pyspark.sql module

Module Context

Important classes of Spark SQL and DataFrames:

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

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

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()
```

class **Builder**

Builder for **SparkSession**.

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.

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.

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.

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") == s1.sparkContext.getConf().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.

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.

SparkSession.**builder** = *<pyspark.sql.session.Builder object at 0x7f8396ea2cd0>*

SparkSession.**catalog**

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

New in version 2.0.

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.

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

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

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

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

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

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

Parameters:
- **data** – an RDD of any kind of SQL data representation(e.g. row, tuple, int, boolean, etc.), or **list**, or **pandas.DataFrame**.
- **schema** – a **DataType** or a datatype string or a list of column names, default is None. The data type string format equals to *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 ByteType. We can also use *int* as a short name for IntegerType.
- **samplingRatio** – the sample ratio of rows used for inferring

Returns: **DataFrame**

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

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

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

```
[Row(age=1, name=u'Alice')]
```

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

```
>>> spark.createDataFrame(df.toPandas()).collect()
[Row(name=u'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=u'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.

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.

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

Create a **DataFrame** with single 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.

SparkSession.**read**

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

Returns: **DataFrameReader**

New in version 2.0.

SparkSession.**readStream**

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

> **Note:** Experimental.

Returns: **DataStreamReader**

New in version 2.0.

SparkSession.**sparkContext**

Returns the underlying **SparkContext**.

New in version 2.0.

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=u'row1'), Row(f1=2, f2=u'row2'), Row(f1=3, f2=u'row3')]
```

New in version 2.0.

SparkSession.**stop**()

Stop the underlying **SparkContext**.

New in version 2.0.

SparkSession.**streams**

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

> **Note:** Experimental.

Returns: **StreamingQueryManager**

New in version 2.0.

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.

SparkSession.**udf**

Returns a **UDFRegistration** for UDF registration.

Returns: **UDFRegistration**

New in version 2.0.

SparkSession.**version**

The version of Spark on which this application is running.

New in version 2.0.

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

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

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

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

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

cacheTable(*tableName*)

Caches the specified table in-memory.

New in version 1.0.

clearCache()

Removes all cached tables from the in-memory cache.

New in version 1.3.

createDataFrame(*data*, *schema=None*, *samplingRatio=None*)

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

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

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

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

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

Parameters:
- **data** – an RDD of any kind of SQL data representation(e.g. row, tuple, int, boolean, etc.), or **list**, or **pandas.DataFrame**.
- **schema** – a **DataType** or a datatype string or a list of column names, default is None. The data type string format equals to *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 ByteType. We can also use *int* as a short name for IntegerType.
- **samplingRatio** – the sample ratio of rows used for inferring

Returns: **DataFrame**

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

```
>>> l = [('Alice', 1)]
```

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

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

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

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

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

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

```
>>> sqlContext.createDataFrame(rdd, "a: string, b: int").collect()
```

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

New in version 1.3.

createExternalTable(*tableName*, *path=None*, *source=None*, *schema=None*, ***options*)

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

It returns the DataFrame associated with the external table.

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

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

Returns: **DataFrame**

New in version 1.3.

dropTempTable(*tableName*)

Remove the temp table from catalog.

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

New in version 1.6.

getConf(*key*, *defaultValue=None*)

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

If the key is not set and defaultValue is not None, return defaultValue. If the key is not set and defaultValue is None, return the system default

value.

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

New in version 1.3.

classmethod **getOrCreate**(*sc*)

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

Parameters: **sc** – SparkContext

New in version 1.6.

newSession()

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

New in version 1.6.

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

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

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

Returns: **DataFrame**

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

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

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

New in version 1.4.

read

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

Returns: **DataFrameReader**

New in version 1.4.

readStream

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

Note: Experimental.

Returns: **DataStreamReader**

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

New in version 2.0.

registerDataFrameAsTable(*df*, *tableName*)

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

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

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

New in version 1.3.

registerFunction(*name*, *f*, *returnType=StringType*)

Registers a python function (including lambda function) as a UDF so it can be used in SQL statements.

In addition to a name and the function itself, the return type can be optionally specified. When the return type is not given it default to a string and conversion will automatically be done. For any other return type, the produced object must match the specified type.

Parameters:
- **name** – name of the UDF
- **f** – python function
- **returnType** – a **DataType** object

```
>>> sqlContext.registerFunction("stringLengthString", lambda x: len(x))
>>> sqlContext.sql("SELECT stringLengthString('test')").collect()
[Row(stringLengthString(test)=u'4')]
```

```
>>> from pyspark.sql.types import IntegerType
>>> sqlContext.registerFunction("stringLengthInt", lambda x: len(x), IntegerType())
>>> sqlContext.sql("SELECT stringLengthInt('test')").collect()
[Row(stringLengthInt(test)=4)]
```

```
>>> from pyspark.sql.types import IntegerType
>>> sqlContext.udf.register("stringLengthInt", lambda x: len(x), IntegerType())
>>> sqlContext.sql("SELECT stringLengthInt('test')").collect()
[Row(stringLengthInt(test)=4)]
```

New in version 1.2.

setConf(*key*, *value*)

Sets the given Spark SQL configuration property.

New in version 1.3.

sql(*sqlQuery*)

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

Returns: `DataFrame`

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

New in version 1.0.

streams

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

Note: Experimental.

New in version 2.0.

table(*tableName*)

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

Returns: `DataFrame`

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

New in version 1.0.

tableNames(*dbName=None*)

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

Parameters: **dbName** – string, name of the database to use. Default to the current database.

Returns: list of table names, in string

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

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

New in version 1.3.

tables(*dbName=None*)

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

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

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

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

Returns: **DataFrame**

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

New in version 1.3.

udf

Returns a **UDFRegistration** for UDF registration.

Returns: **UDFRegistration**

New in version 1.3.1.

uncacheTable(*tableName*)

Removes the specified table from the in-memory cache.

New in version 1.0.

class `pyspark.sql.`**HiveContext**(*sparkContext*, *jhiveContext=None*)

A variant of Spark SQL that integrates with data stored in Hive.

Configuration for Hive is read from `hive-site.xml` on the classpath. It supports running both SQL and HiveQL commands.

Parameters:
- **sparkContext** – The SparkContext to wrap.
- **jhiveContext** – An optional JVM Scala HiveContext. If set, we do not instantiate a new **`HiveContext`** in the JVM, instead we make all calls to this object.

> **Note:** Deprecated in 2.0.0. Use SparkSession.builder.enableHiveSupport().getOrCreate().

`refreshTable`(*tableName*)

Invalidate and refresh all the cached the metadata of the given table. For performance reasons, Spark SQL or the external data source library it uses might cache certain metadata about a table, such as the location of blocks. When those change outside of Spark SQL, users should call this function to invalidate the cache.

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 **`SQLContext`**:

```
people = sqlContext.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 data frame, use the apply method:

```
ageCol = people.age
```

A more concrete example:

```
# To create DataFrame using SQLContext
people = sqlContext.read.parquet("...")
department = sqlContext.read.parquet("...")

people.filter(people.age > 30).join(department, people.deptId == department.id)          .groupBy(department.name, "gender").agg({"sal
```

New in version 1.3.

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.

alias(*alias*)

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

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

New in version 1.3.

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

Calculates the approximate quantiles of a numerical column 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

[[http://dx.doi.org/10.1145/375663.375670 Space-efficient Online Computation of Quantile Summaries]] by Greenwald and Khanna.

Parameters:
- **col** – the name of the numerical column
- **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

New in version 2.0.

`cache`()

Persists with the default storage level (`MEMORY_ONLY`).

New in version 1.3.

`coalesce`(*numPartitions*)

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

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

New in version 1.4.

`collect`()

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

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

New in version 1.3.

columns

Returns all column names as a list.

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

New in version 1.3.

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.

count()

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

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

New in version 1.3.

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.

createOrReplaceTempView(*name*)

Creates or replaces a temporary view with this DataFrame.

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

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

New in version 2.0.

createTempView(*name*)

Creates a 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.

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.

`cube`(**cols*)

Create a multi-dimensional cube for the current `DataFrame` using the specified columns, so we can run aggregation 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.

`describe`(**cols*)

Computes statistics for numeric columns.

This include count, mean, stddev, min, and max. If no columns are given, this function computes statistics for all numerical 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().show()
+-------+------------------+
|summary|               age|
+-------+------------------+
|  count|                 2|
```

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

New in version 1.3.1.

`distinct`()

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

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

New in version 1.3.

`drop`(*col*)

Returns a new `DataFrame` that drops the specified column.

Parameters: **col** – a string name of the column to drop, or a `Column` to drop.

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

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

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

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

New in version 1.4.

dropDuplicates(*subset=None*)

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

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

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

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

New in version 1.4.

drop_duplicates(*subset=None*)

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

New in version 1.4.

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

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.

dtypes

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

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

New in version 1.3.

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.

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

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

New in version 1.3.

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, 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, 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|
+---+------+-----+
```

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

```
| 50|  null|unknown|
+---+------+-------+
```

New in version 1.3.1.

filter(*condition*)

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

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

New in version 1.3.

first()

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

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

New in version 1.3.

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.

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.

