg* are both *Column*, we can use *df.colName* to get a column from a DataFrame. We can also import pyspark.sql.functions, which provides a lot of convenient functions to build a new Column from an old one.

One common data flow pattern is MapReduce, as popularized by Hadoop. Spark can implement MapReduce flows easily:

```
>>> wordCounts = textFile.select(explode(split(textFile.value, "\s+")).alias("word")).
↪groupBy("word").count()
```

Here, we use the *explode* function in *select*, to transform a Dataset of lines to a Dataset of words, and then combine *groupBy* and *count* to compute the per-word counts in the file as a DataFrame of 2 columns: "word" and "count". To collect the word counts in our shell, we can call *collect*:

```
>>> wordCounts.collect()
[Row(word=u'online', count=1), Row(word=u'graphs', count=1), ...]
```

Caching

Spark also supports pulling data sets into a cluster-wide in-memory cache. This is very useful when data is accessed repeatedly, such as when querying a small "hot" dataset or when running an iterative algorithm like PageRank. As a simple example, let's mark our *linesWithSpark* dataset to be cached:

```
>>> linesWithSpark.cache()
>>> linesWithSpark.count()
15

>>> linesWithSpark.count()
15
```

It may seem silly to use Spark to explore and cache a 100-line text file. The interesting part is that these same functions can be used on very large data sets, even when they are striped across tens or hundreds of nodes. You can also do this interactively by connecting *bin/pyspark* to a cluster, as described in the RDD programming guide.

Self-contained Application

Now we will show how to write an application using the Python API (PySpark).

If you are building a packaged PySpark application or library you can add it to your setup.py file as:

```
install_requires=[
    'pyspark=={site.SPARK_VERSION}'
]
```

As an example, we'll create a simple Spark application, *SimpleApp.py*:

```
"""SimpleApp.py"""
from pyspark.sql import SparkSession

logFile = "YOUR_SPARK_HOME/README.md"  # Should be some file on your system
spark = SparkSession.builder.appName("SimpleApp").getOrCreate()
logData = spark.read.text(logFile).cache()

numAs = logData.filter(logData.value.contains('a')).count()
numBs = logData.filter(logData.value.contains('b')).count()

print("Lines with a: %i, lines with b: %i" % (numAs, numBs))

spark.stop()
```

This program just counts the number of lines containing 'a' and the number containing 'b' in a text file. Note that you'll need to replace YOUR_SPARK_HOME with the location where Spark is installed. As with the Scala and Java examples, we use a SparkSession to create Datasets. For applications that use custom classes or third-party libraries, we can also add code dependencies to *spark-submit* through its *–py-files* argument by packaging them into a .zip file (see *spark-submit –help* for details). *SimpleApp* is simple enough that we do not need to specify any code dependencies.

We can run this application using the *bin/spark-submit* script:

```
# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
  --master local[4] \
  SimpleApp.py
```

(continues on next page)

(continued from previous page)

```
...
Lines with a: 46, Lines with b: 23
```

If you have PySpark pip installed into your environment (e.g., *pip install pyspark*), you can run your application with the regular Python interpreter or use the provided 'spark-submit' as you prefer.

```
# Use the Python interpreter to run your application
$ python SimpleApp.py
...
```

1.1.2 Installation

Binary Distribution

PyPi

Enviornments

NumPy, PyArrow and Pandas

1.1.3 Package Overview

Structure

Functions

1.1.4 Basic Functionality

SparkContext

SparkSession

DataFrame

RDD

Reading CSV and Writing JSON

1.1.5 Tutorials

Handling Missing Data

Plotting and Visualization

1.2 User Guide

1.2.1 PySpark Usage Guide for Pandas with Apache Arrow

Apache Arrow in PySpark

Apache Arrow is an in-memory columnar data format that is used in Spark to efficiently transfer data between JVM and Python processes. This currently is most beneficial to Python users that work with Pandas/NumPy data. Its usage is not automatic and might require some minor changes to configuration or code to take full advantage and ensure compatibility. This guide will give a high-level description of how to use Arrow in Spark and highlight any differences when working with Arrow-enabled data.

Ensure PyArrow Installed

To use Apache Arrow in PySpark, *the recommended version of PyArrow* should be installed. If you install PySpark using pip, then PyArrow can be brought in as an extra dependency of the SQL module with the command `pip install pyspark[sql]`. Otherwise, you must ensure that PyArrow is installed and available on all cluster nodes. You can install using pip or conda from the conda-forge channel. See PyArrow installation for details.

Enabling for Conversion to/from Pandas

Arrow is available as an optimization when converting a Spark DataFrame to a Pandas DataFrame using the call `toPandas()` and when creating a Spark DataFrame from a Pandas DataFrame with `createDataFrame(pandas_df)`. To use Arrow when executing these calls, users need to first set the Spark configuration `spark.sql.execution.arrow.pyspark.enabled` to `true`. This is disabled by default.

In addition, optimizations enabled by `spark.sql.execution.arrow.pyspark.enabled` could fallback automatically to non-Arrow optimization implementation if an error occurs before the actual computation within Spark. This can be controlled by `spark.sql.execution.arrow.pyspark.fallback.enabled`.

```
import numpy as np
import pandas as pd

# Enable Arrow-based columnar data transfers
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")

# Generate a Pandas DataFrame
pdf = pd.DataFrame(np.random.rand(100, 3))

# Create a Spark DataFrame from a Pandas DataFrame using Arrow
df = spark.createDataFrame(pdf)

# Convert the Spark DataFrame back to a Pandas DataFrame using Arrow
result_pdf = df.select("*").toPandas()
```

Using the above optimizations with Arrow will produce the same results as when Arrow is not enabled. Note that even with Arrow, `toPandas()` results in the collection of all records in the DataFrame to the driver program and should be done on a small subset of the data. Not all Spark data types are currently supported and an error can be raised if a column has an unsupported type, see Supported SQL Types. If an error occurs during `createDataFrame()`, Spark will fall back to create the DataFrame without Arrow.

Pandas UDFs (a.k.a. Vectorized UDFs)

Pandas UDFs are user defined functions that are executed by Spark using Arrow to transfer data and Pandas to work with the data, which allows vectorized operations. A Pandas UDF is defined using the `pandas_udf` as a decorator or to wrap the function, and no additional configuration is required. A Pandas UDF behaves as a regular PySpark function API in general.

Before Spark 3.0, Pandas UDFs used to be defined with `PandasUDFType`. From Spark 3.0 with Python 3.6+, you can also use Python type hints. Using Python type hints are preferred and using `PandasUDFType` will be deprecated in the future release.

Note that the type hint should use `pandas.Series` in all cases but there is one variant that `pandas.DataFrame` should be used for its input or output type hint instead when the input or output column is of `StructType`. The following example shows a Pandas UDF which takes long column, string column and struct column, and outputs a struct column. It requires the function to specify the type hints of `pandas.Series` and `pandas.DataFrame` as below:

```
import pandas as pd

from pyspark.sql.functions import pandas_udf

@pandas_udf("col1 string, col2 long")
def func(s1: pd.Series, s2: pd.Series, s3: pd.DataFrame) -> pd.DataFrame:
    s3['col2'] = s1 + s2.str.len()
    return s3

# Create a Spark DataFrame that has three columns including a sturct column.
df = spark.createDataFrame(
    [[1, "a string", ("a nested string",)]],
    "long_col long, string_col string, struct_col struct<col1:string>")

df.printSchema()
# root
# |-- long_column: long (nullable = true)
# |-- string_column: string (nullable = true)
# |-- struct_column: struct (nullable = true)
# |    |-- col1: string (nullable = true)

df.select(func("long_col", "string_col", "struct_col")).printSchema()
# |-- func(long_col, string_col, struct_col): struct (nullable = true)
# |    |-- col1: string (nullable = true)
# |    |-- col2: long (nullable = true)
```

In the following sections, it describes the combinations of the supported type hints. For simplicity, `pandas.DataFrame` variant is omitted.

Series to Series

The type hint can be expressed as `pandas.Series, ... -> pandas.Series`.

By using `pandas_udf` with the function having such type hints above, it creates a Pandas UDF where the given function takes one or more `pandas.Series` and outputs one `pandas.Series`. The output of the function should always be of the same length as the input. Internally, PySpark will execute a Pandas UDF by splitting columns into batches and calling the function for each batch as a subset of the data, then concatenating the results together.

The following example shows how to create this Pandas UDF that computes the product of 2 columns.

```
import pandas as pd

from pyspark.sql.functions import col, pandas_udf
from pyspark.sql.types import LongType

# Declare the function and create the UDF
def multiply_func(a: pd.Series, b: pd.Series) -> pd.Series:
    return a * b

multiply = pandas_udf(multiply_func, returnType=LongType())

# The function for a pandas_udf should be able to execute with local Pandas data
x = pd.Series([1, 2, 3])
print(multiply_func(x, x))
# 0    1
# 1    4
# 2    9
# dtype: int64

# Create a Spark DataFrame, 'spark' is an existing SparkSession
df = spark.createDataFrame(pd.DataFrame(x, columns=["x"]))

# Execute function as a Spark vectorized UDF
df.select(multiply(col("x"), col("x"))).show()
# +-------------------+
# |multiply_func(x, x)|
# +-------------------+
# |                  1|
# |                  4|
# |                  9|
# +-------------------+
```

For detailed usage, please see `pyspark.sql.functions.pandas_udf.

Iterator of Series to Iterator of Series

The type hint can be expressed as `Iterator[pandas.Series]` -> `Iterator[pandas.Series]`.

By using `pandas_udf` with the function having such type hints above, it creates a Pandas UDF where the given function takes an iterator of `pandas.Series` and outputs an iterator of `pandas.Series`. The length of the entire output from the function should be the same length of the entire input; therefore, it can prefetch the data from the input iterator as long as the lengths are the same. In this case, the created Pandas UDF requires one input column when the Pandas UDF is called. To use multiple input columns, a different type hint is required. See Iterator of Multiple Series to Iterator of Series.

It is also useful when the UDF execution requires initializing some states although internally it works identically as Series to Series case. The pseudocode below illustrates the example.

{% highlight python %} @pandas_udf("long") def calculate(iterator: Iterator[pd.Series]) -> Iterator[pd.Series]: # Do some expensive initialization with a state state = very_expensive_initialization() for x in iterator: # Use that state for whole iterator. yield calculate_with_state(x, state)

df.select(calculate("value")).show() {% endhighlight %}

The following example shows how to create this Pandas UDF:

```
from typing import Iterator

import pandas as pd

from pyspark.sql.functions import pandas_udf

pdf = pd.DataFrame([1, 2, 3], columns=["x"])
df = spark.createDataFrame(pdf)

# Declare the function and create the UDF
@pandas_udf("long")
def plus_one(iterator: Iterator[pd.Series]) -> Iterator[pd.Series]:
    for x in iterator:
        yield x + 1

df.select(plus_one("x")).show()
# +-----------+
# |plus_one(x)|
# +-----------+
# |          2|
# |          3|
# |          4|
# +-----------+
```

For detailed usage, please see `pyspark.sql.functions.pandas_udf.

Iterator of Multiple Series to Iterator of Series

The type hint can be expressed as `Iterator[Tuple[pandas.Series, ...]] -> Iterator[pandas.Series]`.

By using `pandas_udf` with the function having such type hints above, it creates a Pandas UDF where the given function takes an iterator of a tuple of multiple `pandas.Series` and outputs an iterator of `pandas.Series`. In this case, the created pandas UDF requires multiple input columns as many as the series in the tuple when the Pandas UDF is called. Otherwise, it has the same characteristics and restrictions as Iterator of Series to Iterator of Series case.

The following example shows how to create this Pandas UDF:

```
from typing import Iterator, Tuple

import pandas as pd

from pyspark.sql.functions import pandas_udf

pdf = pd.DataFrame([1, 2, 3], columns=["x"])
df = spark.createDataFrame(pdf)

# Declare the function and create the UDF
@pandas_udf("long")
def multiply_two_cols(
        iterator: Iterator[Tuple[pd.Series, pd.Series]]) -> Iterator[pd.Series]:
    for a, b in iterator:
        yield a * b

df.select(multiply_two_cols("x", "x")).show()
# +-----------------------+
```

(continues on next page)

(continued from previous page)

```
# |multiply_two_cols(x, x)|
# +-----------------------+
# |                      1|
# |                      4|
# |                      9|
# +-----------------------+
```

For detailed usage, please see `pyspark.sql.functions.pandas_udf.

Series to Scalar

The type hint can be expressed as pandas.Series, ... -> Any.

By using pandas_udf with the function having such type hints above, it creates a Pandas UDF similar to PySpark's aggregate functions. The given function takes pandas.Series and returns a scalar value. The return type should be a primitive data type, and 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. Any should ideally be a specific scalar type accordingly.

This UDF can be also used with groupBy().agg() and `pyspark.sql.Window <api/python/pyspark.sql.html#pyspark.sql.Window>`__. It defines an aggregation from one or more pandas.Series to a scalar value, where each pandas.Series represents a column within the group or window.

Note that this type of UDF does not support partial aggregation and all data for a group or window will be loaded into memory. Also, only unbounded window is supported with Grouped aggregate Pandas UDFs currently. The following example shows how to use this type of UDF to compute mean with a group-by and window operations:

```
import pandas as pd

from pyspark.sql.functions import pandas_udf
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"))

# Declare the function and create the UDF
@pandas_udf("double")
def mean_udf(v: pd.Series) -> float:
    return v.mean()

df.select(mean_udf(df['v'])).show()
# +-----------+
# |mean_udf(v)|
# +-----------+
# |        4.2|
# +-----------+

df.groupby("id").agg(mean_udf(df['v'])).show()
# +---+-----------+
# | id|mean_udf(v)|
# +---+-----------+
# |  1|        1.5|
# |  2|        6.0|
# +---+-----------+

w = Window \
```

(continues on next page)

(continued from previous page)

```
    .partitionBy('id') \
    .rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
df.withColumn('mean_v', mean_udf(df['v']).over(w)).show()
# +---+----+------+
# | id|   v|mean_v|
# +---+----+------+
# |  1| 1.0|   1.5|
# |  1| 2.0|   1.5|
# |  2| 3.0|   6.0|
# |  2| 5.0|   6.0|
# |  2|10.0|   6.0|
# +---+----+------+
```

For detailed usage, please see `pyspark.sql.functions.pandas_udf.

Pandas Function APIs

Pandas Function APIs can directly apply a Python native function against the whole DataFrame by using Pandas instances. Internally it works similarly with Pandas UDFs by using Arrow to transfer data and Pandas to work with the data, which allows vectorized operations. However, A Pandas Function API behaves as a regular API under PySpark DataFrame instead of Column, and Python type hints in Pandas Functions APIs are optional and do not affect how it works internally at this moment although they might be required in the future.

From Spark 3.0, grouped map pandas UDF is now categorized as a separate Pandas Function API, DataFrame.groupby().applyInPandas(). It is still possible to use it with PandasUDFType and DataFrame.groupby().apply() as it was; however, it is preferred to use DataFrame.groupby().applyInPandas() directly. Using PandasUDFType will be deprecated in the future.

Grouped Map

Grouped map operations with Pandas instances are supported by DataFrame.groupby().applyInPandas() which requires a Python function that takes a pandas.DataFrame and return another pandas.DataFrame. It maps each group to each pandas.DataFrame in the Python function.

This API implements the "split-apply-combine" pattern which consists of three steps:

- Split the data into groups by using DataFrame.groupBy.
- Apply a function on each group. The input and output of the function are both pandas.DataFrame. The input data contains all the rows and columns for each group.
- Combine the results into a new PySpark DataFrame.

To use groupBy().applyInPandas(), the user needs to define the following:

- A Python function that defines the computation for each group.
- A StructType object or a string that defines the schema of the output PySpark DataFrame.

The column labels of the returned pandas.DataFrame must either match the field names in the defined output schema if specified as strings, or match the field data types by position if not strings, e.g. integer indices. See pandas.DataFrame on how to label columns when constructing a pandas.DataFrame.

Note that all data for a group will be loaded into memory before the function is applied. This can lead to out of memory exceptions, especially if the group sizes are skewed. The configuration for maxRecordsPerBatch is not applied on groups and it is up to the user to ensure that the grouped data will fit into the available memory.

The following example shows how to use `groupby().applyInPandas()` to subtract the mean from each value in the group.

```
df = spark.createDataFrame(
    [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)],
    ("id", "v"))

def subtract_mean(pdf):
    # pdf is a pandas.DataFrame
    v = pdf.v
    return pdf.assign(v=v - v.mean())

df.groupby("id").applyInPandas(subtract_mean, schema="id long, v double").show()
# +---+----+
# | id|   v|
# +---+----+
# |  1|-0.5|
# |  1| 0.5|
# |  2|-3.0|
# |  2|-1.0|
# |  2| 4.0|
# +---+----+
```

For detailed usage, please see `pyspark.sql.GroupedData.applyInPandas`.

Map

Map operations with Pandas instances are supported by `DataFrame.mapInPandas()` which maps an iterator of `pandas.DataFrames` to another iterator of `pandas.DataFrame` that represents the current PySpark `DataFrame` and returns the result as a PySpark `DataFrame`. The functions takes and outputs an iterator of `pandas.DataFrame`. It can return the output of arbitrary length in contrast to some Pandas UDFs although internally it works similarly with Series to Series Pandas UDF.

The following example shows how to use `mapInPandas()`:

```
df = spark.createDataFrame([(1, 21), (2, 30)], ("id", "age"))

def filter_func(iterator):
    for pdf in iterator:
        yield pdf[pdf.id == 1]

df.mapInPandas(filter_func, schema=df.schema).show()
# +---+---+
# | id|age|
# +---+---+
# |  1| 21|
# +---+---+
```

For detailed usage, please see `pyspark.sql.DataFrame.mapsInPandas`.

Co-grouped Map

Co-grouped map operations with Pandas instances are supported by `DataFrame.groupby().cogroup().applyInPandas()` which allows two PySpark `DataFrame`s to be cogrouped by a common key and then a Python function applied to each cogroup. It consists of the following steps: * Shuffle the data such that the groups of each dataframe which share a key are cogrouped together. * Apply a function to each cogroup. The input of the function is two `pandas.DataFrame` (with an optional tuple representing the key). The output of the function is a `pandas.DataFrame`. * Combine the `pandas.DataFrame`s from all groups into a new PySpark `DataFrame`.

To use `groupBy().cogroup().applyInPandas()`, the user needs to define the following: * A Python function that defines the computation for each cogroup. * A `StructType` object or a string that defines the schema of the output PySpark `DataFrame`.

The column labels of the returned `pandas.DataFrame` must either match the field names in the defined output schema if specified as strings, or match the field data types by position if not strings, e.g. integer indices. See pandas.DataFrame on how to label columns when constructing a `pandas.DataFrame`.

Note that all data for a cogroup will be loaded into memory before the function is applied. This can lead to out of memory exceptions, especially if the group sizes are skewed. The configuration for `maxRecordsPerBatch` is not applied and it is up to the user to ensure that the cogrouped data will fit into the available memory.

The following example shows how to use `groupby().cogroup().applyInPandas()` to perform an asof join between two datasets.

```
import pandas as pd

df1 = spark.createDataFrame(
    [(20000101, 1, 1.0), (20000101, 2, 2.0), (20000102, 1, 3.0), (20000102, 2, 4.0)],
    ("time", "id", "v1"))

df2 = spark.createDataFrame(
    [(20000101, 1, "x"), (20000101, 2, "y")],
    ("time", "id", "v2"))

def asof_join(l, r):
    return pd.merge_asof(l, r, on="time", by="id")

df1.groupby("id").cogroup(df2.groupby("id")).applyInPandas(
    asof_join, schema="time int, id int, v1 double, v2 string").show()
# +--------+---+---+---+
# |    time| id| v1| v2|
# +--------+---+---+---+
# |20000101|  1|1.0|  x|
# |20000102|  1|3.0|  x|
# |20000101|  2|2.0|  y|
# |20000102|  2|4.0|  y|
# +--------+---+---+---+
```

For detailed usage, please see `pyspark.sql.PandasCogroupedOps.applyInPandas().

Usage Notes

Supported SQL Types

Currently, all Spark SQL data types are supported by Arrow-based conversion except `MapType`, `ArrayType` of `TimestampType`, and nested `StructType`.

Setting Arrow Batch Size

Data partitions in Spark are converted into Arrow record batches, which can temporarily lead to high memory usage in the JVM. To avoid possible out of memory exceptions, the size of the Arrow record batches can be adjusted by setting the conf "spark.sql.execution.arrow.maxRecordsPerBatch" to an integer that will determine the maximum number of rows for each batch. The default value is 10,000 records per batch. If the number of columns is large, the value should be adjusted accordingly. Using this limit, each data partition will be made into 1 or more record batches for processing.

Timestamp with Time Zone Semantics

Spark internally stores timestamps as UTC values, and timestamp data that is brought in without a specified time zone is converted as local time to UTC with microsecond resolution. When timestamp data is exported or displayed in Spark, the session time zone is used to localize the timestamp values. The session time zone is set with the configuration 'spark.sql.session.timeZone' and will default to the JVM system local time zone if not set. Pandas uses a `datetime64` type with nanosecond resolution, `datetime64[ns]`, with optional time zone on a per-column basis.

When timestamp data is transferred from Spark to Pandas it will be converted to nanoseconds and each column will be converted to the Spark session time zone then localized to that time zone, which removes the time zone and displays values as local time. This will occur when calling `toPandas()` or `pandas_udf` with timestamp columns.

When timestamp data is transferred from Pandas to Spark, it will be converted to UTC microseconds. This occurs when calling `createDataFrame` with a Pandas DataFrame or when returning a timestamp from a `pandas_udf`. These conversions are done automatically to ensure Spark will have data in the expected format, so it is not necessary to do any of these conversions yourself. Any nanosecond values will be truncated.

Note that a standard UDF (non-Pandas) will load timestamp data as Python datetime objects, which is different than a Pandas timestamp. It is recommended to use Pandas time series functionality when working with timestamps in `pandas_udf`s to get the best performance, see here for details.

Recommended Pandas and PyArrow Versions

For usage with pyspark.sql, the supported versions of Pandas is 0.24.2 and PyArrow is 0.15.1. Higher versions may be used, however, compatibility and data correctness can not be guaranteed and should be verified by the user.

Compatibility Setting for PyArrow >= 0.15.0 and Spark 2.3.x, 2.4.x

Since Arrow 0.15.0, a change in the binary IPC format requires an environment variable to be compatible with previous versions of Arrow <= 0.14.1. This is only necessary to do for PySpark users with versions 2.3.x and 2.4.x that have manually upgraded PyArrow to 0.15.0. The following can be added to `conf/spark-env.sh` to use the legacy Arrow IPC format:

```
ARROW_PRE_0_15_IPC_FORMAT=1
```

This will instruct PyArrow >= 0.15.0 to use the legacy IPC format with the older Arrow Java that is in Spark 2.3.x and 2.4.x. Not setting this environment variable will lead to a similar error as described in SPARK-29367 when running `pandas_udfs` or `toPandas()` with Arrow enabled. More information about the Arrow IPC change can be read on the Arrow 0.15.0 release blog.

1.2.2 Pandas UDFs with Type Hints

Python Type Hints

Pandas UDFs In Spark 2.4

Pandas UDFs

Pandas Function APIs

1.2.3 Working with CSV and JSON

Reading CSV

Transformation in SQL

Writing as JSON ——————a

1.2.4 Configurations

Core Configurations

SQL Configurations

PyArrow Configurations

1.2.5 Monitoring

Python Profiler

Python Debugger

Monitoring Python Workers

1.3 API Reference

This page lists an overview of all public PySpark modules, classes, functions and methods.

1.3.1 Spark SQL

Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API. When computing a result the same execution engine is used, independent of which API/language you are using to express the computation.

This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation.

Core Classes

The classes below are core classes in Spark SQL APIs at PySpark.

SparkSession(sparkContext[, jsparkSession])	The entry point to programming Spark with the Dataset and DataFrame API.
DataFrame(jdf, sql_ctx)	A distributed collection of data grouped into named columns.
Column(jc)	A column in a DataFrame.
Row	A row in *DataFrame*.
GroupedData(jgd, df)	A set of methods for aggregations on a *DataFrame*, created by *DataFrame.groupBy()*.
DataFrameNaFunctions(df)	Functionality for working with missing data in *DataFrame*.
DataFrameStatFunctions(df)	Functionality for statistic functions with *DataFrame*.
Window	Utility functions for defining window in DataFrames.

pyspark.sql.SparkSession

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

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.

__init__ (*sparkContext*, *jsparkSession=None*)
Creates a new SparkSession.

```
>>> from datetime import datetime
>>> spark = SparkSession(sc)
>>> allTypes = sc.parallelize([Row(i=1, s="string", d=1.0, l=1,
...     b=True, list=[1, 2, 3], dict={"s": 0}, row=Row(a=1),
...     time=datetime(2014, 8, 1, 14, 1, 5))])
>>> df = allTypes.toDF()
>>> df.createOrReplaceTempView("allTypes")
>>> spark.sql('select i+1, d+1, not b, list[1], dict["s"], time, row.a '
...           'from allTypes where b and i > 0').collect()
[Row((i + CAST(1 AS BIGINT))=2, (d + CAST(1 AS DOUBLE))=2.0, (NOT b)=False,␣
↪list[1]=2,             dict[s]=0, time=datetime.datetime(2014, 8, 1, 14, 1,␣
↪5), a=1)]
>>> df.rdd.map(lambda x: (x.i, x.s, x.d, x.l, x.b, x.time, x.row.a, x.list)).
↪collect()
[(1, 'string', 1.0, 1, True, datetime.datetime(2014, 8, 1, 14, 1, 5), 1, [1,␣
↪2, 3])]
```

Methods

__init__(sparkContext[, jsparkSession])	Creates a new SparkSession.
createDataFrame(data[, schema, ...])	Creates a *DataFrame* from an RDD, a list or a pandas.DataFrame.
getActiveSession()	Returns the active SparkSession for the current thread, returned by the builder.
newSession()	Returns a new SparkSession as new session, that has separate SQLConf, registered temporary views and UDFs, but shared SparkContext and table cache.
range(start[, end, step, numPartitions])	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.
sql(sqlQuery)	Returns a *DataFrame* representing the result of the given query.
stop()	Stop the underlying SparkContext.
table(tableName)	Returns the specified table as a *DataFrame*.

