

Big Data Applications

Contents

- [Linux](#)

Introduction

Pre-requisites

- Be comfortable with basic Python. Familiarity with Numpy and Pandas is beneficial but not required.
- Have an environment ready for the several hands-on exercises (e.g., own laptop or cloud resource)

Syllabus

Something broken?

If you encounter any problem or bug in these materials, please remember to add an issue to the [course repo](#), explaining the problem and, potentially, its solution. By doing this, you will improve the instructions for future users. 

Installation Instructions

Introduction

This document contains instructions for installation of the software we'll be using during the course. If you want to follow the training on your own machines, then please complete these instructions.

If you encounter any problem during installation and you manage to solve them, please remember to add an issue, explaining the problem and solution. By doing this you will be helping to improve the instructions for future users! :tada:

What we're installing

- the [Python](#) programming language (version [3.7](#) or greater)
- [git](#) for version control
- your favourite text editor
- [Apache Spark](#)
- [Jupyter notebooks](#)

Please ensure that you have a computer or a cloud-based workbench with all of these installed.

Linux

Package Manager

Linux users should be able to use their package manager to install all of the software from this page.

However note that if you are running an older **Linux** distribution you may get older versions with different look and features. We target [Ubuntu 21.10](#) for the sake of homogeneity.

💡 Tip

We recommend to upgrade the system and get the latest security packages: `sudo apt-get upgrade`

Please do not forget to restart after the upgrade finishes.

You can test the **Python** version by issuing:

```
python3 --version
```

You should get an output similar to the following:

```
Python 3.9.7
```

Python via package manager

Recent versions of **Ubuntu** come with mostly up to date versions of all needed packages.

The version of **IPython** might be slightly out of date. Thus, you may wish to upgrade this using **pip**.

You should ensure that the following packages are installed using **apt-get**:

- `python3-pip`
- `jupyter`
- `ipython3`

```
sudo apt-update  
sudo apt-get install -y gcc make perl python3-pip jupyter ipython3
```

You can test the installation by issuing:

```
pip --version
```

You should get an output similar to the following:

```
pip 20.3.4 from /usr/lib/python3/dist-packages/pip (python 3.9)
```

Editor

You have many different text editors suitable for programming at your fingertips. Here is an opinionated list of editors in case you do not already have a favourite:

- [Visual Studio Code](#)
- [Atom](#)
- [Neovim](#)
- [Vim](#)

Apache Spark

TL;DR

You can skip the whole [Step-by-step Setup section](#) by issuing the following commands in a terminal.

```
cd ~
wget https://dlcdn.apache.org/spark/spark-3.2.1/spark-3.2.1-bin-hadoop3.2.tgz
sudo tar -xvzf spark-3.2.1-bin-hadoop3.2.tgz
sudo apt-get install openjdk-8-jre -y
echo 'export SPARK_HOME=~/spark-3.2.1-bin-hadoop3.2' >> ~/.bashrc
echo 'export PATH=$SPARK_HOME/bin:$PATH' >> ~/.bashrc
echo 'export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH' >> ~/.bashrc
echo 'export PYSPARK_DRIVER_PYTHON=jupyter' >> ~/.bashrc
echo 'export PYSPARK_DRIVER_PYTHON_OPTS=notebook' >> ~/.bashrc
echo 'export PYSPARK_PYTHON=python3.9' >> ~/.bashrc
echo 'export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64' >> ~/.bashrc
exec bash
jupyter notebook --generate-config
sed -i '/# c.NotebookApp.use_redirect_file/s/$*True/False/g' ~/.jupyter/jupyter_notebook_config.py
sed -i '/c.NotebookApp.use_redirect_file/s/^#*\s*/g' ~/.jupyter/jupyter_notebook_config.py
```

Step-by-step Setup

We will setup [Apache Spark](#) now. You need to open the [downloads](#) page, and download a spark distribution.

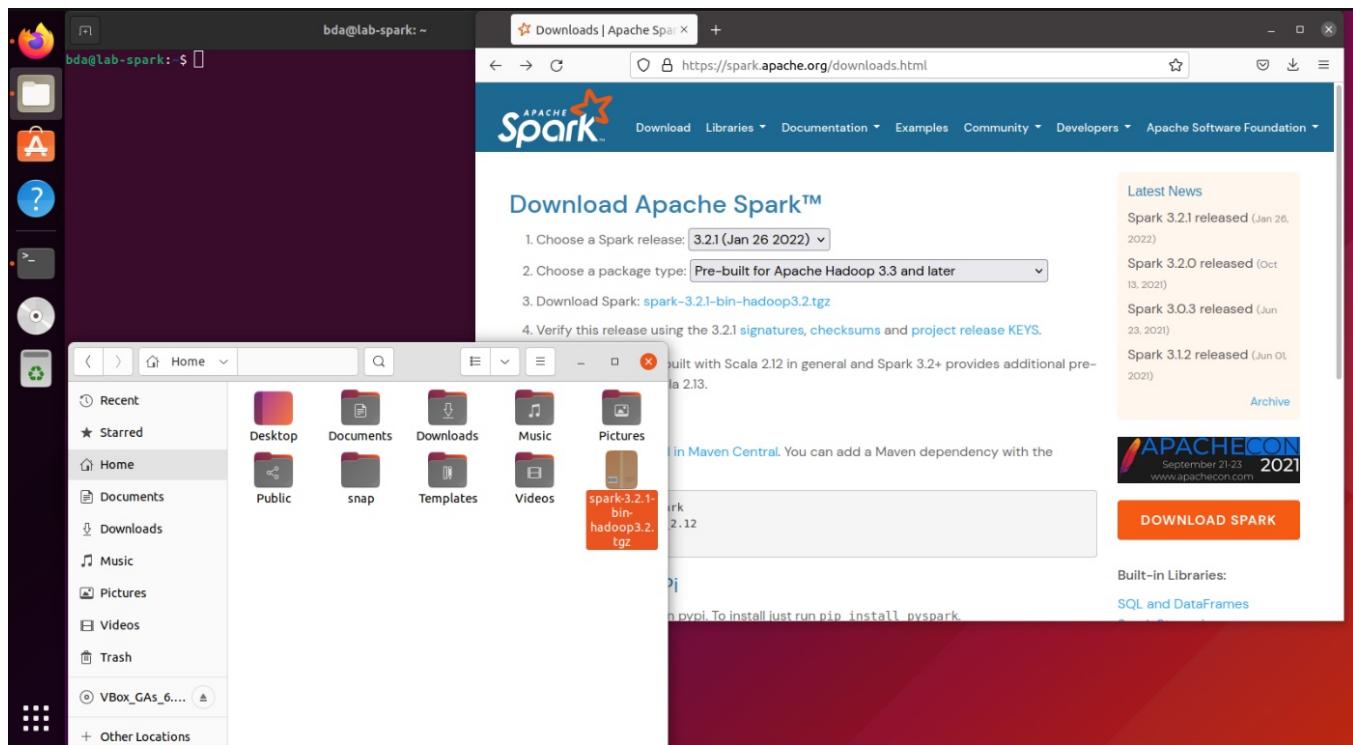
Download Apache Spark

We suggest to choose the same options as on the screenshot below. If you see a newer version is available, please feel free to choose it.

