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*, *hiveContext=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.
- **hiveContext** – 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.

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))
```

> **Note:** Experimental

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

New in version 1.3.

cache()

Persists with the default storage level (**MEMORY_ONLY_SER**).

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.

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().show()
+-----+----+-----+
| name| age|count|
+-----+----+-----+
| null|   2|    1|
|Alice|null|    1|
|  Bob|   5|    1|
|  Bob|null|    1|
| null|   5|    1|
| null|null|    2|
|Alice|   2|    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),
>>> 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*)

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),
>>> 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.

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.

flatMap(*f*)

Returns a new **RDD** by first applying the `f` function to each **Row**, and then flattening the results.

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

```
>>> df.flatMap(lambda p: p.name).collect()
[u'A', u'l', u'i', u'c', u'e', u'B', u'o', u'b']
```

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

New in version 1.3.

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

New in version 1.3.

head(*n=None*)

Returns the first n rows.

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.

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

Inserts the contents of this **DataFrame** into the specified table.

Note: Deprecated in 1.4, use **DataFrameWriter.insertInto()** instead.

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.

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

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

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

Parameters:
- **other** – Right side of the join
- **on** – a string for 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 string 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*.

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

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

```
>>> cond = [df.name == df3.name, df.age == df3.age]
```

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

```
>>> 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.

map(*f*)

Returns a new **RDD** by applying a the `f` function to each **Row**.

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

```
>>> df.map(lambda p: p.name).collect()
[u'Alice', u'Bob']
```

New in version 1.3.

mapPartitions(*f*, *preservesPartitioning=False*)

Returns a new **RDD** by applying the `f` function to each partition.

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

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 4)
>>> def f(iterator): yield 1
>>> rdd.mapPartitions(f).sum()
4
```

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_SER**).

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.

registerAsTable(*name*)

Note: Deprecated in 1.4, use **registerTempTable()** instead.

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 = sqlContext.sql("select * from people")
>>> sorted(df.collect()) == sorted(df2.collect())
True
```

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.unionAll(df).repartition("age")
>>> data.show()
+---+-----+
|age| name|
+---+-----+
|  2|Alice|
|  2|Alice|
```

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

save(*path=None*, *source=None*, *mode='error'*, ***options*)

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

> **Note:** Deprecated in 1.4, use **DataFrameWriter.save()** instead.

New in version 1.3.

saveAsParquetFile(*path*)

Saves the contents as a Parquet file, preserving the schema.

> **Note:** Deprecated in 1.4, use **DataFrameWriter.parquet()** instead.

saveAsTable(*tableName*, *source=None*, *mode='error'*, ***options*)

Saves the contents of this **DataFrame** to a data source as a table.

> **Note:** Deprecated in 1.4, use **DataFrameWriter.saveAsTable()** instead.

schema

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

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

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.

toPandas()

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

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.

unionAll(*other*)

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

This is equivalent to *UNION ALL* in SQL.

New in version 1.3.

unpersist(*blocking=True*)

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

New in version 1.3.

where(*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.

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 **DataFrame** out into external storage.

Returns: **DataFrameWriter**

New in version 1.4.

class pyspark.sql.**GroupedData**(*jdf*, *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*.