Attributes

builder	A class attribute having a Builder to construct *SparkSession* instances.
catalog	Interface through which the user may create, drop, alter or query underlying databases, tables, functions, etc.
conf	Runtime configuration interface for Spark.
read	Returns a DataFrameReader that can be used to read data in as a *DataFrame*.

continues on next page

Table 3 – continued from previous page

readStream	Returns a `DataStreamReader` that can be used to read data streams as a streaming *DataFrame*.
sparkContext	Returns the underlying `SparkContext`.
streams	Returns a `StreamingQueryManager` that allows managing all the `StreamingQuery` instances active on *this* context.
udf	Returns a `UDFRegistration` for UDF registration.
version	The version of Spark on which this application is running.

pyspark.sql.DataFrame

class pyspark.sql.**DataFrame**(*jdf*, *sql_ctx*)

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 == department.id) \
  .groupBy(department.name, "gender").agg({"salary": "avg", "age": "max"})
```

New in version 1.3.

__init__(*jdf*, *sql_ctx*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(jdf, sql_ctx)	Initialize self.
agg(*exprs)	Aggregate on the entire *DataFrame* without groups (shorthand for `df.groupBy.agg()`).
alias(alias)	Returns a new *DataFrame* with an alias set.
approxQuantile(col, probabilities, relativeError)	Calculates the approximate quantiles of numerical columns of a *DataFrame*.
cache()	Persists the *DataFrame* with the default storage level (*MEMORY_AND_DISK*).

continues on next page

Table 4 – continued from previous page

checkpoint([eager])	Returns a checkpointed version of this Dataset.
coalesce(numPartitions)	Returns a new DataFrame that has exactly *numPartitions* partitions.
colRegex(colName)	Selects column based on the column name specified as a regex and returns it as Column.
collect()	Returns all the records as a list of Row.
corr(col1, col2[, method])	Calculates the correlation of two columns of a DataFrame as a double value.
count()	Returns the number of rows in this DataFrame.
cov(col1, col2)	Calculate the sample covariance for the given columns, specified by their names, as a double value.
createGlobalTempView(name)	Creates a global temporary view with this DataFrame.
createOrReplaceGlobalTempView(name)	Creates or replaces a global temporary view using the given name.
createOrReplaceTempView(name)	Creates or replaces a local temporary view with this DataFrame.
createTempView(name)	Creates a local temporary view with this DataFrame.
crossJoin(other)	Returns the cartesian product with another DataFrame.
crosstab(col1, col2)	Computes a pair-wise frequency table of the given columns.
cube(*cols)	Create a multi-dimensional cube for the current DataFrame using the specified columns, so we can run aggregations on them.
describe(*cols)	Computes basic statistics for numeric and string columns.
distinct()	Returns a new DataFrame containing the distinct rows in this DataFrame.
drop(*cols)	Returns a new DataFrame that drops the specified column.
dropDuplicates([subset])	Return a new DataFrame with duplicate rows removed, optionally only considering certain columns.
drop_duplicates([subset])	drop_duplicates() is an alias for dropDuplicates().
dropna([how, thresh, subset])	Returns a new DataFrame omitting rows with null values.
exceptAll(other)	Return a new DataFrame containing rows in this DataFrame but not in another DataFrame while preserving duplicates.
explain([extended, mode])	Prints the (logical and physical) plans to the console for debugging purpose.
fillna(value[, subset])	Replace null values, alias for na.fill().
filter(condition)	Filters rows using the given condition.
first()	Returns the first row as a Row.
foreach(f)	Applies the f function to all Row of this DataFrame.
foreachPartition(f)	Applies the f function to each partition of this DataFrame.

continues on next page

Table 4 – continued from previous page

freqItems(cols[, support])	Finding frequent items for columns, possibly with false positives.
groupBy(*cols)	Groups the DataFrame using the specified columns, so we can run aggregation on them.
groupby(*cols)	groupby() is an alias for groupBy().
head([n])	Returns the first n rows.
hint(name, *parameters)	Specifies some hint on the current DataFrame.
inputFiles()	Returns a best-effort snapshot of the files that compose this DataFrame.
intersect(other)	Return a new DataFrame containing rows only in both this DataFrame and another DataFrame.
intersectAll(other)	Return a new DataFrame containing rows in both this DataFrame and another DataFrame while preserving duplicates.
isLocal()	Returns True if the collect() and take() methods can be run locally (without any Spark executors).
join(other[, on, how])	Joins with another DataFrame, using the given join expression.
limit(num)	Limits the result count to the number specified.
localCheckpoint([eager])	Returns a locally checkpointed version of this Dataset.
mapInPandas(func, schema)	Maps an iterator of batches in the current DataFrame using a Python native function that takes and outputs a pandas DataFrame, and returns the result as a DataFrame.
orderBy(*cols, **kwargs)	Returns a new DataFrame sorted by the specified column(s).
persist([storageLevel])	Sets the storage level to persist the contents of the DataFrame across operations after the first time it is computed.
printSchema()	Prints out the schema in the tree format.
randomSplit(weights[, seed])	Randomly splits this DataFrame with the provided weights.
registerTempTable(name)	Registers this DataFrame as a temporary table using the given name.
repartition(numPartitions, *cols)	Returns a new DataFrame partitioned by the given partitioning expressions.
repartitionByRange(numPartitions, *cols)	Returns a new DataFrame partitioned by the given partitioning expressions.
replace(to_replace[, value, subset])	Returns a new DataFrame replacing a value with another value.
rollup(*cols)	Create a multi-dimensional rollup for the current DataFrame using the specified columns, so we can run aggregation on them.
sameSemantics(other)	Returns *True* when the logical query plans inside both DataFrames are equal and therefore return same results.
sample([withReplacement, fraction, seed])	Returns a sampled subset of this DataFrame.
sampleBy(col, fractions[, seed])	Returns a stratified sample without replacement based on the fraction given on each stratum.

continues on next page

Table 4 – continued from previous page

select(*cols)	Projects a set of expressions and returns a new *DataFrame*.
selectExpr(*expr)	Projects a set of SQL expressions and returns a new *DataFrame*.
semanticHash()	Returns a hash code of the logical query plan against this *DataFrame*.
show([n, truncate, vertical])	Prints the first `n` rows to the console.
sort(*cols, **kwargs)	Returns a new *DataFrame* sorted by the specified column(s).
sortWithinPartitions(*cols, **kwargs)	Returns a new *DataFrame* with each partition sorted by the specified column(s).
subtract(other)	Return a new *DataFrame* containing rows in this *DataFrame* but not in another *DataFrame*.
summary(*statistics)	Computes specified statistics for numeric and string columns.
tail(num)	Returns the last `num` rows as a `list` of *Row*.
take(num)	Returns the first `num` rows as a `list` of *Row*.
toDF(*cols)	Returns a new *DataFrame* that with new specified column names
toJSON([use_unicode])	Converts a *DataFrame* into a `RDD` of string.
toLocalIterator([prefetchPartitions])	Returns an iterator that contains all of the rows in this *DataFrame*.
toPandas()	Returns the contents of this *DataFrame* as Pandas `pandas.DataFrame`.
transform(func)	Returns a new *DataFrame*.
union(other)	Return a new *DataFrame* containing union of rows in this and another *DataFrame*.
unionAll(other)	Return a new *DataFrame* containing union of rows in this and another *DataFrame*.
unionByName(other)	Returns a new *DataFrame* containing union of rows in this and another *DataFrame*.
unpersist([blocking])	Marks the *DataFrame* as non-persistent, and remove all blocks for it from memory and disk.
where(condition)	*where()* is an alias for *filter()*.
withColumn(colName, col)	Returns a new *DataFrame* by adding a column or replacing the existing column that has the same name.
withColumnRenamed(existing, new)	Returns a new *DataFrame* by renaming an existing column.
withWatermark(eventTime, delayThreshold)	Defines an event time watermark for this *DataFrame*.

Attributes

columns	Returns all column names as a list.
dtypes	Returns all column names and their data types as a list.
isStreaming	Returns `True` if this `Dataset` contains one or more sources that continuously return data as it arrives.
na	Returns a *DataFrameNaFunctions* for handling missing values.
rdd	Returns the content as an `pyspark.RDD` of *Row*.
schema	Returns the schema of this *DataFrame* as a *pyspark.sql.types.StructType*.
stat	Returns a *DataFrameStatFunctions* for statistic functions.
storageLevel	Get the *DataFrame*'s current storage level.
write	Interface for saving the content of the non-streaming *DataFrame* out into external storage.
writeStream	Interface for saving the content of the streaming *DataFrame* out into external storage.

pyspark.sql.Column

class pyspark.sql.**Column**(*jc*)

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.

__init__(*jc*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(jc)	Initialize self.
alias(*alias, **kwargs)	Returns this column aliased with a new name or names (in the case of expressions that return more than one column, such as explode).
asc()	Returns a sort expression based on ascending order of the column.

continues on next page

Table 6 – continued from previous page

asc_nulls_first()	Returns a sort expression based on ascending order of the column, and null values return before non-null values.
asc_nulls_last()	Returns a sort expression based on ascending order of the column, and null values appear after non-null values.
astype(dataType)	astype() is an alias for cast().
between(lowerBound, upperBound)	A boolean expression that is evaluated to true if the value of this expression is between the given columns.
bitwiseAND(other)	Compute bitwise AND of this expression with another expression.
bitwiseOR(other)	Compute bitwise OR of this expression with another expression.
bitwiseXOR(other)	Compute bitwise XOR of this expression with another expression.
cast(dataType)	Convert the column into type dataType.
contains(other)	Contains the other element.
desc()	Returns a sort expression based on the descending order of the column.
desc_nulls_first()	Returns a sort expression based on the descending order of the column, and null values appear before non-null values.
desc_nulls_last()	Returns a sort expression based on the descending order of the column, and null values appear after non-null values.
endswith(other)	String ends with.
eqNullSafe(other)	Equality test that is safe for null values.
getField(name)	An expression that gets a field by name in a StructField.
getItem(key)	An expression that gets an item at position ordinal out of a list, or gets an item by key out of a dict.
isNotNull()	True if the current expression is NOT null.
isNull()	True if the current expression is null.
isin(*cols)	A boolean expression that is evaluated to true if the value of this expression is contained by the evaluated values of the arguments.
like(other)	SQL like expression.
name(*alias, **kwargs)	name() is an alias for alias().
otherwise(value)	Evaluates a list of conditions and returns one of multiple possible result expressions.
over(window)	Define a windowing column.
rlike(other)	SQL RLIKE expression (LIKE with Regex).
startswith(other)	String starts with.
substr(startPos, length)	Return a *Column* which is a substring of the column.
when(condition, value)	Evaluates a list of conditions and returns one of multiple possible result expressions.

pyspark.sql.Row

class pyspark.sql.**Row**

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. It is not allowed to omit a named argument to represent that the value is None or missing. This should be explicitly set to None in this case.

NOTE: As of Spark 3.0.0, Rows created from named arguments no longer have field names sorted alphabetically and will be ordered in the position as entered. To enable sorting for Rows compatible with Spark 2.x, set the environment variable "PYSPARK_ROW_FIELD_SORTING_ENABLED" to "true". This option is deprecated and will be removed in future versions of Spark. For Python versions < 3.6, the order of named arguments is not guaranteed to be the same as entered, see https://www.python.org/dev/peps/pep-0468. In this case, a warning will be issued and the Row will fallback to sort the field names automatically.

NOTE: Examples with Row in pydocs are run with the environment variable "PYSPARK_ROW_FIELD_SORTING_ENABLED" set to "true" which results in output where fields are sorted.

```
>>> 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
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
asDict([recursive])	Return as a dict
count	Return number of occurrences of value.
index	Return first index of value.

pyspark.sql.GroupedData

class pyspark.sql.**GroupedData**(*jgd*, *df*)
A set of methods for aggregations on a *DataFrame*, created by *DataFrame.groupBy()*.

New in version 1.3.

__init__(*jgd*, *df*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(jgd, df)	Initialize self.
agg(*exprs)	Compute aggregates and returns the result as a *DataFrame*.
apply(udf)	It is an alias of *pyspark.sql.GroupedData.applyInPandas()*; however, it takes a *pyspark.sql.functions.pandas_udf()* whereas *pyspark.sql.GroupedData.applyInPandas()* takes a Python native function.
applyInPandas(func, schema)	Maps each group of the current *DataFrame* using a pandas udf and returns the result as a *DataFrame*.
avg(*cols)	Computes average values for each numeric columns for each group.
cogroup(other)	Cogroups this group with another group so that we can run cogrouped operations.
count()	Counts the number of records for each group.
max(*cols)	Computes the max value for each numeric columns for each group.
mean(*cols)	Computes average values for each numeric columns for each group.
min(*cols)	Computes the min value for each numeric column for each group.
pivot(pivot_col[, values])	Pivots a column of the current *DataFrame* and perform the specified aggregation.
sum(*cols)	Compute the sum for each numeric columns for each group.

pyspark.sql.DataFrameNaFunctions

class pyspark.sql.**DataFrameNaFunctions**(*df*)
Functionality for working with missing data in *DataFrame*.

New in version 1.4.

__init__(*df*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df)	Initialize self.
drop([how, thresh, subset])	Returns a new *DataFrame* omitting rows with null values.
fill(value[, subset])	Replace null values, alias for `na.fill()`.
replace(to_replace[, value, subset])	Returns a new *DataFrame* replacing a value with another value.

pyspark.sql.DataFrameStatFunctions

class pyspark.sql.**DataFrameStatFunctions**(*df*)
Functionality for statistic functions with *DataFrame*.

New in version 1.4.

__init__(*df*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df)	Initialize self.
approxQuantile(col, probabilities, relativeError)	Calculates the approximate quantiles of numerical columns of a *DataFrame*.
corr(col1, col2[, method])	Calculates the correlation of two columns of a *DataFrame* as a double value.
cov(col1, col2)	Calculate the sample covariance for the given columns, specified by their names, as a double value.
crosstab(col1, col2)	Computes a pair-wise frequency table of the given columns.
freqItems(cols[, support])	Finding frequent items for columns, possibly with false positives.
sampleBy(col, fractions[, seed])	Returns a stratified sample without replacement based on the fraction given on each stratum.

pyspark.sql.Window

class pyspark.sql.**Window**

Utility functions for defining window in DataFrames.

For example:

```
>>> # ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
>>> window = Window.orderBy("date").rowsBetween(Window.unboundedPreceding, Window.
↪currentRow)
```

```
>>> # PARTITION BY country ORDER BY date RANGE BETWEEN 3 PRECEDING AND 3 FOLLOWING
>>> window = Window.orderBy("date").partitionBy("country").rangeBetween(-3, 3)
```

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.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
orderBy(*cols)	Creates a WindowSpec with the ordering defined.
partitionBy(*cols)	Creates a WindowSpec with the partitioning defined.
rangeBetween(start, end)	Creates a WindowSpec with the frame boundaries defined, from *start* (inclusive) to *end* (inclusive).
rowsBetween(start, end)	Creates a WindowSpec with the frame boundaries defined, from *start* (inclusive) to *end* (inclusive).

Attributes

currentRow
unboundedFollowing
unboundedPreceding

Spark Session APIs

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

SparkSession.builder	A class attribute having a Builder to construct *SparkSession* instances.
SparkSession.builder.appName(name)	Sets a name for the application, which will be shown in the Spark web UI.
SparkSession.builder.config([key, value, conf])	Sets a config option.
SparkSession.builder.enableHiveSupport()	Enables Hive support, including connectivity to a persistent Hive metastore, support for Hive SerDes, and Hive user-defined functions.
SparkSession.builder.getOrCreate()	Gets an existing *SparkSession* or, if there is no existing one, creates a new one based on the options set in this builder.
SparkSession.builder.master(master)	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.
SparkSession.catalog	Interface through which the user may create, drop, alter or query underlying databases, tables, functions, etc.
SparkSession.conf	Runtime configuration interface for Spark.
SparkSession.createDataFrame(data[, schema, ...])	Creates a *DataFrame* from an RDD, a list or a pandas.DataFrame.
SparkSession.getActiveSession()	Returns the active SparkSession for the current thread, returned by the builder.
SparkSession.newSession()	Returns a new SparkSession as new session, that has separate SQLConf, registered temporary views and UDFs, but shared SparkContext and table cache.
SparkSession.range(start[, end, step, ...])	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.
SparkSession.read	Returns a DataFrameReader that can be used to read data in as a *DataFrame*.
SparkSession.readStream	Returns a DataStreamReader that can be used to read data streams as a streaming *DataFrame*.
SparkSession.sparkContext	Returns the underlying SparkContext.
SparkSession.sql(sqlQuery)	Returns a *DataFrame* representing the result of the given query.
SparkSession.stop()	Stop the underlying SparkContext.
SparkSession.streams	Returns a StreamingQueryManager that allows managing all the StreamingQuery instances active on *this* context.
SparkSession.table(tableName)	Returns the specified table as a *DataFrame*.
SparkSession.udf	Returns a UDFRegistration for UDF registration.
SparkSession.version	The version of Spark on which this application is running.

pyspark.sql.SparkSession.builder

SparkSession.**builder = <pyspark.sql.session.SparkSession.Builder object>**
A class attribute having a Builder to construct *SparkSession* instances.

pyspark.sql.SparkSession.builder.appName

builder.**appName**(*name*)
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.

pyspark.sql.SparkSession.builder.config

builder.**config**(*key=None*, *value=None*, *conf=None*)
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", "some-value")
<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.

pyspark.sql.SparkSession.builder.enableHiveSupport

builder.**enableHiveSupport**()
Enables Hive support, including connectivity to a persistent Hive metastore, support for Hive SerDes, and Hive user-defined functions.

New in version 2.0.

pyspark.sql.SparkSession.builder.getOrCreate

builder.**getOrCreate**()

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").getOrCreate()
>>> 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").getOrCreate()
>>> s1.conf.get("k1") == s2.conf.get("k1")
True
>>> s1.conf.get("k2") == s2.conf.get("k2")
True
```

New in version 2.0.

pyspark.sql.SparkSession.builder.master

builder.**master**(*master*)

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.

pyspark.sql.SparkSession.catalog

property SparkSession.**catalog**

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

Returns Catalog

New in version 2.0.

pyspark.sql.SparkSession.conf

property SparkSession.**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.

pyspark.sql.SparkSession.createDataFrame

SparkSession.**createDataFrame**(*data*, *schema=None*, *samplingRatio=None*, *verifySchema=True*)

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.

pyspark.sql.SparkSession.getActiveSession

classmethod SparkSession.**getActiveSession**()

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.

pyspark.sql.SparkSession.newSession

SparkSession.**newSession**()

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.

pyspark.sql.SparkSession.range

SparkSession.**range**(*start*, *end=None*, *step=1*, *numPartitions=None*)

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.

pyspark.sql.SparkSession.read

property SparkSession.**read**

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

Returns `DataFrameReader`

New in version 2.0.

pyspark.sql.SparkSession.readStream

property SparkSession.**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.

pyspark.sql.SparkSession.sparkContext

property SparkSession.**sparkContext**
Returns the underlying SparkContext.

New in version 2.0.

pyspark.sql.SparkSession.sql

SparkSession.**sql**(*sqlQuery*)
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 table1")
>>> df2.collect()
[Row(f1=1, f2='row1'), Row(f1=2, f2='row2'), Row(f1=3, f2='row3')]
```

New in version 2.0.

pyspark.sql.SparkSession.stop

SparkSession.**stop**()
Stop the underlying SparkContext.

New in version 2.0.

pyspark.sql.SparkSession.streams

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

Note: Evolving.

Returns StreamingQueryManager

New in version 2.0.

pyspark.sql.SparkSession.table

SparkSession.**table**(*tableName*)
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.

pyspark.sql.SparkSession.udf

property SparkSession.**udf**

Returns a UDFRegistration for UDF registration.

Returns UDFRegistration

New in version 2.0.

pyspark.sql.SparkSession.version

property SparkSession.**version**

The version of Spark on which this application is running.

New in version 2.0.

Input and Output

DataFrameReader.csv(path[, schema, sep, ...])	Loads a CSV file and returns the result as a *DataFrame*.
DataFrameReader.format(source)	Specifies the input data source format.
DataFrameReader.jdbc(url, table[, column, ...])	Construct a *DataFrame* representing the database table named table accessible via JDBC URL url and connection properties.
DataFrameReader.json(path[, schema, ...])	Loads JSON files and returns the results as a *DataFrame*.
DataFrameReader.load([path, format, schema])	Loads data from a data source and returns it as a *DataFrame*.
DataFrameReader.option(key, value)	Adds an input option for the underlying data source.
DataFrameReader.options(**options)	Adds input options for the underlying data source.
DataFrameReader.orc(path[, mergeSchema, ...])	Loads ORC files, returning the result as a *DataFrame*.
DataFrameReader.parquet(*paths, **options)	Loads Parquet files, returning the result as a *DataFrame*.
DataFrameReader.schema(schema)	Specifies the input schema.
DataFrameReader.table(tableName)	Returns the specified table as a *DataFrame*.
DataFrameReader.text(paths[, wholetext, ...])	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.
DataFrameWriter.bucketBy(numBuckets, col, *cols)	Buckets the output by the given columns.If specified, the output is laid out on the file system similar to Hive's bucketing scheme.
DataFrameWriter.csv(path[, mode, ...])	Saves the content of the *DataFrame* in CSV format at the specified path.
DataFrameWriter.format(source)	Specifies the underlying output data source.
DataFrameWriter.insertInto(tableName[, ...])	Inserts the content of the *DataFrame* to the specified table.
DataFrameWriter.jdbc(url, table[, mode, ...])	Saves the content of the *DataFrame* to an external database table via JDBC.

continues on next page

Table 14 – continued from previous page

DataFrameWriter.json(path[, mode, ...])	Saves the content of the DataFrame in JSON format (JSON Lines text format or newline-delimited JSON) at the specified path.
DataFrameWriter.mode(saveMode)	Specifies the behavior when data or table already exists.
DataFrameWriter.option(key, value)	Adds an output option for the underlying data source.
DataFrameWriter.options(**options)	Adds output options for the underlying data source.
DataFrameWriter.orc(path[, mode, ...])	Saves the content of the DataFrame in ORC format at the specified path.
DataFrameWriter.parquet(path[, mode, ...])	Saves the content of the DataFrame in Parquet format at the specified path.
DataFrameWriter.partitionBy(*cols)	Partitions the output by the given columns on the file system.
DataFrameWriter.save([path, format, mode, ...])	Saves the contents of the DataFrame to a data source.
DataFrameWriter.saveAsTable(name[, format, ...])	Saves the content of the DataFrame as the specified table.
DataFrameWriter.sortBy(col, *cols)	Sorts the output in each bucket by the given columns on the file system.
DataFrameWriter.text(path[, compression, ...])	Saves the content of the DataFrame in a text file at the specified path.

pyspark.sql.DataFrameReader.csv

DataFrameReader.**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*, *pathGlobFilter=None*, *recursiveFileLookup=None*)

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 **`datetime pattern`_**. This applies to date type. If None is set, it uses the default value, `yyyy-MM-dd`.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at **`datetime pattern`_**. This applies to timestamp type. If None is set, it uses the default value, `yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX]`.
- **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.
- **`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`_**.
- **`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.

pyspark.sql.DataFrameReader.format

DataFrameReader.**format**(*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/sql/people.json')
>>> df.dtypes
[('age', 'bigint'), ('name', 'string')]
```

New in version 1.4.

pyspark.sql.DataFrameReader.jdbc

DataFrameReader.**jdbc**(*url*, *table*, *column=None*, *lowerBound=None*, *upperBound=None*, *numPartitions=None*, *predicates=None*, *properties=None*)

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.

pyspark.sql.DataFrameReader.json

DataFrameReader.**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*, *pathGlobFilter=None*, *recursiveFileLookup=None*)

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 datetime pattern. This applies to date type. If None is set, it uses the default value, yyyy-MM-dd.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at datetime pattern. This applies to timestamp type. If None is set, it uses the default value, yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX].
- **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.
- **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.
- **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.

pyspark.sql.DataFrameReader.load

DataFrameReader.**load**(*path=None*, *format=None*, *schema=None*, ***options*)

Loads data from a data source and returns it as a *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/sql/parquet_
↪partitioned',
...     opt1=True, opt2=1, opt3='str')
>>> df.dtypes
[('name', 'string'), ('year', 'int'), ('month', 'int'), ('day', 'int')]
```

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

New in version 1.4.

pyspark.sql.DataFrameReader.option

DataFrameReader.**option**(*key*, *value*)

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 time zone ID to be used to parse** timestamps in the JSON/CSV datasources or partition values. The following formats of *timeZone* are supported:
  - Region-based zone ID: It should have the form 'area/city', such as 'America/Los_Angeles'.
  - Zone offset: It should be in the format '(+|-)HH:mm', for example '-08:00' or '+01:00'. Also 'UTC' and 'Z' are supported as aliases of '+00:00'.

  Other short names like 'CST' are not recommended to use because they can be ambiguous. If it isn't set, the current value of the SQL config `spark.sql.session.timeZone` is used by default.
- **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.

pyspark.sql.DataFrameReader.options

DataFrameReader.**options**(***options*)

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 time zone ID to be used to parse** timestamps in the JSON/CSV datasources or partition values. The following formats of *timeZone* are supported:
  - Region-based zone ID: It should have the form 'area/city', such as 'America/Los_Angeles'.
  - Zone offset: It should be in the format '(+|-)HH:mm', for example '-08:00' or '+01:00'. Also 'UTC' and 'Z' are supported as aliases of '+00:00'.

  Other short names like 'CST' are not recommended to use because they can be ambiguous. If it isn't set, the current value of the SQL config `spark.sql.session.timeZone` is used by default.
- **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.

pyspark.sql.DataFrameReader.orc

DataFrameReader.**orc**(*path*, *mergeSchema=None*, *pathGlobFilter=None*, *recursiveFileLookup=None*)

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

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

New in version 1.5.

pyspark.sql.DataFrameReader.parquet

DataFrameReader.**parquet**(**paths*, ***options*)

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

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

New in version 1.4.

pyspark.sql.DataFrameReader.schema

DataFrameReader.**schema**(*schema*)

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.

pyspark.sql.DataFrameReader.table

DataFrameReader.**table**(*tableName*)

Returns the specified table as a *DataFrame*.

Parameters **tableName** – string, name of the table.

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

New in version 1.4.

pyspark.sql.DataFrameReader.text

DataFrameReader.**text**(*paths*, *wholetext=False*, *lineSep=None*, *pathGlobFilter=None*, *recursiveFileLookup=None*)

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.
- **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`_**.
- **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', wholetext=True)
>>> df.collect()
[Row(value='hello\nthis')]
```

New in version 1.6.

pyspark.sql.DataFrameWriter.bucketBy

DataFrameWriter.**bucketBy**(*numBuckets*, *col*, **cols*)

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.

pyspark.sql.DataFrameWriter.csv

DataFrameWriter.**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*)

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 **`datetime pattern`_**. This applies to date type. If None is set, it uses the default value, `yyyy-MM-dd`.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at **`datetime pattern`_**. This applies to timestamp type. If None is set, it uses the default value, `yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX]`.
- **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.

pyspark.sql.DataFrameWriter.format

`DataFrameWriter.`**`format`**(*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(), 'data'))
```

New in version 1.4.

pyspark.sql.DataFrameWriter.insertInto

`DataFrameWriter.`**`insertInto`**(*tableName*, *overwrite=None*)

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

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

Optionally overwriting any existing data.

New in version 1.4.

pyspark.sql.DataFrameWriter.jdbc

`DataFrameWriter.`**`jdbc`**(*url*, *table*, *mode=None*, *properties=None*)

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.

pyspark.sql.DataFrameWriter.json

DataFrameWriter.**json**(*path*, *mode=None*, *compression=None*, *dateFormat=None*, *timestampFormat=None*, *lineSep=None*, *encoding=None*, *ignoreNullFields=None*)

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 **`datetime pattern`_**. This applies to date type. If None is set, it uses the default value, yyyy-MM-dd.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at **`datetime pattern`_**. This applies to timestamp type. If None is set, it uses the default value, yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX].
- **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.

pyspark.sql.DataFrameWriter.mode

DataFrameWriter.**mode**(*saveMode*)

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.mkdtemp(), 'data'))
```

New in version 1.4.

pyspark.sql.DataFrameWriter.option

DataFrameWriter.**option**(*key*, *value*)

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 time zone ID to be used to format** timestamps in the JSON/CSV datasources or partition values. The following formats of *timeZone* are supported:
  - Region-based zone ID: It should have the form 'area/city', such as 'America/Los_Angeles'.
  - Zone offset: It should be in the format '(+|-)HH:mm', for example '-08:00' or '+01:00'. Also 'UTC' and 'Z' are supported as aliases of '+00:00'.

  Other short names like 'CST' are not recommended to use because they can be ambiguous. If it isn't set, the current value of the SQL config `spark.sql.session.timeZone` is used by default.

New in version 1.5.

pyspark.sql.DataFrameWriter.options

DataFrameWriter.**options**(***options*)

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 time zone ID to be used to format** timestamps in the JSON/CSV datasources or partition values. The following formats of *timeZone* are supported:
  - Region-based zone ID: It should have the form 'area/city', such as 'America/Los_Angeles'.
  - Zone offset: It should be in the format '(+|-)HH:mm', for example '-08:00' or '+01:00'. Also 'UTC' and 'Z' are supported as aliases of '+00:00'.

  Other short names like 'CST' are not recommended to use because they can be ambiguous. If it isn't set, the current value of the SQL config `spark.sql.session.timeZone` is used by default.

New in version 1.4.

pyspark.sql.DataFrameWriter.orc

DataFrameWriter.**orc**(*path*, *mode=None*, *partitionBy=None*, *compression=None*)

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_partitioned')
>>> orc_df.write.orc(os.path.join(tempfile.mkdtemp(), 'data'))
```

New in version 1.5.

pyspark.sql.DataFrameWriter.parquet

DataFrameWriter.**parquet**(*path*, *mode=None*, *partitionBy=None*, *compression=None*)

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.

pyspark.sql.DataFrameWriter.partitionBy

DataFrameWriter.**partitionBy**(**cols*)

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(tempfile.mkdtemp(),
↪ 'data'))
```

New in version 1.4.

pyspark.sql.DataFrameWriter.save

DataFrameWriter.**save**(*path=None*, *format=None*, *mode=None*, *partitionBy=None*, ***options*)

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(), 'data'))
```

New in version 1.4.

pyspark.sql.DataFrameWriter.saveAsTable

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

Saves the content of the *DataFrame* as the specified table.

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.

pyspark.sql.DataFrameWriter.sortBy

DataFrameWriter.**sortBy**(*col*, **cols*)

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.

pyspark.sql.DataFrameWriter.text

DataFrameWriter.**text**(*path*, *compression=None*, *lineSep=None*)

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.