The screenshot shows the Apache Spark website's download section. At the top, there's a navigation bar with links for Download, Libraries, Documentation, Examples, Community, Developers, and the Apache Software Foundation. Below the navigation, the main heading is "Download Apache Spark™". There are four numbered steps: 1. Choose a Spark release: a dropdown menu set to "3.2.1 (Jan 26 2022)". 2. Choose a package type: a dropdown menu set to "Pre-built for Apache Hadoop 3.3 and later". 3. Download Spark: a link to "spark-3.2.1-bin-hadoop3.2.tgz". 4. Verify this release using the 3.2.1 signatures, checksums and project release KEYS. A note below says "Note that Spark 3 is pre-built with Scala 2.12 in general and Spark 3.2+ provides additional pre-built distribution with Scala 2.13." On the right side, there's a "Latest News" sidebar with entries for Spark 3.2.1, 3.2.0, 3.0.3, and 3.1.2 releases, each with a date. At the bottom left, there's a "Image Link" button.

Tip

We want the package in the right location, so open the file explorer and place it into your home folder.



Then go to your command line and issue the following to unzip the downloaded file:

```
sudo tar -xzvf spark-3.2.1-bin-hadoop3.2.tgz
```

Important

The above command assumes you downloaded version 3.2.1 with hadoop 3.2 binaries. Please amend it as needed.

Setup JRE

Next step is installing a JRE (Java Runtime Engine), so that we can use the PySpark shell (or submit a job to a cluster):

```
sudo apt-get install openjdk-8-jre -y
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/
```

Configure PySpark

We need to define several environment variables to let Spark find `Python` and use Jupyter notebooks when initiating the PySpark shell:

```
echo 'export SPARK_HOME=~/spark-3.2.1-bin-hadoop3.2' >> ~/.bashrc
echo 'export PATH=$SPARK_HOME/bin:$PATH' >> ~/.bashrc
echo 'export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH' >> ~/.bashrc
echo 'export PYSPARK_DRIVER_PYTHON=jupyter' >> ~/.bashrc
echo 'export PYSPARK_DRIVER_PYTHON_OPTS=notebook' >> ~/.bashrc
echo 'export PYSPARK_PYTHON=python3.9' >> ~/.bashrc
```

Note

PySpark requires the same minor version of Python in both driver and workers. This is why we specify what exact version to use in the `PYSPARK_PYTHON` variable.

! Important

Please do not forget to reload your interactive shell session. [1]

Workaround for Jupyter Notebooks in Linux

If you start a PySpark shell (i.e., `pyspark`) and get a “Access to the file was denied after starting...”, then you could try the following workaround. [2]

```
jupyter notebook --generate-config  
sed -i '/# c.NotebookApp.use_redirect_file/s/$*True/False/g' ~/.jupyter/jupyter_notebook_config.py  
sed -i '/c.NotebookApp.use_redirect_file/s/^#\*\s*//g' ~/.jupyter/jupyter_notebook_config.py
```

[1] <https://www.delftstack.com/howto/linux/reload-bashrc/>

[2] <https://github.com/jupyter/notebook/issues/4353#issuecomment-570564277>

Introduction to RDD

In this first session, we will introduce what is a [Resilient Distributed Dataset \(RDD\)](#). Please refer to the original paper for in-depth details [1].

To understand how Spark works, we need to understand the essence of RDD. Long story short, an RDD **represents a fault-tolerant collection of elements partitioned across the nodes of a cluster that can be operated in parallel**. We will chop the last sentence into pieces now.

Processing Logic Expressed as RDD Operations

We use RDDs to express a certain processic logic we want to apply to a dataset. For example, finding the average number of days required to recover from a certain disease.

Then, Spark makes its magic to **schedule** and **execute** our data processing logic on a distributed fashion.

If we look behind the scenes, the Spark runtime requires to determine the order of execution of RDDs, and make sure the execution is fault-tolerant. It uses the following pieces of information for the aforementioned purposes:

- Lineage
 - Dependencies on parent RDDs
- Fault-tolerance
 - The partitions that makes the whole dataset: used to execute in **parallel** to speed up the computation with **executors**.
 - The function for computing all the rows in the dataset: provided by users (i.e., “you”). Each **executor** in the cluster execute this function against each row in each partition.

With the above information, Spark can reproduce the RDD in case of failure scenarios.

Quick Demo

Prepare a Jupyter Kernel

You will need to prepare a Jupyter kernel in order to run this notebook on your local environment.

i Note

This course relies on [Poetry](#) as package manager. Commands below can might slightly differ if you are using a different strategy (e.g., [Pipenv](#) or [Conda](#), just to mention a few).

```
poetry shell  
ipython kernel install --name "bda-labs" --user
```

► Details

Initialize Spark

We need to do some preparation first. In the next snippet, we will import the [SparkContext](#) and [SparkConf](#) classes.

Now we proceed to create the **configuration** for our application and a **context** which tells Spark how to access the execution environment (local or a cluster).

```
from pyspark import SparkContext, SparkConf

# we use "local" to indicate we're running in local mode
conf = SparkConf().setAppName("intro-to-rdd").setMaster("local")
sc = SparkContext(conf=conf)
```

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform
(file:/opt/hostedtoolcache/Python/3.9.18/x64/lib/python3.9/site-
packages/pyspark/jars/spark-unsafe_2.12-3.2.1.jar) to constructor
java.nio.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of
org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective
access operations
WARNING: All illegal access operations will be denied in a future release
```

```
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
```

```
23/09/13 14:03:20 WARN NativeCodeLoader: Unable to load native-hadoop library for
your platform... using builtin-java classes where applicable
```

SparkConf

We triggered a couple of actions with the above code. To start with, we created a [SparkConfig](#) object and chained two important configurations: - The Spark application name. - The URL where the [master node](#) can be located. **Setting it to "local" is a special configuration to indicate we are running on the same host where this code is being executed.**

Note: it is important to notice that Spark runs on a JVM. When a [SparkConf](#) is created, it will load values from `spark.*` Java system properties. If we make such properties available to the process running Spark, then they would be picked up.

Let's inspect a couple of common Spark configuration properties:

```
print(f"spark.app.name: {conf.get('spark.app.name')}")
print(f"spark.master: {conf.get('spark.master')}")
print(f"spark.home: {conf.get('spark.home')}")
```

```
spark.app.name: intro-to-rdd
spark.master: local
spark.home: None
```

So far so good. We get the same value we chained to the [SparkConf](#) object before.

Homework/self-research:

- Why don't we have a value for `spark.home`?
- Should we care?

Should we want to know all the properties available, then the following snippet might help.