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

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

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

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

New in version 1.4.

groupBy(**cols*)

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=u'Alice', avg(age)=2.0), Row(name=u'Bob', avg(age)=5.0)]
>>> sorted(df.groupBy(df.name).avg().collect())
[Row(name=u'Alice', avg(age)=2.0), Row(name=u'Bob', avg(age)=5.0)]
>>> sorted(df.groupBy(['name', df.age]).count().collect())
[Row(name=u'Alice', age=2, count=1), Row(name=u'Bob', age=5, count=1)]
```

New in version 1.3.

groupby(**cols*)

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

New in version 1.4.

head(*n=None*)

Returns the first `n` rows.

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

New in version 1.3.

intersect(*other*)

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

This is equivalent to *INTERSECT* in SQL.

New in version 1.3.

isLocal()

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

New in version 1.3.

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:** Experimental

New in version 2.0.

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'. One of *inner*, *outer*, *left_outer*, *right_outer*, *leftsemi*.

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

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

```
>>> df.join(df2, 'name', 'outer').select('name', 'height').collect()
```

```
[Row(name=u'Tom', height=80), Row(name=u'Bob', height=85), Row(name=u'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=u'Alice', age=2), Row(name=u'Bob', age=5)]
```

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

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

New in version 1.3.

limit(*num*)

Limits the result count to the number specified.

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

New in version 1.3.

na

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

New in version 1.3.1.

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

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

Parameters:
- **cols** – list of **Column** or column names to sort by.
- **ascending** – boolean or list of boolean (default True). Sort ascending vs. descending. Specify list for multiple sort orders. If a

list is specified, length of the list must equal length of the *cols*.

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

New in version 1.3.

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

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

New in version 1.3.

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.

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()
1
```

```
>>> splits[1].count()
3
```

New in version 1.4.

rdd

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

New in version 1.3.

registerTempTable(*name*)

Registers this RDD as a temporary table using the given name.

The lifetime of this temporary table is tied to the **SQLContext** 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.

repartition(*numPartitions*, **cols*)

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

`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|
+---+-----+
|  5|  Bob|
|  5|  Bob|
|  2|Alice|
|  2|Alice|
+---+-----+
>>> 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.

replace(*to_replace*, *value*, *subset=None*)

Returns a new **DataFrame** replacing a value with another value. **DataFrame.replace()** and **DataFrameNaFunctions.replace()** are aliases of each other.

Parameters:
- **to_replace** – int, long, float, string, or list. Value to be replaced. If the value is a dict, then *value* is ignored and *to_replace* must be a mapping from column name (string) to replacement value. The value to be replaced must be an int, long, float, or string.
- **value** – int, long, float, string, or list. Value to use to replace holes. The replacement value must be an int, long, float, or string. If *value* is a list or tuple, *value* should be of the same length with *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', 'Bob'], ['A', 'B'], 'name').show()
+----+------+----+
| age|height|name|
+----+------+----+
|  10|    80|   A|
|   5|  null|   B|
|null|  null| Tom|
|null|  null|null|
+----+------+----+
```

New in version 1.4.

rollup(**cols*)

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.

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

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

```
>>> df.sample(False, 0.5, 42).count()
2
```

New in version 1.3.

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|    5|
|  1|    9|
```

```
+---+-----+
```

New in version 1.5.

schema

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

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

New in version 1.3.

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

New in version 1.3.

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.

show(*n=20*, *truncate=True*)

Prints the first `n` rows to the console.

Parameters:
- **n** – Number of rows to show.
- **truncate** – Whether truncate long strings and align cells right.

```
>>> df
DataFrame[age: int, name: string]
>>> df.show()
+---+-----+
|age| name|
+---+-----+
|  2|Alice|
|  5|  Bob|
+---+-----+
```

New in version 1.3.

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

```
[Row(age=5, name=u'Bob'), Row(age=2, name=u'Alice')]
```

New in version 1.3.

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.

stat

Returns a **DataFrameStatFunctions** for statistic functions.

New in version 1.4.

subtract(*other*)

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

This is equivalent to *EXCEPT* in SQL.

New in version 1.3.

take(*num*)

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

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

New in version 1.3.

toDF(**cols*)

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

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

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

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()
u'{"age":2,"name":"Alice"}'
```

New in version 1.3.

toLocalIterator()

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.

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

New in version 2.0.

toPandas()

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

Note that 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.

This is only available if Pandas is installed and available.

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

New in version 1.3.

union(*other*)

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

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 a distinct.

New in version 2.0.

unionAll(*other*)

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

> **Note:** Deprecated in 2.0, use union instead.

New in version 1.3.

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.

where(*condition*)

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

New in version 1.3.

withColumn(*colName*, *col*)

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

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

```
>>> df.withColumn('age2', df.age + 2).collect()
[Row(age=2, name=u'Alice', age2=4), Row(age=5, name=u'Bob', age2=7)]
```

New in version 1.3.

withColumnRenamed(*existing*, *new*)

Returns a new **DataFrame** by renaming an existing column.

Parameters:
- **existing** – string, name of the existing column to rename.
- **col** – string, new name of the column.

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

New in version 1.3.

write

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

Returns: **DataFrameWriter**

New in version 1.4.

writeStream

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

> **Note:** Experimental.

Returns: **DataStreamWriter**

New in version 2.0.

class pyspark.sql.**GroupedData**(*jgd*, *sql_ctx*)

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

> **Note:** Experimental

New in version 1.3.

agg(**exprs*)

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

The available aggregate functions are *avg*, *max*, *min*, *sum*, *count*.

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.

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=u'Alice', count(1)=1), Row(name=u'Bob', count(1)=1)]
```

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

New in version 1.3.

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.

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.

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.

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.

`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.

`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)]
```

New in version 1.6.

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.

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.

alias(**alias*)

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

```
>>> df.select(df.age.alias("age2")).collect()
[Row(age2=2), Row(age2=5)]
```

New in version 1.3.

asc()

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

astype(*dataType*)

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

New in version 1.4.

between(*lowerBound*, *upperBound*)

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

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

New in version 1.3.

bitwiseAND(*other*)

binary operator

bitwiseOR(*other*)

binary operator

bitwiseXOR(*other*)

binary operator

cast(*dataType*)

Convert the column into type `dataType`.

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

New in version 1.3.

desc()

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

endswith(*other*)

binary operator

getField(*name*)

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

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

New in version 1.3.

getItem(*key*)

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

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

New in version 1.3.

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.

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

New in version 1.5.

like(*other*)

binary operator

name(**alias*)

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

New in version 2.0.

otherwise(*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.

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

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

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

New in version 1.4.

over(*window*)

Define a windowing column.

Parameters: **window** – a **WindowSpec**

Returns: a Column

```
>>> from pyspark.sql import Window
>>> window = Window.partitionBy("name").orderBy("age").rowsBetween(-1, 1)
>>> from pyspark.sql.functions import rank, min
>>> # df.select(rank().over(window), min('age').over(window))
```

New in version 1.4.

rlike(*other*)

binary operator

startswith(*other*)

binary operator

substr(*startPos*, *length*)

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

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

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

New in version 1.3.

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.

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

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

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

New in version 1.4.

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, the fields will be sorted by names.

```
>>> 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)
```

asDict(*recursive=False*)

Return as an dict

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

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

class pyspark.sql.**DataFrameNaFunctions**(*df*)

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

New in version 1.4.

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.

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, 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, 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|
+---+------+-----+
```

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

New in version 1.3.1.

replace(*to_replace*, *value*, *subset=None*)

Returns a new **DataFrame** replacing a value with another value. **DataFrame.replace()** and **DataFrameNaFunctions.replace()** are aliases of each other.

Parameters:

- **to_replace** – int, long, float, string, or list. Value to be replaced. If the value is a dict, then *value* is ignored and *to_replace* must be a mapping from column name (string) to replacement value. The value to be replaced must be an int, long, float, or string.
- **value** – int, long, float, string, or list. Value to use to replace holes. The replacement value must be an int, long, float, or string. If *value* is a list or tuple, *value* should be of the same length with *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', 'Bob'], ['A', 'B'], 'name').show()
+----+------+----+
| age|height|name|
+----+------+----+
|  10|    80|   A|
|   5|  null|   B|
|null|  null| Tom|
|null|  null|null|
+----+------+----+
```

New in version 1.4.

class pyspark.sql.**DataFrameStatFunctions**(*df*)

Functionality for statistic functions with **DataFrame**.

New in version 1.4.

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

Calculates the approximate quantiles of a numerical column 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 [[http://dx.doi.org/10.1145/375663.375670 Space-efficient Online Computation of Quantile Summaries]] by Greenwald and Khanna.

Parameters:
- **col** – the name of the numerical column
- **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

New in version 2.0.

`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.

`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.

`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.

`freqItems`(*cols*, *support=None*)

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

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

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

New in version 1.4.

`sampleBy`(*col*, *fractions*, *seed=None*)

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|    5|
|  1|    9|
+---+-----+
```

New in version 1.5.

class pyspark.sql.**Window**

Utility functions for defining window in DataFrames.

For example:

```
>>> # PARTITION BY country ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
>>> window = Window.partitionBy("country").orderBy("date").rowsBetween(-sys.maxsize, 0)
```

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

Note: Experimental

New in version 1.4.

static **orderBy**(**cols*)

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

New in version 1.4.

static **partitionBy**(**cols*)

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

New in version 1.4.

class pyspark.sql.**WindowSpec**(*jspec*)

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

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

Note: Experimental

New in version 1.4.

orderBy(**cols*)

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

Parameters: **cols** – names of columns or expressions

New in version 1.4.

partitionBy(**cols*)

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

Parameters: **cols** – names of columns or expressions

New in version 1.4.

rangeBetween(*start*, *end*)

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

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

Parameters:
- **start** – boundary start, inclusive. The frame is unbounded if this is `-sys.maxsize` (or lower).
- **end** – boundary end, inclusive. The frame is unbounded if this is `sys.maxsize` (or higher).