DataFrame APIs

DataFrame.agg(*exprs)	Aggregate on the entire DataFrame without groups (shorthand for df.groupBy.agg()).
DataFrame.alias(alias)	Returns a new DataFrame with an alias set.
DataFrame.approxQuantile(col, probabilities, ...)	Calculates the approximate quantiles of numerical columns of a DataFrame.
DataFrame.cache()	Persists the DataFrame with the default storage level (*MEMORY_AND_DISK*).
DataFrame.checkpoint([eager])	Returns a checkpointed version of this Dataset.
DataFrame.coalesce(numPartitions)	Returns a new DataFrame that has exactly *numPartitions* partitions.
DataFrame.colRegex(colName)	Selects column based on the column name specified as a regex and returns it as Column.
DataFrame.collect()	Returns all the records as a list of Row.
DataFrame.columns	Returns all column names as a list.
DataFrame.corr(col1, col2[, method])	Calculates the correlation of two columns of a DataFrame as a double value.
DataFrame.count()	Returns the number of rows in this DataFrame.
DataFrame.cov(col1, col2)	Calculate the sample covariance for the given columns, specified by their names, as a double value.
DataFrame.createGlobalTempView(name)	Creates a global temporary view with this DataFrame.
DataFrame.createOrReplaceGlobalTempView(name)	Creates or replaces a global temporary view using the given name.
DataFrame.createOrReplaceTempView(name)	Creates or replaces a local temporary view with this DataFrame.
DataFrame.createTempView(name)	Creates a local temporary view with this DataFrame.
DataFrame.crossJoin(other)	Returns the cartesian product with another DataFrame.
DataFrame.crosstab(col1, col2)	Computes a pair-wise frequency table of the given columns.
DataFrame.cube(*cols)	Create a multi-dimensional cube for the current DataFrame using the specified columns, so we can run aggregations on them.
DataFrame.describe(*cols)	Computes basic statistics for numeric and string columns.
DataFrame.distinct()	Returns a new DataFrame containing the distinct rows in this DataFrame.
DataFrame.drop(*cols)	Returns a new DataFrame that drops the specified column.
DataFrame.dropDuplicates([subset])	Return a new DataFrame with duplicate rows removed, optionally only considering certain columns.
DataFrame.drop_duplicates([subset])	drop_duplicates() is an alias for dropDuplicates().
DataFrame.dropna([how, thresh, subset])	Returns a new DataFrame omitting rows with null values.
DataFrame.dtypes	Returns all column names and their data types as a list.
DataFrame.exceptAll(other)	Return a new DataFrame containing rows in this DataFrame but not in another DataFrame while preserving duplicates.

continues on next page

Table 15 – continued from previous page

DataFrame.explain([extended, mode])	Prints the (logical and physical) plans to the console for debugging purpose.
DataFrame.fillna(value[, subset])	Replace null values, alias for na.fill().
DataFrame.filter(condition)	Filters rows using the given condition.
DataFrame.first()	Returns the first row as a Row.
DataFrame.foreach(f)	Applies the f function to all Row of this DataFrame.
DataFrame.foreachPartition(f)	Applies the f function to each partition of this DataFrame.
DataFrame.freqItems(cols[, support])	Finding frequent items for columns, possibly with false positives.
DataFrame.groupBy(*cols)	Groups the DataFrame using the specified columns, so we can run aggregation on them.
DataFrame.groupby(*cols)	groupby() is an alias for groupBy().
DataFrame.head([n])	Returns the first n rows.
DataFrame.hint(name, *parameters)	Specifies some hint on the current DataFrame.
DataFrame.inputFiles()	Returns a best-effort snapshot of the files that compose this DataFrame.
DataFrame.intersect(other)	Return a new DataFrame containing rows only in both this DataFrame and another DataFrame.
DataFrame.intersectAll(other)	Return a new DataFrame containing rows in both this DataFrame and another DataFrame while preserving duplicates.
DataFrame.isLocal()	Returns True if the collect() and take() methods can be run locally (without any Spark executors).
DataFrame.isStreaming	Returns True if this Dataset contains one or more sources that continuously return data as it arrives.
DataFrame.join(other[, on, how])	Joins with another DataFrame, using the given join expression.
DataFrame.limit(num)	Limits the result count to the number specified.
DataFrame.localCheckpoint([eager])	Returns a locally checkpointed version of this Dataset.
DataFrame.mapInPandas(func, schema)	Maps an iterator of batches in the current DataFrame using a Python native function that takes and outputs a pandas DataFrame, and returns the result as a DataFrame.
DataFrame.na	Returns a DataFrameNaFunctions for handling missing values.
DataFrame.orderBy(*cols, **kwargs)	Returns a new DataFrame sorted by the specified column(s).
DataFrame.persist([storageLevel])	Sets the storage level to persist the contents of the DataFrame across operations after the first time it is computed.
DataFrame.printSchema()	Prints out the schema in the tree format.
DataFrame.randomSplit(weights[, seed])	Randomly splits this DataFrame with the provided weights.
DataFrame.rdd	Returns the content as an pyspark.RDD of Row.
DataFrame.registerTempTable(name)	Registers this DataFrame as a temporary table using the given name.
DataFrame.repartition(numPartitions, *cols)	Returns a new DataFrame partitioned by the given partitioning expressions.
DataFrame.repartitionByRange(numPartitions, ...)	Returns a new DataFrame partitioned by the given partitioning expressions.

continues on next page

Table 15 – continued from previous page

DataFrame.replace(to_replace[, value, subset])	Returns a new *DataFrame* replacing a value with another value.
DataFrame.rollup(*cols)	Create a multi-dimensional rollup for the current *DataFrame* using the specified columns, so we can run aggregation on them.
DataFrame.sameSemantics(other)	Returns *True* when the logical query plans inside both *DataFrame*s are equal and therefore return same results.
DataFrame.sample([withReplacement, ...])	Returns a sampled subset of this *DataFrame*.
DataFrame.sampleBy(col, fractions[, seed])	Returns a stratified sample without replacement based on the fraction given on each stratum.
DataFrame.schema	Returns the schema of this *DataFrame* as a *pyspark.sql.types.StructType*.
DataFrame.select(*cols)	Projects a set of expressions and returns a new *DataFrame*.
DataFrame.selectExpr(*expr)	Projects a set of SQL expressions and returns a new *DataFrame*.
DataFrame.semanticHash()	Returns a hash code of the logical query plan against this *DataFrame*.
DataFrame.show([n, truncate, vertical])	Prints the first `n` rows to the console.
DataFrame.sort(*cols, **kwargs)	Returns a new *DataFrame* sorted by the specified column(s).
DataFrame.sortWithinPartitions(*cols, **kwargs)	Returns a new *DataFrame* with each partition sorted by the specified column(s).
DataFrame.stat	Returns a *DataFrameStatFunctions* for statistic functions.
DataFrame.storageLevel	Get the *DataFrame*'s current storage level.
DataFrame.subtract(other)	Return a new *DataFrame* containing rows in this *DataFrame* but not in another *DataFrame*.
DataFrame.summary(*statistics)	Computes specified statistics for numeric and string columns.
DataFrame.tail(num)	Returns the last `num` rows as a `list` of *Row*.
DataFrame.take(num)	Returns the first `num` rows as a `list` of *Row*.
DataFrame.toDF(*cols)	Returns a new *DataFrame* that with new specified column names
DataFrame.toJSON([use_unicode])	Converts a *DataFrame* into a `RDD` of string.
DataFrame.toLocalIterator([prefetchPartitions])	Returns an iterator that contains all of the rows in this *DataFrame*.
DataFrame.toPandas()	Returns the contents of this *DataFrame* as Pandas `pandas.DataFrame`.
DataFrame.transform(func)	Returns a new *DataFrame*.
DataFrame.union(other)	Return a new *DataFrame* containing union of rows in this and another *DataFrame*.
DataFrame.unionAll(other)	Return a new *DataFrame* containing union of rows in this and another *DataFrame*.
DataFrame.unionByName(other)	Returns a new *DataFrame* containing union of rows in this and another *DataFrame*.
DataFrame.unpersist([blocking])	Marks the *DataFrame* as non-persistent, and remove all blocks for it from memory and disk.
DataFrame.where(condition)	`where()` is an alias for `filter()`.

continues on next page

Table 15 – continued from previous page

DataFrame.withColumn(colName, col)	Returns a new *DataFrame* by adding a column or replacing the existing column that has the same name.
DataFrame.withColumnRenamed(existing, new)	Returns a new *DataFrame* by renaming an existing column.
DataFrame.withWatermark(eventTime, ...)	Defines an event time watermark for this *DataFrame*.
DataFrame.write	Interface for saving the content of the non-streaming *DataFrame* out into external storage.
DataFrame.writeStream	Interface for saving the content of the streaming *DataFrame* out into external storage.
DataFrameNaFunctions.drop([how, thresh, subset])	Returns a new *DataFrame* omitting rows with null values.
DataFrameNaFunctions.fill(value[, subset])	Replace null values, alias for `na.fill()`.
DataFrameNaFunctions.replace(to_replace[, ...])	Returns a new *DataFrame* replacing a value with another value.
DataFrameStatFunctions.approxQuantile(col, ...)	Calculates the approximate quantiles of numerical columns of a *DataFrame*.
DataFrameStatFunctions.corr(col1, col2[, method])	Calculates the correlation of two columns of a *DataFrame* as a double value.
DataFrameStatFunctions.cov(col1, col2)	Calculate the sample covariance for the given columns, specified by their names, as a double value.
DataFrameStatFunctions.crosstab(col1, col2)	Computes a pair-wise frequency table of the given columns.
DataFrameStatFunctions.freqItems(cols[, support])	Finding frequent items for columns, possibly with false positives.
DataFrameStatFunctions.sampleBy(col, fractions)	Returns a stratified sample without replacement based on the fraction given on each stratum.

pyspark.sql.DataFrame.agg

DataFrame.**agg**(**exprs*)

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.

pyspark.sql.DataFrame.alias

DataFrame.**alias**(*alias*)

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("df_as2.name"),
↪'inner')
```

(continues on next page)

(continued from previous page)

```
>>> joined_df.select("df_as1.name", "df_as2.name", "df_as2.age")                .
↪sort(desc("df_as1.name")).collect()
[Row(name='Bob', name='Bob', age=5), Row(name='Alice', name='Alice', age=2)]
```

New in version 1.3.

pyspark.sql.DataFrame.approxQuantile

DataFrame.**approxQuantile**(*col*, *probabilities*, *relativeError*)

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 [[https://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.

pyspark.sql.DataFrame.cache

DataFrame.**cache**()

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.

pyspark.sql.DataFrame.checkpoint

DataFrame.**checkpoint**(*eager=True*)

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.

pyspark.sql.DataFrame.coalesce

DataFrame.**coalesce**(*numPartitions*)

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.

pyspark.sql.DataFrame.colRegex

DataFrame.**colRegex**(*colName*)

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)], ["Col1", "Col2"])
>>> df.select(df.colRegex("`(Col1)?+.+`")).show()
+----+
|Col2|
+----+
|   1|
|   2|
|   3|
+----+
```

New in version 2.3.

pyspark.sql.DataFrame.collect

DataFrame.**collect**()

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.

pyspark.sql.DataFrame.columns

property DataFrame.**columns**

Returns all column names as a list.

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

New in version 1.3.

pyspark.sql.DataFrame.corr

DataFrame.**corr**(*col1*, *col2*, *method=None*)

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.

pyspark.sql.DataFrame.count

DataFrame.**count**()

Returns the number of rows in this *DataFrame*.

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

New in version 1.3.

pyspark.sql.DataFrame.cov

DataFrame.**cov**(*col1*, *col2*)

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.

pyspark.sql.DataFrame.createGlobalTempView

DataFrame.**createGlobalTempView**(*name*)

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.

pyspark.sql.DataFrame.createOrReplaceGlobalTempView

DataFrame.**createOrReplaceGlobalTempView**(*name*)

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.

pyspark.sql.DataFrame.createOrReplaceTempView

DataFrame.**createOrReplaceTempView**(*name*)

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.

pyspark.sql.DataFrame.createTempView

DataFrame.**createTempView**(*name*)

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.

pyspark.sql.DataFrame.crossJoin

DataFrame.**crossJoin**(*other*)

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", "height").collect()
[Row(age=2, name='Alice', height=80), Row(age=2, name='Alice', height=85),
 Row(age=5, name='Bob', height=80), Row(age=5, name='Bob', height=85)]
```

New in version 2.1.

pyspark.sql.DataFrame.crosstab

DataFrame.**crosstab**(*col1*, *col2*)

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.

pyspark.sql.DataFrame.cube

DataFrame.**cube**(**cols*)

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").show()
+-----+----+-----+
| 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.

pyspark.sql.DataFrame.describe

DataFrame.**describe**(**cols*)

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|
```

(continues on next page)

(continued from previous page)

```
+-------+------------------+
|  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.

pyspark.sql.DataFrame.distinct

DataFrame.**distinct**()

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

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

New in version 1.3.

pyspark.sql.DataFrame.drop

DataFrame.**drop**(**cols*)

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).collect()
[Row(age=5, height=85, name='Bob')]
```

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

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

New in version 1.4.

pyspark.sql.DataFrame.dropDuplicates

DataFrame.**dropDuplicates**(*subset=None*)

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.

pyspark.sql.DataFrame.drop_duplicates

DataFrame.**drop_duplicates**(*subset=None*)

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

New in version 1.4.

pyspark.sql.DataFrame.dropna

DataFrame.**dropna**(*how='any'*, *thresh=None*, *subset=None*)

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.

pyspark.sql.DataFrame.dtypes

property DataFrame.**dtypes**

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

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

New in version 1.3.

pyspark.sql.DataFrame.exceptAll

DataFrame.**exceptAll**(*other*)

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), ("c", 4)], ["C1",
↪"C2"])
>>> df2 = spark.createDataFrame([("a", 1), ("b", 3)], ["C1", "C2"])
```

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

(continues on next page)

(continued from previous page)

```
|  c|  4|
+---+---+
```

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

New in version 2.4.

pyspark.sql.DataFrame.explain

DataFrame.**explain**(*extended=None*, *mode=None*)

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 [2]: [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.

pyspark.sql.DataFrame.fillna

DataFrame.**fillna**(*value*, *subset=None*)

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.

pyspark.sql.DataFrame.filter

DataFrame.**filter**(*condition*)

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.

pyspark.sql.DataFrame.first

DataFrame.**first**()

Returns the first row as a *Row*.

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

New in version 1.3.

pyspark.sql.DataFrame.foreach

DataFrame.**foreach**(*f*)

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.

pyspark.sql.DataFrame.foreachPartition

DataFrame.**foreachPartition**(*f*)

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.

pyspark.sql.DataFrame.freqItems

DataFrame.**freqItems**(*cols*, *support=None*)

Finding frequent items for columns, possibly with false positives. Using the frequent element count algorithm described in "https://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.

pyspark.sql.DataFrame.groupBy

DataFrame.**groupBy**(**cols*)

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, count=1)]
```

New in version 1.3.

pyspark.sql.DataFrame.groupby

DataFrame.**groupby**(**cols*)

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

New in version 1.4.

pyspark.sql.DataFrame.head

DataFrame.**head**(*n=None*)

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.

pyspark.sql.DataFrame.hint

DataFrame.**hint**(*name*, **parameters*)

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.

pyspark.sql.DataFrame.inputFiles

DataFrame.**inputFiles**()

Returns a best-effort snapshot of the files that compose this *DataFrame*. This method simply asks each constituent BaseRelation for its respective files and takes the union of all results. Depending on the source relations, this may not find all input files. Duplicates are removed.

```
>>> df = spark.read.load("examples/src/main/resources/people.json", format="json")
>>> len(df.inputFiles())
1
```

New in version 3.1.

pyspark.sql.DataFrame.intersect

DataFrame.**intersect**(*other*)

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.

pyspark.sql.DataFrame.intersectAll

DataFrame.**intersectAll**(*other*)

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.

pyspark.sql.DataFrame.isLocal

DataFrame.**isLocal**()

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

New in version 1.3.

pyspark.sql.DataFrame.isStreaming

property DataFrame.**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.

pyspark.sql.DataFrame.join

DataFrame.**join**(*other*, *on=None*, *how=None*)

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. >>> from pyspark.sql.functions import desc >>> df.join(df2, df.name == df2.name, 'outer').select(df.name, df2.height) .sort(desc("name")).collect() [Row(name='Bob', height=85), Row(name='Alice', height=None), Row(name=None, height=80)]

```
>>> df.join(df2, 'name', 'outer').select('name', 'height').sort(desc("name")).
↪collect()
[Row(name='Tom', height=80), Row(name='Bob', height=85), Row(name='Alice',␣
↪height=None)]
```

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

New in version 1.3.

pyspark.sql.DataFrame.limit

DataFrame.**limit**(*num*)

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.

pyspark.sql.DataFrame.localCheckpoint

DataFrame.**localCheckpoint**(*eager=True*)

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.

pyspark.sql.DataFrame.mapInPandas

DataFrame.**mapInPandas**(*func*, *schema*)

Maps an iterator of batches in the current *DataFrame* using a Python native function that takes and outputs a pandas DataFrame, and returns the result as a *DataFrame*.

The 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 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*.

Parameters

- **func** – a Python native function that takes an iterator of *pandas.DataFrame*s, and outputs an iterator of *pandas.DataFrame*s.
- **schema** – the return type of the *func* in PySpark. The value can be either a *pyspark.sql.types.DataType* object or a DDL-formatted type string.

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

(continues on next page)

(continued from previous page)

```
|  1| 21|
+---+---+
```

See also:

pyspark.sql.functions.pandas_udf()

Note: Experimental

New in version 3.0.

pyspark.sql.DataFrame.na

property DataFrame.**na**

Returns a *DataFrameNaFunctions* for handling missing values.

New in version 1.3.1.

pyspark.sql.DataFrame.orderBy

DataFrame.**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.

pyspark.sql.DataFrame.persist

DataFrame.**persist**(*storageLevel=StorageLevel(True, True, False, False, 1)*)

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.

pyspark.sql.DataFrame.printSchema

DataFrame.**printSchema**()

Prints out the schema in the tree format.

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

New in version 1.3.

pyspark.sql.DataFrame.randomSplit

DataFrame.**randomSplit**(*weights*, *seed=None*)

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.

pyspark.sql.DataFrame.rdd

property DataFrame.**rdd**
Returns the content as an pyspark.RDD of *Row*.

New in version 1.3.

pyspark.sql.DataFrame.registerTempTable

DataFrame.**registerTempTable**(*name*)
Registers this DataFrame as a temporary table using the given name.

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

```
>>> df.registerTempTable("people")
>>> df2 = spark.sql("select * from people")
>>> sorted(df.collect()) == sorted(df2.collect())
True
>>> spark.catalog.dropTempView("people")
```

Note: Deprecated in 2.0, use createOrReplaceTempView instead.

New in version 1.3.

pyspark.sql.DataFrame.repartition

DataFrame.**repartition**(*numPartitions*, **cols*)
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|
```

(continues on next page)

(continued from previous page)

```
+---+-----+
|  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.

pyspark.sql.DataFrame.repartitionByRange

DataFrame.**repartitionByRange**(*numPartitions*, **cols*)

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|
```

(continues on next page)

(continued from previous page)

```
|  5|  Bob|
+---+-----+
```

New in version 2.4.0.

pyspark.sql.DataFrame.replace

DataFrame.**replace**(*to_replace*, *value=<no value>*, *subset=None*)

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|
```

(continues on next page)

(continued from previous page)

```
|   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.

pyspark.sql.DataFrame.rollup

DataFrame.**rollup**(**cols*)

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").show()
+-----+----+-----+
| name| age|count|
+-----+----+-----+
| null|null|    2|
|Alice|null|    1|
|Alice|   2|    1|
|  Bob|null|    1|
|  Bob|   5|    1|
+-----+----+-----+
```

New in version 1.4.

pyspark.sql.DataFrame.sameSemantics

DataFrame.**sameSemantics**(*other*)

Returns *True* when the logical query plans inside both *DataFrame*s are equal and therefore return same results.

Note: The equality comparison here is simplified by tolerating the cosmetic differences such as attribute names.

Note: This API can compare both *DataFrame*s very fast but can still return *False* on the *DataFrame* that return the same results, for instance, from different plans. Such false negative semantic can be useful when caching as an example.

Note: DeveloperApi

```
>>> df1 = spark.range(10)
>>> df2 = spark.range(10)
>>> df1.withColumn("col1", df1.id * 2).sameSemantics(df2.withColumn("col1", df2.
↪id * 2))
True
>>> df1.withColumn("col1", df1.id * 2).sameSemantics(df2.withColumn("col1", df2.
↪id + 2))
False
>>> df1.withColumn("col1", df1.id * 2).sameSemantics(df2.withColumn("col0", df2.
↪id * 2))
True
```

New in version 3.1.

pyspark.sql.DataFrame.sample

DataFrame.**sample**(*withReplacement=None*, *fraction=None*, *seed=None*)

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.

pyspark.sql.DataFrame.sampleBy

DataFrame.**sampleBy**(*col*, *fractions*, *seed=None*)

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).alias("key"))
>>> sampled = dataset.sampleBy("key", fractions={0: 0.1, 1: 0.2}, seed=0)
>>> sampled.groupBy("key").count().orderBy("key").show()
+---+-----+
|key|count|
+---+-----+
|  0|    3|
|  1|    6|
+---+-----+
>>> dataset.sampleBy(col("key"), fractions={2: 1.0}, seed=0).count()
33
```

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

New in version 1.5.

pyspark.sql.DataFrame.schema

property DataFrame.**schema**

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

```
>>> df.schema
StructType(List(StructField(age,IntegerType,true),StructField(name,StringType,
↪true)))
```

New in version 1.3.

pyspark.sql.DataFrame.select

DataFrame.**select**(**cols*)

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)]
```

(continues on next page)

(continued from previous page)

```
>>> 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.

pyspark.sql.DataFrame.selectExpr

DataFrame.**selectExpr**(**expr*)

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.

pyspark.sql.DataFrame.semanticHash

DataFrame.**semanticHash**()

Returns a hash code of the logical query plan against this *DataFrame*.

Note: Unlike the standard hash code, the hash is calculated against the query plan simplified by tolerating the cosmetic differences such as attribute names.

Note: DeveloperApi

```
>>> spark.range(10).selectExpr("id as col0").semanticHash()
1855039936
>>> spark.range(10).selectExpr("id as col1").semanticHash()
1855039936
```

New in version 3.1.

pyspark.sql.DataFrame.show

DataFrame.**show**(*n=20*, *truncate=True*, *vertical=False*)

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.

pyspark.sql.DataFrame.sort

DataFrame.**sort**(**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.

pyspark.sql.DataFrame.sortWithinPartitions

DataFrame.**sortWithinPartitions**(**cols*, ***kwargs*)

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.

pyspark.sql.DataFrame.stat

property DataFrame.**stat**

Returns a *DataFrameStatFunctions* for statistic functions.

New in version 1.4.

pyspark.sql.DataFrame.storageLevel

property DataFrame.**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.

pyspark.sql.DataFrame.subtract

DataFrame.**subtract**(*other*)

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.

pyspark.sql.DataFrame.summary

DataFrame.**summary**(**statistics*)

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.

pyspark.sql.DataFrame.tail

DataFrame.**tail**(*num*)

Returns the last num rows as a list of *Row*.

Running tail requires moving data into the application's driver process, and doing so with a very large num can crash the driver process with OutOfMemoryError.

```
>>> df.tail(1)
[Row(age=5, name='Bob')]
```

New in version 3.0.

pyspark.sql.DataFrame.take

DataFrame.**take**(*num*)

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.

pyspark.sql.DataFrame.toDF

DataFrame.**toDF**(**cols*)

Returns a new *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')]
```

pyspark.sql.DataFrame.toJSON

DataFrame.**toJSON**(*use_unicode=True*)

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.

pyspark.sql.DataFrame.toLocalIterator

DataFrame.**toLocalIterator**(*prefetchPartitions=False*)

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.

pyspark.sql.DataFrame.toPandas

DataFrame.**toPandas**()

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.

pyspark.sql.DataFrame.transform

DataFrame.**transform**(*func*)

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

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

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

(continues on next page)

(continued from previous page)

```
+-----+---+
|    1|  1|
|    2|  2|
+-----+---+
```

New in version 3.0.

pyspark.sql.DataFrame.union

DataFrame.**union**(*other*)

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.

pyspark.sql.DataFrame.unionAll

DataFrame.**unionAll**(*other*)

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.

pyspark.sql.DataFrame.unionByName

DataFrame.**unionByName**(*other*)

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

New in version 2.3.

pyspark.sql.DataFrame.unpersist

DataFrame.**unpersist**(*blocking=False*)

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.

pyspark.sql.DataFrame.where

DataFrame.**where**(*condition*)

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

New in version 1.3.

pyspark.sql.DataFrame.withColumn

DataFrame.**withColumn**(*colName*, *col*)

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=7)]
```

New in version 1.3.

pyspark.sql.DataFrame.withColumnRenamed

DataFrame.**withColumnRenamed**(*existing*, *new*)

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.

pyspark.sql.DataFrame.withWatermark

DataFrame.**withWatermark**(*eventTime*, *delayThreshold*)

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')).withWatermark('time', '10␣
↪minutes')
DataFrame[name: string, time: timestamp]
```

New in version 2.1.

pyspark.sql.DataFrame.write

property DataFrame.**write**

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

Returns DataFrameWriter

New in version 1.4.

pyspark.sql.DataFrame.writeStream

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

Note: Evolving.

Returns DataStreamWriter

New in version 2.0.

pyspark.sql.DataFrameNaFunctions.drop

DataFrameNaFunctions.**drop**(*how='any'*, *thresh=None*, *subset=None*)
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.

pyspark.sql.DataFrameNaFunctions.fill

DataFrameNaFunctions.**fill**(*value*, *subset=None*)
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.

pyspark.sql.DataFrameNaFunctions.replace

DataFrameNaFunctions.**replace**(*to_replace*, *value=<no value>*, *subset=None*)

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.

pyspark.sql.DataFrameStatFunctions.approxQuantile

DataFrameStatFunctions.**approxQuantile**(*col*, *probabilities*, *relativeError*)

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 [[https://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.

pyspark.sql.DataFrameStatFunctions.corr

DataFrameStatFunctions.**corr**(*col1*, *col2*, *method=None*)

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.

pyspark.sql.DataFrameStatFunctions.cov

DataFrameStatFunctions.**cov**(*col1*, *col2*)

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.

pyspark.sql.DataFrameStatFunctions.crosstab

DataFrameStatFunctions.**crosstab**(*col1*, *col2*)

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.

pyspark.sql.DataFrameStatFunctions.freqItems

DataFrameStatFunctions.**freqItems**(*cols*, *support=None*)

Finding frequent items for columns, possibly with false positives. Using the frequent element count algorithm described in "https://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.

pyspark.sql.DataFrameStatFunctions.sampleBy

DataFrameStatFunctions.**sampleBy**(*col*, *fractions*, *seed=None*)

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).alias("key"))
>>> sampled = dataset.sampleBy("key", fractions={0: 0.1, 1: 0.2}, seed=0)
>>> sampled.groupBy("key").count().orderBy("key").show()
+---+-----+
|key|count|
+---+-----+
|  0|    3|
|  1|    6|
+---+-----+
>>> dataset.sampleBy(col("key"), fractions={2: 1.0}, seed=0).count()
33
```

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

New in version 1.5.

Data Types

ArrayType(elementType[, containsNull])	Array data type.
BinaryType	Binary (byte array) data type.
BooleanType	Boolean data type.
ByteType	Byte data type, i.e.
DataType	Base class for data types.
DateType	Date (datetime.date) data type.
DecimalType([precision, scale])	Decimal (decimal.Decimal) data type.
DoubleType	Double data type, representing double precision floats.
FloatType	Float data type, representing single precision floats.
IntegerType	Int data type, i.e.
LongType	Long data type, i.e.
MapType(keyType, valueType[, valueContainsNull])	Map data type.
NullType	Null type.
Row	A row in `DataFrame`.
ShortType	Short data type, i.e.
StringType	String data type.
StructField(name, dataType[, nullable, metadata])	A field in *StructType*.
StructType([fields])	Struct type, consisting of a list of *StructField*.
TimestampType	Timestamp (datetime.datetime) data type.

pyspark.sql.types.ArrayType

class pyspark.sql.types.**ArrayType**(*elementType*, *containsNull=True*)

Array data type.

Parameters

- **elementType** – *DataType* of each element in the array.
- **containsNull** – boolean, whether the array can contain null (None) values.

__init__(*elementType*, *containsNull=True*)

```
>>> ArrayType(StringType()) == ArrayType(StringType(), True)
True
>>> ArrayType(StringType(), False) == ArrayType(StringType())
False
```

Methods

__init__(elementType[, containsNull])	
	`>>> ArrayType(StringType()) == ArrayType(StringType(), True)`
fromInternal(obj)	Converts an internal SQL object into a native Python object.
fromJson(json)	
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.BinaryType

class pyspark.sql.types.**BinaryType**

Binary (byte array) data type.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.BooleanType

class pyspark.sql.types.**BooleanType**
Boolean data type.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.ByteType

class pyspark.sql.types.**ByteType**
Byte data type, i.e. a signed integer in a single byte.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.DataType

class pyspark.sql.types.**DataType**
Base class for data types.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.DateType

class pyspark.sql.types.**DateType**
Date (datetime.date) data type.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(v)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(d)	Converts a Python object into an internal SQL object.
typeName()	

Attributes

EPOCH_ORDINAL

pyspark.sql.types.DecimalType

class pyspark.sql.types.**DecimalType**(*precision=10*, *scale=0*)

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 creating a DecimalType, the default precision and scale is (10, 0). When inferring schema from decimal.Decimal objects, it will be DecimalType(38, 18).

Parameters

- **precision** – the maximum (i.e. total) number of digits (default: 10)
- **scale** – the number of digits on right side of dot. (default: 0)

__init__(*precision=10*, *scale=0*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([precision, scale])	Initialize self.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.DoubleType

class pyspark.sql.types.**DoubleType**

Double data type, representing double precision floats.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.FloatType

class pyspark.sql.types.**FloatType**

Float data type, representing single precision floats.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.IntegerType

class pyspark.sql.types.**IntegerType**

Int data type, i.e. a signed 32-bit integer.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.LongType

class pyspark.sql.types.**LongType**
Long data type, i.e. a signed 64-bit integer.

If the values are beyond the range of [-9223372036854775808, 9223372036854775807], please use *DecimalType*.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.MapType

class pyspark.sql.types.**MapType**(*keyType*, *valueType*, *valueContainsNull=True*)
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).

__init__(*keyType*, *valueType*, *valueContainsNull=True*)

```
>>> (MapType(StringType(), IntegerType())
...         == MapType(StringType(), IntegerType(), True))
True
>>> (MapType(StringType(), IntegerType(), False)
...         == MapType(StringType(), FloatType()))
False
```

Methods

__init__(keyType, valueType[, valueContainsNull])	`>>> (MapType(StringType(), IntegerType())`
`fromInternal`(obj)	Converts an internal SQL object into a native Python object.
`fromJson`(json)	
`json`()	
`jsonValue`()	
`needConversion`()	Does this type needs conversion between Python object and internal SQL object.
`simpleString`()	
`toInternal`(obj)	Converts a Python object into an internal SQL object.
`typeName`()	

pyspark.sql.types.NullType

class pyspark.sql.types.**NullType**

Null type.

The data type representing None, used for the types that cannot be inferred.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
`fromInternal`(obj)	Converts an internal SQL object into a native Python object.
`json`()	
`jsonValue`()	
`needConversion`()	Does this type needs conversion between Python object and internal SQL object.
`simpleString`()	
`toInternal`(obj)	Converts a Python object into an internal SQL object.
`typeName`()	

pyspark.sql.types.Row

class pyspark.sql.types.**Row**

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. It is not allowed to omit a named argument to represent that the value is None or missing. This should be explicitly set to None in this case.

NOTE: As of Spark 3.0.0, Rows created from named arguments no longer have field names sorted alphabetically and will be ordered in the position as entered. To enable sorting for Rows compatible with Spark 2.x, set the environment variable "PYSPARK_ROW_FIELD_SORTING_ENABLED" to "true". This option is deprecated and will be removed in future versions of Spark. For Python versions < 3.6, the order of named arguments is not guaranteed to be the same as entered, see https://www.python.org/dev/peps/pep-0468. In this case, a warning will be issued and the Row will fallback to sort the field names automatically.

NOTE: Examples with Row in pydocs are run with the environment variable "PYSPARK_ROW_FIELD_SORTING_ENABLED" set to "true" which results in output where fields are sorted.