```
for p in sorted(conf.getAll(), key=lambda p: p[0]):
    print(p)
```

```
('spark.app.name', 'intro-to-rdd')
('spark.master', 'local')
```

SparkContext

`SparkContext` is at the heart of Spark. It is the main entry point for Spark that represents the connection of the “driver program” to a Spark cluster. **You must get familiarized with it.**

We can pass several parameters to it, from which two of them are mandatory: `master` and `appName`. In our case, we passed such information wrapped into a `SparkConf` object.

Attention

- `SparkContext` is **always** created on the driver and **it is not serialized**. Thus, it cannot be shipped to workers.
- We can have one `SparkContext` per application at most. Try to run the cell where we created the `SparkContext` and see what happens!

Note

Actually, only one `SparkContext` can be active per JVM. If we want to create a new one, we must call the `stop()` method to our existing `SparkContext` object first.

Another consequence of passing a `SparkConf` object to the `SparkContext` constructor is **immutable configuration**. The `SparkConf` object is cloned and can no longer be modified. Let's see it in action:

```
# Is our SparkConf object the same we get from SparkContext?
print(f"conf == sc.getConf() --> {conf == sc.getConf()}")
```

```
conf == sc.getConf() --> False
```

Define an RDD

Let's start by defining a very simple RDD. We can think of it as data points indicating the recovery days for a certain disease.

```
recovery_days_per_disease = [("disease_1", [3, 6, 9, 10]), ("disease_2", [11, 11, 10, 9])]
rdd = sc.parallelize(recovery_days_per_disease)
```

Now we can operate on it. For example, we can compute the average.

```
rdd.mapValues(lambda x: sum(x) / len(x)).collect()
```

```
[Stage 0:> (0 + 1) / 1]
```

```
[('disease_1', 7.0), ('disease_2', 10.25)]
```

References

^[1] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 15–28. 2012.

Introduction to the DataFrame API

In this section, we will introduce the [DataFrame and Dataset APIs](#).

We will use a small subset from the [Record Linkage Comparison Data Set](#), borrowed from UC Irvine Machine Learning Repository. It consists of several CSV files with match scores for patients in a German hospital, but we will use only one of them for the sake of simplicity. Please consult [\[1\]](#) and [\[2\]](#) for more details regarding the data sets and research.

Setup

- Setup a `SparkSession` to work with the Dataset and DataFrame API
- Unzip the `scores.zip` file located under `data` folder.

```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName("intro-to-df").setMaster("local")
sc = SparkContext(conf=conf)
# Avoid polluting the console with warning messages
sc.setLogLevel("ERROR")
```

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform
(file:/opt/hostedtoolcache/Python/3.9.18/x64/lib/python3.9/site-
packages/pyspark/jars/spark-unsafe_2.12-3.2.1.jar) to constructor
java.nio.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of
org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective
access operations
WARNING: All illegal access operations will be denied in a future release
```

```
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
23/09/13 14:03:26 WARN NativeCodeLoader: Unable to load native-hadoop library for
your platform... using builtin-java classes where applicable
```

Create a SparkSession to work with the DataFrame API

```
from pyspark.sql import SparkSession
spark = SparkSession(sc)
```

```
help(SparkSession)
```

```
Help on class SparkSession in module pyspark.sql.session:

class SparkSession(pyspark.sql.pandas.conversion.SparkConversionMixin)
|   SparkSession(sparkContext, jsparkSession=None)
|
|   The entry point to programming Spark with the Dataset and DataFrame API.
|
|   A SparkSession can be used create :class:`DataFrame`, register
:class:`DataFrame` as
|       tables, execute SQL over tables, cache tables, and read parquet files.
|       To create a :class:`SparkSession`, use the following builder pattern:
|
|       .. autoattribute:: builder
|          :annotation:
|
| Examples
| -----
>>> spark = SparkSession.builder \
...     .master("local") \
...     .appName("Word Count") \
...     .config("spark.some.config.option", "some-value") \
...     .getOrCreate()

>>> from datetime import datetime
>>> from pyspark.sql import Row
>>> spark = SparkSession(sc)
>>> allTypes = sc.parallelize([Row(i=1, s="string", d=1.0, l=1,
...     b=True, list=[1, 2, 3], dict={"s": 0}, row=Row(a=1),
...     time=datetime(2014, 8, 1, 14, 1, 5))])
>>> df = allTypes.toDF()
>>> df.createOrReplaceTempView("allTypes")
```

```

| >>> spark.sql('select i+1, d+1, not b, list[1], dict["s"], time, row.a '
| | ...
| |     'from allTypes where b and i > 0').collect()
| | [Row((i + 1)=2, (d + 1)=2.0, (NOT b)=False, list[1]=2,           dict[s]=0,
| | time=datetime.datetime(2014, 8, 1, 14, 1, 5), a=1]
| |   >>> df.rdd.map(lambda x: (x.i, x.s, x.d, x.l, x.b, x.time, x.row.a,
| | x.list)).collect()
| | [(1, 'string', 1.0, 1, True, datetime.datetime(2014, 8, 1, 14, 1, 5), 1, [1, 2,
| | 3])]

| Method resolution order:
|     SparkSession
|     pyspark.sql.pandas.conversion.SparkConversionMixin
|     builtins.object

| Methods defined here:

|     __enter__(self)
|         Enable 'with SparkSession.builder(...).getOrCreate() as session: app'
syntax.

|         .. versionadded:: 2.0

|     __exit__(self, exc_type, exc_val, exc_tb)
|         Enable 'with SparkSession.builder(...).getOrCreate() as session: app'
syntax.

|             Specifically stop the SparkSession on exit of the with block.

|         .. versionadded:: 2.0

|     __init__(self, sparkContext, jsparkSession=None)
|         Initialize self. See help(type(self)) for accurate signature.

|     createDataFrame(self, data, schema=None, samplingRatio=None, verifySchema=True)
|         Creates a :class:`DataFrame` from an :class:`RDD`, a list or a
:class:`pandas.DataFrame`.

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

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

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

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

|         .. versionadded:: 2.0.0

|         .. versionchanged:: 2.1.0
|             Added verifySchema.

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