New in version 1.4.

rowsBetween(*start*, *end*)

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

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

Parameters:
- **start** – boundary start, inclusive. The frame is unbounded if this is `-sys.maxsize` (or lower).
- **end** – boundary end, inclusive. The frame is unbounded if this is `sys.maxsize` (or higher).

New in version 1.4.

class `pyspark.sql.`**DataFrameReader**(*spark*)

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

New in version 1.4.

csv(*path*, *schema=None*, *sep=None*, *encoding=None*, *quote=None*, *escape=None*, *comment=None*, *header=None*, *inferSchema=None*, *ignoreLeadingWhiteSpace=None*, *ignoreTrailingWhiteSpace=None*, *nullValue=None*, *nanValue=None*, *positiveInf=None*, *negativeInf=None*, *dateFormat=None*, *maxColumns=None*, *maxCharsPerColumn=None*, *maxMalformedLogPerPartition=None*, *mode=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).
- **schema** – an optional **StructType** for the input schema.
- **sep** – sets the single character as a separator 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 the 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 the single character used for escaping quotes inside an already quoted value. If None is set, it uses the default value, `\`.
- **comment** – sets the 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`.
- **ignoreLeadingWhiteSpace** – defines whether or not leading whitespaces from values being read should be skipped. If None is set, it uses the default value, `false`.
- **ignoreTrailingWhiteSpace** – defines 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.
- **nanValue** – sets the string representation of a non-number value. If None is set, it uses the default value, `NaN`.
- **positiveInf** – sets the string representation of a positive infinity value. If None is set, it uses the default value, `Inf`.
- **negativeInf** – sets the string representation of a negative infinity value. If None is set, it uses the default value, `Inf`.
- **dateFormat** – sets the string that indicates a date format. Custom date formats follow the formats at `java.text.SimpleDateFormat`. This applies to both date type and timestamp type. By default, it is None which means trying to parse times and date by `java.sql.Timestamp.valueOf()` and `java.sql.Date.valueOf()`.

- **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, `1000000`.
- **maxMalformedLogPerPartition** – sets the maximum number of malformed rows Spark will log for each partition. Malformed records beyond this number will be ignored. If None is set, it uses the default value, `10`.
- **mode** –

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

  - `PERMISSIVE` : *sets other fields to `null` when it meets a corrupted record.* When a schema is set by user, it sets `null` for extra fields.
  - `DROPMALFORMED` : ignores the whole corrupted records.
  - `FAILFAST` : throws an exception when it meets corrupted records.

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

New in version 2.0.

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.

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.

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 an integer column 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 a "user" and "password" property should be included

Returns: a DataFrame

New in version 1.4.

json(*path*, *schema=None*, *primitivesAsString=None*, *prefersDecimal=None*, *allowComments=None*, *allowUnquotedFieldNames=None*, *allowSingleQuotes=None*, *allowNumericLeadingZero=None*, *allowBackslashEscapingAnyCharacter=None*, *mode=None*, *columnNameOfCorruptRecord=None*)

Loads a JSON file (one object per line) or an RDD of Strings storing JSON objects (one object per record) and returns the result as a :class`DataFrame`.

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 RDD of Strings storing JSON objects.
- **schema** – an optional **StructType** for the input schema.
- **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` : sets other fields to `null` when it meets a corrupted record and puts the malformed string into a new field configured by `columnNameOfCorruptRecord`. When a schema is set by user, it sets `null` for extra fields.
  - `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`.

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

New in version 1.4.

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

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

Parameters:

- **path** – optional string or a list of string for file-system backed data sources.
- **format** – optional string for format of the data source. Default to 'parquet'.
- **schema** – optional **`StructType`** for the input schema.

- **options** – all other string options

```
>>> df = spark.read.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.

option(*key*, *value*)

Adds an input option for the underlying data source.

New in version 1.5.

options(***options*)

Adds input options for the underlying data source.

New in version 1.4.

orc(*path*)

Loads an ORC file, returning the result as a **DataFrame**.

Note: Currently ORC support is only available together with Hive support.

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

New in version 1.5.

parquet(**paths*)

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

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

- `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`.

```
>>> 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.

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 StructType object

New in version 1.4.

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.

text(*paths*)

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.

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).

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

New in version 1.6.

class pyspark.sql.**DataFrameWriter**(*df*)

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

New in version 1.4.

csv(*path*, *mode=None*, *compression=None*, *sep=None*, *quote=None*, *escape=None*, *header=None*, *nullValue=None*, *escapeQuotes=None*, *quoteAll=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 (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 the single character as a separator for each field and value. If None is set, it uses the default value, ,.
- **quote** – sets the 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 the 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.

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

New in version 2.0.

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.

insertInto(*tableName*, *overwrite=False*)

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

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

Optionally overwriting any existing data.

New in version 1.4.

jdbc(*url*, *table*, *mode=None*, *properties=None*)

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` (default case): Throw an exception if data already exists.
- **properties** – JDBC database connection arguments, a list of arbitrary string tag/value. Normally at least a "user" and "password" property should be included.

New in version 1.4.

`json`(*path*, *mode=None*, *compression=None*)

Saves the content of the **`DataFrame`** in JSON 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` (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).

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

New in version 1.4.

`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*: 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.

option(*key*, *value*)

Adds an output option for the underlying data source.

New in version 1.5.

options(***options*)

Adds output options for the underlying data source.

New in version 1.4.

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

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

Note: Currently ORC support is only available together with Hive support.

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` (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`. If None is set, it uses the default value, `snappy`.

```
>>> 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.

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` (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, gzip, and lzo). This will override `spark.sql.parquet.compression.codec`. If None is set, it uses the value specified in `spark.sql.parquet.compression.codec`.

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

New in version 1.4.

partitionBy(**cols*)

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.

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` (default case): Throw an exception if data already exists.
- **partitionBy** – names of partitioning columns
- **options** – all other string options

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

New in version 1.4.

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*: 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*, *ignore* (default: error)
- **partitionBy** – names of partitioning columns
- **options** – all other string options

New in version 1.4.

text(*path*, *compression=None*)

Saves the content of the DataFrame in a text file at the specified path.

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).

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.

pyspark.sql.types module

class pyspark.sql.types.**DataType** [source]

Base class for data types.

fromInternal(*obj*) [source]

Converts an internal SQL object into a native Python object.

json() [source]

jsonValue() [source]

needConversion() [source]

Does this type need to conversion between Python object and internal SQL object.

This is used to avoid the unnecessary conversion for ArrayType/MapType/StructType.

simpleString() [source]

toInternal(*obj*) [source]

Converts a Python object into an internal SQL object.

classmethod **typeName**() [source]

class pyspark.sql.types.**NullType** [source]

Null type.

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

class pyspark.sql.types.**StringType** [source]

String data type.

class pyspark.sql.types.**BinaryType** [source]

Binary (byte array) data type.

class pyspark.sql.types.**BooleanType** [source]

Boolean data type.

class pyspark.sql.types.**DateType** [source]

Date (datetime.date) data type.

EPOCH_ORDINAL = *719163*

fromInternal(*v*) [source]

needConversion() [source]

toInternal(*d*) [source]

class pyspark.sql.types.**TimestampType** [source]

Timestamp (datetime.datetime) data type.

fromInternal(*ts*) [source]

needConversion() [source]

toInternal(*dt*) [source]

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

Decimal (decimal.Decimal) data type.

The DecimalType must have fixed precision (the maximum total number of digits) and scale (the number of digits on the right of dot). For example, (5, 2) can support the value from [-999.99 to 999.99].

The precision can be up to 38, the scale must less or equal to precision.

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

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

jsonValue() [source]

simpleString() [source]

class pyspark.sql.types.**DoubleType** [source]

Double data type, representing double precision floats.

class pyspark.sql.types.**FloatType** [source]

Float data type, representing single precision floats.

class pyspark.sql.types.**ByteType** [source]

Byte data type, i.e. a signed integer in a single byte.

simpleString() [source]

class pyspark.sql.types.**IntegerType** [source]

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

simpleString() [source]

class pyspark.sql.types.**LongType** [source]

Long data type, i.e. a signed 64-bit integer.

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

simpleString() [source]

class pyspark.sql.types.**ShortType** [source]

Short data type, i.e. a signed 16-bit integer.

simpleString() [source]

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

Array data type.

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

fromInternal(*obj*) [source]

classmethod **fromJson**(*json*) [source]

jsonValue() [source]

needConversion() [source]

simpleString() [source]

toInternal(*obj*) [source]

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

Map data type.

Parameters:
- **keyType** – **DataType** of the keys in the map.
- **valueType** – **DataType** of the values in the map.

- **valueContainsNull** – indicates whether values can contain null (None) values.

Keys in a map data type are not allowed to be null (None).

fromInternal(*obj*) [source]

classmethod **fromJson**(*json*) [source]

jsonValue() [source]

needConversion() [source]

simpleString() [source]

toInternal(*obj*) [source]

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

A field in **StructType**.

Parameters:
- **name** – string, name of the field.
- **dataType** – **DataType** of the field.
- **nullable** – boolean, whether the field can be null (None) or not.
- **metadata** – a dict from string to simple type that can be toInternald to JSON automatically

fromInternal(*obj*) [source]

classmethod **fromJson**(*json*) [source]

jsonValue() [source]

needConversion() [source]

simpleString() [source]

toInternal(*obj*) [source]

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

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

This is the data type representing a **`Row`**.

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

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

add(*field*, *data_type=None*, *nullable=True*, *metadata=None*) [source]

Construct a StructType by adding new elements to it to define the schema. The method accepts either:

a. A single parameter which is a StructField object.
b. Between 2 and 4 parameters as (name, data_type, nullable (optional), metadata(optional). The data_type parameter may be either a String or a DataType object.