```
>>> 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
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
asDict([recursive])	Return as a dict
count	Return number of occurrences of value.
index	Return first index of value.

pyspark.sql.types.ShortType

class pyspark.sql.types.**ShortType**
Short data type, i.e. a signed 16-bit integer.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.StringType

class pyspark.sql.types.**StringType**
String data type.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.StructField

class pyspark.sql.types.**StructField**(*name*, *dataType*, *nullable=True*, *metadata=None*)
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

__init__(*name*, *dataType*, *nullable=True*, *metadata=None*)

```
>>> (StructField("f1", StringType(), True)
...       == StructField("f1", StringType(), True))
True
>>> (StructField("f1", StringType(), True)
...       == StructField("f2", StringType(), True))
False
```

Methods

__init__(name, dataType[, nullable, metadata])	
	`>>> (StructField("f1", StringType(), ↪True)`
fromInternal(obj)	Converts an internal SQL object into a native Python object.
fromJson(json)	
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	

continues on next page

Table 34 – continued from previous page

toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.StructType

class pyspark.sql.types.**StructType**(*fields=None*)

Struct type, consisting of a list of *StructField*.

This is the data type representing a *Row*.

Iterating a *StructType* will iterate over its *StructField*s. A contained *StructField* can be accessed by its name or position.

```
>>> struct1 = StructType([StructField("f1", StringType(), True)])
>>> struct1["f1"]
StructField(f1,StringType,true)
>>> struct1[0]
StructField(f1,StringType,true)
```

__init__(*fields=None*)

```
>>> struct1 = StructType([StructField("f1", StringType(), True)])
>>> struct2 = StructType([StructField("f1", StringType(), True)])
>>> struct1 == struct2
True
>>> struct1 = StructType([StructField("f1", StringType(), True)])
>>> struct2 = StructType([StructField("f1", StringType(), True),
...     StructField("f2", IntegerType(), False)])
>>> struct1 == struct2
False
```

Methods

__init__([fields])	`>>> struct1 = StructType([StructField( ↪"f1", StringType(), True)])`
add(field[, data_type, nullable, metadata])	Construct a StructType by adding new elements to it, to define the schema.
fieldNames()	Returns all field names in a list.
fromInternal(obj)	Converts an internal SQL object into a native Python object.
fromJson(json)	
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(obj)	Converts a Python object into an internal SQL object.
typeName()	

pyspark.sql.types.TimestampType

class pyspark.sql.types.**TimestampType**
Timestamp (datetime.datetime) data type.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
fromInternal(ts)	Converts an internal SQL object into a native Python object.
json()	
jsonValue()	
needConversion()	Does this type needs conversion between Python object and internal SQL object.
simpleString()	
toInternal(dt)	Converts a Python object into an internal SQL object.
typeName()	

Functions

abs(col)	Computes the absolute value.
acos(col)	**return** inverse cosine of *col*, as if computed by *java.lang.Math.acos()*
add_months(start, months)	Returns the date that is *months* months after *start*
aggregate(col, zero, merge[, finish])	Applies a binary operator to an initial state and all elements in the array, and reduces this to a single state.
approxCountDistinct(col[, rsd])	**Note:** Deprecated in 2.1, use *approx_count_distinct()* instead.
approx_count_distinct(col[, rsd])	Aggregate function: returns a new Column for approximate distinct count of column *col*.
array(*cols)	Creates a new array column.
array_contains(col, value)	Collection function: returns null if the array is null, true if the array contains the given value, and false otherwise.
array_distinct(col)	Collection function: removes duplicate values from the array.
array_except(col1, col2)	Collection function: returns an array of the elements in col1 but not in col2, without duplicates.
array_intersect(col1, col2)	Collection function: returns an array of the elements in the intersection of col1 and col2, without duplicates.

continues on next page

Table 37 – continued from previous page

array_join(col, delimiter[, null_replacement])	Concatenates the elements of *column* using the *delimiter*.
array_max(col)	Collection function: returns the maximum value of the array.
array_min(col)	Collection function: returns the minimum value of the array.
array_position(col, value)	Collection function: Locates the position of the first occurrence of the given value in the given array.
array_remove(col, element)	Collection function: Remove all elements that equal to element from the given array.
array_repeat(col, count)	Collection function: creates an array containing a column repeated count times.
array_sort(col)	Collection function: sorts the input array in ascending order.
array_union(col1, col2)	Collection function: returns an array of the elements in the union of col1 and col2, without duplicates.
arrays_overlap(a1, a2)	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.
arrays_zip(*cols)	Collection function: Returns a merged array of structs in which the N-th struct contains all N-th values of input arrays.
asc(col)	Returns a sort expression based on the ascending order of the given column name.
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.
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.
ascii(col)	Computes the numeric value of the first character of the string column.
asin(col)	**return** inverse sine of *col*, as if computed by *java.lang.Math.asin()*
atan(col)	**return** inverse tangent of *col*, as if computed by *java.lang.Math.atan()*
atan2(col1, col2)	**param col1** coordinate on y-axis
avg(col)	Aggregate function: returns the average of the values in a group.
base64(col)	Computes the BASE64 encoding of a binary column and returns it as a string column.
basestring	alias of `builtins.str`
bin(col)	Returns the string representation of the binary value of the given column.

continues on next page

Table 37 – continued from previous page

bitwiseNOT(col)	Computes bitwise not.
blacklist	Built-in mutable sequence.
broadcast(df)	Marks a DataFrame as small enough for use in broadcast joins.
bround(col[, scale])	Round the given value to *scale* decimal places using HALF_EVEN rounding mode if *scale* >= 0 or at integral part when *scale* < 0.
cbrt(col)	Computes the cube-root of the given value.
ceil(col)	Computes the ceiling of the given value.
coalesce(*cols)	Returns the first column that is not null.
col(col)	Returns a `Column` based on the given column name.
collect_list(col)	Aggregate function: returns a list of objects with duplicates.
collect_set(col)	Aggregate function: returns a set of objects with duplicate elements eliminated.
column(col)	Returns a `Column` based on the given column name.
concat(*cols)	Concatenates multiple input columns together into a single column.
concat_ws(sep, *cols)	Concatenates multiple input string columns together into a single string column, using the given separator.
conv(col, fromBase, toBase)	Convert a number in a string column from one base to another.
corr(col1, col2)	Returns a new `Column` for the Pearson Correlation Coefficient for `col1` and `col2`.
cos(col) **param col** angle in radians	
cosh(col) **param col** hyperbolic angle	
count(col)	Aggregate function: returns the number of items in a group.
countDistinct(col, *cols)	Returns a new `Column` for distinct count of `col` or `cols`.
covar_pop(col1, col2)	Returns a new `Column` for the population covariance of `col1` and `col2`.
covar_samp(col1, col2)	Returns a new `Column` for the sample covariance of `col1` and `col2`.
crc32(col)	Calculates the cyclic redundancy check value (CRC32) of a binary column and returns the value as a bigint.
create_map(*cols)	Creates a new map column.
cume_dist()	Window function: returns the cumulative distribution of values within a window partition, i.e.
current_date()	Returns the current date as a `DateType` column.
current_timestamp()	Returns the current timestamp as a `TimestampType` column.
date_add(start, days)	Returns the date that is *days* days after *start*
date_format(date, format)	Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument.
date_sub(start, days)	Returns the date that is *days* days before *start*

continues on next page

Table 37 – continued from previous page

date_trunc(format, timestamp)	Returns timestamp truncated to the unit specified by the format.
datediff(end, start)	Returns the number of days from *start* to *end*.
dayofmonth(col)	Extract the day of the month of a given date as integer.
dayofweek(col)	Extract the day of the week of a given date as integer.
dayofyear(col)	Extract the day of the year of a given date as integer.
decode(col, charset)	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').
degrees(col)	Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
dense_rank()	Window function: returns the rank of rows within a window partition, without any gaps.
desc(col)	Returns a sort expression based on the descending order of the given column name.
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.
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
element_at(col, extraction)	Collection function: Returns element of array at given index in extraction if col is array.
encode(col, charset)	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').
exists(col, f)	Returns whether a predicate holds for one or more elements in the array.
exp(col)	Computes the exponential of the given value.
explode(col)	Returns a new row for each element in the given array or map.
explode_outer(col)	Returns a new row for each element in the given array or map.
expm1(col)	Computes the exponential of the given value minus one.
expr(str)	Parses the expression string into the column that it represents
factorial(col)	Computes the factorial of the given value.
filter(col, f)	Returns an array of elements for which a predicate holds in a given array.
first(col[, ignorenulls])	Aggregate function: returns the first value in a group.
flatten(col)	Collection function: creates a single array from an array of arrays.
floor(col)	Computes the floor of the given value.
forall(col, f)	Returns whether a predicate holds for every element in the array.
format_number(col, d)	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.

continues on next page

Table 37 – continued from previous page

format_string(format, *cols)	Formats the arguments in printf-style and returns the result as a string column.
from_csv(col, schema[, options])	Parses a column containing a CSV string to a row with the specified schema.
from_json(col, schema[, options])	Parses a column containing a JSON string into a `MapType` with `StringType` as keys type, `StructType` or `ArrayType` with the specified schema.
from_unixtime(timestamp[, format])	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.
from_utc_timestamp(timestamp, tz)	This is a common function for databases supporting TIMESTAMP WITHOUT TIMEZONE.
functools	functools.py - Tools for working with functions and callable objects
get_json_object(col, path)	Extracts json object from a json string based on json path specified, and returns json string of the extracted json object.
greatest(*cols)	Returns the greatest value of the list of column names, skipping null values.
grouping(col)	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.
grouping_id(*cols)	Aggregate function: returns the level of grouping, equals to
hash(*cols)	Calculates the hash code of given columns, and returns the result as an int column.
hex(col)	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*.
hour(col)	Extract the hours of a given date as integer.
hypot(col1, col2)	Computes `sqrt(a^2 + b^2)` without intermediate overflow or underflow.
ignore_unicode_prefix(f)	Ignore the 'u' prefix of string in doc tests, to make it works in both python 2 and 3
initcap(col)	Translate the first letter of each word to upper case in the sentence.
input_file_name()	Creates a string column for the file name of the current Spark task.
instr(str, substr)	Locate the position of the first occurrence of substr column in the given string.
isnan(col)	An expression that returns true iff the column is NaN.
isnull(col)	An expression that returns true iff the column is null.
json_tuple(col, *fields)	Creates a new row for a json column according to the given field names.
kurtosis(col)	Aggregate function: returns the kurtosis of the values in a group.

continues on next page

Table 37 – continued from previous page

lag(col[, offset, default])	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.
last(col[, ignorenulls])	Aggregate function: returns the last value in a group.
last_day(date)	Returns the last day of the month which the given date belongs to.
lead(col[, offset, default])	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.
least(*cols)	Returns the least value of the list of column names, skipping null values.
length(col)	Computes the character length of string data or number of bytes of binary data.
levenshtein(left, right)	Computes the Levenshtein distance of the two given strings.
lit(col)	Creates a `Column` of literal value.
locate(substr, str[, pos])	Locate the position of the first occurrence of substr in a string column, after position pos.
log(arg1[, arg2])	Returns the first argument-based logarithm of the second argument.
log10(col)	Computes the logarithm of the given value in Base 10.
log1p(col)	Computes the natural logarithm of the given value plus one.
log2(col)	Returns the base-2 logarithm of the argument.
lower(col)	Converts a string expression to lower case.
lpad(col, len, pad)	Left-pad the string column to width *len* with *pad*.
ltrim(col)	Trim the spaces from left end for the specified string value.
map_concat(*cols)	Returns the union of all the given maps.
map_entries(col)	Collection function: Returns an unordered array of all entries in the given map.
map_filter(col, f)	Returns a map whose key-value pairs satisfy a predicate.
map_from_arrays(col1, col2)	Creates a new map from two arrays.
map_from_entries(col)	Collection function: Returns a map created from the given array of entries.
map_keys(col)	Collection function: Returns an unordered array containing the keys of the map.
map_values(col)	Collection function: Returns an unordered array containing the values of the map.
map_zip_with(col1, col2, f)	Merge two given maps, key-wise into a single map using a function.
max(col)	Aggregate function: returns the maximum value of the expression in a group.
md5(col)	Calculates the MD5 digest and returns the value as a 32 character hex string.
mean(col)	Aggregate function: returns the average of the values in a group.
min(col)	Aggregate function: returns the minimum value of the expression in a group.
minute(col)	Extract the minutes of a given date as integer.

continues on next page

Table 37 – continued from previous page

monotonically_increasing_id()	A column that generates monotonically increasing 64-bit integers.
month(col)	Extract the month of a given date as integer.
months_between(date1, date2[, roundOff])	Returns number of months between dates date1 and date2.
nanvl(col1, col2)	Returns col1 if it is not NaN, or col2 if col1 is NaN.
next_day(date, dayOfWeek)	Returns the first date which is later than the value of the date column.
ntile(n)	Window function: returns the ntile group id (from 1 to *n* inclusive) in an ordered window partition.
overlay(src, replace, pos[, len])	Overlay the specified portion of *src* with *replace*, starting from byte position *pos* of *src* and proceeding for *len* bytes.
pandas_udf([f, returnType, functionType])	Creates a pandas user defined function (a.k.a.
percent_rank()	Window function: returns the relative rank (i.e.
percentile_approx(col, percentage[, accuracy])	Returns the approximate percentile value of numeric column col at the given percentage.
posexplode(col)	Returns a new row for each element with position in the given array or map.
posexplode_outer(col)	Returns a new row for each element with position in the given array or map.
pow(col1, col2)	Returns the value of the first argument raised to the power of the second argument.
quarter(col)	Extract the quarter of a given date as integer.
radians(col)	Converts an angle measured in degrees to an approximately equivalent angle measured in radians.
rand([seed])	Generates a random column with independent and identically distributed (i.i.d.) samples uniformly distributed in [0.0, 1.0).
randn([seed])	Generates a column with independent and identically distributed (i.i.d.) samples from the standard normal distribution.
rank()	Window function: returns the rank of rows within a window partition.
regexp_extract(str, pattern, idx)	Extract a specific group matched by a Java regex, from the specified string column.
regexp_replace(str, pattern, replacement)	Replace all substrings of the specified string value that match regexp with rep.
repeat(col, n)	Repeats a string column n times, and returns it as a new string column.
reverse(col)	Collection function: returns a reversed string or an array with reverse order of elements.
rint(col)	Returns the double value that is closest in value to the argument and is equal to a mathematical integer.
round(col[, scale])	Round the given value to *scale* decimal places using HALF_UP rounding mode if *scale* >= 0 or at integral part when *scale* < 0.
row_number()	Window function: returns a sequential number starting at 1 within a window partition.
rpad(col, len, pad)	Right-pad the string column to width *len* with *pad*.

continues on next page

Table 37 – continued from previous page

rtrim(col)	Trim the spaces from right end for the specified string value.
schema_of_csv(csv[, options])	Parses a CSV string and infers its schema in DDL format.
schema_of_json(json[, options])	Parses a JSON string and infers its schema in DDL format.
second(col)	Extract the seconds of a given date as integer.
sequence(start, stop[, step])	Generate a sequence of integers from *start* to *stop*, incrementing by *step*.
sha1(col)	Returns the hex string result of SHA-1.
sha2(col, numBits)	Returns the hex string result of SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, and SHA-512).
shiftLeft(col, numBits)	Shift the given value numBits left.
shiftRight(col, numBits)	(Signed) shift the given value numBits right.
shiftRightUnsigned(col, numBits)	Unsigned shift the given value numBits right.
shuffle(col)	Collection function: Generates a random permutation of the given array.
signum(col)	Computes the signum of the given value.
sin(col)	**param col** angle in radians
since(version)	A decorator that annotates a function to append the version of Spark the function was added.
sinh(col)	**param col** hyperbolic angle
size(col)	Collection function: returns the length of the array or map stored in the column.
skewness(col)	Aggregate function: returns the skewness of the values in a group.
slice(x, start, length)	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*.
sort_array(col[, asc])	Collection function: sorts the input array in ascending or descending order according to the natural ordering of the array elements.
soundex(col)	Returns the SoundEx encoding for a string
spark_partition_id()	A column for partition ID.
split(str, pattern[, limit])	Splits str around matches of the given pattern.
sqrt(col)	Computes the square root of the specified float value.
stddev(col)	Aggregate function: alias for stddev_samp.
stddev_pop(col)	Aggregate function: returns population standard deviation of the expression in a group.
stddev_samp(col)	Aggregate function: returns the unbiased sample standard deviation of the expression in a group.
struct(*cols)	Creates a new struct column.
substring(str, pos, len)	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.

continues on next page

Table 37 – continued from previous page

substring_index(str, delim, count)	Returns the substring from string str before count occurrences of the delimiter delim.
sum(col)	Aggregate function: returns the sum of all values in the expression.
sumDistinct(col)	Aggregate function: returns the sum of distinct values in the expression.
sys	This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.
tan(col)	**param col** angle in radians
tanh(col)	**param col** hyperbolic angle
toDegrees(col)	**Note:** Deprecated in 2.1, use degrees() instead.
toRadians(col)	**Note:** Deprecated in 2.1, use radians() instead.
to_csv(col[, options])	Converts a column containing a StructType into a CSV string.
to_date(col[, format])	Converts a Column into pyspark.sql.types.DateType using the optionally specified format.
to_json(col[, options])	Converts a column containing a StructType, ArrayType or a MapType into a JSON string.
to_str(value)	A wrapper over str(), but converts bool values to lower case strings.
to_timestamp(col[, format])	Converts a Column into pyspark.sql.types.TimestampType using the optionally specified format.
to_utc_timestamp(timestamp, tz)	This is a common function for databases supporting TIMESTAMP WITHOUT TIMEZONE.
transform(col, f)	Returns an array of elements after applying a transformation to each element in the input array.
transform_keys(col, f)	Applies a function to every key-value pair in a map and returns a map with the results of those applications as the new keys for the pairs.
transform_values(col, f)	Applies a function to every key-value pair in a map and returns a map with the results of those applications as the new values for the pairs.
translate(srcCol, matching, replace)	A function translate any character in the *srcCol* by a character in *matching*.
trim(col)	Trim the spaces from both ends for the specified string column.

continues on next page

Table 37 – continued from previous page

trunc(date, format)	Returns date truncated to the unit specified by the format.
udf([f, returnType])	Creates a user defined function (UDF).
unbase64(col)	Decodes a BASE64 encoded string column and returns it as a binary column.
unhex(col)	Inverse of hex.
unix_timestamp([timestamp, format])	Convert time string with given pattern ('yyyy-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.
upper(col)	Converts a string expression to upper case.
var_pop(col)	Aggregate function: returns the population variance of the values in a group.
var_samp(col)	Aggregate function: returns the unbiased sample variance of the values in a group.
variance(col)	Aggregate function: alias for var_samp.
warnings	Python part of the warnings subsystem.
weekofyear(col)	Extract the week number of a given date as integer.
when(condition, value)	Evaluates a list of conditions and returns one of multiple possible result expressions.
window(timeColumn, windowDuration[, ...])	Bucketize rows into one or more time windows given a timestamp specifying column.
xxhash64(*cols)	Calculates the hash code of given columns using the 64-bit variant of the xxHash algorithm, and returns the result as a long column.
year(col)	Extract the year of a given date as integer.
zip_with(col1, col2, f)	Merge two given arrays, element-wise, into a single array using a function.

pyspark.sql.functions.abs

pyspark.sql.functions.**abs**(*col*)

Computes the absolute value.

New in version 1.3.

pyspark.sql.functions.acos

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

pyspark.sql.functions.**add_months**(*start*, *months*)

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.aggregate

pyspark.sql.functions.**aggregate**(*col*, *zero*, *merge*, *finish=None*)

Applies a binary operator to an initial state and all elements in the array, and reduces this to a single state. The final state is converted into the final result by applying a finish function.

Both functions can use methods of *pyspark.sql.Column*, functions defined in `pyspark.sql.functions` and Scala `UserDefinedFunctions`. Python `UserDefinedFunctions` are not supported (SPARK-27052).

Parameters

- **col** – name of column or expression
- **zero** – initial value. Name of column or expression
- **merge** – a binary function `(acc: Column, x: Column) -> Column...` returning expression of the same type as `zero`
- **finish** – an optional unary function `(x: Column) -> Column: ...` used to convert accumulated value.

Returns a *pyspark.sql.Column*

```
>>> df = spark.createDataFrame([(1, [20.0, 4.0, 2.0, 6.0, 10.0])], ("id", "values
↪"))
>>> df.select(aggregate("values", lit(0.0), lambda acc, x: acc + x).alias("sum")).
↪show()
+----+
| sum|
+----+
|42.0|
+----+
```

```
>>> def merge(acc, x):
...     count = acc.count + 1
...     sum = acc.sum + x
...     return struct(count.alias("count"), sum.alias("sum"))
```

(continues on next page)

(continued from previous page)

```
>>> df.select(
...     aggregate(
...         "values",
...         struct(lit(0).alias("count"), lit(0.0).alias("sum")),
...         merge,
...         lambda acc: acc.sum / acc.count,
...     ).alias("mean")
... ).show()
+----+
|mean|
+----+
| 8.4|
+----+
```

New in version 3.1.

pyspark.sql.functions.approxCountDistinct

pyspark.sql.functions.**approxCountDistinct**(*col*, *rsd=None*)

Note: Deprecated in 2.1, use *approx_count_distinct()* instead.

New in version 1.3.

pyspark.sql.functions.approx_count_distinct

pyspark.sql.functions.**approx_count_distinct**(*col*, *rsd=None*)

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')).collect()
[Row(distinct_ages=2)]
```

New in version 2.1.

pyspark.sql.functions.array

pyspark.sql.functions.**array**(**cols*)

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

pyspark.sql.functions.**array_contains**(*col*, *value*)

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)=False)]
>>> df.select(array_contains(df.data, lit("a"))).collect()
[Row(array_contains(data, a)=True), Row(array_contains(data, a)=False)]
```

New in version 1.5.

pyspark.sql.functions.array_distinct

pyspark.sql.functions.**array_distinct**(*col*)

Collection function: removes duplicate values from the array.

Parameters **col** – name of column or expression

```
>>> df = spark.createDataFrame([([1, 2, 3, 2],), ([4, 5, 5, 4],)], ['data'])
>>> df.select(array_distinct(df.data)).collect()
[Row(array_distinct(data)=[1, 2, 3]), Row(array_distinct(data)=[4, 5])]
```

New in version 2.4.

pyspark.sql.functions.array_except

pyspark.sql.functions.**array_except**(*col1*, *col2*)

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", "d", "a", "f"])])
>>> 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

pyspark.sql.functions.**array_intersect**(*col1*, *col2*)

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", "d", "a", "f"])])
>>> 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

pyspark.sql.functions.**array_join**(*col*, *delimiter*, *null_replacement=None*)

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],)], ['data'])
>>> 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")).collect()
[Row(joined='a,b,c'), Row(joined='a,NULL')]
```

New in version 2.4.

pyspark.sql.functions.array_max

pyspark.sql.functions.**array_max**(*col*)

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],)], ['data'])
>>> 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

pyspark.sql.functions.**array_min**(*col*)

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],)], ['data'])
>>> 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

pyspark.sql.functions.**array_position**(*col*, *value*)

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

pyspark.sql.functions.**array_remove**(*col*, *element*)

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

pyspark.sql.functions.**array_repeat**(*col*, *count*)

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

pyspark.sql.functions.**array_sort**(*col*)

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],),([],)], ['data'])
>>> 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

pyspark.sql.functions.**array_union**(*col1*, *col2*)

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", "d", "a", "f"])])
>>> 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

pyspark.sql.functions.**arrays_overlap**(*a1*, *a2*)

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"], ["b", "c"])], [
↪'x', 'y'])
>>> 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

pyspark.sql.functions.**arrays_zip**(**cols*)

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', 'vals2'])
>>> df.select(arrays_zip(df.vals1, df.vals2).alias('zipped')).collect()
[Row(zipped=[Row(vals1=1, vals2=2), Row(vals1=2, vals2=3), Row(vals1=3,␣
↪vals2=4)])]
```

New in version 2.4.

pyspark.sql.functions.asc

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

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

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

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

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

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

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

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

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

pyspark.sql.functions.**basestring**
alias of builtins.str

pyspark.sql.functions.bin

pyspark.sql.functions.**bin**(*col*)
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

pyspark.sql.functions.**bitwiseNOT**(*col*)
Computes bitwise not.

New in version 1.4.

pyspark.sql.functions.blacklist

pyspark.sql.functions.**blacklist = ['map', 'since', 'ignore_unicode_prefix']**
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

pyspark.sql.functions.broadcast

pyspark.sql.functions.**broadcast**(*df*)
Marks a DataFrame as small enough for use in broadcast joins.

New in version 1.6.

pyspark.sql.functions.bround

pyspark.sql.functions.**bround**(*col*, *scale=0*)
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).alias('r')).
↪collect()
[Row(r=2.0)]
```

New in version 2.0.

pyspark.sql.functions.cbrt

pyspark.sql.functions.**cbrt**(*col*)

Computes the cube-root of the given value.

New in version 1.4.

pyspark.sql.functions.ceil

pyspark.sql.functions.**ceil**(*col*)

Computes the ceiling of the given value.

New in version 1.4.

pyspark.sql.functions.coalesce

pyspark.sql.functions.**coalesce**(**cols*)

Returns the first column that is not null.

```
>>> cDf = spark.createDataFrame([(None, None), (1, None), (None, 2)], ("a", "b"))
>>> 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

pyspark.sql.functions.**col**(*col*)
Returns a Column based on the given column name.

New in version 1.3.

pyspark.sql.functions.collect_list

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 the order of the 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

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 the order of the 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

pyspark.sql.functions.**column**(*col*)
Returns a Column based on the given column name.

New in version 1.3.

pyspark.sql.functions.concat

pyspark.sql.functions.**concat**(**cols*)

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], None, [3])], ['a',
↪ 'b', 'c'])
>>> 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

pyspark.sql.functions.**concat_ws**(*sep*, **cols*)

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

pyspark.sql.functions.**conv**(*col*, *fromBase*, *toBase*)

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

pyspark.sql.functions.**corr**(*col1*, *col2*)

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

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

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

pyspark.sql.functions.**count**(*col*)

Aggregate function: returns the number of items in a group.

New in version 1.3.

pyspark.sql.functions.countDistinct

pyspark.sql.functions.**countDistinct**(*col*, **cols*)

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

pyspark.sql.functions.**covar_pop**(*col1*, *col2*)

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

pyspark.sql.functions.**covar_samp**(*col1*, *col2*)

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

pyspark.sql.functions.**crc32**(*col*)

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').alias('crc32')).
↪collect()
[Row(crc32=2743272264)]
```

New in version 1.5.

pyspark.sql.functions.create_map

pyspark.sql.functions.**create_map**(**cols*)

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

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

pyspark.sql.functions.**current_date**()
Returns the current date as a DateType column.

New in version 1.5.

pyspark.sql.functions.current_timestamp

pyspark.sql.functions.**current_timestamp**()
Returns the current timestamp as a TimestampType column.

pyspark.sql.functions.date_add

pyspark.sql.functions.**date_add**(*start*, *days*)
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

pyspark.sql.functions.**date_format**(*date*, *format*)
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 datetime pattern. 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')).collect()
[Row(date='04/08/2015')]
```

New in version 1.5.

pyspark.sql.functions.date_sub

pyspark.sql.functions.**date_sub**(*start*, *days*)
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

pyspark.sql.functions.**date_trunc**(*format*, *timestamp*)

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

pyspark.sql.functions.**datediff**(*end*, *start*)

Returns the number of days from *start* to *end*.

```
>>> df = spark.createDataFrame([('2015-04-08','2015-05-10')], ['d1', 'd2'])
>>> df.select(datediff(df.d2, df.d1).alias('diff')).collect()
[Row(diff=32)]
```

New in version 1.5.

pyspark.sql.functions.dayofmonth

pyspark.sql.functions.**dayofmonth**(*col*)

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

pyspark.sql.functions.**dayofweek**(*col*)

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

pyspark.sql.functions.**dayofyear**(*col*)

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

pyspark.sql.functions.**decode**(*col*, *charset*)

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

pyspark.sql.functions.**degrees**(*col*)

Converts an angle measured in radians to an approximately equivalent angle measured in degrees.