```

```

    Returns
    -----
    :class:`DataFrame`


    Notes
    -----
    Usage with spark.sql.execution.arrow.pyspark.enabled=True is experimental.


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

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

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

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

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

    >>> spark.createDataFrame(df.toPandas()).collect() # doctest: +SKIP
    [Row(name='Alice', age=1)]
    >>> spark.createDataFrame(pandas.DataFrame([[1, 2]])).collect() # doctest:
+SKIP
[Row(0=1, 1=2)]

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

    newSession(self)
        Returns a new :class:`SparkSession` as new session, that has separate
SQLConf,
        registered temporary views and UDFs, but shared :class:`SparkContext` and
        table cache.

        .. versionadded:: 2.0

    | range(self, start, end=None, step=1, numPartitions=None)
    |     Create a :class:`DataFrame` with single :class:`pyspark.sql.types.LongType`-
    |     column named
    |     ``id``, containing elements in a range from ``start`` to ``end``-
    |     (exclusive) with
    |     step value ``step``.

        .. versionadded:: 2.0.0

    Parameters
    -----
    start : int
        the start value
    end : int, optional
        the end value (exclusive)
    step : int, optional
        the incremental step (default: 1)
    numPartitions : int, optional
        the number of partitions of the DataFrame

```

```
Returns
-----
:class:`DataFrame`


Examples
-----
>>> 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)]


sql(self, sqlQuery)
    Returns a :class:`DataFrame` representing the result of the given query.

    .. versionadded:: 2.0.0


Returns
-----
:class:`DataFrame`


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


stop(self)
    Stop the underlying :class:`SparkContext`.

    .. versionadded:: 2.0


table(self, tableName)
    Returns the specified table as a :class:`DataFrame`.

    .. versionadded:: 2.0.0


Returns
-----
:class:`DataFrame`


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


-----Class methods defined here:-----


getActiveSession() from builtins.type
    Returns the active :class:`SparkSession` for the current thread, returned
by the builder

    .. versionadded:: 3.0.0


Returns
-----
:class:`SparkSession`
    Spark session if an active session exists for the current thread


Examples
-----
>>> s = SparkSession.getActiveSession()
>>> l = [('Alice', 1)]
>>> rdd = s.sparkContext.parallelize(l)
>>> df = s.createDataFrame(rdd, ['name', 'age'])
>>> df.select("age").collect()
[Row(age=1)]


-----Readonly properties defined here:-----


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

    .. versionadded:: 2.0.0


Returns
-----
:class:`Catalog`
```

```
| 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 :class:`SparkContext`, if
any.
|
|     Returns
|     -----
|     :class:`pyspark.sql.conf.RuntimeConfig`
|
|     .. versionadded:: 2.0
|
read
|     Returns a :class:`DataFrameReader` that can be used to read data
in as a :class:`DataFrame`.
|
|     .. versionadded:: 2.0.0
|
|     Returns
|     -----
|     :class:`DataFrameReader`
|
readStream
|     Returns a :class:`DataStreamReader` that can be used to read data streams
as a streaming :class:`DataFrame`.
|
|     .. versionadded:: 2.0.0
|
|     Notes
|     -----
|     This API is evolving.
|
|     Returns
|     -----
|     :class:`DataStreamReader`
|
sparkContext
|     Returns the underlying :class:`SparkContext`.
|
|     .. versionadded:: 2.0
|
streams
|     Returns a :class:`StreamingQueryManager` that allows managing all the
:class:`StreamingQuery` instances active on `this` context.
|
|     .. versionadded:: 2.0.0
|
|     Notes
|     -----
|     This API is evolving.
|
|     Returns
|     -----
|     :class:`StreamingQueryManager`
|
udf
|     Returns a :class:`UDFRegistration` for UDF registration.
|
|     .. versionadded:: 2.0.0
|
|     Returns
|     -----
|     :class:`UDFRegistration`
|
version
|     The version of Spark on which this application is running.
|
|     .. versionadded:: 2.0
|
|----- Data and other attributes defined here:
|
Builder = <class 'pyspark.sql.session.SparkSession.Builder'>
|     Builder for :class:`SparkSession`.
|
builder = <pyspark.sql.session.SparkSession.Builder object>
|
|----- Data descriptors inherited from
pyspark.sql.pandas.conversion.SparkConversionMixin:
|
```

```
| __dict__  
|     dictionary for instance variables (if defined)  
|  
| __weakref__  
|     list of weak references to the object (if defined)
```

Unzip the scores file, if it was not done already

```
from os import path  
scores_zip = path.join("data", "scores.zip")  
scores_csv = path.join("data", "scores.csv")  
  
%set_env SCORES_ZIP=$scores_zip  
%set_env SCORES_CSV=$scores_csv
```

```
env: SCORES_ZIP=data/scores.zip  
env: SCORES_CSV=data/scores.csv
```

```
%%bash  
command -v unzip >/dev/null 2>&1 || { echo >&2 "unzip command is not installed.  
Aborting."; exit 1; }  
[[ -f "$SCORES_CSV" ]] && { echo "file data/$SCORES_CSV already exist. Skipping.";  
exit 0; }  
  
[[ -f "$SCORES_ZIP" ]] || { echo "file data/$SCORES_ZIP does not exist.  
Aborting."; exit 1; }  
  
echo "Unzip file $SCORES_ZIP"  
unzip "$SCORES_ZIP" -d data
```

```
Unzip file data/scores.zip
```

```
Archive: data/scores.zip
```

```
inflating: data/scores.csv
```

```
inflating: data/__MACOSX/.__scores.csv
```

```
! head "$SCORES_CSV"
```

```
"id_1","id_2","cmp_fname_c1","cmp_fname_c2","cmp_lname_c1","cmp_lname_c2","cmp_sex"  
,"cmp_bd","cmp_bm","cmp_by","cmp_plz","is_match"  
37291,53113,0,83333333333333,?,1,?,1,1,1,0,TRUE  
39086,47614,1,?,1,?,1,1,1,1,1,TRUE  
70031,70237,1,?,1,?,1,1,1,1,1,TRUE  
84795,97439,1,?,1,?,1,1,1,1,1,TRUE  
36950,42116,1,?,1,1,1,1,1,1,1,TRUE  
42413,48491,1,?,1,?,1,1,1,1,1,1,TRUE  
25965,64753,1,?,1,?,1,1,1,1,1,1,1,TRUE  
49451,90407,1,?,1,?,1,1,1,1,1,0,TRUE  
39932,40902,1,?,1,?,1,1,1,1,1,1,1,TRUE
```