```
>>> struct1 = StructType().add("f1", StringType(), True).add("f2", StringType(), True, None)
>>> struct2 = StructType([StructField("f1", StringType(), True), \
...     StructField("f2", StringType(), True, None)])
>>> struct1 == struct2
True
>>> struct1 = StructType().add(StructField("f1", StringType(), True))
>>> struct2 = StructType([StructField("f1", StringType(), True)])
>>> struct1 == struct2
True
>>> struct1 = StructType().add("f1", "string", True)
>>> struct2 = StructType([StructField("f1", StringType(), True)])
>>> struct1 == struct2
True
```

Parameters:
- **field** – Either the name of the field or a StructField object
- **data_type** – If present, the DataType of the StructField to create
- **nullable** – Whether the field to add should be nullable (default True)
- **metadata** – Any additional metadata (default None)

Returns: a new updated StructType

fromInternal(*obj*) [source]

classmethod **fromJson**(*json*) [source]

jsonValue() [source]

needConversion() [source]

simpleString() [source]

toInternal(*obj*) [source]

pyspark.sql.functions module

A collections of builtin functions

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

Computes the absolute value.

New in version 1.3.

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

Computes the cosine inverse of the given value; the returned angle is in the range0.0 through pi.

New in version 1.4.

pyspark.sql.functions.**add_months**(*start*, *months*) [source]

Returns the date that is *months* months after *start*

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['d'])
>>> df.select(add_months(df.d, 1).alias('d')).collect()
[Row(d=datetime.date(2015, 5, 8))]
```

New in version 1.5.

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

Returns a new **Column** for approximate distinct count of col.

```
>>> df.agg(approxCountDistinct(df.age).alias('c')).collect()
[Row(c=2)]
```

New in version 1.3.

pyspark.sql.functions.**array**(**cols*) [source]

Creates a new array column.

Parameters: **cols** – list of column names (string) or list of **Column** expressions that have the same data type.

```
>>> df.select(array('age', 'age').alias("arr")).collect()
[Row(arr=[2, 2]), Row(arr=[5, 5])]
>>> df.select(array([df.age, df.age]).alias("arr")).collect()
[Row(arr=[2, 2]), Row(arr=[5, 5])]
```

New in version 1.4.

pyspark.sql.functions.**array_contains**(*col*, *value*) [source]

Collection function: returns True if the array contains the given value. The collection elements and value must be of the same type.

Parameters:
- **col** – name of column containing array
- **value** – value 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)]
```

New in version 1.5.

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.**ascii**(*col*)

Computes the numeric value of the first character of the string column.

New in version 1.5.

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

Computes the sine inverse of the given value; the returned angle is in the range-pi/2 through pi/2.

New in version 1.4.

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

Computes the tangent inverse of the given value.

New in version 1.4.

pyspark.sql.functions.**atan2**(*col1*, *col2*)

Returns the angle theta from the conversion of rectangular coordinates (x, y) topolar coordinates (r, theta).

New in version 1.4.

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

Aggregate function: returns the average of the values in a group.

New in version 1.3.

pyspark.sql.functions.**base64**(*col*)

Computes the BASE64 encoding of a binary column and returns it as a string column.

New in version 1.5.

pyspark.sql.functions.**bin**(*col*) [source]

Returns the string representation of the binary value of the given column.

```
>>> df.select(bin(df.age).alias('c')).collect()
[Row(c=u'10'), Row(c=u'101')]
```

New in version 1.5.

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

Computes bitwise not.

New in version 1.4.

pyspark.sql.functions.**broadcast**(*df*) [source]

Marks a DataFrame as small enough for use in broadcast joins.

New in version 1.6.

pyspark.sql.functions.**bround**(*col*, *scale=0*) [source]

Round the given value to *scale* decimal places using HALF_EVEN rounding mode if *scale* >= 0 or at integral part when *scale* < 0.

```
>>> spark.createDataFrame([(2.5,)], ['a']).select(bround('a', 0).alias('r')).collect()
[Row(r=2.0)]
```

New in version 2.0.

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

Computes the cube-root of the given value.

New in version 1.4.

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

Computes the ceiling of the given value.

New in version 1.4.

pyspark.sql.functions.**coalesce**(**cols*) [source]

Returns the first column that is not null.

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

Returns a **Column** based on the given column name.

New in version 1.3.

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

Aggregate function: returns a list of objects with duplicates.

New in version 1.6.

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

Aggregate function: returns a set of objects with duplicate elements eliminated.

New in version 1.6.

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

Returns a **Column** based on the given column name.

New in version 1.3.

pyspark.sql.functions.**concat**(**cols*) [source]

Concatenates multiple input string columns together into a single string column.

```
>>> df = spark.createDataFrame([('abcd','123')], ['s', 'd'])
>>> df.select(concat(df.s, df.d).alias('s')).collect()
[Row(s=u'abcd123')]
```

New in version 1.5.

pyspark.sql.functions.**concat_ws**(*sep*, **cols*) [source]

Concatenates multiple input string columns together into a single string column, using the given separator.

```
>>> df = spark.createDataFrame([('abcd','123')], ['s', 'd'])
>>> df.select(concat_ws('-', df.s, df.d).alias('s')).collect()
[Row(s=u'abcd-123')]
```

New in version 1.5.

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

Convert a number in a string column from one base to another.

```
>>> df = spark.createDataFrame([("010101",)], ['n'])
>>> df.select(conv(df.n, 2, 16).alias('hex')).collect()
[Row(hex=u'15')]
```

New in version 1.5.

pyspark.sql.functions.**corr**(*col1*, *col2*) [source]

Returns a new **Column** for the Pearson Correlation Coefficient for `col1` and `col2`.

```
>>> a = range(20)
>>> b = [2 * x for x in range(20)]
>>> df = spark.createDataFrame(zip(a, b), ["a", "b"])
>>> df.agg(corr("a", "b").alias('c')).collect()
[Row(c=1.0)]
```

New in version 1.6.

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

Computes the cosine of the given value.

New in version 1.4.

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

Computes the hyperbolic cosine of the given value.

New in version 1.4.

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

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

New in version 1.3.

pyspark.sql.functions.**countDistinct**(*col*, **cols*) [source]

Returns a new **Column** for distinct count of `col` or `cols`.

```
>>> df.agg(countDistinct(df.age, df.name).alias('c')).collect()
[Row(c=2)]
```

```
>>> df.agg(countDistinct("age", "name").alias('c')).collect()
[Row(c=2)]
```

New in version 1.3.

pyspark.sql.functions.**covar_pop**(*col1*, *col2*) [source]

Returns a new **Column** for the population covariance of `col1` and `col2`.

```
>>> a = [1] * 10
>>> b = [1] * 10
>>> df = spark.createDataFrame(zip(a, b), ["a", "b"])
>>> df.agg(covar_pop("a", "b").alias('c')).collect()
[Row(c=0.0)]
```

New in version 2.0.

pyspark.sql.functions.**covar_samp**(*col1*, *col2*) [source]

Returns a new **Column** for the sample covariance of `col1` and `col2`.

```
>>> a = [1] * 10
>>> b = [1] * 10
>>> df = spark.createDataFrame(zip(a, b), ["a", "b"])
>>> df.agg(covar_samp("a", "b").alias('c')).collect()
[Row(c=0.0)]
```

New in version 2.0.

pyspark.sql.functions.**crc32**(*col*) [source]

Calculates the cyclic redundancy check value (CRC32) of a binary column and returns the value as a bigint.

```
>>> spark.createDataFrame([('ABC',)], ['a']).select(crc32('a').alias('crc32')).collect()
[Row(crc32=2743272264)]
```

New in version 1.5.

pyspark.sql.functions.**create_map**(**cols*) [source]

Creates a new map column.

Parameters: **cols** – list of column names (string) or list of `Column` expressions that grouped as key-value pairs, e.g. (key1, value1, key2, value2, ...).

```
>>> df.select(create_map('name', 'age').alias("map")).collect()
[Row(map={u'Alice': 2}), Row(map={u'Bob': 5})]
>>> df.select(create_map([df.name, df.age]).alias("map")).collect()
[Row(map={u'Alice': 2}), Row(map={u'Bob': 5})]
```

New in version 2.0.

pyspark.sql.functions.**cume_dist**()

Window function: returns the cumulative distribution of values within a window partition, i.e. the fraction of rows that are below the current row.

New in version 1.6.

pyspark.sql.functions.**current_date**() [source]

Returns the current date as a date column.

New in version 1.5.

pyspark.sql.functions.**current_timestamp**() [source]

Returns the current timestamp as a timestamp column.

pyspark.sql.functions.**date_add**(*start*, *days*) [source]

Returns the date that is *days* days after *start*

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['d'])
>>> df.select(date_add(df.d, 1).alias('d')).collect()
[Row(d=datetime.date(2015, 4, 9))]
```

New in version 1.5.

pyspark.sql.functions.**date_format**(*date*, *format*) [source]

Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument.

A pattern could be for instance *dd.MM.yyyy* and could return a string like '18.03.1993'. All pattern letters of the Java class *java.text.SimpleDateFormat* can be used.