Parameters **col** – angle in radians

Returns angle in degrees, as if computed by *java.lang.Math.toDegrees()*

New in version 2.1.

pyspark.sql.functions.dense_rank

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

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

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

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

pyspark.sql.functions.**element_at**(*col*, *extraction*)

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},), ({},)], ['data'])
>>> 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

pyspark.sql.functions.**encode**(*col*, *charset*)

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.exists

pyspark.sql.functions.**exists**(*col*, *f*)

Returns whether a predicate holds for one or more elements in the array.

Parameters

- **col** – name of column or expression
- **f** – an function `(x: Column) -> Column: ...` returning the Boolean expression. Can use methods of *pyspark.sql.Column*, functions defined in `pyspark.sql.functions` and Scala `UserDefinedFunctions`. Python `UserDefinedFunctions` are not supported (SPARK-27052).

Returns a *pyspark.sql.Column*

```
>>> df = spark.createDataFrame([(1, [1, 2, 3, 4]), (2, [3, -1, 0])],("key",
↪"values"))
>>> df.select(exists("values", lambda x: x < 0).alias("any_negative")).show()
+------------+
|any_negative|
+------------+
|       false|
|        true|
+------------+
```

New in version 3.1.

pyspark.sql.functions.exp

pyspark.sql.functions.**exp**(*col*)

Computes the exponential of the given value.

New in version 1.4.

pyspark.sql.functions.explode

pyspark.sql.functions.**explode**(*col*)

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], mapfield={"a": "b"})])
>>> 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

pyspark.sql.functions.**explode_outer**(*col*)

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, None)],
...     ("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

pyspark.sql.functions.**expm1**(*col*)

Computes the exponential of the given value minus one.

New in version 1.4.

pyspark.sql.functions.expr

pyspark.sql.functions.**expr**(*str*)

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

pyspark.sql.functions.**factorial**(*col*)

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.filter

pyspark.sql.functions.**filter**(*col*, *f*)

Returns an array of elements for which a predicate holds in a given array.

Parameters

- **col** – name of column or expression
- **f** – A function that returns the Boolean expression. Can take one of the following forms:
  - Unary `(x: Column) -> Column: ...`
  - **Binary `(x: Column, i: Column) -> Column...`, where the second argument is** a 0-based index of the element.

  and can use methods of *pyspark.sql.Column*, functions defined in `pyspark.sql.functions` and Scala `UserDefinedFunctions`. Python `UserDefinedFunctions` are not supported (SPARK-27052).

Returns a *pyspark.sql.Column*

```
>>> df = spark.createDataFrame(
...     [(1, ["2018-09-20",  "2019-02-03", "2019-07-01", "2020-06-01"])],
...     ("key", "values")
... )
>>> def after_second_quarter(x):
...     return month(to_date(x)) > 6
>>> df.select(
...     filter("values", after_second_quarter).alias("after_second_quarter")
... ).show(truncate=False)
+------------------------+
|after_second_quarter    |
+------------------------+
|[2018-09-20, 2019-07-01]|
+------------------------+
```

New in version 3.1.

pyspark.sql.functions.first

pyspark.sql.functions.**first**(*col*, *ignorenulls=False*)

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 the order of the rows which may be non-deterministic after a shuffle.

New in version 1.3.

pyspark.sql.functions.flatten

pyspark.sql.functions.**flatten**(*col*)

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]],), ([None, [4, 5]],)], [
↪'data'])
>>> 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

pyspark.sql.functions.**floor**(*col*)

Computes the floor of the given value.

New in version 1.4.

pyspark.sql.functions.forall

pyspark.sql.functions.**forall**(*col*, *f*)

Returns whether a predicate holds for every element in the array.

Parameters

- **col** – name of column or expression
- **f** – an function `(x: Column) -> Column: ...` returning the Boolean expression. Can use methods of *pyspark.sql.Column*, functions defined in `pyspark.sql.functions` and Scala `UserDefinedFunctions`. Python `UserDefinedFunctions` are not supported (SPARK-27052).

Returns a *pyspark.sql.Column*

```
>>> df = spark.createDataFrame(
...     [(1, ["bar"]), (2, ["foo", "bar"]), (3, ["foobar", "foo"])],
...     ("key", "values")
... )
>>> df.select(forall("values", lambda x: x.rlike("foo")).alias("all_foo")).show()
+-------+
|all_foo|
+-------+
|  false|
|  false|
|   true|
+-------+
```

New in version 3.1.

pyspark.sql.functions.format_number

pyspark.sql.functions.**format_number**(*col*, *d*)

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', 4).alias('v')).
↪collect()
[Row(v='5.0000')]
```

New in version 1.5.

pyspark.sql.functions.format_string

pyspark.sql.functions.**format_string**(*format*, **cols*)

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')).collect()
[Row(v='5 hello')]
```

New in version 1.5.

pyspark.sql.functions.from_csv

pyspark.sql.functions.**from_csv**(*col*, *schema*, *options={}*)

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")).collect()
[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")).collect()
[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")).collect()
[Row(csv=Row(s='abc'))]
```

New in version 3.0.

pyspark.sql.functions.from_json

pyspark.sql.functions.**from_json**(*col*, *schema*, *options={}*)

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")).collect()
[Row(json={'a': 1})]
>>> data = [(1, '''[{"a": 1}]''')]
>>> schema = ArrayType(StructType([StructField("a", IntegerType())]))
```

(continues on next page)

(continued from previous page)

```
>>> 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

pyspark.sql.functions.**from_unixtime**(*timestamp*, *format='yyyy-MM-dd HH:mm:ss'*)

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_Angeles")
>>> 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

pyspark.sql.functions.**from_utc_timestamp**(*timestamp*, *tz*)

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 detailing the time zone ID that the input should be adjusted to. It should be in the format of either region-based zone IDs or zone offsets. Region IDs must have the form 'area/city', such as 'America/Los_Angeles'. Zone offsets must be in the format '(+|-)HH:mm', for example '-08:00' or '+01:00'. Also 'UTC' and 'Z' are supported as aliases of '+00:00'. Other short names are not recommended to use because they can be ambiguous.

Changed in version 2.4: *tz* can take a `Column` containing timezone ID strings.

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00', 'JST')], ['ts', 'tz'])
>>> df.select(from_utc_timestamp(df.ts, "PST").alias('local_time')).collect()
[Row(local_time=datetime.datetime(1997, 2, 28, 2, 30))]
>>> df.select(from_utc_timestamp(df.ts, df.tz).alias('local_time')).collect()
[Row(local_time=datetime.datetime(1997, 2, 28, 19, 30))]
```

New in version 1.5.

pyspark.sql.functions.functools

functools.py - Tools for working with functions and callable objects

Functions

lru_cache([maxsize, typed])	Least-recently-used cache decorator.
singledispatch(func)	Single-dispatch generic function decorator.
total_ordering(cls)	Class decorator that fills in missing ordering methods
update_wrapper(wrapper, wrapped[, assigned, ...])	Update a wrapper function to look like the wrapped function
wraps(wrapped[, assigned, updated])	Decorator factory to apply update_wrapper() to a wrapper function

Classes

partial	partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.
partialmethod(func, *args, **keywords)	Method descriptor with partial application of the given arguments and keywords.

pyspark.sql.functions.get_json_object

pyspark.sql.functions.**get_json_object**(*col*, *path*)

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", '''{"f1":
↪"value12"}''')]
>>> df = spark.createDataFrame(data, ("key", "jstring"))
>>> df.select(df.key, get_json_object(df.jstring, '$.f1').alias("c0"), \
...                   get_json_object(df.jstring, '$.f2').alias("c1") ).collect()
[Row(key='1', c0='value1', c1='value2'), Row(key='2', c0='value12', c1=None)]
```

New in version 1.6.

pyspark.sql.functions.greatest

pyspark.sql.functions.**greatest**(**cols*)

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")).collect()
[Row(greatest=4)]
```

New in version 1.5.

pyspark.sql.functions.grouping

pyspark.sql.functions.**grouping**(*col*)

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").show()
+-----+--------------+--------+
| name|grouping(name)|sum(age)|
+-----+--------------+--------+
| null|             1|       7|
|Alice|             0|       2|
|  Bob|             0|       5|
+-----+--------------+--------+
```

New in version 2.0.

pyspark.sql.functions.grouping_id

pyspark.sql.functions.**grouping_id**(**cols*)

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").show()
+-----+-------------+--------+
| name|grouping_id()|sum(age)|
+-----+-------------+--------+
| null|            1|       7|
|Alice|            0|       2|
|  Bob|            0|       5|
+-----+-------------+--------+
```

New in version 2.0.

pyspark.sql.functions.hash

pyspark.sql.functions.**hash**(**cols*)

Calculates the hash code of given columns, and returns the result as an int column.

```
>>> spark.createDataFrame([('ABC',)], ['a']).select(hash('a').alias('hash')).
↪collect()
[Row(hash=-757602832)]
```

New in version 2.0.

pyspark.sql.functions.hex

pyspark.sql.functions.**hex**(*col*)

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'), hex('b')).
↪collect()
[Row(hex(a)='414243', hex(b)='3')]
```

New in version 1.5.

pyspark.sql.functions.hour

pyspark.sql.functions.**hour**(*col*)

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

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.ignore_unicode_prefix

pyspark.sql.functions.**ignore_unicode_prefix**(*f*)

Ignore the 'u' prefix of string in doc tests, to make it works in both python 2 and 3

pyspark.sql.functions.initcap

pyspark.sql.functions.**initcap**(*col*)

Translate the first letter of each word to upper case in the sentence.

```
>>> spark.createDataFrame([('ab cd',)], ['a']).select(initcap("a").alias('v')).
↪collect()
[Row(v='Ab Cd')]
```

New in version 1.5.

pyspark.sql.functions.input_file_name

pyspark.sql.functions.**input_file_name**()

Creates a string column for the file name of the current Spark task.

New in version 1.6.

pyspark.sql.functions.instr

pyspark.sql.functions.**instr**(*str*, *substr*)

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

pyspark.sql.functions.**isnan**(*col*)

An expression that returns true iff the column is NaN.

```
>>> df = spark.createDataFrame([(1.0, float('nan')), (float('nan'), 2.0)], ("a",
↪"b"))
>>> df.select(isnan("a").alias("r1"), isnan(df.a).alias("r2")).collect()
[Row(r1=False, r2=False), Row(r1=True, r2=True)]
```

New in version 1.6.

pyspark.sql.functions.isnull

pyspark.sql.functions.**isnull**(*col*)

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")).collect()
[Row(r1=False, r2=False), Row(r1=True, r2=True)]
```

New in version 1.6.

pyspark.sql.functions.json_tuple

pyspark.sql.functions.**json_tuple**(*col*, **fields*)

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", '''{"f1":
↪"value12"}''')]
>>> 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', c1=None)]
```

New in version 1.6.

pyspark.sql.functions.kurtosis

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

pyspark.sql.functions.**lag**(*col*, *offset=1*, *default=None*)

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

pyspark.sql.functions.**last**(*col*, *ignorenulls=False*)

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 the order of the rows which may be non-deterministic after a shuffle.

New in version 1.3.

pyspark.sql.functions.last_day

pyspark.sql.functions.**last_day**(*date*)

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

pyspark.sql.functions.**lead**(*col*, *offset=1*, *default=None*)

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

pyspark.sql.functions.**least**(**cols*)

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

pyspark.sql.functions.**length**(*col*)

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').alias('length')).
↪collect()
[Row(length=4)]
```

New in version 1.5.

pyspark.sql.functions.levenshtein

pyspark.sql.functions.**levenshtein**(*left*, *right*)

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

pyspark.sql.functions.**lit**(*col*)

Creates a Column of literal value.

```
>>> df.select(lit(5).alias('height')).withColumn('spark_user', lit(True)).take(1)
[Row(height=5, spark_user=True)]
```

New in version 1.3.

pyspark.sql.functions.locate

pyspark.sql.functions.**locate**(*substr*, *str*, *pos=1*)

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

pyspark.sql.functions.**log**(*arg1*, *arg2=None*)

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(l.ten)[:7]).
↪collect()
['0.30102', '0.69897']
```

```
>>> df.select(log(df.age).alias('e')).rdd.map(lambda l: str(l.e)[:7]).collect()
['0.69314', '1.60943']
```

New in version 1.5.

pyspark.sql.functions.log10

pyspark.sql.functions.**log10**(*col*)

Computes the logarithm of the given value in Base 10.

New in version 1.4.

pyspark.sql.functions.log1p

pyspark.sql.functions.**log1p**(*col*)

Computes the natural logarithm of the given value plus one.

New in version 1.4.

pyspark.sql.functions.log2

pyspark.sql.functions.**log2**(*col*)

Returns the base-2 logarithm of the argument.

```
>>> spark.createDataFrame([(4,)], ['a']).select(log2('a').alias('log2')).collect()
[Row(log2=2.0)]
```

New in version 1.5.

pyspark.sql.functions.lower

pyspark.sql.functions.**lower**(*col*)

Converts a string expression to lower case.

New in version 1.5.

pyspark.sql.functions.lpad

pyspark.sql.functions.**lpad**(*col*, *len*, *pad*)
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

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

pyspark.sql.functions.**map_concat**(**cols*)
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') as map2")
>>> df.select(map_concat("map1", "map2").alias("map3")).show(truncate=False)
+------------------------+
|map3                    |
+------------------------+
|[1 -> a, 2 -> b, 3 -> c]|
+------------------------+
```

New in version 2.4.

pyspark.sql.functions.map_entries

pyspark.sql.functions.**map_entries**(*col*)
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_filter

pyspark.sql.functions.**map_filter**(*col*, *f*)

Returns a map whose key-value pairs satisfy a predicate.

Parameters

- **col** – name of column or expression
- **f** – a binary function `(k: Column, v: Column) -> Column...` Can use methods of *pyspark.sql.Column*, functions defined in `pyspark.sql.functions` and Scala `UserDefinedFunctions`. Python `UserDefinedFunctions` are not supported (SPARK-27052).

Returns a *pyspark.sql.Column*

```
>>> df = spark.createDataFrame([(1, {"foo": 42.0, "bar": 1.0, "baz": 32.0})], ("id
↪", "data"))
>>> df.select(map_filter(
...     "data", lambda _, v: v > 30.0).alias("data_filtered")
... ).show(truncate=False)
+--------------------------+
|data_filtered             |
+--------------------------+
|[baz -> 32.0, foo -> 42.0]|
+--------------------------+
```

New in version 3.1.

pyspark.sql.functions.map_from_arrays

pyspark.sql.functions.**map_from_arrays**(*col1*, *col2*)

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

pyspark.sql.functions.**map_from_entries**(*col*)

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 data")
>>> df.select(map_from_entries("data").alias("map")).show()
+----------------+
|             map|
+----------------+
|[1 -> a, 2 -> b]|
+----------------+
```

New in version 2.4.

pyspark.sql.functions.map_keys

pyspark.sql.functions.**map_keys**(*col*)

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

pyspark.sql.functions.**map_values**(*col*)

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.map_zip_with

pyspark.sql.functions.**map_zip_with**(*col1*, *col2*, *f*)

Merge two given maps, key-wise into a single map using a function.

Parameters

- **col1** – name of the first column or expression
- **col2** – name of the second column or expression
- **f** – a ternary function `(k: Column, v1: Column, v2: Column) -> Column...` Can use methods of *pyspark.sql.Column*, functions defined in `pyspark.sql.functions` and Scala `UserDefinedFunctions`. Python `UserDefinedFunctions` are not supported (SPARK-27052).

Returns a *pyspark.sql.Column*

```
>>> df = spark.createDataFrame([
...     (1, {"IT": 24.0, "SALES": 12.00}, {"IT": 2.0, "SALES": 1.4})],
...     ("id", "base", "ratio")
... )
>>> df.select(map_zip_with(
...     "base", "ratio", lambda k, v1, v2: round(v1 * v2, 2)).alias("updated_data
↪")
... ).show(truncate=False)
+---------------------------+
|updated_data               |
+---------------------------+
|[SALES -> 16.8, IT -> 48.0]|
+---------------------------+
```

New in version 3.1.

pyspark.sql.functions.max

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

pyspark.sql.functions.**md5**(*col*)

Calculates the MD5 digest and returns the value as a 32 character hex string.

```
>>> spark.createDataFrame([('ABC',)], ['a']).select(md5('a').alias('hash')).
↪collect()
[Row(hash='902fbdd2b1df0c4f70b4a5d23525e932')]
```

New in version 1.5.

pyspark.sql.functions.mean

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

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

pyspark.sql.functions.**minute**(*col*)
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

pyspark.sql.functions.**monotonically_increasing_id**()
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,), (2,), (3,)]).
↪toDF(['col1'])
>>> df0.select(monotonically_increasing_id().alias('id')).collect()
[Row(id=0), Row(id=1), Row(id=2), Row(id=8589934592), Row(id=8589934593),␣
↪Row(id=8589934594)]
```

New in version 1.6.

pyspark.sql.functions.month

pyspark.sql.functions.**month**(*col*)

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

pyspark.sql.functions.**months_between**(*date1*, *date2*, *roundOff=True*)

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')], ['date1',
↪'date2'])
>>> df.select(months_between(df.date1, df.date2).alias('months')).collect()
[Row(months=3.94959677)]
>>> df.select(months_between(df.date1, df.date2, False).alias('months')).collect()
[Row(months=3.9495967741935485)]
```

New in version 1.5.

pyspark.sql.functions.nanvl

pyspark.sql.functions.**nanvl**(*col1*, *col2*)

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'), 2.0)], ("a",
↪"b"))
>>> df.select(nanvl("a", "b").alias("r1"), nanvl(df.a, df.b).alias("r2")).
↪collect()
[Row(r1=1.0, r2=1.0), Row(r1=2.0, r2=2.0)]
```

New in version 1.6.

pyspark.sql.functions.next_day

pyspark.sql.functions.**next_day**(*date*, *dayOfWeek*)

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

pyspark.sql.functions.**ntile**(*n*)

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.overlay

pyspark.sql.functions.**overlay**(*src*, *replace*, *pos*, *len=- 1*)

Overlay the specified portion of *src* with *replace*, starting from byte position *pos* of *src* and proceeding for *len* bytes.

```
>>> df = spark.createDataFrame([("SPARK_SQL", "CORE")], ("x", "y"))
>>> df.select(overlay("x", "y", 7).alias("overlayed")).show()
+----------+
| overlayed|
+----------+
|SPARK_CORE|
+----------+
```

New in version 3.0.

pyspark.sql.functions.pandas_udf

pyspark.sql.functions.**pandas_udf**(*f=None*, *returnType=None*, *functionType=None*)

Creates a pandas user defined function (a.k.a. vectorized user defined function).

Pandas UDFs are user defined functions that are executed by Spark using Arrow to transfer data and Pandas to work with the data, which allows vectorized operations. A Pandas UDF is defined using the *pandas_udf* as a decorator or to wrap the function, and no additional configuration is required. A Pandas UDF behaves as a regular PySpark function API in general.

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.

Note: This parameter exists for compatibility. Using Python type hints is encouraged.

In order to use this API, customarily the below are imported:

```
>>> import pandas as pd
>>> from pyspark.sql.functions import pandas_udf
```

From Spark 3.0 with Python 3.6+, Python type hints detect the function types as below:

```
>>> @pandas_udf(IntegerType())
... def slen(s: pd.Series) -> pd.Series:
...     return s.str.len()
```

Prior to Spark 3.0, the pandas UDF used *functionType* to decide the execution type as below:

```
>>> from pyspark.sql.functions import PandasUDFType
>>> from pyspark.sql.types import IntegerType
>>> @pandas_udf(IntegerType(), PandasUDFType.SCALAR)
... def slen(s):
...     return s.str.len()
```

It is preferred to specify type hints for the pandas UDF instead of specifying pandas UDF type via *functionType* which will be deprecated in the future releases.

Note that the type hint should use *pandas.Series* in all cases but there is one variant that *pandas.DataFrame* should be used for its input or output type hint instead when the input or output column is of *pyspark.sql.types.StructType*. The following example shows a Pandas UDF which takes long column, string column and struct column, and outputs a struct column. It requires the function to specify the type hints of *pandas.Series* and *pandas.DataFrame* as below:

```
>>> @pandas_udf("col1 string, col2 long")
>>> def func(s1: pd.Series, s2: pd.Series, s3: pd.DataFrame) -> pd.DataFrame:
...     s3['col2'] = s1 + s2.str.len()
...     return s3
...
>>> # Create a Spark DataFrame that has three columns including a sturct column.
... df = spark.createDataFrame(
...     [[1, "a string", ("a nested string",)]],
...     "long_col long, string_col string, struct_col struct<col1:string>")
>>> df.printSchema()
root
|-- long_column: long (nullable = true)
|-- string_column: string (nullable = true)
|-- struct_column: struct (nullable = true)
|    |-- col1: string (nullable = true)
>>> df.select(func("long_col", "string_col", "struct_col")).printSchema()
|-- func(long_col, string_col, struct_col): struct (nullable = true)
|    |-- col1: string (nullable = true)
|    |-- col2: long (nullable = true)
```

In the following sections, it describes the cominations of the supported type hints. For simplicity, *pandas.DataFrame* variant is omitted.

- **Series to Series** *pandas.Series, ...* -> *pandas.Series*

  The function takes one or more *pandas.Series* and outputs one *pandas.Series*. The output of the function should always be of the same length as the input.

  ```
  >>> @pandas_udf("string")
  ... def to_upper(s: pd.Series) -> pd.Series:
  ...     return s.str.upper()
  ...
  ```

(continues on next page)

(continued from previous page)

```
>>> df = spark.createDataFrame([("John Doe",)], ("name",))
>>> df.select(to_upper("name")).show()
+--------------+
|to_upper(name)|
+--------------+
|      JOHN DOE|
+--------------+
```

```
>>> @pandas_udf("first string, last string")
... def split_expand(s: pd.Series) -> pd.DataFrame:
...     return s.str.split(expand=True)
...
>>> df = spark.createDataFrame([("John Doe",)], ("name",))
>>> df.select(split_expand("name")).show()
+------------------+
|split_expand(name)|
+------------------+
|       [John, Doe]|
+------------------+
```

Note: The length of the input is not that of the whole input column, but is the length of an internal batch used for each call to the function.

- **Iterator of Series to Iterator of Series** *Iterator[pandas.Series] -> Iterator[pandas.Series]*

  The function takes an iterator of *pandas.Series* and outputs an iterator of *pandas.Series*. In this case, the created pandas UDF instance requires one input column when this is called as a PySpark column. The length of the entire output from the function should be the same length of the entire input; therefore, it can prefetch the data from the input iterator as long as the lengths are the same.

  It is also useful when the UDF execution requires initializing some states although internally it works identically as Series to Series case. The pseudocode below illustrates the example.

```
@pandas_udf("long")
def calculate(iterator: Iterator[pd.Series]) -> Iterator[pd.Series]:
    # Do some expensive initialization with a state
    state = very_expensive_initialization()
    for x in iterator:
        # Use that state for whole iterator.
        yield calculate_with_state(x, state)

df.select(calculate("value")).show()
```

```
>>> from typing import Iterator
>>> @pandas_udf("long")
... def plus_one(iterator: Iterator[pd.Series]) -> Iterator[pd.Series]:
...     for s in iterator:
...         yield s + 1
...
>>> df = spark.createDataFrame(pd.DataFrame([1, 2, 3], columns=["v"]))
>>> df.select(plus_one(df.v)).show()
+-----------+
|plus_one(v)|
+-----------+
```

(continues on next page)

(continued from previous page)

```
|           2|
|           3|
|           4|
+------------+
```

Note: The length of each series is the length of a batch internally used.

- **Iterator of Multiple Series to Iterator of Series** *Iterator[Tuple[pandas.Series, ...]] -> Iterator[pandas.Series]*

  The function takes an iterator of a tuple of multiple *pandas.Series* and outputs an iterator of *pandas.Series*. In this case, the created pandas UDF instance requires input columns as many as the series when this is called as a PySpark column. Otherwise, it has the same characteristics and restrictions as Iterator of Series to Iterator of Series case.

```
>>> from typing import Iterator, Tuple
>>> from pyspark.sql.functions import struct, col
>>> @pandas_udf("long")
... def multiply(iterator: Iterator[Tuple[pd.Series, pd.DataFrame]]) ->␣
↪Iterator[pd.Series]:
...     for s1, df in iterator:
...         yield s1 * df.v
...
>>> df = spark.createDataFrame(pd.DataFrame([1, 2, 3], columns=["v"]))
>>> df.withColumn('output', multiply(col("v"), struct(col("v")))).show()
+---+------+
|  v|output|
+---+------+
|  1|     1|
|  2|     4|
|  3|     9|
+---+------+
```

Note: The length of each series is the length of a batch internally used.

- **Series to Scalar** *pandas.Series, ... -> Any*

  The function takes *pandas.Series* and returns a scalar value. The *returnType* should be a primitive data type, and 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. *Any* should ideally be a specific scalar type accordingly.

```
>>> @pandas_udf("double")
... def mean_udf(v: pd.Series) -> float:
...     return v.mean()
...
>>> df = spark.createDataFrame(
...     [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)], ("id", "v"))
>>> df.groupby("id").agg(mean_udf(df['v'])).show()
+---+-----------+
| id|mean_udf(v)|
+---+-----------+
|  1|        1.5|
```

(continues on next page)

(continued from previous page)

```
|  2|        6.0|
+---+-----------+
```

This UDF can also be used as window functions as below:

```
>>> from pyspark.sql import Window
>>> @pandas_udf("double")
... def mean_udf(v: pd.Series) -> float:
...     return v.mean()
...
>>> df = spark.createDataFrame(
...     [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)], ("id", "v"))
>>> w = Window.partitionBy('id').orderBy('v').rowsBetween(-1, 0)
>>> df.withColumn('mean_v', mean_udf("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*

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.

Note: Currently, *pyspark.sql.types.MapType*, *pyspark.sql.types.ArrayType* of *pyspark.sql.types.TimestampType* and nested *pyspark.sql.types.StructType* are currently not supported as output types.

See also:

pyspark.sql.DataFrame.mapInPandas()

See also:

pyspark.sql.GroupedData.applyInPandas()

See also:

pyspark.sql.PandasCogroupedOps.applyInPandas()

See also:

pyspark.sql.UDFRegistration.register()

New in version 2.3.

pyspark.sql.functions.percent_rank

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.percentile_approx

pyspark.sql.functions.**percentile_approx**(*col*, *percentage*, *accuracy=10000*)
Returns the approximate percentile value of numeric column col at the given percentage. The value of percentage must be between 0.0 and 1.0.

The accuracy parameter (default: 10000) is a positive numeric literal which controls approximation accuracy at the cost of memory. Higher value of accuracy yields better accuracy, 1.0/accuracy is the relative error of the approximation.

When percentage is an array, each value of the percentage array must be between 0.0 and 1.0. In this case, returns the approximate percentile array of column col at the given percentage array.

```
>>> key = (col("id") % 3).alias("key")
>>> value = (randn(42) + key * 10).alias("value")
>>> df = spark.range(0, 1000, 1, 1).select(key, value)
>>> df.select(
...     percentile_approx("value", [0.25, 0.5, 0.75], 1000000).alias("quantiles")
... ).printSchema()
root
 |-- quantiles: array (nullable = true)
 |    |-- element: double (containsNull = false)
```

```
>>> df.groupBy("key").agg(
...     percentile_approx("value", 0.5, lit(1000000)).alias("median")
... ).printSchema()
root
 |-- key: long (nullable = true)
 |-- median: double (nullable = true)
```

New in version 3.1.

pyspark.sql.functions.posexplode

pyspark.sql.functions.**posexplode**(*col*)

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], mapfield={"a": "b"})])
>>> 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

pyspark.sql.functions.**posexplode_outer**(*col*)

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, None)],
...     ("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

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

pyspark.sql.functions.**quarter**(*col*)
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

pyspark.sql.functions.**radians**(*col*)
Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

Parameters **col** – angle in degrees

Returns angle in radians, as if computed by *java.lang.Math.toRadians()*

New in version 2.1.

pyspark.sql.functions.rand

pyspark.sql.functions.**rand**(*seed=None*)
Generates a random column with independent and identically distributed (i.i.d.) samples uniformly distributed in [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

pyspark.sql.functions.**randn**(*seed=None*)

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

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

pyspark.sql.functions.**regexp_extract**(*str*, *pattern*, *idx*)

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')).collect()
[Row(d='100')]
>>> df = spark.createDataFrame([('foo',)], ['str'])
>>> df.select(regexp_extract('str', r'(\d+)', 1).alias('d')).collect()
[Row(d='')]
>>> df = spark.createDataFrame([('aaaac',)], ['str'])
>>> df.select(regexp_extract('str', '(a+)(b)?(c)', 2).alias('d')).collect()
[Row(d='')]
```

New in version 1.5.

pyspark.sql.functions.regexp_replace

pyspark.sql.functions.**regexp_replace**(*str*, *pattern*, *replacement*)

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')).collect()
[Row(d='-----')]
```

New in version 1.5.

pyspark.sql.functions.repeat

pyspark.sql.functions.**repeat**(*col*, *n*)

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

pyspark.sql.functions.**reverse**(*col*)

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],) ,([],)], ['data'])
>>> 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

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

pyspark.sql.functions.**round**(*col*, *scale=0*)

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).alias('r')).
↪collect()
[Row(r=3.0)]
```

New in version 1.5.

pyspark.sql.functions.row_number

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

pyspark.sql.functions.**rpad**(*col*, *len*, *pad*)

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

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

pyspark.sql.functions.**schema_of_csv**(*csv*, *options={}*)

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")).collect()
[Row(csv='struct<_c0:int,_c1:string>')]
>>> df.select(schema_of_csv('1|a', {'sep':'|'}).alias("csv")).collect()
[Row(csv='struct<_c0:int,_c1:string>')]
```

New in version 3.0.

pyspark.sql.functions.schema_of_json

pyspark.sql.functions.**schema_of_json**(*json*, *options={}*)

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")).collect()
[Row(json='struct<a:bigint>')]
>>> schema = schema_of_json('{a: 1}', {'allowUnquotedFieldNames':'true'})
>>> df.select(schema.alias("json")).collect()
[Row(json='struct<a:bigint>')]
```

New in version 2.4.

pyspark.sql.functions.second

pyspark.sql.functions.**second**(*col*)

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

pyspark.sql.functions.**sequence**(*start*, *stop*, *step=None*)

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

pyspark.sql.functions.**sha1**(*col*)

Returns the hex string result of SHA-1.

```
>>> spark.createDataFrame([('ABC',)], ['a']).select(sha1('a').alias('hash')).
↪collect()
[Row(hash='3c01bdbb26f358bab27f267924aa2c9a03fcfdb8')]
```

New in version 1.5.

pyspark.sql.functions.sha2

pyspark.sql.functions.**sha2**(*col*, *numBits*)

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='3bc51062973c458d5a6f2d8d64a023246354ad7e064b1e4e009ec8a0699a3043')
>>> digests[1]
Row(s='cd9fb1e148ccd8442e5aa74904cc73bf6fb54d1d54d333bd596aa9bb4bb4e961')
```

New in version 1.5.

pyspark.sql.functions.shiftLeft

pyspark.sql.functions.**shiftLeft**(*col*, *numBits*)

Shift the given value numBits left.