Loading the Scores CSV file into a DataFrame

We are going to use the Reader API

```
help(spark.read)
```

```
Help on DataFrameReader in module pyspark.sql.readwriter object:

class DataFrameReader(OptionUtils)
|   DataFrameReader(spark)
|
|   Interface used to load a :class:`DataFrame` from external storage systems
|   (e.g. file systems, key-value stores, etc). Use :attr:`SparkSession.read`
|   to access this.
|
|   .. versionadded:: 1.4
```

```
| Method resolution order:
|     DataFrameReader
|     OptionUtils
|     builtins.object
|
| Methods defined here:
|
|     __init__(self, spark)
|         Initialize self. See help(type(self)) for accurate signature.
|
|     csv(self, path, schema=None, sep=None, encoding=None, quote=None, escape=None,
|          comment=None, header=None, inferSchema=None, ignoreLeadingWhiteSpace=None,
|          ignoreTrailingWhiteSpace=None, nullValue=None, nanValue=None, positiveInf=None,
|          negativeInf=None, dateFormat=None, timestampFormat=None, maxColumns=None,
|          maxCharsPerColumn=None, maxMalformedLogPerPartition=None, mode=None,
|          columnNameOfCorruptRecord=None, multiLine=None, charToEscapeQuoteEscaping=None,
|          samplingRatio=None, enforceSchema=None, emptyValue=None, locale=None, lineSep=None,
|          pathGlobFilter=None, recursiveFileLookup=None, modifiedBefore=None,
|          modifiedAfter=None, unescapedQuoteHandling=None)
|         Loads a CSV file and returns the result as a :class:`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``.
|
|     .. versionadded:: 2.0.0
|
| Parameters
| -----
|     path : str or list
|         string, or list of strings, for input path(s),
|         or RDD of Strings storing CSV rows.
|     schema : :class:`pyspark.sql.types.StructType` or str, optional
|         an optional :class:`pyspark.sql.types.StructType` for the input schema
|         or a DDL-formatted string (For example ``col0 INT, col1 DOUBLE``).
|
| Other Parameters
| -----
| Extra options
|     For the extra options, refer to
|     `Data Source Option <https://spark.apache.org/docs/latest/sql-data-
sources-csv.html#data-source-option>`_
|     in the version you use.
|
|     .. # noqa
|
| Examples
| -----
|     >>> df = spark.read.csv('python/test_support/sql/ages.csv')
|     >>> df.dtypes
|     [('c0', 'string'), ('c1', 'string')]
|     >>> rdd = sc.textFile('python/test_support/sql/ages.csv')
|     >>> df2 = spark.read.csv(rdd)
|     >>> df2.dtypes
|     [('c0', 'string'), ('c1', 'string')]
|
|     format(self, source)
|         Specifies the input data source format.
|
|     .. versionadded:: 1.4.0
|
| Parameters
| -----
|     source : str
|         string, name of the data source, e.g. 'json', 'parquet'.
|
| Examples
| -----
|     >>> df =
| spark.read.format('json').load('python/test_support/sql/people.json')
|     >>> df.dtypes
|     [('age', 'bigint'), ('name', 'string')]
|
|     jdbc(self, url, table, column=None, lowerBound=None, upperBound=None,
|           numPartitions=None, predicates=None, properties=None)
|         Construct a :class:`DataFrame` representing the database table named
|         ``table``
|         accessible via JDBC URL ``url`` and connection ``properties``.
|
|         Partitions of the table will be retrieved in parallel if either ``column``
|         or
|         ``predicates`` is specified. ``lowerBound``, ``upperBound`` and
|         ``numPartitions``
|         is needed when ``column`` is specified.
|
```

```

|     If both ``column`` and ``predicates`` are specified, ``column`` will be
| used.
|     .. versionadded:: 1.4.0
|
| Parameters
| -----
|     table : str
|         the name of the table
|     column : str, optional
|         alias of ``partitionColumn`` option. Refer to ``partitionColumn`` in
|         `Data Source Option <https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html#data-source-option>`_
|         in the version you use.
|     predicates : list, optional
|         a list of expressions suitable for inclusion in WHERE clauses;
|         each one defines one partition of the :class:`DataFrame`
|     properties : dict, optional
|         a dictionary of JDBC database connection arguments. Normally at
|         least properties "user" and "password" with their corresponding values.
|         For example { 'user' : 'SYSTEM', 'password' : 'mypassword' }
|
| Other Parameters
| -----
| Extra options
|     For the extra options, refer to
|     `Data Source Option <https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html#data-source-option>`_
|     in the version you use.
|
|     .. # noqa
|
| Notes
| -----
| Don't create too many partitions in parallel on a large cluster;
| otherwise Spark might crash your external database systems.
|
| Returns
| -----
|     :class:`DataFrame`
|
|     json(self, path, schema=None, primitivesAsString=None, prefersDecimal=None,
|     allowComments=None, allowUnquotedFieldNames=None, allowSingleQuotes=None,
|     allowNumericLeadingZero=None, allowBackslashEscapingAnyCharacter=None, mode=None,
|     columnNameOfCorruptRecord=None, dateFormat=None, timestampFormat=None,
|     multiLine=None, allowUnquotedControlChars=None, lineSep=None, samplingRatio=None,
|     dropFieldIfAllNull=None, encoding=None, locale=None, pathGlobFilter=None,
|     recursiveFileLookup=None, allowNonNumericNumbers=None, modifiedBefore=None,
|     modifiedAfter=None)
|         Loads JSON files and returns the results as a :class:`DataFrame`.
|
|         `JSON Lines <http://jsonlines.org/>`_ (newline-delimited JSON) is supported
|         by default.
|         For JSON (one record per file), set the ``multiLine`` parameter to
|         ``true``.
|
|         If the ``schema`` parameter is not specified, this function goes
|         through the input once to determine the input schema.
|
|     .. versionadded:: 1.4.0
|
| Parameters
| -----
|     path : str, list or :class:`RDD`
|         string represents path to the JSON dataset, or a list of paths,
|         or RDD of Strings storing JSON objects.
|     schema : :class:`pyspark.sql.types.StructType` or str, optional
|         an optional :class:`pyspark.sql.types.StructType` for the input schema
|     or
|         a DDL-formatted string (For example ``col0 INT, col1 DOUBLE``).
|
| Other Parameters
| -----
| Extra options
|     For the extra options, refer to
|     `Data Source Option <https://spark.apache.org/docs/latest/sql-data-sources-json.html#data-source-option>`_
|     in the version you use.
|
|     .. # noqa
|
| Examples
| -----
|>>> 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')]

load(self, path=None, format=None, schema=None, **options)
    Loads data from a data source and returns it as a :class:`DataFrame`.

    .. versionadded:: 1.4.0

    Parameters
    -----
    path : str or list, optional
        optional string or a list of string for file-system backed data
sources.
    format : str, optional
        optional string for format of the data source. Default to 'parquet'.
    schema : :class:`pyspark.sql.types.StructType` or str, optional
        optional :class:`pyspark.sql.types.StructType` for the input schema
        or a DDL-formatted string (For example ```col0 INT, col1 DOUBLE``').
    **options : dict
        all other string options

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

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

option(self, key, value)
    Adds an input option for the underlying data source.

    .. versionadded:: 1.5

options(self, **options)
    Adds input options for the underlying data source.

    .. versionadded:: 1.4

orc(self, path, mergeSchema=None, pathGlobFilter=None,
recursiveFileLookup=None, modifiedBefore=None, modifiedAfter=None)
    Loads ORC files, returning the result as a :class:`DataFrame`.

    .. versionadded:: 1.5.0

    Parameters
    -----
    path : str or list

    Other Parameters
    -----
    Extra options
        For the extra options, refer to
        `Data Source Option <https://spark.apache.org/docs/latest/sql-data-sources-orc.html#data-source-option>`_
        in the version you use.

    ... # noqa

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

parquet(self, *paths, **options)
    Loads Parquet files, returning the result as a :class:`DataFrame`.

    .. versionadded:: 1.4.0

    Parameters
    -----
    paths : str

    Other Parameters
    -----
    **options
        For the extra options, refer to
        `Data Source Option <https://spark.apache.org/docs/latest/sql-data-sources-parquet.html#data-source-option>`_
```

```

    in the version you use.

    ... # noqa

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

schema(self, 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.

.. versionadded:: 1.4.0

Parameters
-----
schema : :class:`pyspark.sql.types.StructType` or str
    a :class:`pyspark.sql.types.StructType` object or a DDL-formatted
string
    (For example ``col0 INT, col1 DOUBLE``).

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

table(self, tableName)
    Returns the specified table as a :class:`DataFrame`.

.. versionadded:: 1.4.0

Parameters
-----
tableName : str
    string, name of the table.

Examples
-----
>>> 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')]

text(self, paths, wholertext=False, lineSep=None, pathGlobFilter=None,
recursiveFileLookup=None, modifiedBefore=None, modifiedAfter=None)
    Loads text files and returns a :class:`DataFrame` whose schema starts with
a
    string column named "value", and followed by partitioned columns if there
are any.
    The text files must be encoded as UTF-8.

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

.. versionadded:: 1.6.0

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

Other Parameters
-----
Extra options
    For the extra options, refer to
    `Data Source Option <https://spark.apache.org/docs/latest/sql-data-sources-text.html#data-source-option>`_
    in the version you use.

    ... # noqa

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

-----
Data descriptors inherited from OptionUtils:

```

```
| __dict__  
|     dictionary for instance variables (if defined)  
| __weakref__  
|     list of weak references to the object (if defined)
```

```
help(spark.read.csv)
```

```
Help on method csv in module pyspark.sql.readwriter:  
  
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, dateFormats=None, timestampFormat=None, maxColumns=None,  
maxCharsPerColumn=None, maxMalformedLogPerPartition=None, mode=None,  
columnNameOfCorruptRecord=None, multiLine=None, charToEscapeQuoteEscaping=None,  
samplingRatio=None, enforceSchema=None, emptyValue=None, locale=None, lineSep=None,  
pathGlobFilter=None, recursiveFileLookup=None, modifiedBefore=None,  
modifiedAfter=None, unescapedQuoteHandling=None) method of  
pyspark.sql.readwriter.DataFrameReader instance  
    Loads a CSV file and returns the result as a :class:`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``.  
  
.. versionadded:: 2.0.0  
  
Parameters  
-----  
path : str or list  
    string, or list of strings, for input path(s),  
    or RDD of Strings storing CSV rows.  
schema : :class:`pyspark.sql.types.StructType` or str, optional  
    an optional :class:`pyspark.sql.types.StructType` for the input schema  
    or a DDL-formatted string (For example ``col0 INT, col1 DOUBLE``).  
  
Other Parameters  
-----  
Extra options  
For the extra options, refer to  
`Data Source Option <https://spark.apache.org/docs/latest/sql-data-sources-csv.html#data-source-option>`_  
in the version you use.  
  
.. # noqa  
  
Examples  
-----  
>>> df = spark.read.csv('python/test_support/sql/ages.csv')  
>>> df.dtypes  
[('_c0', 'string'), ('_c1', 'string')]  
>>> rdd = sc.textFile('python/test_support/sql/ages.csv')  
>>> df2 = spark.read.csv(rdd)  
>>> df2.dtypes  
[('_c0', 'string'), ('_c1', 'string')]
```

```
scores = spark.read.csv(scores_csv)
```

```
scores
```

```
DataFrame[_c0: string, _c1: string, _c2: string, _c3: string, _c4: string, _c5:  
string, _c6: string, _c7: string, _c8: string, _c9: string, _c10: string, _c11:  
string]
```

```
help(scores.show)
```

```

Help on method show in module pyspark.sql.dataframe:

show(n=20, truncate=True, vertical=False) method of pyspark.sql.dataframe.DataFrame
instance
    Prints the first ``n`` rows to the console.

.. versionadded:: 1.3.0

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

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

```

We can look at the head of the DataFrame calling the `show` method.

```
scores.show()
```

Can anyone spot what's wrong with the above data?

- Question marks
- Column names
- `Float` and `Int` in the same column

Let's check the schema of our DataFrame

```
help(scores.printSchema)
```

```

Help on method printSchema in module pyspark.sql.dataframe:

printSchema() method of pyspark.sql.dataframe.DataFrame instance
    Prints out the schema in the tree format.

.. versionadded:: 1.3.0

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

```

```
scores.printSchema()
```

```

root
|-- _c0: string (nullable = true)
|-- _c1: string (nullable = true)
|-- _c2: string (nullable = true)
|-- _c3: string (nullable = true)
|-- _c4: string (nullable = true)
|-- _c5: string (nullable = true)
|-- _c6: string (nullable = true)
|-- _c7: string (nullable = true)
|-- _c8: string (nullable = true)
|-- _c9: string (nullable = true)
|-- _c10: string (nullable = true)
|-- _c11: string (nullable = true)

```

Why everythin is a **String**?

Managing Schema and Null Values

```

scores_df = (
    spark.read
        .option("header", "true")
        .option("nullValue", "?")
        .option("inferSchema", "true")
        .csv(scores_csv)
)

```

[Stage 2:> (0 + 1) / 1]

```
scores_df.printSchema()
```

```

root
|-- id_1: integer (nullable = true)
|-- id_2: integer (nullable = true)
|-- cmp_fname_c1: double (nullable = true)
|-- cmp_fname_c2: double (nullable = true)
|-- cmp_lname_c1: double (nullable = true)
|-- cmp_lname_c2: double (nullable = true)
|-- cmp_sex: integer (nullable = true)
|-- cmp_bd: integer (nullable = true)
|-- cmp_bm: integer (nullable = true)
|-- cmp_by: integer (nullable = true)
|-- cmp_plz: integer (nullable = true)
|-- is_match: boolean (nullable = true)

```

```
scores_df.show(5)
```

id_1	id_2	cmp_fname_c1	cmp_fname_c2	cmp_lname_c1	cmp_lname_c2	cmp_sex	cmp_bd	cmp_bm	cmp_by	cmp_plz	is_match
37291	53113	0.8333333333333333		null	1.0	null				1	
1	1	0	true								
39086	47614		1.0	null	1.0	null				1	
1	1	1	true								
70031	70237		1.0	null	1.0	null				1	
1	1	1	true								
84795	97439		1.0	null	1.0	null				1	
1	1	1	true								
36950	42116		1.0	null	1.0	1.0				1	
1	1	1	true								

only showing top 5 rows

Transformations and Actions

Creating a DataFrame does not cause any distributed computation in the cluster. **A DataFrame is un data set representing an intermediate step in a computation.**