NOTE: Use when ever possible specialized functions like *year*. These benefit from a specialized implementation.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['a'])
>>> df.select(date_format('a', 'MM/dd/yyy').alias('date')).collect()
[Row(date=u'04/08/2015')]
```

New in version 1.5.

pyspark.sql.functions.**date_sub**(*start*, *days*) [source]

Returns the date that is *days* days before *start*

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['d'])
>>> df.select(date_sub(df.d, 1).alias('d')).collect()
[Row(d=datetime.date(2015, 4, 7))]
```

New in version 1.5.

pyspark.sql.functions.**datediff**(*end*, *start*) [source]

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

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

New in version 1.5.

pyspark.sql.functions.**dayofmonth**(*col*) [source]

Extract the day of the month of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['a'])
>>> df.select(dayofmonth('a').alias('day')).collect()
[Row(day=8)]
```

New in version 1.5.

pyspark.sql.functions.**dayofyear**(*col*) [source]

Extract the day of the year of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['a'])
>>> df.select(dayofyear('a').alias('day')).collect()
[Row(day=98)]
```

New in version 1.5.

pyspark.sql.functions.**decode**(*col*, *charset*) [source]

Computes the first argument into a string from a binary using the provided character set (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16').

New in version 1.5.

pyspark.sql.functions.**dense_rank**()

Window function: returns the rank of rows within a window partition, without any gaps.

The difference between rank and denseRank is that denseRank leaves no gaps in ranking sequence when there are ties. That is, if you were ranking a competition using denseRank 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.

New in version 1.6.

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

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

New in version 1.3.

pyspark.sql.functions.**encode**(*col*, *charset*) [source]

Computes the first argument into a binary from a string using the provided character set (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16').

New in version 1.5.

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

Computes the exponential of the given value.

New in version 1.4.

pyspark.sql.functions.**explode**(*col*) [source]

Returns a new row for each element in the given array or map.

```
>>> 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.**expm1**(*col*)

Computes the exponential of the given value minus one.

New in version 1.4.

pyspark.sql.functions.**expr**(*str*) [source]

Parses the expression string into the column that it represents

```
>>> df.select(expr("length(name)")).collect()
[Row(length(name)=5), Row(length(name)=3)]
```

New in version 1.5.

pyspark.sql.functions.**factorial**(*col*) [source]

Computes the factorial of the given value.

```
>>> df = spark.createDataFrame([(5,)], ['n'])
>>> df.select(factorial(df.n).alias('f')).collect()
[Row(f=120)]
```

New in version 1.5.

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

Aggregate function: returns the first value in a group.

The function by default returns the first values it sees. It will return the first non-null value it sees when ignoreNulls is set to true. If all values are null, then null is returned.

New in version 1.3.

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

Computes the floor of the given value.

New in version 1.4.

pyspark.sql.functions.**format_number**(*col*, *d*) [source]

Formats the number X to a format like '#,–#,–#.–', rounded to d decimal places, 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=u'5.0000')]
```

New in version 1.5.

pyspark.sql.functions.**format_string**(*format*, **cols*) [source]

Formats the arguments in printf-style and returns the result as a string column.

Parameters:
- **col** – the column name of the numeric value to be formatted
- **d** – the N decimal places

```
>>> df = spark.createDataFrame([(5, "hello")], ['a', 'b'])
>>> df.select(format_string('%d %s', df.a, df.b).alias('v')).collect()
[Row(v=u'5 hello')]
```

New in version 1.5.

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

Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the given format.

New in version 1.5.

pyspark.sql.functions.**from_utc_timestamp**(*timestamp*, *tz*) [source]

Assumes given timestamp is UTC and converts to given timezone.

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00',)], ['t'])
>>> df.select(from_utc_timestamp(df.t, "PST").alias('t')).collect()
[Row(t=datetime.datetime(1997, 2, 28, 2, 30))]
```

New in version 1.5.

pyspark.sql.functions.**get_json_object**(*col*, *path*) [source]

Extracts json object from a json string based on json path specified, and returns json string of the extracted json object. It will return null if the input json string is invalid.

Parameters:
- **col** – string column in json format
- **path** – path to the json object to extract

```
>>> data = [("1", '''{"f1": "value1", "f2": "value2"}'''), ("2", '''{"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=u'1', c0=u'value1', c1=u'value2'), Row(key=u'2', c0=u'value12', c1=None)]
```

New in version 1.6.

pyspark.sql.functions.**greatest**(**cols*) [source]

Returns the greatest value of the list of column names, skipping null values. This function takes at least 2 parameters. It will return null iff all parameters are null.

```
>>> df = spark.createDataFrame([(1, 4, 3)], ['a', 'b', 'c'])
>>> df.select(greatest(df.a, df.b, df.c).alias("greatest")).collect()
[Row(greatest=4)]
```

New in version 1.5.

pyspark.sql.functions.**grouping**(*col*) [source]

Aggregate function: indicates whether a specified column in a GROUP BY list is aggregated or not, returns 1 for aggregated or 0 for not aggregated in the result set.

```
>>> df.cube("name").agg(grouping("name"), sum("age")).orderBy("name").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**(**cols*) [source]

Aggregate function: returns the level of grouping, equals to

(grouping(c1) << (n-1)) + (grouping(c2) << (n-2)) + ... + grouping(cn)

Note: the list of columns should match with grouping columns exactly, or empty (means all the grouping columns).

```
>>> df.cube("name").agg(grouping_id(), sum("age")).orderBy("name").show()
+-----+-------------+--------+
| name|grouping_id()|sum(age)|
+-----+-------------+--------+
| null|            1|       7|
|Alice|            0|       2|
|  Bob|            0|       5|
+-----+-------------+--------+
```

New in version 2.0.

pyspark.sql.functions.**hash**(**cols*) [source]

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

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

New in version 2.0.

pyspark.sql.functions.**hex**(*col*) [source]

Computes hex value of the given column, which could be StringType, BinaryType, IntegerType or LongType.

```
>>> spark.createDataFrame([('ABC', 3)], ['a', 'b']).select(hex('a'), hex('b')).collect()
[Row(hex(a)=u'414243', hex(b)=u'3')]
```

New in version 1.5.

pyspark.sql.functions.**hour**(*col*) [source]

Extract the hours of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08 13:08:15',)], ['a'])
>>> df.select(hour('a').alias('hour')).collect()
[Row(hour=13)]
```

New in version 1.5.

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

Computes *sqrt(a^2 + b^2)* without intermediate overflow or underflow.

New in version 1.4.

pyspark.sql.functions.**initcap**(*col*) [source]

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

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

New in version 1.5.

pyspark.sql.functions.**input_file_name**() [source]

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

New in version 1.6.

pyspark.sql.functions.**instr**(*str*, *substr*) [source]

Locate the position of the first occurrence of substr column in the given string. Returns null if either of the arguments are null.

NOTE: The position is not zero based, but 1 based index, returns 0 if substr could not be found in str.

```
>>> df = spark.createDataFrame([('abcd',)], ['s',])
>>> df.select(instr(df.s, 'b').alias('s')).collect()
[Row(s=2)]
```

New in version 1.5.

pyspark.sql.functions.**isnan**(*col*) [source]

An expression that returns true iff the column is NaN.

```
>>> df = spark.createDataFrame([(1.0, float('nan')), (float('nan'), 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**(*col*) [source]

An expression that returns true iff the column is null.

```
>>> df = spark.createDataFrame([(1, None), (None, 2)], ("a", "b"))
>>> df.select(isnull("a").alias("r1"), isnull(df.a).alias("r2")).collect()
[Row(r1=False, r2=False), Row(r1=True, r2=True)]
```

New in version 1.6.

pyspark.sql.functions.**json_tuple**(*col*, **fields*) [source]

Creates a new row for a json column according to the given field names.

Parameters:
- **col** – string column in json format
- **fields** – list of fields to extract

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

New in version 1.6.

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

Aggregate function: returns the kurtosis of the values in a group.

New in version 1.6.

pyspark.sql.functions.**lag**(*col*, *count=1*, *default=None*) [source]

Window function: returns the value that is *offset* rows before the current row, and *defaultValue* if there is less than *offset* rows before the current row. For example, an *offset* of one will return the previous row at any given point in the window partition.

This is equivalent to the LAG function in SQL.

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

- **count** – number of row to extend
- **default** – default value

New in version 1.4.

`pyspark.sql.functions.`**`last`**(*col*, *ignorenulls=False*) [source]

Aggregate function: returns the last value in a group.

The function by default returns the last values it sees. It will return the last non-null value it sees when ignoreNulls is set to true. If all values are null, then null is returned.

New in version 1.3.

`pyspark.sql.functions.`**`last_day`**(*date*) [source]

Returns the last day of the month which the given date belongs to.

```
>>> df = spark.createDataFrame([('1997-02-10',)], ['d'])
>>> df.select(last_day(df.d).alias('date')).collect()
[Row(date=datetime.date(1997, 2, 28))]
```

New in version 1.5.

`pyspark.sql.functions.`**`lead`**(*col*, *count=1*, *default=None*) [source]

Window function: returns the value that is *offset* rows after the current row, and *defaultValue* if there is less than *offset* rows after the current row. For example, an *offset* of one will return the next row at any given point in the window partition.

This is equivalent to the LEAD function in SQL.

Parameters:
- **col** – name of column or expression
- **count** – number of row to extend
- **default** – default value

New in version 1.4.