```
>>> spark.createDataFrame([(21,)], ['a']).select(shiftLeft('a', 1).alias('r')).
↪collect()
[Row(r=42)]
```

New in version 1.5.

pyspark.sql.functions.shiftRight

pyspark.sql.functions.**shiftRight**(*col*, *numBits*)

(Signed) shift the given value numBits right.

```
>>> spark.createDataFrame([(42,)], ['a']).select(shiftRight('a', 1).alias('r')).
↪collect()
[Row(r=21)]
```

New in version 1.5.

pyspark.sql.functions.shiftRightUnsigned

pyspark.sql.functions.**shiftRightUnsigned**(*col*, *numBits*)

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

pyspark.sql.functions.**shuffle**(*col*)

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],)], ['data'])
>>> 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

pyspark.sql.functions.**signum**(*col*)

Computes the signum of the given value.

New in version 1.4.

pyspark.sql.functions.sin

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.since

pyspark.sql.functions.**since**(*version*)
A decorator that annotates a function to append the version of Spark the function was added.

pyspark.sql.functions.sinh

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

pyspark.sql.functions.**size**(*col*)
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

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

pyspark.sql.functions.**slice**(*x*, *start*, *length*)
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

pyspark.sql.functions.**sort_array**(*col*, *asc=True*)

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],),([],)], ['data'])
>>> 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

pyspark.sql.functions.**soundex**(*col*)

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

pyspark.sql.functions.**spark_partition_id**()

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")).collect()
[Row(pid=0), Row(pid=0)]
```

New in version 1.6.

pyspark.sql.functions.split

pyspark.sql.functions.**split**(*str*, *pattern*, *limit=-1*)

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

pyspark.sql.functions.**sqrt**(*col*)

Computes the square root of the specified float value.

New in version 1.3.

pyspark.sql.functions.stddev

pyspark.sql.functions.**stddev**(*col*)

Aggregate function: alias for stddev_samp.

New in version 1.6.

pyspark.sql.functions.stddev_pop

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

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

pyspark.sql.functions.**struct**(**cols*)

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

New in version 1.4.

pyspark.sql.functions.substring

pyspark.sql.functions.**substring**(*str*, *pos*, *len*)

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

pyspark.sql.functions.**substring_index**(*str*, *delim*, *count*)

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

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

pyspark.sql.functions.**sumDistinct**(*col*)

Aggregate function: returns the sum of distinct values in the expression.

New in version 1.3.

pyspark.sql.functions.sys

This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv – command line arguments; argv[0] is the script pathname if known path – module search path; path[0] is the script directory, else '' modules – dictionary of loaded modules

displayhook – called to show results in an interactive session excepthook – called to handle any uncaught exception other than SystemExit

To customize printing in an interactive session or to install a custom top-level exception handler, assign other functions to replace these.

stdin – standard input file object; used by input() stdout – standard output file object; used by print() stderr – standard error object; used for error messages

By assigning other file objects (or objects that behave like files) to these, it is possible to redirect all of the interpreter's I/O.

last_type – type of last uncaught exception last_value – value of last uncaught exception last_traceback – traceback of last uncaught exception

These three are only available in an interactive session after a traceback has been printed.

Static objects:

builtin_module_names – tuple of module names built into this interpreter copyright – copyright notice pertaining to this interpreter exec_prefix – prefix used to find the machine-specific Python library executable – absolute path of the executable binary of the Python interpreter float_info – a struct sequence with information about the float implementation. float_repr_style – string indicating the style of repr() output for floats hash_info – a struct sequence with information about the hash algorithm. hexversion – version information encoded as a single integer implementation – Python implementation information. int_info – a struct sequence with information about the int implementation. maxsize – the largest supported length of containers. maxunicode – the value of the largest Unicode code point platform – platform identifier prefix – prefix used to find the Python library thread_info – a struct sequence with information about the thread implementation. version – the version of this interpreter as a string version_info – version information as a named tuple __stdin__ – the original stdin; don't touch! __stdout__ – the original stdout; don't touch! __stderr__ – the original stderr; don't touch! __displayhook__ – the original displayhook; don't touch! __excepthook__ – the original excepthook; don't touch!

Functions:

displayhook() – print an object to the screen, and save it in builtins._ excepthook() – print an exception and its traceback to sys.stderr exc_info() – return thread-safe information about the current exception exit() – exit the interpreter by raising SystemExit getdlopenflags() – returns flags to be used for dlopen() calls getprofile() – get the global profiling function getrefcount() – return the reference count for an object (plus one :-) getrecursionlimit() – return the max recursion depth for the interpreter getsizeof() – return the size of an object in bytes gettrace() – get the global debug tracing function setcheckinterval() – control how often the interpreter checks for events setdlopenflags() – set the flags to be used for dlopen() calls setprofile() – set the global profiling function setrecursionlimit() – set the max recursion depth for the interpreter settrace() – set the global debug tracing function

Functions

`breakpointhook`(*args, **kws)	This hook function is called by built-in breakpoint().
`call_tracing`(func, args)	Call func(*args), while tracing is enabled.The tracing state is saved, and restored afterwards.This is intended to be called from a debugger from a checkpoint, to recursively debug some other code..
`callstats`()	Return a tuple of function call statistics, if CALL_PROFILE was defined when Python was built.
`displayhook`(object)	Print an object to sys.stdout and also save it in builtins._
`exc_info`()	Return information about the most recent exception caught by an except clause in the current stack frame or in an older stack frame.
`excepthook`(exctype, value, traceback)	Handle an exception by displaying it with a traceback on sys.stderr.
`exit`([status])	Exit the interpreter by raising SystemExit(status).
`get_asyncgen_hooks`()	Return a namedtuple of installed asynchronous generators hooks (firstiter, finalizer).
`get_coroutine_origin_tracking_depth`	Check status of origin tracking for coroutine objects in this thread.
`get_coroutine_wrapper`()	Return the wrapper for coroutine objects set by sys.set_coroutine_wrapper.
`getallocatedblocks`()	Return the number of memory blocks currently allocated, regardless of their size.
`getcheckinterval`()	
`getdefaultencoding`()	Return the current default string encoding used by the Unicode implementation.
`getdlopenflags`()	Return the current value of the flags that are used for dlopen calls.
`getfilesystemencodeerrors`()	Return the error mode used to convert Unicode filenames in operating system filenames.
`getfilesystemencoding`()	Return the encoding used to convert Unicode filenames in operating system filenames.
`getprofile`()	Return the profiling function set with sys.setprofile.
`getrecursionlimit`()	Return the current value of the recursion limit, the maximum depth of the Python interpreter stack.
`getrefcount`(object)	Return the reference count of object.
`getsizeof`(object, default)	Return the size of object in bytes.
`getswitchinterval`()	
`gettrace`()	Return the global debug tracing function set with sys.settrace.

continues on next page

Table 40 – continued from previous page

intern(string)	``Intern" the given string.This enters the string in the (global) table of interned strings whose purpose is to speed up dictionary lookups.Return the string itself or the previously interned string object with the same value..
is_finalizing()	Return True if Python is exiting.
set_asyncgen_hooks(*[, firstiter, finalizer])	Set a finalizer for async generators objects.
set_coroutine_origin_tracking_depth	Enable or disable origin tracking for coroutine objects in this thread.
set_coroutine_wrapper(wrapper)	Set a wrapper for coroutine objects.
setcheckinterval(n)	Tell the Python interpreter to check for asynchronous events every n instructions.
setdlopenflags(n)	Set the flags used by the interpreter for dlopen calls, such as when the interpreter loads extension modules.
setprofile(function)	Set the profiling function.
setrecursionlimit(n)	Set the maximum depth of the Python interpreter stack to n.
setswitchinterval(n)	Set the ideal thread switching delay inside the Python interpreter The actual frequency of switching threads can be lower if the interpreter executes long sequences of uninterruptible code (this is implementation-specific and workload-dependent).
settrace(function)	Set the global debug tracing function.

pyspark.sql.functions.tan

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

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

pyspark.sql.functions.**toDegrees**(*col*)

> **Note:** Deprecated in 2.1, use *degrees()* instead.

New in version 1.4.

pyspark.sql.functions.toRadians

pyspark.sql.functions.**toRadians**(*col*)

> **Note:** Deprecated in 2.1, use *radians()* instead.

New in version 1.4.

pyspark.sql.functions.to_csv

pyspark.sql.functions.**to_csv**(*col*, *options={}*)

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

pyspark.sql.functions.**to_date**(*col*, *format=None*)

Converts a `Column` into *pyspark.sql.types.DateType* using the optionally specified format. Specify formats according to **`datetime pattern`_**. 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')).collect()
[Row(date=datetime.date(1997, 2, 28))]
```

New in version 2.2.

pyspark.sql.functions.to_json

pyspark.sql.functions.**to_json**(*col*, *options={}*)

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

pyspark.sql.functions.**to_str**(*value*)

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

pyspark.sql.functions.**to_timestamp**(*col*, *format=None*)

Converts a Column into *pyspark.sql.types.TimestampType* using the optionally specified format. Specify formats according to **`datetime pattern`_**. 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')).collect()
[Row(dt=datetime.datetime(1997, 2, 28, 10, 30))]
```

New in version 2.2.

pyspark.sql.functions.to_utc_timestamp

pyspark.sql.functions.**to_utc_timestamp**(*timestamp*, *tz*)

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 detailing the time zone ID that the input should be adjusted to. It should be in the format of either region-based zone IDs or zone offsets. Region IDs must have the form 'area/city', such as 'America/Los_Angeles'. Zone offsets must be in the format '(+|-)HH:mm', for example '-08:00' or '+01:00'. Also 'UTC' and 'Z' are supported as aliases of '+00:00'. Other short names are not recommended to use because they can be ambiguous.

Changed in version 2.4: *tz* can take a Column containing timezone ID strings.

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00', 'JST')], ['ts', 'tz'])
>>> df.select(to_utc_timestamp(df.ts, "PST").alias('utc_time')).collect()
[Row(utc_time=datetime.datetime(1997, 2, 28, 18, 30))]
>>> df.select(to_utc_timestamp(df.ts, df.tz).alias('utc_time')).collect()
[Row(utc_time=datetime.datetime(1997, 2, 28, 1, 30))]
```

New in version 1.5.

pyspark.sql.functions.transform

pyspark.sql.functions.**transform**(*col*, *f*)

Returns an array of elements after applying a transformation to each element in the input array.

Parameters

- **col** – name of column or expression
- **f** – a function that is applied to each element of the input array. Can take one of the following forms:
  - Unary `(x: Column) -> Column: ...`
  - **Binary `(x: Column, i: Column) -> Column...`, where the second argument is** a 0-based index of the element.

  and can use methods of *pyspark.sql.Column*, functions defined in `pyspark.sql.functions` and Scala `UserDefinedFunctions`. Python `UserDefinedFunctions` are not supported (SPARK-27052).

Returns a *pyspark.sql.Column*

```
>>> df = spark.createDataFrame([(1, [1, 2, 3, 4])], ("key", "values"))
>>> df.select(transform("values", lambda x: x * 2).alias("doubled")).show()
+------------+
|     doubled|
+------------+
|[2, 4, 6, 8]|
+------------+
```

```
>>> def alternate(x, i):
...     return when(i % 2 == 0, x).otherwise(-x)
>>> df.select(transform("values", alternate).alias("alternated")).show()
+--------------+
|    alternated|
+--------------+
|[1, -2, 3, -4]|
+--------------+
```

New in version 3.1.

pyspark.sql.functions.transform_keys

pyspark.sql.functions.**transform_keys**(*col*, *f*)

Applies a function to every key-value pair in a map and returns a map with the results of those applications as the new keys for the pairs.

Parameters

- **col** – name of column or expression
- **f** – a binary function `(k: Column, v: Column) -> Column...` Can use methods of *pyspark.sql.Column*, functions defined in `pyspark.sql.functions` and Scala `UserDefinedFunctions`. Python `UserDefinedFunctions` are not supported (SPARK-27052).

Returns a *pyspark.sql.Column*

```
>>> df = spark.createDataFrame([(1, {"foo": -2.0, "bar": 2.0})], ("id", "data"))
>>> df.select(transform_keys(
...     "data", lambda k, _: upper(k)).alias("data_upper")
... ).show(truncate=False)
+-------------------------+
|data_upper               |
+-------------------------+
|[BAR -> 2.0, FOO -> -2.0]|
+-------------------------+
```

New in version 3.1.

pyspark.sql.functions.transform_values

pyspark.sql.functions.**transform_values**(*col*, *f*)

Applies a function to every key-value pair in a map and returns a map with the results of those applications as the new values for the pairs.

Parameters

- **col** – name of column or expression
- **f** – a binary function `(k: Column, v: Column) -> Column...` Can use methods of *pyspark.sql.Column*, functions defined in `pyspark.sql.functions` and Scala `UserDefinedFunctions`. Python `UserDefinedFunctions` are not supported (SPARK-27052).

Returns a *pyspark.sql.Column*

```
>>> df = spark.createDataFrame([(1, {"IT": 10.0, "SALES": 2.0, "OPS": 24.0})], (
↪"id", "data"))
>>> df.select(transform_values(
...     "data", lambda k, v: when(k.isin("IT", "OPS"), v + 10.0).otherwise(v)
... ).alias("new_data")).show(truncate=False)
+---------------------------------------+
|new_data                               |
+---------------------------------------+
|[OPS -> 34.0, IT -> 20.0, SALES -> 2.0]|
+---------------------------------------+
```

New in version 3.1.

pyspark.sql.functions.translate

pyspark.sql.functions.**translate**(*srcCol*, *matching*, *replace*)

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('a', "rnlt",
↪"123") \
...     .alias('r')).collect()
[Row(r='1a2s3ae')]
```

New in version 1.5.

pyspark.sql.functions.trim

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

pyspark.sql.functions.**trunc**(*date*, *format*)
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

pyspark.sql.functions.**udf**(*f=None*, *returnType=StringType*)
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), IntegerType()).
↪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.

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", "age"))
>>> df.select(slen("name").alias("slen(name)"), to_upper("name"), add_one("age")).
↪show()
+----------+--------------+------------+
|slen(name)|to_upper(name)|add_one(age)|
+----------+--------------+------------+
|         8|      JOHN DOE|          22|
+----------+--------------+------------+
```

New in version 1.3.

pyspark.sql.functions.unbase64

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

pyspark.sql.functions.**unhex**(*col*)

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')).collect()
[Row(unhex(a)=bytearray(b'ABC'))]
```

New in version 1.5.

pyspark.sql.functions.unix_timestamp

pyspark.sql.functions.**unix_timestamp**(*timestamp=None*, *format='yyyy-MM-dd HH:mm:ss'*)

Convert time string with given pattern ('yyyy-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_Angeles")
>>> time_df = spark.createDataFrame([('2015-04-08',)], ['dt'])
>>> time_df.select(unix_timestamp('dt', 'yyyy-MM-dd').alias('unix_time')).
↪collect()
```

(continues on next page)

(continued from previous page)

```
[Row(unix_time=1428476400)]
>>> spark.conf.unset("spark.sql.session.timeZone")
```

New in version 1.5.

pyspark.sql.functions.upper

pyspark.sql.functions.**upper**(*col*)
Converts a string expression to upper case.

New in version 1.5.

pyspark.sql.functions.var_pop

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

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

pyspark.sql.functions.**variance**(*col*)
Aggregate function: alias for var_samp.

New in version 1.6.

pyspark.sql.functions.warnings

Python part of the warnings subsystem.

Functions

filterwarnings(action[, message, category, ...])	Insert an entry into the list of warnings filters (at the front).
formatwarning(message, category, filename, ...)	Function to format a warning the standard way.
resetwarnings()	Clear the list of warning filters, so that no filters are active.
showwarning(message, category, filename, lineno)	Hook to write a warning to a file; replace if you like.
simplefilter(action[, category, lineno, append])	Insert a simple entry into the list of warnings filters (at the front).

Classes

WarningMessage(message, category, filename, ...)	
catch_warnings(*[, record, module])	A context manager that copies and restores the warnings filter upon exiting the context.

pyspark.sql.functions.weekofyear

pyspark.sql.functions.**weekofyear**(*col*)

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

pyspark.sql.functions.**when**(*condition*, *value*)

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")).collect()
[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

pyspark.sql.functions.**window**(*timeColumn*, *windowDuration*, *slideDuration=None*, *startTime=None*)

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("date", "val")
>>> w = df.groupBy(window("date", "5 seconds")).agg(sum("val").alias("sum"))
>>> w.select(w.window.start.cast("string").alias("start"),
...           w.window.end.cast("string").alias("end"), "sum").collect()
[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

pyspark.sql.functions.**xxhash64**(**cols*)

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').alias('hash')).
↪collect()
[Row(hash=4105715581806190027)]
```

New in version 3.0.

pyspark.sql.functions.year

pyspark.sql.functions.**year**(*col*)

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.functions.zip_with

pyspark.sql.functions.**zip_with**(*col1*, *col2*, *f*)

Merge two given arrays, element-wise, into a single array using a function. If one array is shorter, nulls are appended at the end to match the length of the longer array, before applying the function.

Parameters

- **col1** – name of the first column or expression
- **col2** – name of the second column or expression
- **f** – a binary function `(x1: Column, x2: Column) -> Column...` Can use methods of *pyspark.sql.Column*, functions defined in `pyspark.sql.functions` and Scala `UserDefinedFunctions`. Python `UserDefinedFunctions` are not supported (SPARK-27052).

Returns a *pyspark.sql.Column*

```
>>> df = spark.createDataFrame([(1, [1, 3, 5, 8], [0, 2, 4, 6])], ("id", "xs", "ys
↪"))
>>> df.select(zip_with("xs", "ys", lambda x, y: x ** y).alias("powers")).
↪show(truncate=False)
+---------------------------+
|powers                     |
+---------------------------+
|[1.0, 9.0, 625.0, 262144.0]|
+---------------------------+
```

```
>>> df = spark.createDataFrame([(1, ["foo", "bar"], [1, 2, 3])], ("id", "xs", "ys
↪"))
>>> df.select(zip_with("xs", "ys", lambda x, y: concat_ws("_", x, y)).alias("xs_ys
↪")).show()
+-----------------+
|            xs_ys|
+-----------------+
|[foo_1, bar_2, 3]|
+-----------------+
```

New in version 3.1.

from_avro(data, jsonFormatSchema[, options])	Converts a binary column of Avro format into its corresponding catalyst value.
to_avro(data[, jsonFormatSchema])	Converts a column into binary of avro format.

pyspark.sql.avro.functions.from_avro

pyspark.sql.avro.functions.**from_avro**(*data*, *jsonFormatSchema*, *options={}*)

Converts a binary column of Avro format into its corresponding catalyst value. The specified schema must match the read data, otherwise the behavior is undefined: it may fail or return arbitrary result. To deserialize the data with a compatible and evolved schema, the expected Avro schema can be set via the option avroSchema.

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","fields":
...     [{"name":"avro","type":[{"type":"record","name":"value","namespace":
↪"topLevelRecord",
...     "fields":[{"name":"age","type":["long","null"]},
...     {"name":"name","type":["string","null"]}]},"null"]}]}'''
```

(continues on next page)

(continued from previous page)

```
>>> avroDf.select(from_avro(avroDf.avro, jsonFormatSchema).alias("value")).
↪collect()
[Row(value=Row(avro=Row(age=2, name='Alice')))]
```

New in version 3.0.

pyspark.sql.avro.functions.to_avro

pyspark.sql.avro.functions.**to_avro**(*data*, *jsonFormatSchema=''*)

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")).collect()
[Row(suite=bytearray(b'\x02\x00'))]
```

New in version 3.0.

Window

Window.currentRow	
Window.orderBy(*cols)	Creates a WindowSpec with the ordering defined.
Window.partitionBy(*cols)	Creates a WindowSpec with the partitioning defined.
Window.rangeBetween(start, end)	Creates a WindowSpec with the frame boundaries defined, from *start* (inclusive) to *end* (inclusive).
Window.rowsBetween(start, end)	Creates a WindowSpec with the frame boundaries defined, from *start* (inclusive) to *end* (inclusive).
Window.unboundedFollowing	
Window.unboundedPreceding	

pyspark.sql.Window.currentRow

```
Window.currentRow = 0
```

pyspark.sql.Window.orderBy

static Window.**orderBy**(**cols*)

Creates a WindowSpec with the ordering defined.

New in version 1.4.

pyspark.sql.Window.partitionBy

static Window.**partitionBy**(**cols*)

Creates a WindowSpec with the partitioning defined.

New in version 1.4.

pyspark.sql.Window.rangeBetween

static Window.**rangeBetween**(*start*, *end*)

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"), (3, "b")]
>>> df = sqlContext.createDataFrame(tup, ["id", "category"])
>>> window = Window.partitionBy("category").orderBy("id").rangeBetween(Window.
↪currentRow, 1)
>>> df.withColumn("sum", func.sum("id").over(window)).sort("id", "category").
↪show()
+---+--------+---+
| id|category|sum|
+---+--------+---+
|  1|       a|  4|
|  1|       a|  4|
|  1|       b|  3|
|  2|       a|  2|
```

(continues on next page)

(continued from previous page)

```
|  2|       b|  5|
|  3|       b|  3|
+---+--------+---+
```

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.

pyspark.sql.Window.rowsBetween

static `Window.`**rowsBetween**(*start*, *end*)

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"), (3, "b")]
>>> df = sqlContext.createDataFrame(tup, ["id", "category"])
>>> window = Window.partitionBy("category").orderBy("id").rowsBetween(Window.
↪currentRow, 1)
>>> df.withColumn("sum", func.sum("id").over(window)).sort("id", "category", "sum
↪").show()
+---+--------+---+
| id|category|sum|
+---+--------+---+
|  1|       a|  2|
|  1|       a|  3|
|  1|       b|  3|
|  2|       a|  2|
|  2|       b|  5|
|  3|       b|  3|
+---+--------+---+
```

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.

pyspark.sql.Window.unboundedFollowing

`Window.`**`unboundedFollowing = 9223372036854775807`**

pyspark.sql.Window.unboundedPreceding

`Window.`**`unboundedPreceding = -9223372036854775808`**

Grouping

GroupedData.agg(*exprs)	Compute aggregates and returns the result as a *DataFrame*.
GroupedData.apply(udf)	It is an alias of *pyspark.sql.GroupedData.applyInPandas()*; however, it takes a *pyspark.sql.functions.pandas_udf()* whereas *pyspark.sql.GroupedData.applyInPandas()* takes a Python native function.
GroupedData.applyInPandas(func, schema)	Maps each group of the current *DataFrame* using a pandas udf and returns the result as a *DataFrame*.
GroupedData.avg(*cols)	Computes average values for each numeric columns for each group.
GroupedData.cogroup(other)	Cogroups this group with another group so that we can run cogrouped operations.
GroupedData.count()	Counts the number of records for each group.
GroupedData.max(*cols)	Computes the max value for each numeric columns for each group.
GroupedData.mean(*cols)	Computes average values for each numeric columns for each group.
GroupedData.min(*cols)	Computes the min value for each numeric column for each group.
GroupedData.pivot(pivot_col[, values])	Pivots a column of the current *DataFrame* and perform the specified aggregation.
GroupedData.sum(*cols)	Compute the sum for each numeric columns for each group.

pyspark.sql.GroupedData.agg

GroupedData.**agg**(**exprs*)

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)=5)]
```

New in version 1.3.

pyspark.sql.GroupedData.apply

GroupedData.**apply**(*udf*)

It is an alias of *pyspark.sql.GroupedData.applyInPandas()*; however, it takes a *pyspark.sql.functions.pandas_udf()* whereas *pyspark.sql.GroupedData.applyInPandas()* takes a Python native function.

Note: It is preferred to use *pyspark.sql.GroupedData.applyInPandas()* over this API. This API will be deprecated in the future releases.

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.

pyspark.sql.GroupedData.applyInPandas

GroupedData.**applyInPandas**(*func*, *schema*)

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

The 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 *schema* 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 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.

Parameters

- **func** – a Python native function that takes a *pandas.DataFrame*, and outputs a *pandas.DataFrame*.
- **schema** – the return type of the *func* in PySpark. The value can be either a *pyspark.sql.types.DataType* object or a DDL-formatted type string.

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

Alternatively, the user can pass 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.