For operating data (in a distributed manner), we have two type of operations: **transformations** and **actions**:

- Transformations: lazy evaluation. They're not computed immediately, but they are recorded as a **lineage** for query plan optimization.
- Actions: distributed computation occurs after invoking an action

```
# how many?  
scores_df.count()
```

```
574913
```

We can use the `collect` action to return `Array` with all the `Row` objects in our DataFrame.

```
scores_df.collect()
```

[Stage 7:>

(0 + 1) / 1]


```

Row(id_1=35327, id_2=47055, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=50928, id_2=50929, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=86259, id_2=86260, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=31413, id_2=39918, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=25518, id_2=31391, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=18033, id_2=30504, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=87985, id_2=87990, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=22128, id_2=23052, cmp_fname_c1=1.0, cmp_fname_c2=1.0, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=28910, id_2=32363, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=90239, id_2=91588, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=24369, id_2=31627, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=27525, id_2=29593, cmp_fname_c1=1.0, cmp_fname_c2=1.0, cmp_lname_c1=1.0,
    cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=38456, id_2=44565, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0,
    cmp_lname_c2=1.0, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1,
    is_match=True),
Row(id_1=56197, id_2=88150, cmp_fname_c1=1.0, cmp_fname_c2=None,
    cmp_lname_c1=0.88888888888889, cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1,
    cmp_by=1, cmp_plz=0, is_match=True),
...

```

The **Array** will reside in local memory!!

Write to Disk

We are going to save the DataFrame into a different format: Parquet

```
scores_df.write.format("parquet").save("data/scores-parquet")
```

```
[Stage 8:> (0 + 1) / 1]
```

```
! ls data/scores-parquet
```

```
_SUCCESS part-00000-c264247c-4a23-4210-b148-b27a700fc207-c000.snappy.parquet
```

```
scores_parquet = spark.read.parquet("data/scores-parquet")
```

```
scores_parquet.printSchema()
```

```

root
|-- id_1: integer (nullable = true)
|-- id_2: integer (nullable = true)
|-- cmp_fname_c1: double (nullable = true)
|-- cmp_fname_c2: double (nullable = true)
|-- cmp_lname_c1: double (nullable = true)
|-- cmp_lname_c2: double (nullable = true)
|-- cmp_sex: integer (nullable = true)
|-- cmp_bd: integer (nullable = true)
|-- cmp_bm: integer (nullable = true)
|-- cmp_by: integer (nullable = true)
|-- cmp_plz: integer (nullable = true)
|-- is_match: boolean (nullable = true)

```

```
scores_parquet.show(5)
```

id_1	id_2	cmp_fname_c1	cmp_fname_c2	cmp_lname_c1	cmp_lname_c2	cmp_sex	cmp_bd	cmp_bm	cmp_by	cmp_plz	is_match
37291	53113	0.8333333333333333		null	1.0	null	1				
1	1	1	0	true							
39086	47614		1.0		null	1.0	null	1			
1	1	1	1	true							
70031	70237		1.0		null	1.0	null	1			
1	1	1	1	true							
84795	97439		1.0		null	1.0	null	1			
1	1	1	1	true							
36950	42116		1.0		null	1.0	1.0	1			
1	1	1	1	true							

only showing top 5 rows

Analyzing Data

All good for now, but we don't load data for the sake of i, we do it because we want to run some analysis.

- First two column are Integer IDs. There represent the patients that were matched in the record.
- The next nine column are numeric values (int and double). They represent match scores on different fields, such name, sex, birthday, and locations.
- The last column is a boolean value indicating whether or not the pair of patient records represented by the line was a match.

We could use this dataset to build a simple classifier that allows us to predict whether a record will be a match based on the values of the match scores for patient records.

Caching

Each time we process data (e.g., calling the `collect` method), Spark re-opens the file, parses the rows, and then execute the requested action. It does not matter if we have filtered the data and created a smaller set of record.

We can use the `cache` method to indicate to store the DataFrame in memory.

```
help(scores_df.cache)
```

```

Help on method cache in module pyspark.sql.dataframe:

cache() method of pyspark.sql.dataframe.DataFrame instance
    Persists the :class:`DataFrame` with the default storage level
    (`MEMORY_AND_DISK`).

    .. versionadded:: 1.3.0

    Notes
    -----
    The default storage level has changed to `MEMORY_AND_DISK` to match Scala in
    2.0.

```

Spark is in-memory only. Myth or misconception?

"Spill" Storage levels:

- MEMORY_AND_DISK
- MEMORY
- MEMORY_SER

```
scores_cached = scores_df.cache()
```

```
scores_cached.count()
```

```
[Stage 11:>
```

```
(0 + 1) / 1]
```

```
574913
```

```
scores_cached.take(10)
```

```
[Row(id_1=37291, id_2=53113, cmp_fname_c1=0.83333333333333, cmp_fname_c2=None, cmp_lname_c1=1.0, cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=0, is_match=True),  
 Row(id_1=39086, id_2=47614, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0, cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1, is_match=True),  
 Row(id_1=70031, id_2=70237, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0, cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1, is_match=True),  
 Row(id_1=84795, id_2=97439, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0, cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1, is_match=True),  
 Row(id_1=36950, id_2=42116, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0, cmp_lname_c2=1.0, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1, is_match=True),  
 Row(id_1=42413, id_2=48491, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0, cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1, is_match=True),  
 Row(id_1=25965, id_2=64753, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0, cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1, is_match=True),  
 Row(id_1=49451, id_2=90407, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0, cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=0, is_match=True),  
 Row(id_1=39932, id_2=40902, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0, cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1, is_match=True),  
 Row(id_1=46626, id_2=47940, cmp_fname_c1=1.0, cmp_fname_c2=None, cmp_lname_c1=1.0, cmp_lname_c2=None, cmp_sex=1, cmp_bd=1, cmp_bm=1, cmp_by=1, cmp_plz=1, is_match=True)]
```

Query Plan

```
scores_cached.explain()
```

```
== Physical Plan ==  
FileScan csv  
[id_1#56,id_2#57,cmp_fname_c1#58,cmp_fname_c2#59,cmp_lname_c1#60,cmp_lname_c2#61,cmp_p_sex#62,cmp_bd#63,cmp_bm#64,cmp_by#65,cmp_plz#66,is_match#67] Batched: false,  
DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 paths)  
[file:/home/runner/work/bda-course/bda-course/coursebook/modules/m2/data...],  
PartitionFilters: [], PushedFilters: [], ReadSchema:  
struct<id_1:int,id_2:int,cmp_fname_c1:double,cmp_fname_c2:double,cmp_lname_c1:double,cmp_lname_c2:double>
```