`pyspark.sql.functions.`**`least`**(**cols*) [source]

Returns the least value of the list of column names, skipping null values. This function takes at least 2 parameters. It will return null iff all parameters

are null.

```
>>> df = spark.createDataFrame([(1, 4, 3)], ['a', 'b', 'c'])
>>> df.select(least(df.a, df.b, df.c).alias("least")).collect()
[Row(least=1)]
```

New in version 1.5.

pyspark.sql.functions.**length**(*col*) [source]

Calculates the length of a string or binary expression.

```
>>> spark.createDataFrame([('ABC',)], ['a']).select(length('a').alias('length')).collect()
[Row(length=3)]
```

New in version 1.5.

pyspark.sql.functions.**levenshtein**(*left*, *right*) [source]

Computes the Levenshtein distance of the two given strings.

```
>>> df0 = spark.createDataFrame([('kitten', 'sitting',)], ['l', 'r'])
>>> df0.select(levenshtein('l', 'r').alias('d')).collect()
[Row(d=3)]
```

New in version 1.5.

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

Creates a **Column** of literal value.

New in version 1.3.

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

Locate the position of the first occurrence of substr in a string column, after position pos.

NOTE: The position is not zero based, but 1 based index. returns 0 if substr could not be found in str.

Parameters:
- **substr** – a string
- **str** – a Column of StringType
- **pos** – start position (zero based)

```
>>> df = spark.createDataFrame([('abcd',)], ['s',])
>>> df.select(locate('b', df.s, 1).alias('s')).collect()
[Row(s=2)]
```

New in version 1.5.

`pyspark.sql.functions.`**log**(*arg1*, *arg2=None*) [source]

Returns the first argument-based logarithm of the second argument.

If there is only one argument, then this takes the natural logarithm of the argument.

```
>>> df.select(log(10.0, df.age).alias('ten')).rdd.map(lambda l: str(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**(*col*)

Computes the logarithm of the given value in Base 10.

New in version 1.4.

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

Computes the natural logarithm of the given value plus one.

New in version 1.4.

`pyspark.sql.functions.`**log2**(*col*) [source]

Returns the base-2 logarithm of the argument.

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

New in version 1.5.

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

Converts a string column to lower case.

New in version 1.5.

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

Left-pad the string column to width *len* with *pad*.

```
>>> df = spark.createDataFrame([('abcd',)], ['s',])
>>> df.select(lpad(df.s, 6, '#').alias('s')).collect()
[Row(s=u'##abcd')]
```

New in version 1.5.

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

Trim the spaces from left end for the specified string value.

New in version 1.5.

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

Aggregate function: returns the maximum value of the expression in a group.

New in version 1.3.

pyspark.sql.functions.**md5**(*col*) [source]

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

```
>>> spark.createDataFrame([('ABC',)], ['a']).select(md5('a').alias('hash')).collect()
```

```
[Row(hash=u'902fbdd2b1df0c4f70b4a5d23525e932')]
```

New in version 1.5.

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

Aggregate function: returns the average of the values in a group.

New in version 1.3.

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

Aggregate function: returns the minimum value of the expression in a group.

New in version 1.3.

pyspark.sql.functions.**minute**(*col*) [source]

Extract the minutes of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08 13:08:15',)], ['a'])
>>> df.select(minute('a').alias('minute')).collect()
[Row(minute=8)]
```

New in version 1.5.

pyspark.sql.functions.**monotonically_increasing_id**() [source]

A column that generates monotonically increasing 64-bit integers.

The generated ID is guaranteed to be monotonically increasing and unique, but not consecutive. The current implementation puts the partition ID in the upper 31 bits, and the record number within each partition in the lower 33 bits. The assumption is that the data frame has less than 1 billion partitions, and each partition has less than 8 billion records.

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**(*col*) [source]

Extract the month of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['a'])
>>> df.select(month('a').alias('month')).collect()
[Row(month=4)]
```

New in version 1.5.

pyspark.sql.functions.**months_between**(*date1*, *date2*) [source]

Returns the number of months between date1 and date2.

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00', '1996-10-30')], ['t', 'd'])
>>> df.select(months_between(df.t, df.d).alias('months')).collect()
[Row(months=3.9495967...)]
```

New in version 1.5.

pyspark.sql.functions.**nanvl**(*col1*, *col2*) [source]

Returns col1 if it is not NaN, or col2 if col1 is NaN.

Both inputs should be floating point columns (DoubleType or FloatType).

```
>>> df = spark.createDataFrame([(1.0, float('nan')), (float('nan'), 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**(*date*, *dayOfWeek*) [source]

Returns the first date which is later than the value of the date column.

Day of the week parameter is case insensitive, and accepts:

"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun".

```
>>> df = spark.createDataFrame([('2015-07-27',)], ['d'])
>>> df.select(next_day(df.d, 'Sun').alias('date')).collect()
[Row(date=datetime.date(2015, 8, 2))]
```

New in version 1.5.

`pyspark.sql.functions.`**`ntile`**`(`*`n`*`)` [source]

Window function: returns the ntile group id (from 1 to *n* inclusive) in an ordered window partition. For example, if *n* is 4, the first quarter of the rows will get value 1, the second quarter will get 2, the third quarter will get 3, and the last quarter will get 4.

This is equivalent to the NTILE function in SQL.

Parameters: **n** – an integer

New in version 1.4.

`pyspark.sql.functions.`**`percent_rank`**`()`

Window function: returns the relative rank (i.e. percentile) of rows within a window partition.

New in version 1.6.

`pyspark.sql.functions.`**`posexplode`**`(`*`col`*`)` [source]

Returns a new row for each element with position in the given array or map.

```
>>> 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.**pow**(*col1*, *col2*)

Returns the value of the first argument raised to the power of the second argument.

New in version 1.4.

pyspark.sql.functions.**quarter**(*col*) [source]

Extract the quarter of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['a'])
>>> df.select(quarter('a').alias('quarter')).collect()
[Row(quarter=2)]
```

New in version 1.5.

pyspark.sql.functions.**rand**(*seed=None*) [source]

Generates a random column with i.i.d. samples from U[0.0, 1.0].

New in version 1.4.

pyspark.sql.functions.**randn**(*seed=None*) [source]

Generates a column with i.i.d. samples from the standard normal distribution.

New in version 1.4.

pyspark.sql.functions.**rank**()

Window function: returns the rank of rows within a window partition.

The difference between rank and denseRank is that denseRank leaves no gaps in ranking sequence when there are ties. That is, if you were ranking a competition using denseRank 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.

This is equivalent to the RANK function in SQL.

New in version 1.6.

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

Extract a specific(idx) group identified by a java regex, from the specified string column.

```
>>> df = spark.createDataFrame([('100-200',)], ['str'])
>>> df.select(regexp_extract('str', '(\d+)-(\d+)', 1).alias('d')).collect()
[Row(d=u'100')]
```

New in version 1.5.

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

Replace all substrings of the specified string value that match regexp with rep.

```
>>> df = spark.createDataFrame([('100-200',)], ['str'])
>>> df.select(regexp_replace('str', '(\d+)', '--').alias('d')).collect()
[Row(d=u'-----')]
```

New in version 1.5.

pyspark.sql.functions.**repeat**(*col*, *n*) [source]

Repeats a string column n times, and returns it as a new string column.

```
>>> df = spark.createDataFrame([('ab',)], ['s',])
>>> df.select(repeat(df.s, 3).alias('s')).collect()
[Row(s=u'ababab')]
```

New in version 1.5.

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

Reverses the string column and returns it as a new string column.

New in version 1.5.

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

Returns the double value that is closest in value to the argument and is equal to a mathematical integer.

New in version 1.4.

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

Round the given value to *scale* decimal places using HALF_UP rounding mode if *scale* >= 0 or at integral part when *scale* < 0.

```
>>> spark.createDataFrame([(2.5,)], ['a']).select(round('a', 0).alias('r')).collect()
[Row(r=3.0)]
```

New in version 1.5.

pyspark.sql.functions.**row_number**()

Window function: returns a sequential number starting at 1 within a window partition.

New in version 1.6.

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

Right-pad the string column to width *len* with *pad*.

```
>>> df = spark.createDataFrame([('abcd',)], ['s',])
>>> df.select(rpad(df.s, 6, '#').alias('s')).collect()
[Row(s=u'abcd##')]
```

New in version 1.5.

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

Trim the spaces from right end for the specified string value.

New in version 1.5.

pyspark.sql.functions.**second**(*col*) [source]

Extract the seconds of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08 13:08:15',)], ['a'])
>>> df.select(second('a').alias('second')).collect()
[Row(second=15)]
```

New in version 1.5.

pyspark.sql.functions.**sha1**(*col*) [source]

Returns the hex string result of SHA-1.

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

New in version 1.5.

pyspark.sql.functions.**sha2**(*col*, *numBits*) [source]

Returns the hex string result of SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, and SHA-512). The numBits indicates the desired bit length of the result, which must have a value of 224, 256, 384, 512, or 0 (which is equivalent to 256).

```
>>> digests = df.select(sha2(df.name, 256).alias('s')).collect()
>>> digests[0]
Row(s=u'3bc51062973c458d5a6f2d8d64a023246354ad7e064b1e4e009ec8a0699a3043')
>>> digests[1]
Row(s=u'cd9fb1e148ccd8442e5aa74904cc73bf6fb54d1d54d333bd596aa9bb4bb4e961')
```

New in version 1.5.

pyspark.sql.functions.**shiftLeft**(*col*, *numBits*) [source]

Shift the given value numBits left.