```
>>> df = spark.createDataFrame(
...     [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)],
...     ("id", "v"))
>>> def mean_func(key, pdf):
...     # key is a tuple of one numpy.int64, which is the value
...     # of 'id' for the current group
...     return pd.DataFrame([key + (pdf.v.mean(),)])
>>> df.groupby('id').applyInPandas(
...     mean_func, schema="id long, v double").show()
+---+---+
| id|  v|
+---+---+
|  1|1.5|
|  2|6.0|
+---+---+
>>> def sum_func(key, pdf):
...     # key is a tuple of two numpy.int64s, which is the values
...     # 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)).applyInPandas(
...     sum_func, schema="id long, `ceil(v / 2)` long, v double").show()
+---+-----------+----+
| id|ceil(v / 2)|   v|
+---+-----------+----+
|  2|          5|10.0|
|  1|          1| 3.0|
```

(continues on next page)

(continued from previous page)

```
|  2|           3| 5.0|
|  2|           2| 3.0|
+---+------------+----+
```

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.

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)]))*.

Note: Experimental

See also:

pyspark.sql.functions.pandas_udf()

New in version 3.0.

pyspark.sql.GroupedData.avg

GroupedData.**avg**(**cols*)

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.

pyspark.sql.GroupedData.cogroup

GroupedData.**cogroup**(*other*)

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.

pyspark.sql.GroupedData.count

GroupedData.**count**()

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.

pyspark.sql.GroupedData.max

GroupedData.**max**(**cols*)

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.

pyspark.sql.GroupedData.mean

GroupedData.**mean**(**cols*)

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.

pyspark.sql.GroupedData.min

GroupedData.**min**(**cols*)

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.

pyspark.sql.GroupedData.pivot

GroupedData.**pivot**(*pivot_col*, *values=None*)

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("earnings").
↪collect()
[Row(year=2012, dotNET=15000, Java=20000), Row(year=2013, dotNET=48000,␣
↪Java=30000)]
```

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=30000,␣
↪dotNET=48000)]
>>> df5.groupBy("sales.year").pivot("sales.course").sum("sales.earnings").
↪collect()
[Row(year=2012, Java=20000, dotNET=15000), Row(year=2013, Java=30000,␣
↪dotNET=48000)]
```

New in version 1.6.

pyspark.sql.GroupedData.sum

GroupedData.**sum**(**cols*)

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.

1.3.2 Structured Streaming

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive. You can use the Dataset/DataFrame to express streaming aggregations, event-time windows, stream-to-batch joins, etc. The computation is executed on the same optimized Spark SQL engine. Finally, the system ensures end-to-end exactly-once fault-tolerance guarantees through checkpointing and Write-Ahead Logs. In short, Structured Streaming provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming.

Note that you can leverage almost all Spark SQL APIs in PySpark. This page describes Structured Streaming specific features.

Core Classes

The classes below are core classes in Structured Streaming APIs at PySpark.

DataStreamReader(spark)	Interface used to load a streaming *DataFrame* from external storage systems (e.g.
DataStreamWriter(df)	Interface used to write a streaming *DataFrame* to external storage systems (e.g.
ForeachBatchFunction(sql_ctx, func)	This is the Python implementation of Java interface 'ForeachBatchFunction'.
StreamingQuery(jsq)	A handle to a query that is executing continuously in the background as new data arrives.
StreamingQueryException(desc, stackTrace[, ...])	Exception that stopped a *StreamingQuery*.
StreamingQueryManager(jsqm)	A class to manage all the *StreamingQuery* StreamingQueries active.

pyspark.sql.streaming.DataStreamReader

class pyspark.sql.streaming.**DataStreamReader**(*spark*)

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

Note: Evolving.

New in version 2.0.

__init__(*spark*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(spark)	Initialize self.
csv(path[, schema, sep, encoding, quote, ...])	Loads a CSV file stream and returns the result as a DataFrame.
format(source)	Specifies the input data source format.
json(path[, schema, primitivesAsString, ...])	Loads a JSON file stream and returns the results as a DataFrame.
load([path, format, schema])	Loads a data stream from a data source and returns it as a *DataFrame*.
option(key, value)	Adds an input option for the underlying data source.
options(**options)	Adds input options for the underlying data source.
orc(path[, mergeSchema, pathGlobFilter, ...])	Loads a ORC file stream, returning the result as a DataFrame.
parquet(path[, mergeSchema, pathGlobFilter, ...])	Loads a Parquet file stream, returning the result as a DataFrame.
schema(schema)	Specifies the input schema.
text(path[, wholetext, lineSep, ...])	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.

pyspark.sql.streaming.DataStreamWriter

class pyspark.sql.streaming.**DataStreamWriter**(*df*)

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.

__init__(*df*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(df)	Initialize self.
foreach(f)	Sets the output of the streaming query to be processed using the provided writer f.
foreachBatch(func)	Sets the output of the streaming query to be processed using the provided function.
format(source)	Specifies the underlying output data source.
option(key, value)	Adds an output option for the underlying data source.
options(**options)	Adds output options for the underlying data source.
outputMode(outputMode)	Specifies how data of a streaming DataFrame/Dataset is written to a streaming sink.

continues on next page

Table 48 – continued from previous page

partitionBy(*cols)	Partitions the output by the given columns on the file system.
queryName(queryName)	Specifies the name of the *StreamingQuery* that can be started with *start()*.
start([path, format, outputMode, ...])	Streams the contents of the DataFrame to a data source.
trigger([processingTime, once, continuous])	Set the trigger for the stream query.

pyspark.sql.streaming.ForeachBatchFunction

class pyspark.sql.streaming.**ForeachBatchFunction**(*sql_ctx*, *func*)

This is the Python implementation of Java interface 'ForeachBatchFunction'. This wraps the user-defined 'foreachBatch' function such that it can be called from the JVM when the query is active.

__init__(*sql_ctx*, *func*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(sql_ctx, func)	Initialize self.
call(jdf, batch_id)	

pyspark.sql.streaming.StreamingQuery

class pyspark.sql.streaming.**StreamingQuery**(*jsq*)

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.

__init__(*jsq*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(jsq)	Initialize self.
awaitTermination([timeout])	Waits for the termination of *this* query, either by query.stop() or by an exception.
exception()	**return** the StreamingQueryException if the query was terminated by an exception, or None.

continues on next page

Table 50 – continued from previous page

explain([extended])	Prints the (logical and physical) plans to the console for debugging purpose.
processAllAvailable()	Blocks until all available data in the source has been processed and committed to the sink.
stop()	Stop this streaming query.

Attributes

id	Returns the unique id of this query that persists across restarts from checkpoint data.
isActive	Whether this streaming query is currently active or not.
lastProgress	Returns the most recent `StreamingQueryProgress` update of this streaming query or None if there were no progress updates
name	Returns the user-specified name of the query, or null if not specified.
recentProgress	Returns an array of the most recent [[StreamingQueryProgress]] updates for this query.
runId	Returns the unique id of this query that does not persist across restarts.
status	Returns the current status of the query.

pyspark.sql.streaming.StreamingQueryException

exception pyspark.sql.streaming.**StreamingQueryException**(*desc*, *stackTrace*, *cause=None*)

Exception that stopped a *StreamingQuery*.

pyspark.sql.streaming.StreamingQueryManager

class pyspark.sql.streaming.**StreamingQueryManager**(*jsqm*)

A class to manage all the *StreamingQuery* StreamingQueries active.

Note: Evolving

New in version 2.0.

__init__(*jsqm*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(jsqm)	Initialize self.
awaitAnyTermination([timeout])	Wait until any of the queries on the associated SQLContext has terminated since the creation of the context, or since *resetTerminated()* was called.
get(id)	Returns an active query from this SQLContext or throws exception if an active query with this name doesn't exist.
resetTerminated()	Forget about past terminated queries so that *awaitAnyTermination()* can be used again to wait for new terminations.

Attributes

active	Returns a list of active queries associated with this SQLContext

Input and Output

DataStreamReader.csv(path[, schema, sep, ...])	Loads a CSV file stream and returns the result as a `DataFrame`.
DataStreamReader.format(source)	Specifies the input data source format.
DataStreamReader.json(path[, schema, ...])	Loads a JSON file stream and returns the results as a `DataFrame`.
DataStreamReader.load([path, format, schema])	Loads a data stream from a data source and returns it as a *DataFrame*.
DataStreamReader.option(key, value)	Adds an input option for the underlying data source.
DataStreamReader.options(**options)	Adds input options for the underlying data source.
DataStreamReader.orc(path[, mergeSchema, ...])	Loads a ORC file stream, returning the result as a `DataFrame`.
DataStreamReader.parquet(path[, ...])	Loads a Parquet file stream, returning the result as a `DataFrame`.
DataStreamReader.schema(schema)	Specifies the input schema.
DataStreamReader.text(path[, wholetext, ...])	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.
DataStreamWriter.foreach(f)	Sets the output of the streaming query to be processed using the provided writer `f`.
DataStreamWriter.foreachBatch(func)	Sets the output of the streaming query to be processed using the provided function.
DataStreamWriter.format(source)	Specifies the underlying output data source.
DataStreamWriter.option(key, value)	Adds an output option for the underlying data source.
DataStreamWriter.options(**options)	Adds output options for the underlying data source.
DataStreamWriter.outputMode(outputMode)	Specifies how data of a streaming DataFrame/Dataset is written to a streaming sink.

continues on next page

Table 54 – continued from previous page

DataStreamWriter.partitionBy(*cols)	Partitions the output by the given columns on the file system.
DataStreamWriter.queryName(queryName)	Specifies the name of the *StreamingQuery* that can be started with `start()`.
DataStreamWriter.start([path, format, ...])	Streams the contents of the `DataFrame` to a data source.
DataStreamWriter.trigger([processingTime, ...])	Set the trigger for the stream query.

pyspark.sql.streaming.DataStreamReader.csv

`DataStreamReader.`**`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*, *pathGlobFilter=None*, *recursiveFileLookup=None*)

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 **`datetime pattern`_**. This applies to date type. If None is set, it uses the default value, `yyyy-MM-dd`.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at **`datetime pattern`_**. This applies to timestamp type. If None is set, it uses the default value, `yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX]`.
- **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.
- **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`_**.
- **recursiveFileLookup** – recursively scan a directory for files. Using this option disables **`partition discovery`_**.

```
>>> csv_sdf = spark.readStream.csv(tempfile.mkdtemp(), schema = sdf_schema)
>>> csv_sdf.isStreaming
True
>>> csv_sdf.schema == sdf_schema
True
```

New in version 2.0.

pyspark.sql.streaming.DataStreamReader.format

DataStreamReader.**format**(*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.

pyspark.sql.streaming.DataStreamReader.json

DataStreamReader.**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*, *pathGlobFilter=None*, *recursiveFileLookup=None*)

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 datetime pattern. This applies to date type. If None is set, it uses the default value, `yyyy-MM-dd`.
- **`timestampFormat`** – sets the string that indicates a timestamp format. Custom date formats follow the formats at datetime pattern. This applies to timestamp type. If None is set, it uses the default value, `yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX]`.
- **`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`.
- **`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.
- **`recursiveFileLookup`** – recursively scan a directory for files. Using this option disables partition discovery.

```
>>> json_sdf = spark.readStream.json(tempfile.mkdtemp(), schema = sdf_schema)
>>> json_sdf.isStreaming
True
>>> json_sdf.schema == sdf_schema
True
```

New in version 2.0.

pyspark.sql.streaming.DataStreamReader.load

DataStreamReader.**load**(*path=None*, *format=None*, *schema=None*, ***options*)

Loads a data stream from a data source and returns it as a *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.

pyspark.sql.streaming.DataStreamReader.option

DataStreamReader.**option**(*key*, *value*)

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 time zone ID to be used to parse** timestamps in the JSON/CSV datasources or partition values. The following formats of *timeZone* are supported:
  - Region-based zone ID: It should have the form 'area/city', such as 'America/Los_Angeles'.
  - Zone offset: It should be in the format '(+|-)HH:mm', for example '-08:00' or '+01:00'. Also 'UTC' and 'Z' are supported as aliases of '+00:00'.

  Other short names like 'CST' are not recommended to use because they can be ambiguous. If it isn't set, the current value of the SQL config `spark.sql.session.timeZone` is used by default.

Note: Evolving.

```
>>> s = spark.readStream.option("x", 1)
```

New in version 2.0.

pyspark.sql.streaming.DataStreamReader.options

DataStreamReader.**options**(***options*)

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 time zone ID to be used to parse** timestamps in the JSON/CSV datasources or partition values. The following formats of *timeZone* are supported:
  - Region-based zone ID: It should have the form 'area/city', such as 'America/Los_Angeles'.
  - Zone offset: It should be in the format '(+|-)HH:mm', for example '-08:00' or '+01:00'. Also 'UTC' and 'Z' are supported as aliases of '+00:00'.

  Other short names like 'CST' are not recommended to use because they can be ambiguous. If it isn't set, the current value of the SQL config `spark.sql.session.timeZone` is used by default.

Note: Evolving.

```
>>> s = spark.readStream.options(x="1", y=2)
```

New in version 2.0.

pyspark.sql.streaming.DataStreamReader.orc

DataStreamReader.**orc**(*path*, *mergeSchema=None*, *pathGlobFilter=None*, *recursiveFileLookup=None*)

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

```
>>> orc_sdf = spark.readStream.schema(sdf_schema).orc(tempfile.mkdtemp())
>>> orc_sdf.isStreaming
True
>>> orc_sdf.schema == sdf_schema
True
```

New in version 2.3.

pyspark.sql.streaming.DataStreamReader.parquet

DataStreamReader.**parquet**(*path*, *mergeSchema=None*, *pathGlobFilter=None*, *recursiveFileLookup=None*)

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

```
>>> parquet_sdf = spark.readStream.schema(sdf_schema).parquet(tempfile.mkdtemp())
>>> parquet_sdf.isStreaming
True
>>> parquet_sdf.schema == sdf_schema
True
```

New in version 2.0.

pyspark.sql.streaming.DataStreamReader.schema

DataStreamReader.**schema**(*schema*)

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.

pyspark.sql.streaming.DataStreamReader.text

DataStreamReader.**text**(*path*, *wholetext=False*, *lineSep=None*, *pathGlobFilter=None*, *recursiveFileLookup=None*)

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.
- **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`_**.
- **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.

pyspark.sql.streaming.DataStreamWriter.foreach

DataStreamWriter.**foreach**(*f*)

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.

pyspark.sql.streaming.DataStreamWriter.foreachBatch

DataStreamWriter.**foreachBatch**(*func*)

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.

pyspark.sql.streaming.DataStreamWriter.format

DataStreamWriter.**format**(*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.

pyspark.sql.streaming.DataStreamWriter.option

DataStreamWriter.**option**(*key*, *value*)

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 time zone ID to be used to format** timestamps in the JSON/CSV datasources or partition values. The following formats of *timeZone* are supported:
  - Region-based zone ID: It should have the form 'area/city', such as 'America/Los_Angeles'.
  - Zone offset: It should be in the format '(+|-)HH:mm', for example '-08:00' or '+01:00'. Also 'UTC' and 'Z' are supported as aliases of '+00:00'.

  Other short names like 'CST' are not recommended to use because they can be ambiguous. If it isn't set, the current value of the SQL config `spark.sql.session.timeZone` is used by default.

Note: Evolving.

New in version 2.0.

pyspark.sql.streaming.DataStreamWriter.options

`DataStreamWriter.`**`options`**`(**options)`

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 time zone ID to be used to format** timestamps in the JSON/CSV datasources or partition values. The following formats of *timeZone* are supported:
  - Region-based zone ID: It should have the form 'area/city', such as 'America/Los_Angeles'.
  - Zone offset: It should be in the format '(+|-)HH:mm', for example '-08:00' or '+01:00'. Also 'UTC' and 'Z' are supported as aliases of '+00:00'.

  Other short names like 'CST' are not recommended to use because they can be ambiguous. If it isn't set, the current value of the SQL config `spark.sql.session.timeZone` is used by default.

Note: Evolving.

New in version 2.0.

pyspark.sql.streaming.DataStreamWriter.outputMode

`DataStreamWriter.`**`outputMode`**`(outputMode)`

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.

pyspark.sql.streaming.DataStreamWriter.partitionBy

DataStreamWriter.**partitionBy**(**cols*)

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.

pyspark.sql.streaming.DataStreamWriter.queryName

DataStreamWriter.**queryName**(*queryName*)

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.

pyspark.sql.streaming.DataStreamWriter.start

DataStreamWriter.**start**(*path=None*, *format=None*, *outputMode=None*, *partitionBy=None*, *queryName=None*, ***options*)

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').start()
>>> sq.isActive
True
>>> sq.name
'this_query'
>>> sq.stop()
>>> sq.isActive
False
>>> sq = sdf.writeStream.trigger(processingTime='5 seconds').start(
...     queryName='that_query', outputMode="append", format='memory')
>>> sq.name
'that_query'
>>> sq.isActive
True
>>> sq.stop()
```

New in version 2.0.

pyspark.sql.streaming.DataStreamWriter.trigger

DataStreamWriter.**trigger**(*processingTime=None*, *once=None*, *continuous=None*)

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')
```

(continues on next page)

(continued from previous page)

```
>>> # 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.

Query Management

StreamingQuery.awaitTermination([timeout])	Waits for the termination of *this* query, either by `query.stop()` or by an exception.
StreamingQuery.exception()	**return** the StreamingQueryException if the query was terminated by an exception, or None.
StreamingQuery.explain([extended])	Prints the (logical and physical) plans to the console for debugging purpose.
StreamingQuery.id	Returns the unique id of this query that persists across restarts from checkpoint data.
StreamingQuery.isActive	Whether this streaming query is currently active or not.
StreamingQuery.lastProgress	Returns the most recent `StreamingQueryProgress` update of this streaming query or None if there were no progress updates
StreamingQuery.name	Returns the user-specified name of the query, or null if not specified.
StreamingQuery.processAllAvailable()	Blocks until all available data in the source has been processed and committed to the sink.
StreamingQuery.recentProgress	Returns an array of the most recent [[StreamingQueryProgress]] updates for this query.
StreamingQuery.runId	Returns the unique id of this query that does not persist across restarts.
StreamingQuery.status	Returns the current status of the query.
StreamingQuery.stop()	Stop this streaming query.
StreamingQueryManager.active	Returns a list of active queries associated with this SQLContext
StreamingQueryManager.awaitAnyTermination([...])	Wait until any of the queries on the associated SQLContext has terminated since the creation of the context, or since `resetTerminated()` was called.
StreamingQueryManager.get(id)	Returns an active query from this SQLContext or throws exception if an active query with this name doesn't exist.
StreamingQueryManager.resetTerminated()	Forget about past terminated queries so that `awaitAnyTermination()` can be used again to wait for new terminations.

pyspark.sql.streaming.StreamingQuery.awaitTermination

StreamingQuery.**awaitTermination**(*timeout=None*)

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.

pyspark.sql.streaming.StreamingQuery.exception

StreamingQuery.**exception**()

Returns the StreamingQueryException if the query was terminated by an exception, or None.

New in version 2.1.

pyspark.sql.streaming.StreamingQuery.explain

StreamingQuery.**explain**(*extended=False*)

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_explain').start()
>>> sq.processAllAvailable() # Wait a bit to generate the runtime plans.
>>> 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.

pyspark.sql.streaming.StreamingQuery.id

property StreamingQuery.**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.

pyspark.sql.streaming.StreamingQuery.isActive

property StreamingQuery.**isActive**

Whether this streaming query is currently active or not.

New in version 2.0.

pyspark.sql.streaming.StreamingQuery.lastProgress

property StreamingQuery.**lastProgress**

Returns the most recent `StreamingQueryProgress` update of this streaming query or None if there were no progress updates

Returns a map

New in version 2.1.

pyspark.sql.streaming.StreamingQuery.name

property StreamingQuery.**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.

pyspark.sql.streaming.StreamingQuery.processAllAvailable

StreamingQuery.**processAllAvailable**()

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.

pyspark.sql.streaming.StreamingQuery.recentProgress

property StreamingQuery.**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.

pyspark.sql.streaming.StreamingQuery.runId

property StreamingQuery.**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.

pyspark.sql.streaming.StreamingQuery.status

property StreamingQuery.**status**

Returns the current status of the query.

New in version 2.1.

pyspark.sql.streaming.StreamingQuery.stop

StreamingQuery.**stop**()

Stop this streaming query.

New in version 2.0.

pyspark.sql.streaming.StreamingQueryManager.active

property StreamingQueryManager.**active**

Returns a list of active queries associated with this SQLContext

```
>>> sq = sdf.writeStream.format('memory').queryName('this_query').start()
>>> 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.

pyspark.sql.streaming.StreamingQueryManager.awaitAnyTermination

StreamingQueryManager.**awaitAnyTermination**(*timeout=None*)

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.

pyspark.sql.streaming.StreamingQueryManager.get

StreamingQueryManager.**get**(*id*)

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').start()
>>> 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.

pyspark.sql.streaming.StreamingQueryManager.resetTerminated

StreamingQueryManager.**resetTerminated**()

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.

1.3.3 ML

Classification

Clustering

Recommendation

Reression

Statistics

Turning

Evaluation

Image

1.3.4 Spark Streaming

Core Classes

Input and Output

Streaming Context

DStream

Kinesis

1.3.5 MLlib

Classification

Clustering

Recommendation

Reression

Statistics

Turning

Evaluation

Image

1.3.6 Spark Core

Core Classes

Spark Context APIs

Input and Output

RDD APIs

1.3.7 Resource Management

Resource Allocation

Resource Profile

1.4 Development

1.4.1 Developer Tools

Testing PySpark

To run individual PySpark tests, you can use run-tests script under python directory. Test cases are located at tests package under each PySpark packages. Note that, if you add some changes into Scala or Python side in Apache Spark, you need to manually build Apache Spark again before running PySpark tests in order to apply the changes. Running PySpark testing script does not automatically build it.

Also, note that there is an ongoing issue to use PySpark on macOS High Serria+. `OBJC_DISABLE_INITIALIZE_FORK_SAFETY` should be set to `YES` in order to run some of tests. See PySpark issue and Python issue for more details.

To run test cases in a specific module:

```
$ python/run-tests --testnames pyspark.sql.tests.test_arrow
```

To run test cases in a specific class:

```
$ python/run-tests --testnames 'pyspark.sql.tests.test_arrow ArrowTests'
```

To run single test case in a specific class:

```
$ python/run-tests --testnames 'pyspark.sql.tests.test_arrow ArrowTests.test_null_
↪conversion'
```

You can also run doctests in a specific module:

```
$ python/run-tests --testnames pyspark.sql.dataframe
```

Lastly, there is another script called run-tests-with-coverage in the same location, which generates coverage report for PySpark tests. It accepts same arguments with run-tests.

```
$ python/run-tests-with-coverage --testnames pyspark.sql.tests.test_arrow --python-
↪executables=python
...
Name                                Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------------
pyspark/__init__.py                    42      4      8      2    84%
pyspark/_globals.py                    16      3      4      2    75%
...
```

Generating HTML files for PySpark coverage under /. . . /spark/python/test_coverage/htmlcov You can check the coverage report visually by HTMLs under /. . . /spark/python/test_coverage/htmlcov.

Please check other available options via `python/run-tests[-with-coverage] --help`.

Setup Pycharm with PySpark

Import a project to PyCharm: File –> Open –> path_to_project.

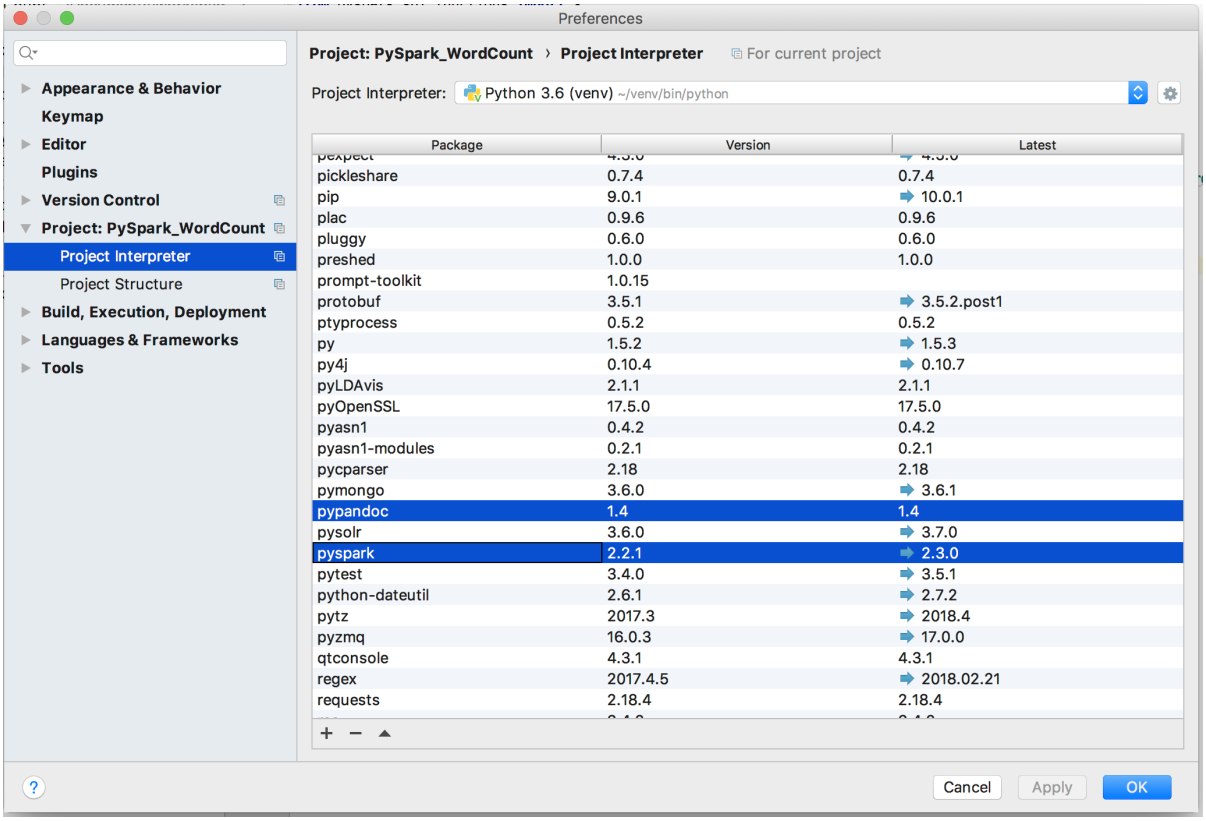
After that, open `~/.bash_profile` and add the below lines.

```
# Spark home
export SPARK_HOME=/.../spark-2.4.5-bin-hadoop2.7
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
export PYTHONPATH=$SPARK_HOME/python/:$PYTHONPATH
export PYTHONPATH=$SPARK_HOME/python/lib/py4j-0.10.4-src.zip:$PYTHONPATH
```

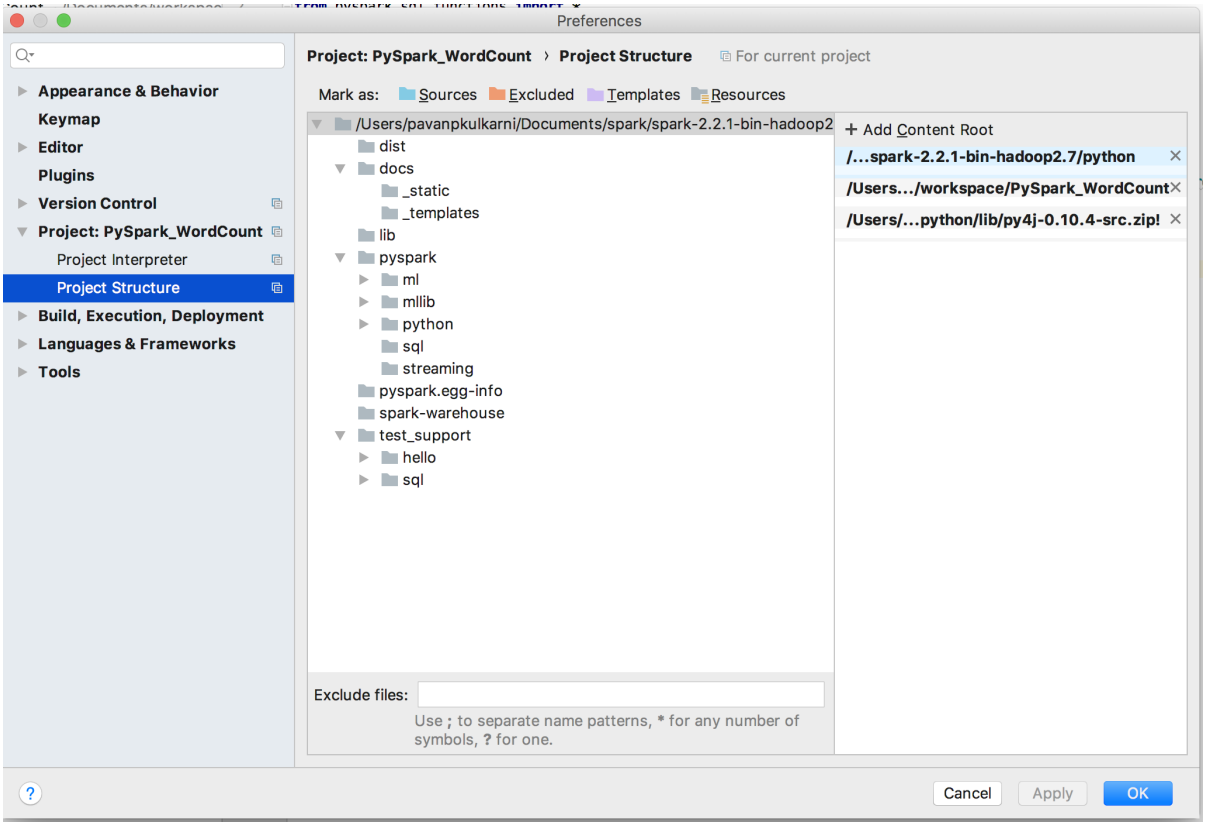
Source the `~/.bash_profile` to reflect the changes.

```
source ~/.bash_profile
```

Install pyspark and pypandoc: PyCharm –> Preferences –> Project Interpreter



Go to PyCharm –> Preferences –> Project Interpreter. Click on Add Content Root. Here you need to add paths



Restart PyCharm, and then run the project. You should be able to see output.

```
+---------+-----+
|     word|count|
+---------+-----+
|      ...|    5|
|     July|    2|
|       By|    1|
|   North,|    1|
|   taking|    1|
|    harry|   18|
|     #TBT|    1|
|  Potter:|    3|
|character|    1|
|        7|    2|
|  Phoenix|    1|
|   Number|    1|
|      day|    1|
|   (Video|    1|
| seconds)|    1|
| Hermione|    3|
|    Which|    1|
|      did|    1|
|   Potter|   38|
|Voldemort|    1|
+---------+-----+
only showing top 20 rows

Process finished with exit code 0
```

1.4.2 Contribution Guide

Style Guide

Filing an Issue

Review Process

1.4.3 Release Process

setup.py

Official Release Channel

PyPi and PIP

1.4.4 Roadmaps

Apache Spark 3.0

Pandas UDF and Pandas Function APIs

Extendability

1.5 Release Notes

1.5.1 Apache Spark 3.0

1.5.2 Apache Spark 2.4.6

1.5.3 Apache Spark 2.4.5

1.5.4 Spark News

Spark 2.4.5 released

We are happy to announce the availability of Spark 2.4.5! Visit the release notes to read about the new features, or download the release today.

Preview release of Spark 3.0

To enable wide-scale community testing of the upcoming Spark 3.0 release, the Apache Spark community has posted a Spark 3.0.0 preview2 release. This preview is not a stable release in terms of either API or functionality, but it is meant to give the community early access to try the code that will become Spark 3.0. If you would like to test the release, please download it, and send feedback using either the mailing lists or JIRA. The documentation is available at the link.

PYTHON MODULE INDEX

p

pyspark.sql.functions.functools, 144
pyspark.sql.functions.sys, 176
pyspark.sql.functions.warnings, 186

INDEX

Symbols

__init__() (*pyspark.sql.Column method*), 23
__init__() (*pyspark.sql.DataFrame method*), 19
__init__() (*pyspark.sql.DataFrameNaFunctions method*), 27
__init__() (*pyspark.sql.DataFrameStatFunctions method*), 27
__init__() (*pyspark.sql.GroupedData method*), 26
__init__() (*pyspark.sql.Row method*), 25
__init__() (*pyspark.sql.SparkSession method*), 17
__init__() (*pyspark.sql.Window method*), 28
__init__() (*pyspark.sql.streaming.DataStreamReader method*), 200
__init__() (*pyspark.sql.streaming.DataStreamWriter method*), 201
__init__() (*pyspark.sql.streaming.ForeachBatchFunction method*), 202
__init__() (*pyspark.sql.streaming.StreamingQuery method*), 202
__init__() (*pyspark.sql.streaming.StreamingQueryManager method*), 203
__init__() (*pyspark.sql.types.ArrayType method*), 97
__init__() (*pyspark.sql.types.BinaryType method*), 98
__init__() (*pyspark.sql.types.BooleanType method*), 99
__init__() (*pyspark.sql.types.ByteType method*), 99
__init__() (*pyspark.sql.types.DataType method*), 100
__init__() (*pyspark.sql.types.DateType method*), 100
__init__() (*pyspark.sql.types.DecimalType method*), 101
__init__() (*pyspark.sql.types.DoubleType method*), 101
__init__() (*pyspark.sql.types.FloatType method*), 102
__init__() (*pyspark.sql.types.IntegerType method*), 102
__init__() (*pyspark.sql.types.LongType method*), 103
__init__() (*pyspark.sql.types.MapType method*), 103
__init__() (*pyspark.sql.types.NullType method*), 104
__init__() (*pyspark.sql.types.Row method*), 105
__init__() (*pyspark.sql.types.ShortType method*), 106
__init__() (*pyspark.sql.types.StringType method*), 106
__init__() (*pyspark.sql.types.StructField method*), 107
__init__() (*pyspark.sql.types.StructType method*), 108
__init__() (*pyspark.sql.types.TimestampType method*), 109

A

abs() (*in module pyspark.sql.functions*), 118
acos() (*in module pyspark.sql.functions*), 119
active() (*pyspark.sql.streaming.StreamingQueryManager property*), 222
add_months() (*in module pyspark.sql.functions*), 119
agg() (*pyspark.sql.DataFrame method*), 57
agg() (*pyspark.sql.GroupedData method*), 194
aggregate() (*in module pyspark.sql.functions*), 119
alias() (*pyspark.sql.DataFrame method*), 57
apply() (*pyspark.sql.GroupedData method*), 195
applyInPandas() (*pyspark.sql.GroupedData method*), 195
appName() (*pyspark.sql.SparkSession.builder method*), 30
approx_count_distinct() (*in module pyspark.sql.functions*), 120
approxCountDistinct() (*in module pyspark.sql.functions*), 120
approxQuantile() (*pyspark.sql.DataFrame method*), 58
approxQuantile() (*pyspark.sql.DataFrameStatFunctions method*), 94
array() (*in module pyspark.sql.functions*), 120
array_contains() (*in module pyspark.sql.functions*), 121
array_distinct() (*in module pyspark.sql.functions*), 121
array_except() (*in module pyspark.sql.functions*), 121

array_intersect() (*in module pyspark.sql.functions*), 122
array_join() (*in module pyspark.sql.functions*), 122
array_max() (*in module pyspark.sql.functions*), 122
array_min() (*in module pyspark.sql.functions*), 122
array_position() (*in module pyspark.sql.functions*), 123
array_remove() (*in module pyspark.sql.functions*), 123
array_repeat() (*in module pyspark.sql.functions*), 123
array_sort() (*in module pyspark.sql.functions*), 124
array_union() (*in module pyspark.sql.functions*), 124
arrays_overlap() (*in module pyspark.sql.functions*), 124
arrays_zip() (*in module pyspark.sql.functions*), 125
ArrayType (*class in pyspark.sql.types*), 97
asc() (*in module pyspark.sql.functions*), 125
asc_nulls_first() (*in module pyspark.sql.functions*), 125
asc_nulls_last() (*in module pyspark.sql.functions*), 125
ascii() (*in module pyspark.sql.functions*), 125
asin() (*in module pyspark.sql.functions*), 126
atan() (*in module pyspark.sql.functions*), 126
atan2() (*in module pyspark.sql.functions*), 126
avg() (*in module pyspark.sql.functions*), 126
avg() (*pyspark.sql.GroupedData method*), 197
awaitAnyTermination() (*pyspark.sql.streaming.StreamingQueryManager method*), 223
awaitTermination() (*pyspark.sql.streaming.StreamingQuery method*), 220

B

base64() (*in module pyspark.sql.functions*), 126
basestring (*in module pyspark.sql.functions*), 127
bin() (*in module pyspark.sql.functions*), 127
BinaryType (*class in pyspark.sql.types*), 98
bitwiseNOT() (*in module pyspark.sql.functions*), 127
blacklist (*in module pyspark.sql.functions*), 127
BooleanType (*class in pyspark.sql.types*), 99
broadcast() (*in module pyspark.sql.functions*), 127
bround() (*in module pyspark.sql.functions*), 127
bucketBy() (*pyspark.sql.DataFrameWriter method*), 46
builder (*pyspark.sql.SparkSession attribute*), 17, 30
ByteType (*class in pyspark.sql.types*), 99

C

cache() (*pyspark.sql.DataFrame method*), 58
catalog() (*pyspark.sql.SparkSession property*), 31
cbrt() (*in module pyspark.sql.functions*), 128
ceil() (*in module pyspark.sql.functions*), 128
checkpoint() (*pyspark.sql.DataFrame method*), 59
coalesce() (*in module pyspark.sql.functions*), 128
coalesce() (*pyspark.sql.DataFrame method*), 59
cogroup() (*pyspark.sql.GroupedData method*), 197
col() (*in module pyspark.sql.functions*), 129
collect() (*pyspark.sql.DataFrame method*), 60
collect_list() (*in module pyspark.sql.functions*), 129
collect_set() (*in module pyspark.sql.functions*), 129
colRegex() (*pyspark.sql.DataFrame method*), 59
Column (*class in pyspark.sql*), 23
column() (*in module pyspark.sql.functions*), 129
columns() (*pyspark.sql.DataFrame property*), 60
concat() (*in module pyspark.sql.functions*), 130
concat_ws() (*in module pyspark.sql.functions*), 130
conf() (*pyspark.sql.SparkSession property*), 31
config() (*pyspark.sql.SparkSession.builder method*), 30
conv() (*in module pyspark.sql.functions*), 130
corr() (*in module pyspark.sql.functions*), 130
corr() (*pyspark.sql.DataFrame method*), 60
corr() (*pyspark.sql.DataFrameStatFunctions method*), 95
cos() (*in module pyspark.sql.functions*), 131
cosh() (*in module pyspark.sql.functions*), 131
count() (*in module pyspark.sql.functions*), 131
count() (*pyspark.sql.DataFrame method*), 60
count() (*pyspark.sql.GroupedData method*), 198
countDistinct() (*in module pyspark.sql.functions*), 131
cov() (*pyspark.sql.DataFrame method*), 61
cov() (*pyspark.sql.DataFrameStatFunctions method*), 95
covar_pop() (*in module pyspark.sql.functions*), 131
covar_samp() (*in module pyspark.sql.functions*), 132
crc32() (*in module pyspark.sql.functions*), 132
create_map() (*in module pyspark.sql.functions*), 132
createDataFrame() (*pyspark.sql.SparkSession method*), 32
createGlobalTempView() (*pyspark.sql.DataFrame method*), 61
createOrReplaceGlobalTempView() (*pyspark.sql.DataFrame method*), 61
createOrReplaceTempView() (*pyspark.sql.DataFrame method*), 62
createTempView() (*pyspark.sql.DataFrame method*), 62
crossJoin() (*pyspark.sql.DataFrame method*), 62
crosstab() (*pyspark.sql.DataFrame method*), 63
crosstab() (*pyspark.sql.DataFrameStatFunctions method*), 96

csv() (*pyspark.sql.DataFrameReader method*), 37
csv() (*pyspark.sql.DataFrameWriter method*), 47
csv() (*pyspark.sql.streaming.DataStreamReader method*), 205
cube() (*pyspark.sql.DataFrame method*), 63
cume_dist() (*in module pyspark.sql.functions*), 132
current_date() (*in module pyspark.sql.functions*), 133
current_timestamp() (*in module pyspark.sql.functions*), 133
currentRow (*pyspark.sql.Window attribute*), 191

D

DataFrame (*class in pyspark.sql*), 19
DataFrameNaFunctions (*class in pyspark.sql*), 27
DataFrameStatFunctions (*class in pyspark.sql*), 27
DataStreamReader (*class in pyspark.sql.streaming*), 200
DataStreamWriter (*class in pyspark.sql.streaming*), 201
DataType (*class in pyspark.sql.types*), 100
date_add() (*in module pyspark.sql.functions*), 133
date_format() (*in module pyspark.sql.functions*), 133
date_sub() (*in module pyspark.sql.functions*), 133
date_trunc() (*in module pyspark.sql.functions*), 134
datediff() (*in module pyspark.sql.functions*), 134
DateType (*class in pyspark.sql.types*), 100
dayofmonth() (*in module pyspark.sql.functions*), 134
dayofweek() (*in module pyspark.sql.functions*), 134
dayofyear() (*in module pyspark.sql.functions*), 135
DecimalType (*class in pyspark.sql.types*), 101
decode() (*in module pyspark.sql.functions*), 135
degrees() (*in module pyspark.sql.functions*), 135
dense_rank() (*in module pyspark.sql.functions*), 135
desc() (*in module pyspark.sql.functions*), 135
desc_nulls_first() (*in module pyspark.sql.functions*), 136
desc_nulls_last() (*in module pyspark.sql.functions*), 136
describe() (*pyspark.sql.DataFrame method*), 63
distinct() (*pyspark.sql.DataFrame method*), 64
DoubleType (*class in pyspark.sql.types*), 101
drop() (*pyspark.sql.DataFrame method*), 64
drop() (*pyspark.sql.DataFrameNaFunctions method*), 92
drop_duplicates() (*pyspark.sql.DataFrame method*), 65
dropDuplicates() (*pyspark.sql.DataFrame method*), 65
dropna() (*pyspark.sql.DataFrame method*), 66
dtypes() (*pyspark.sql.DataFrame property*), 66

E

element_at() (*in module pyspark.sql.functions*), 136
enableHiveSupport() (*pyspark.sql.SparkSession.builder method*), 30
encode() (*in module pyspark.sql.functions*), 136
exceptAll() (*pyspark.sql.DataFrame method*), 66
exception() (*pyspark.sql.streaming.StreamingQuery method*), 220
exists() (*in module pyspark.sql.functions*), 137
exp() (*in module pyspark.sql.functions*), 137
explain() (*pyspark.sql.DataFrame method*), 67
explain() (*pyspark.sql.streaming.StreamingQuery method*), 220
explode() (*in module pyspark.sql.functions*), 137
explode_outer() (*in module pyspark.sql.functions*), 138
expm1() (*in module pyspark.sql.functions*), 138
expr() (*in module pyspark.sql.functions*), 138

F

factorial() (*in module pyspark.sql.functions*), 139
fill() (*pyspark.sql.DataFrameNaFunctions method*), 92
fillna() (*pyspark.sql.DataFrame method*), 68
filter() (*in module pyspark.sql.functions*), 139
filter() (*pyspark.sql.DataFrame method*), 69
first() (*in module pyspark.sql.functions*), 140
first() (*pyspark.sql.DataFrame method*), 69
flatten() (*in module pyspark.sql.functions*), 140
FloatType (*class in pyspark.sql.types*), 102
floor() (*in module pyspark.sql.functions*), 140
forall() (*in module pyspark.sql.functions*), 140
foreach() (*pyspark.sql.DataFrame method*), 69
foreach() (*pyspark.sql.streaming.DataStreamWriter method*), 213
foreachBatch() (*pyspark.sql.streaming.DataStreamWriter method*), 215
ForeachBatchFunction (*class in pyspark.sql.streaming*), 202
foreachPartition() (*pyspark.sql.DataFrame method*), 69
format() (*pyspark.sql.DataFrameReader method*), 40
format() (*pyspark.sql.DataFrameWriter method*), 48
format() (*pyspark.sql.streaming.DataStreamReader method*), 207
format() (*pyspark.sql.streaming.DataStreamWriter method*), 215
format_number() (*in module pyspark.sql.functions*), 141
format_string() (*in module pyspark.sql.functions*), 141
freqItems() (*pyspark.sql.DataFrame method*), 70

freqItems() (*pyspark.sql.DataFrameStatFunctions method*), 96
from_avro() (*in module pyspark.sql.avro.functions*), 189
from_csv() (*in module pyspark.sql.functions*), 142
from_json() (*in module pyspark.sql.functions*), 142
from_unixtime() (*in module pyspark.sql.functions*), 143
from_utc_timestamp() (*in module pyspark.sql.functions*), 143

G

get() (*pyspark.sql.streaming.StreamingQueryManager method*), 223
get_json_object() (*in module pyspark.sql.functions*), 144
getActiveSession() (*pyspark.sql.SparkSession class method*), 33
getOrCreate() (*pyspark.sql.SparkSession.builder method*), 31
greatest() (*in module pyspark.sql.functions*), 145
groupBy() (*pyspark.sql.DataFrame method*), 70
groupby() (*pyspark.sql.DataFrame method*), 71
GroupedData (*class in pyspark.sql*), 26
grouping() (*in module pyspark.sql.functions*), 145
grouping_id() (*in module pyspark.sql.functions*), 145

H

hash() (*in module pyspark.sql.functions*), 146
head() (*pyspark.sql.DataFrame method*), 71
hex() (*in module pyspark.sql.functions*), 146
hint() (*pyspark.sql.DataFrame method*), 71
hour() (*in module pyspark.sql.functions*), 146
hypot() (*in module pyspark.sql.functions*), 146

I

id() (*pyspark.sql.streaming.StreamingQuery property*), 221
ignore_unicode_prefix() (*in module pyspark.sql.functions*), 146
initcap() (*in module pyspark.sql.functions*), 147
input_file_name() (*in module pyspark.sql.functions*), 147
inputFiles() (*pyspark.sql.DataFrame method*), 72
insertInto() (*pyspark.sql.DataFrameWriter method*), 48
instr() (*in module pyspark.sql.functions*), 147
IntegerType (*class in pyspark.sql.types*), 102
intersect() (*pyspark.sql.DataFrame method*), 72
intersectAll() (*pyspark.sql.DataFrame method*), 72
isActive() (*pyspark.sql.streaming.StreamingQuery property*), 221
isLocal() (*pyspark.sql.DataFrame method*), 72
isnan() (*in module pyspark.sql.functions*), 147
isnull() (*in module pyspark.sql.functions*), 148
isStreaming() (*pyspark.sql.DataFrame property*), 73

J

jdbc() (*pyspark.sql.DataFrameReader method*), 40
jdbc() (*pyspark.sql.DataFrameWriter method*), 48
join() (*pyspark.sql.DataFrame method*), 73
json() (*pyspark.sql.DataFrameReader method*), 41
json() (*pyspark.sql.DataFrameWriter method*), 49
json() (*pyspark.sql.streaming.DataStreamReader method*), 208
json_tuple() (*in module pyspark.sql.functions*), 148

K

kurtosis() (*in module pyspark.sql.functions*), 148

L

lag() (*in module pyspark.sql.functions*), 148
last() (*in module pyspark.sql.functions*), 149
last_day() (*in module pyspark.sql.functions*), 149
lastProgress() (*pyspark.sql.streaming.StreamingQuery property*), 221
lead() (*in module pyspark.sql.functions*), 149
least() (*in module pyspark.sql.functions*), 149
length() (*in module pyspark.sql.functions*), 150
levenshtein() (*in module pyspark.sql.functions*), 150
limit() (*pyspark.sql.DataFrame method*), 74
lit() (*in module pyspark.sql.functions*), 150
load() (*pyspark.sql.DataFrameReader method*), 43
load() (*pyspark.sql.streaming.DataStreamReader method*), 210
localCheckpoint() (*pyspark.sql.DataFrame method*), 74
locate() (*in module pyspark.sql.functions*), 150
log() (*in module pyspark.sql.functions*), 151
log10() (*in module pyspark.sql.functions*), 151
log1p() (*in module pyspark.sql.functions*), 151
log2() (*in module pyspark.sql.functions*), 151
LongType (*class in pyspark.sql.types*), 103
lower() (*in module pyspark.sql.functions*), 151
lpad() (*in module pyspark.sql.functions*), 152
ltrim() (*in module pyspark.sql.functions*), 152

M

map_concat() (*in module pyspark.sql.functions*), 152
map_entries() (*in module pyspark.sql.functions*), 152
map_filter() (*in module pyspark.sql.functions*), 153

map_from_arrays() (*in module pyspark.sql.functions*), 153
map_from_entries() (*in module pyspark.sql.functions*), 154
map_keys() (*in module pyspark.sql.functions*), 154
map_values() (*in module pyspark.sql.functions*), 154
map_zip_with() (*in module pyspark.sql.functions*), 155
mapInPandas() (*pyspark.sql.DataFrame method*), 74
MapType (*class in pyspark.sql.types*), 103
master() (*pyspark.sql.SparkSession.builder method*), 31
max() (*in module pyspark.sql.functions*), 155
max() (*pyspark.sql.GroupedData method*), 198
md5() (*in module pyspark.sql.functions*), 155
mean() (*in module pyspark.sql.functions*), 156
mean() (*pyspark.sql.GroupedData method*), 198
min() (*in module pyspark.sql.functions*), 156
min() (*pyspark.sql.GroupedData method*), 198
minute() (*in module pyspark.sql.functions*), 156
mode() (*pyspark.sql.DataFrameWriter method*), 50
module
    pyspark.sql.functions.functools, 144
    pyspark.sql.functions.sys, 176
    pyspark.sql.functions.warnings, 186
monotonically_increasing_id() (*in module pyspark.sql.functions*), 156
month() (*in module pyspark.sql.functions*), 157
months_between() (*in module pyspark.sql.functions*), 157

N

na() (*pyspark.sql.DataFrame property*), 75
name() (*pyspark.sql.streaming.StreamingQuery property*), 221
nanvl() (*in module pyspark.sql.functions*), 157
newSession() (*pyspark.sql.SparkSession method*), 34
next_day() (*in module pyspark.sql.functions*), 157
ntile() (*in module pyspark.sql.functions*), 158
NullType (*class in pyspark.sql.types*), 104

O

option() (*pyspark.sql.DataFrameReader method*), 43
option() (*pyspark.sql.DataFrameWriter method*), 50
option() (*pyspark.sql.streaming.DataStreamReader method*), 210
option() (*pyspark.sql.streaming.DataStreamWriter method*), 215
options() (*pyspark.sql.DataFrameReader method*), 44
options() (*pyspark.sql.DataFrameWriter method*), 50
options() (*pyspark.sql.streaming.DataStreamReader method*), 211
options() (*pyspark.sql.streaming.DataStreamWriter method*), 216
orc() (*pyspark.sql.DataFrameReader method*), 44
orc() (*pyspark.sql.DataFrameWriter method*), 51
orc() (*pyspark.sql.streaming.DataStreamReader method*), 211
orderBy() (*pyspark.sql.DataFrame method*), 75
orderBy() (*pyspark.sql.Window static method*), 191
outputMode() (*pyspark.sql.streaming.DataStreamWriter method*), 216
overlay() (*in module pyspark.sql.functions*), 158

P

pandas_udf() (*in module pyspark.sql.functions*), 158
parquet() (*pyspark.sql.DataFrameReader method*), 45
parquet() (*pyspark.sql.DataFrameWriter method*), 51
parquet() (*pyspark.sql.streaming.DataStreamReader method*), 212
partitionBy() (*pyspark.sql.DataFrameWriter method*), 52
partitionBy() (*pyspark.sql.streaming.DataStreamWriter method*), 217
partitionBy() (*pyspark.sql.Window static method*), 191
percent_rank() (*in module pyspark.sql.functions*), 163
percentile_approx() (*in module pyspark.sql.functions*), 163
persist() (*pyspark.sql.DataFrame method*), 76
pivot() (*pyspark.sql.GroupedData method*), 199
posexplode() (*in module pyspark.sql.functions*), 164
posexplode_outer() (*in module pyspark.sql.functions*), 164
pow() (*in module pyspark.sql.functions*), 165
printSchema() (*pyspark.sql.DataFrame method*), 76
processAllAvailable() (*pyspark.sql.streaming.StreamingQuery method*), 221
pyspark.sql.functions.functools
    module, 144
pyspark.sql.functions.sys
    module, 176
pyspark.sql.functions.warnings
    module, 186

Q

quarter() (*in module pyspark.sql.functions*), 165
queryName() (*pyspark.sql.streaming.DataStreamWriter method*), 217

R

radians() (*in module pyspark.sql.functions*), 165
rand() (*in module pyspark.sql.functions*), 165
randn() (*in module pyspark.sql.functions*), 166
randomSplit() (*pyspark.sql.DataFrame method*), 76
range() (*pyspark.sql.SparkSession method*), 34
rangeBetween() (*pyspark.sql.Window static method*), 191
rank() (*in module pyspark.sql.functions*), 166
rdd() (*pyspark.sql.DataFrame property*), 77
read() (*pyspark.sql.SparkSession property*), 34
readStream() (*pyspark.sql.SparkSession property*), 34
recentProgress() (*pyspark.sql.streaming.StreamingQuery property*), 222
regexp_extract() (*in module pyspark.sql.functions*), 166
regexp_replace() (*in module pyspark.sql.functions*), 167
registerTempTable() (*pyspark.sql.DataFrame method*), 77
repartition() (*pyspark.sql.DataFrame method*), 77
repartitionByRange() (*pyspark.sql.DataFrame method*), 78
repeat() (*in module pyspark.sql.functions*), 167
replace() (*pyspark.sql.DataFrame method*), 79
replace() (*pyspark.sql.DataFrameNaFunctions method*), 93
resetTerminated() (*pyspark.sql.streaming.StreamingQueryManager method*), 223
reverse() (*in module pyspark.sql.functions*), 167
rint() (*in module pyspark.sql.functions*), 167
rollup() (*pyspark.sql.DataFrame method*), 80
round() (*in module pyspark.sql.functions*), 168
Row (*class in pyspark.sql*), 25
Row (*class in pyspark.sql.types*), 105
row_number() (*in module pyspark.sql.functions*), 168
rowsBetween() (*pyspark.sql.Window static method*), 192
rpad() (*in module pyspark.sql.functions*), 168
rtrim() (*in module pyspark.sql.functions*), 168
runId() (*pyspark.sql.streaming.StreamingQuery property*), 222

S

sameSemantics() (*pyspark.sql.DataFrame method*), 80
sample() (*pyspark.sql.DataFrame method*), 81
sampleBy() (*pyspark.sql.DataFrame method*), 82
sampleBy() (*pyspark.sql.DataFrameStatFunctions method*), 96
save() (*pyspark.sql.DataFrameWriter method*), 52
saveAsTable() (*pyspark.sql.DataFrameWriter method*), 52
schema() (*pyspark.sql.DataFrame property*), 82
schema() (*pyspark.sql.DataFrameReader method*), 45
schema() (*pyspark.sql.streaming.DataStreamReader method*), 212
schema_of_csv() (*in module pyspark.sql.functions*), 168
schema_of_json() (*in module pyspark.sql.functions*), 169
second() (*in module pyspark.sql.functions*), 169
select() (*pyspark.sql.DataFrame method*), 82
selectExpr() (*pyspark.sql.DataFrame method*), 83
semanticHash() (*pyspark.sql.DataFrame method*), 83
sequence() (*in module pyspark.sql.functions*), 169
sha1() (*in module pyspark.sql.functions*), 170
sha2() (*in module pyspark.sql.functions*), 170
shiftLeft() (*in module pyspark.sql.functions*), 170
shiftRight() (*in module pyspark.sql.functions*), 170
shiftRightUnsigned() (*in module pyspark.sql.functions*), 171
ShortType (*class in pyspark.sql.types*), 106
show() (*pyspark.sql.DataFrame method*), 83
shuffle() (*in module pyspark.sql.functions*), 171
signum() (*in module pyspark.sql.functions*), 171
sin() (*in module pyspark.sql.functions*), 171
since() (*in module pyspark.sql.functions*), 172
sinh() (*in module pyspark.sql.functions*), 172
size() (*in module pyspark.sql.functions*), 172
skewness() (*in module pyspark.sql.functions*), 172
slice() (*in module pyspark.sql.functions*), 172
sort() (*pyspark.sql.DataFrame method*), 84
sort_array() (*in module pyspark.sql.functions*), 173
sortBy() (*pyspark.sql.DataFrameWriter method*), 53
sortWithinPartitions() (*pyspark.sql.DataFrame method*), 85
soundex() (*in module pyspark.sql.functions*), 173
spark_partition_id() (*in module pyspark.sql.functions*), 173
sparkContext() (*pyspark.sql.SparkSession property*), 35
SparkSession (*class in pyspark.sql*), 17
split() (*in module pyspark.sql.functions*), 173
sql() (*pyspark.sql.SparkSession method*), 35
sqrt() (*in module pyspark.sql.functions*), 174
start() (*pyspark.sql.streaming.DataStreamWriter method*), 217
stat() (*pyspark.sql.DataFrame property*), 85
status() (*pyspark.sql.streaming.StreamingQuery property*), 222
stddev() (*in module pyspark.sql.functions*), 174
stddev_pop() (*in module pyspark.sql.functions*), 174

stddev_samp() (*in module pyspark.sql.functions*), 174
stop() (*pyspark.sql.SparkSession method*), 35
stop() (*pyspark.sql.streaming.StreamingQuery method*), 222
storageLevel() (*pyspark.sql.DataFrame property*), 85
StreamingQuery (*class in pyspark.sql.streaming*), 202
StreamingQueryException, 203
StreamingQueryManager (*class in pyspark.sql.streaming*), 203
streams() (*pyspark.sql.SparkSession property*), 35
StringType (*class in pyspark.sql.types*), 106
struct() (*in module pyspark.sql.functions*), 175
StructField (*class in pyspark.sql.types*), 107
StructType (*class in pyspark.sql.types*), 108
substring() (*in module pyspark.sql.functions*), 175
substring_index() (*in module pyspark.sql.functions*), 175
subtract() (*pyspark.sql.DataFrame method*), 85
sum() (*in module pyspark.sql.functions*), 176
sum() (*pyspark.sql.GroupedData method*), 199
sumDistinct() (*in module pyspark.sql.functions*), 176
summary() (*pyspark.sql.DataFrame method*), 86

T

table() (*pyspark.sql.DataFrameReader method*), 45
table() (*pyspark.sql.SparkSession method*), 35
tail() (*pyspark.sql.DataFrame method*), 87
take() (*pyspark.sql.DataFrame method*), 87
tan() (*in module pyspark.sql.functions*), 178
tanh() (*in module pyspark.sql.functions*), 178
text() (*pyspark.sql.DataFrameReader method*), 46
text() (*pyspark.sql.DataFrameWriter method*), 53
text() (*pyspark.sql.streaming.DataStreamReader method*), 213
TimestampType (*class in pyspark.sql.types*), 109
to_avro() (*in module pyspark.sql.avro.functions*), 190
to_csv() (*in module pyspark.sql.functions*), 179
to_date() (*in module pyspark.sql.functions*), 179
to_json() (*in module pyspark.sql.functions*), 180
to_str() (*in module pyspark.sql.functions*), 180
to_timestamp() (*in module pyspark.sql.functions*), 181
to_utc_timestamp() (*in module pyspark.sql.functions*), 181
toDegrees() (*in module pyspark.sql.functions*), 179
toDF() (*pyspark.sql.DataFrame method*), 87
toJSON() (*pyspark.sql.DataFrame method*), 87
toLocalIterator() (*pyspark.sql.DataFrame method*), 88
toPandas() (*pyspark.sql.DataFrame method*), 88
toRadians() (*in module pyspark.sql.functions*), 179
transform() (*in module pyspark.sql.functions*), 182
transform() (*pyspark.sql.DataFrame method*), 88
transform_keys() (*in module pyspark.sql.functions*), 182
transform_values() (*in module pyspark.sql.functions*), 183
translate() (*in module pyspark.sql.functions*), 183
trigger() (*pyspark.sql.streaming.DataStreamWriter method*), 218
trim() (*in module pyspark.sql.functions*), 184
trunc() (*in module pyspark.sql.functions*), 184

U

udf() (*in module pyspark.sql.functions*), 184
udf() (*pyspark.sql.SparkSession property*), 36
unbase64() (*in module pyspark.sql.functions*), 185
unboundedFollowing (*pyspark.sql.Window attribute*), 193
unboundedPreceding (*pyspark.sql.Window attribute*), 193
unhex() (*in module pyspark.sql.functions*), 185
union() (*pyspark.sql.DataFrame method*), 89
unionAll() (*pyspark.sql.DataFrame method*), 89
unionByName() (*pyspark.sql.DataFrame method*), 89
unix_timestamp() (*in module pyspark.sql.functions*), 185
unpersist() (*pyspark.sql.DataFrame method*), 90
upper() (*in module pyspark.sql.functions*), 186

V

var_pop() (*in module pyspark.sql.functions*), 186
var_samp() (*in module pyspark.sql.functions*), 186
variance() (*in module pyspark.sql.functions*), 186
version() (*pyspark.sql.SparkSession property*), 36

W

weekofyear() (*in module pyspark.sql.functions*), 187
when() (*in module pyspark.sql.functions*), 187
where() (*pyspark.sql.DataFrame method*), 90
Window (*class in pyspark.sql*), 28
window() (*in module pyspark.sql.functions*), 187
withColumn() (*pyspark.sql.DataFrame method*), 90
withColumnRenamed() (*pyspark.sql.DataFrame method*), 90
withWatermark() (*pyspark.sql.DataFrame method*), 91
write() (*pyspark.sql.DataFrame property*), 91
writeStream() (*pyspark.sql.DataFrame property*), 92

X

xxhash64() (*in module pyspark.sql.functions*), 188

Y

year() (*in module pyspark.sql.functions*), 188

Z

zip_with() (*in module pyspark.sql.functions*), 188