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

New in version 1.5.

pyspark.sql.functions.**shiftRight**(*col*, *numBits*) [source]

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**(*col*, *numBits*) [source]

Unsigned shift the given value numBits right.

```
>>> df = spark.createDataFrame([(-42,)], ['a'])
>>> df.select(shiftRightUnsigned('a', 1).alias('r')).collect()
[Row(r=9223372036854775787)]
```

New in version 1.5.

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

Computes the signum of the given value.

New in version 1.4.

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

Computes the sine of the given value.

New in version 1.4.

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

Computes the hyperbolic sine of the given value.

New in version 1.4.

pyspark.sql.functions.**size**(*col*) [source]

Collection function: returns the length of the array or map stored in the column.

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

```
>>> df = spark.createDataFrame([([1, 2, 3],),([1],),([],)], ['data'])
>>> df.select(size(df.data)).collect()
[Row(size(data)=3), Row(size(data)=1), Row(size(data)=0)]
```

New in version 1.5.

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

Aggregate function: returns the skewness of the values in a group.

New in version 1.6.

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

Collection function: sorts the input array for the given column in ascending order.

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

```
>>> df = spark.createDataFrame([([2, 1, 3],),([1],),([],)], ['data'])
>>> df.select(sort_array(df.data).alias('r')).collect()
[Row(r=[1, 2, 3]), Row(r=[1]), Row(r=[])]
>>> df.select(sort_array(df.data, asc=False).alias('r')).collect()
[Row(r=[3, 2, 1]), Row(r=[1]), Row(r=[])]
```

New in version 1.5.

pyspark.sql.functions.**soundex**(*col*) [source]

Returns the SoundEx encoding for a string

```
>>> df = spark.createDataFrame([("Peters",),("Uhrbach",)], ['name'])
>>> df.select(soundex(df.name).alias("soundex")).collect()
[Row(soundex=u'P362'), Row(soundex=u'U612')]
```

New in version 1.5.

pyspark.sql.functions.**spark_partition_id**() [source]

A column for partition ID of the Spark task.

Note that 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**(*str*, *pattern*) [source]

Splits str around pattern (pattern is a regular expression).

NOTE: pattern is a string represent the regular expression.

```
>>> df = spark.createDataFrame([('ab12cd',)], ['s',])
>>> df.select(split(df.s, '[0-9]+').alias('s')).collect()
[Row(s=[u'ab', u'cd'])]
```

New in version 1.5.

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

Computes the square root of the specified float value.

New in version 1.3.

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

Aggregate function: returns the unbiased sample standard deviation of the expression in a group.

New in version 1.6.

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

Aggregate function: returns population standard deviation of the expression in a group.

New in version 1.6.

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

Aggregate function: returns the unbiased sample standard deviation of the expression in a group.

New in version 1.6.

pyspark.sql.functions.**struct**(**cols*) [source]

Creates a new struct column.

Parameters: **cols** – list of column names (string) or list of `Column` expressions

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

New in version 1.4.

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

Substring starts at *pos* and is of length *len* when str is String type or returns the slice of byte array that starts at *pos* in byte and is of length *len* when str is Binary type

```
>>> df = spark.createDataFrame([('abcd',)], ['s',])
>>> df.select(substring(df.s, 1, 2).alias('s')).collect()
[Row(s=u'ab')]
```

New in version 1.5.

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

Returns the substring from string str before count occurrences of the delimiter delim. If count is positive, everything the left of the final delimiter (counting from left) is returned. If count is negative, every to the right of the final delimiter (counting from the right) is returned. substring_index performs a case-sensitive match when searching for delim.

```
>>> df = spark.createDataFrame([('a.b.c.d',)], ['s'])
>>> df.select(substring_index(df.s, '.', 2).alias('s')).collect()
[Row(s=u'a.b')]
```

```
>>> df.select(substring_index(df.s, '.', -3).alias('s')).collect()
[Row(s=u'b.c.d')]
```

New in version 1.5.

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

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

New in version 1.3.

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

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

New in version 1.3.

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

Computes the tangent of the given value.

New in version 1.4.

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

Computes the hyperbolic tangent of the given value.

New in version 1.4.

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

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

New in version 1.4.

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

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

New in version 1.4.

pyspark.sql.functions.**to_date**(*col*) [source]

Converts the column of StringType or TimestampType into DateType.

```
>>> 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))]
```

New in version 1.5.

pyspark.sql.functions.**to_utc_timestamp**(*timestamp*, *tz*) [source]

Assumes given timestamp is in given timezone and converts to UTC.

```
>>> df = spark.createDataFrame([('1997-02-28 10:30:00',)], ['t'])
>>> df.select(to_utc_timestamp(df.t, "PST").alias('t')).collect()
[Row(t=datetime.datetime(1997, 2, 28, 18, 30))]
```

New in version 1.5.

pyspark.sql.functions.**translate**(*srcCol*, *matching*, *replace*) [source]

A function translate any character in the *srcCol* by a character in *matching*. The characters in *replace* is corresponding to the characters in *matching*. The translate will happen when any character in the string matching with the character in the *matching*.

```
>>> spark.createDataFrame([('translate',)], ['a']).select(translate('a', "rnlt", "123") \
...     .alias('r')).collect()
[Row(r=u'1a2s3ae')]
```

New in version 1.5.

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

Trim the spaces from both ends for the specified string column.

New in version 1.5.

pyspark.sql.functions.**trunc**(*date*, *format*) [source]

Returns date truncated to the unit specified by the format.

Parameters: **format** – ‘year’, ‘YYYY’, ‘yy’ or ‘month’, ‘mon’, ‘mm’

```
>>> df = spark.createDataFrame([('1997-02-28',)], ['d'])
>>> df.select(trunc(df.d, 'year').alias('year')).collect()
[Row(year=datetime.date(1997, 1, 1))]
>>> df.select(trunc(df.d, 'mon').alias('month')).collect()
[Row(month=datetime.date(1997, 2, 1))]
```

New in version 1.5.

pyspark.sql.functions.**udf**(*f*, *returnType=StringType*) [source]

Creates a **Column** expression representing a user defined function (UDF). Note that the user-defined functions must be deterministic. Due to optimization, duplicate invocations may be eliminated or the function may even be invoked more times than it is present in the query.

```
>>> from pyspark.sql.types import IntegerType
>>> slen = udf(lambda s: len(s), IntegerType())
>>> df.select(slen(df.name).alias('slen')).collect()
[Row(slen=5), Row(slen=3)]
```

New in version 1.3.

pyspark.sql.functions.**unbase64**(*col*)

Decodes a BASE64 encoded string column and returns it as a binary column.

New in version 1.5.

pyspark.sql.functions.**unhex**(*col*) [source]

Inverse of hex. Interprets each pair of characters as a hexadecimal number and converts to the byte representation of number.

```
>>> spark.createDataFrame([('414243',)], ['a']).select(unhex('a')).collect()
[Row(unhex(a)=bytearray(b'ABC'))]
```

New in version 1.5.

pyspark.sql.functions.**unix_timestamp**(*timestamp=None*, *format='yyyy-MM-dd HH:mm:ss'*) [source]

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.

New in version 1.5.

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

Converts a string column to upper case.

New in version 1.5.

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

Aggregate function: returns the population variance of the values in a group.

New in version 1.6.

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

Aggregate function: returns the unbiased variance of the values in a group.

New in version 1.6.

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

Aggregate function: returns the population variance of the values in a group.

New in version 1.6.

pyspark.sql.functions.**weekofyear**(*col*) [source]

Extract the week number of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['a'])
>>> df.select(weekofyear(df.a).alias('week')).collect()
[Row(week=15)]
```

New in version 1.5.

pyspark.sql.functions.**when**(*condition*, *value*) [source]

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

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

```
>>> df.select(when(df['age'] == 2, 3).otherwise(4).alias("age")).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`**(*timeColumn*, *windowDuration*, *slideDuration=None*, *startTime=None*) [source]

Bucketize rows into one or more time windows given a timestamp specifying column. Window starts are inclusive but the window ends are exclusive, e.g. 12:05 will be in the window [12:05,12:10) but not in [12:00,12:05). Windows can support microsecond precision. Windows in the order of months are not supported.

The time column must be of 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 *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=u'2016-03-11 09:00:05', end=u'2016-03-11 09:00:10', sum=1)]
```

New in version 2.0.

`pyspark.sql.functions.`**`year`**(*col*) [source]

Extract the year of a given date as integer.

```
>>> df = spark.createDataFrame([('2015-04-08',)], ['a'])
>>> df.select(year('a').alias('year')).collect()
[Row(year=2015)]
```

New in version 1.5.

pyspark.sql.streaming module

class pyspark.sql.streaming.**StreamingQuery**(*jsq*) [source]

A handle to a query that is executing continuously in the background as new data arrives. All these methods are thread-safe.

> **Note:** Experimental

New in version 2.0.

awaitTermination(*timeout=None*) [source]

Waits for the termination of *this* query, either by **query.stop()** or by an exception. If the query has terminated with an exception, then the exception will be thrown. If *timeout* is set, it returns whether the query has terminated or not within the *timeout* seconds.

If the query has terminated, then all subsequent calls to this method will either return immediately (if the query was terminated by **stop()**), or throw the exception immediately (if the query has terminated with exception).

throws **StreamingQueryException**, if *this* query has terminated with an exception

New in version 2.0.

id [source]

The id of the streaming query. This id is unique across all queries that have been started in the current process.

New in version 2.0.

isActive [source]

Whether this streaming query is currently active or not.

New in version 2.0.

name [source]

The name of the streaming query. This name is unique across all active queries.

New in version 2.0.

processAllAvailable() [source]

Blocks until all available data in the source has been processed and committed to the sink. This method is intended for testing. Note that 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.

stop() [source]

Stop this streaming query.

New in version 2.0.

class pyspark.sql.streaming.**StreamingQueryManager**(*jsqm*) [source]

A class to manage all the **StreamingQuery** StreamingQueries active.

> **Note:** Experimental

New in version 2.0.

active [source]

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]
[u'this_query']
>>> sq.stop()
```

New in version 2.0.

awaitAnyTermination(*timeout=None*) [source]

Wait until any of the queries on the associated SQLContext has terminated since the creation of the context, or since **resetTerminated()** was called. If any query was terminated with an exception, then the exception will be thrown. If *timeout* is set, it returns whether the query has terminated or not within the *timeout* seconds.

If a query has terminated, then subsequent calls to **awaitAnyTermination()** will either return immediately (if the query was terminated by **query.stop()**), or throw the exception immediately (if the query was terminated with exception). Use **resetTerminated()** to clear past terminations and wait for new terminations.

In the case where multiple queries have terminated since **resetTermination()** was called, if any query has terminated with exception, then **awaitAnyTermination()** will throw any of the exception. For correctly documenting exceptions across multiple queries, users need to stop all of them after any of them terminates with exception, and then check the *query.exception()* for each query.

throws **StreamingQueryException**, if *this* query has terminated with an exception

New in version 2.0.

get(*id*) [source]

Returns an active query from this SQLContext or throws exception if an active query with this name doesn't exist.

```
>>> sq = sdf.writeStream.format('memory').queryName('this_query').start()
>>> sq.name
u'this_query'
>>> sq = spark.streams.get(sq.id)
>>> sq.isActive
True
>>> sq = sqlContext.streams.get(sq.id)
>>> sq.isActive
True
>>> sq.stop()
```

New in version 2.0.

resetTerminated() [source]

Forget about past terminated queries so that **awaitAnyTermination()** can be used again to wait for new terminations.

```
>>> spark.streams.resetTerminated()
```

New in version 2.0.

class pyspark.sql.streaming.**DataStreamReader**(*spark*) [source]

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

> **Note:** Experimental.

New in version 2.0.

csv(*path*, *schema=None*, *sep=None*, *encoding=None*, *quote=None*, *escape=None*, *comment=None*, *header=None*, *inferSchema=None*, *ignoreLeadingWhiteSpace=None*, *ignoreTrailingWhiteSpace=None*, *nullValue=None*, *nanValue=None*, *positiveInf=None*, *negativeInf=None*, *dateFormat=None*, *maxColumns=None*, *maxCharsPerColumn=None*, *maxMalformedLogPerPartition=None*, *mode=None*) [source]

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

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

> **Note:** Experimental.

Parameters:

- **path** – string, or list of strings, for input path(s).
- **schema** – an optional **StructType** for the input schema.
- **sep** – sets the single character as a separator 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 the 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 the single character used for escaping quotes inside an already quoted value. If None is set, it uses the default value, `\`.
- **comment** – sets the 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`.
- **ignoreLeadingWhiteSpace** – defines whether or not leading whitespaces from values being read should be skipped. If None is set, it uses the default value, `false`.
- **ignoreTrailingWhiteSpace** – defines 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.
- **nanValue** – sets the string representation of a non-number value. If None is set, it uses the default value, `NaN`.
- **positiveInf** – sets the string representation of a positive infinity value. If None is set, it uses the default value, `Inf`.
- **negativeInf** – sets the string representation of a negative infinity value. If None is set, it uses the default value, `Inf`.
- **dateFormat** – sets the string that indicates a date format. Custom date formats follow the formats at `java.text.SimpleDateFormat`. This applies to both date type and timestamp type. By default, it is None which means trying to parse times and date by `java.sql.Timestamp.valueOf()` and `java.sql.Date.valueOf()`.
- **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, `1000000`.
- **mode** –

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

  - `PERMISSIVE` : *sets other fields to `null` when it meets a corrupted record.* When a schema is set by user, it sets `null` for extra fields.
  - `DROPMALFORMED` : ignores the whole corrupted records.
  - `FAILFAST` : throws an exception when it meets corrupted records.

```
>>> 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.

`format`(*source*) [source]

Specifies the input data source format.

> **Note:** Experimental.

Parameters: **source** – string, name of the data source, e.g. 'json', 'parquet'.

```
>>> s = spark.readStream.format("text")
```

New in version 2.0.

`json`(*path*, *schema=None*, *primitivesAsString=None*, *prefersDecimal=None*, *allowComments=None*, *allowUnquotedFieldNames=None*, *allowSingleQuotes=None*, *allowNumericLeadingZero=None*, *allowBackslashEscapingAnyCharacter=None*, *mode=None*, *columnNameOfCorruptRecord=None*) [source]

Loads a JSON file stream (one object per line) and returns a :class`DataFrame`.

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

> **Note:** Experimental.

Parameters:

- **path** – string represents path to the JSON dataset, or RDD of Strings storing JSON objects.
- **schema** – an optional `StructType` for the input schema.
- **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` : sets other fields to `null` when it meets a corrupted record and puts the malformed string into a new field configured by `columnNameOfCorruptRecord`. When a schema is set by user, it sets `null` for extra fields.
- `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`.

```
>>> 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.

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

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

Note: Experimental.

Parameters:
- **path** – optional string for file-system backed data sources.
- **format** – optional string for format of the data source. Default to ‘parquet’.
- **schema** – optional **`StructType`** for the input schema.
- **options** – all other string options

```
>>> json_sdf = spark.readStream.format("json")                                 .schema(sdf_schema)
>>> json_sdf.isStreaming
True
>>> json_sdf.schema == sdf_schema
True
```

New in version 2.0.

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

Adds an input option for the underlying data source.

> **Note:** Experimental.

```
>>> s = spark.readStream.option("x", 1)
```

New in version 2.0.

options(***options*) [source]

Adds input options for the underlying data source.

> **Note:** Experimental.

```
>>> s = spark.readStream.options(x="1", y=2)
```

New in version 2.0.

parquet(*path*) [source]

Loads a Parquet file stream, returning the result as a **DataFrame**.

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

- `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`.

> **Note:** Experimental.

```
>>> 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.

schema(*schema*) [source]

Specifies the input schema.

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

> **Note:** Experimental.

Parameters: **schema** – a StructType object

```
>>> s = spark.readStream.schema(sdf_schema)
```

New in version 2.0.

text(*path*) [source]

Loads a text file stream and returns a **DataFrame** whose schema starts with a string column named “value”, and followed by partitioned columns if there are any.

Each line in the text file is a new row in the resulting DataFrame.

> **Note:** Experimental.

Parameters: **paths** – string, or list of strings, for input path(s).

```
>>> text_sdf = spark.readStream.text(tempfile.mkdtemp())
>>> text_sdf.isStreaming
True
>>> "value" in str(text_sdf.schema)
True
```

New in version 2.0.

class pyspark.sql.streaming.**DataStreamWriter**(*df*) [source]

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

> **Note:** Experimental.

New in version 2.0.

format(*source*) [source]

Specifies the underlying output data source.

> **Note:** Experimental.

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

```
>>> writer = sdf.writeStream.format('json')
```

New in version 2.0.

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

Adds an output option for the underlying data source.

> **Note:** Experimental.

New in version 2.0.

options(***options*) [source]

Adds output options for the underlying data source.

> **Note:** Experimental.

New in version 2.0.

outputMode(*outputMode*) [source]

Specifies how data of a streaming DataFrame/Dataset is written to a streaming sink.

Options include:

- *append*:Only the new rows in the streaming DataFrame/Dataset will be written to the sink
- *complete*:All the rows in the streaming DataFrame/Dataset will be written to the sink every time these is some updates

Note: Experimental.

```
>>> writer = sdf.writeStream.outputMode('append')
```

New in version 2.0.

partitionBy(**cols*) [source]

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

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

Note: Experimental.

Parameters: **cols** – name of columns

New in version 2.0.

queryName(*queryName*) [source]

Specifies the name of the **StreamingQuery** that can be started with **start()**. This name must be unique among all the currently active queries in the associated SparkSession.

Note: Experimental.

Parameters: **queryName** – unique name for the query

```
>>> writer = sdf.writeStream.queryName('streaming_query')
```

New in version 2.0.

start(*path=None*, *format=None*, *partitionBy=None*, *queryName=None*, ***options*) [source]

Streams the contents of the **DataFrame** to a data source.

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

Note: Experimental.

Parameters:
- **path** – the path in a Hadoop supported file system
- **format** –
  the format used to save
  - `append`: Append contents of this **DataFrame** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` (default case): Throw an exception if data already exists.
- **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
u'this_query'
>>> sq.stop()
>>> sq.isActive
False
>>> sq = sdf.writeStream.trigger(processingTime='5 seconds').start(
...     queryName='that_query', format='memory')
```

```
>>> sq.name
u'that_query'
>>> sq.isActive
True
>>> sq.stop()
```

New in version 2.0.

trigger(**args*, ***kwargs*) [source]

Set the trigger for the stream query. If this is not set it will run the query as fast as possible, which is equivalent to setting the trigger to `processingTime='0 seconds'`.

> **Note:** Experimental.

Parameters: **processingTime** – a processing time interval as a string, e.g. '5 seconds', '1 minute'.

```
>>> # trigger the query for execution every 5 seconds
>>> writer = sdf.writeStream.trigger(processingTime='5 seconds')
```

New in version 2.0.