GroupBy + OrderBy

```
from pyspark.sql.functions import col  
scores_cached.groupBy("is_match").count().orderBy(col("count").desc()).show()
```

```
+-----+-----+
|is_match| count|
+-----+-----+
|  false|572820|
|  true | 2093|
+-----+-----+
```

Aggregation Functions

In addition to `count`, we can also compute more complex aggregation like sums, mins, maxes, means, and standard deviation. How? we use `agg` method of the DataFrame API.

```
from pyspark.sql.functions import avg, stddev

aggregated = scores_cached.agg(avg("cmp_sex"), stddev("cmp_sex"))

aggregated.explain()

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[], functions=[avg(cmp_sex#62), stddev_samp(cast(cmp_sex#62 as double))])
   +- Exchange SinglePartition, ENSURE_REQUIREMENTS, [id=#220]
      +- HashAggregate(keys=[], functions=[partial_avg(cmp_sex#62),
      partial_stddev_samp(cast(cmp_sex#62 as double))])
         +- InMemoryTableScan [cmp_sex#62]
            +- InMemoryRelation [id_1#56, id_2#57, cmp_fname_c1#58,
           cmp_fname_c2#59, cmp_lname_c1#60, cmp_lname_c2#61, cmp_sex#62, cmp_bd#63,
           cmp_bm#64, cmp_by#65, cmp_plz#66, is_match#67], StorageLevel(disk, memory,
           serialized, 1 replicas)
               +- FileScan csv
[id_1#56,id_2#57,cmp_fname_c1#58,cmp_fname_c2#59,cmp_lname_c1#60,cmp_lname_c2#61,cm
p_sex#62,cmp_bd#63,cmp_bm#64,cmp_by#65,cmp_plz#66,is_match#67] Batched: false,
DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 paths)
[file:/home/runner/work/bda-course/bda-course/coursebook/modules/m2/data...,
PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<id_1:int,id_2:int,cmp_fname_c1:double,cmp_fname_c2:double,cmp_lname_c1:double,cmp_lname_c2...]
```

```
aggregated.show()
```

```
+-----+-----+
|    avg(cmp_sex)|stddev_samp(cmp_sex)|
+-----+-----+
|0.9550923357099248| 0.20710152240504734|
+-----+-----+
```

SQL

ANSI 2003-compliant version or HiveQL.

```
scores_df.createOrReplaceTempView("scores")

# scores_cached.groupBy("is_match").count().orderBy(col("count").desc()).show()
spark.sql("""
    SELECT is_match, COUNT(*) cnt
    FROM scores
    GROUP BY is_match
    ORDER BY cnt DESC
""").show()
```

```
+-----+-----+
|is_match|  cnt|
+-----+-----+
|  false|572820|
|  true | 2093|
+-----+-----+
```

```
spark.sql("""
    SELECT is_match, COUNT(*) cnt
    FROM scores
    GROUP BY is_match
    ORDER BY cnt DESC
""").explain()
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [cnt#2165L DESC NULLS LAST], true, 0
   +- Exchange rangepartitioning(cnt#2165L DESC NULLS LAST, 200),
   ENSURE_REQUIREMENTS, [id=#330]
      +- HashAggregate(keys=[is_match#67], functions=[count(1)])
         +- Exchange hashpartitioning(is_match#67, 200), ENSURE_REQUIREMENTS,
   [id=#327]
      +- HashAggregate(keys=[is_match#67], functions=[partial_count(1)])
         +- InMemoryTableScan [is_match#67]
            +- InMemoryRelation [id_1#56, id_2#57, cmp_fname_c1#58,
   cmp_fname_c2#59, cmp_lname_c1#60, cmp_lname_c2#61, cmp_sex#62, cmp_bd#63,
   cmp_bm#64, cmp_by#65, cmp_plz#66, is_match#67], StorageLevel(disk, memory,
   serialized, 1 replicas)
               +- FileScan csv
[id_1#56,id_2#57,cmp_fname_c1#58,cmp_fname_c2#59,cmp_lname_c1#60,cmp_lname_c2#61,cm
p_sex#62,cmp_bd#63,cmp_bm#64,cmp_by#65,cmp_plz#66,is_match#67] Batched: false,
DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 paths)
[file:/home/runner/work/bda-course/bda-course/coursebook/modules/m2/dat...,
PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<id_1:int,id_2:int,cmp_fname_c1:double,cmp_fname_c2:double,cmp_lname_c1:double,cmp_lname_c2...
```

```
scores_cached.groupBy("is_match").count().orderBy(col("count").desc()).explain()
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [count#2364L DESC NULLS LAST], true, 0
   +- Exchange rangepartitioning(count#2364L DESC NULLS LAST, 200),
   ENSURE_REQUIREMENTS, [id=#350]
      +- HashAggregate(keys=[is_match#67], functions=[count(1)])
         +- Exchange hashpartitioning(is_match#67, 200), ENSURE_REQUIREMENTS,
   [id=#347]
      +- HashAggregate(keys=[is_match#67], functions=[partial_count(1)])
         +- InMemoryTableScan [is_match#67]
            +- InMemoryRelation [id_1#56, id_2#57, cmp_fname_c1#58,
   cmp_fname_c2#59, cmp_lname_c1#60, cmp_lname_c2#61, cmp_sex#62, cmp_bd#63,
   cmp_bm#64, cmp_by#65, cmp_plz#66, is_match#67], StorageLevel(disk, memory,
   serialized, 1 replicas)
               +- FileScan csv
[id_1#56,id_2#57,cmp_fname_c1#58,cmp_fname_c2#59,cmp_lname_c1#60,cmp_lname_c2#61,cm
p_sex#62,cmp_bd#63,cmp_bm#64,cmp_by#65,cmp_plz#66,is_match#67] Batched: false,
DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 paths)
[file:/home/runner/work/bda-course/bda-course/coursebook/modules/m2/dat...,
PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<id_1:int,id_2:int,cmp_fname_c1:double,cmp_fname_c2:double,cmp_lname_c1:double,cmp_lname_c2...
```

Should I use Spark SQL or the DataFrame API

Depends on the query.

Pandas is my friend!

required packages: `pandas` and `numpy`

- poetry add pandas numpy
- pip install pandas numpy

```
scores_pandas = scores_df.toPandas()
```

[Stage 24:>

(0 + 1) / 1]

```
scores_pandas.head()
```

	id_1	id_2	cmp_fname_c1	cmp_fname_c2	cmp_lname_c1	cmp_lname_c2	cmp_
0	37291	53113	0.833333	NaN	1.0	NaN	
1	39086	47614	1.000000	NaN	1.0	NaN	
2	70031	70237	1.000000	NaN	1.0	NaN	
3	84795	97439	1.000000	NaN	1.0	NaN	
4	36950	42116	1.000000	NaN	1.0	1.0	

```
scores_pandas.shape
```

```
(574913, 12)
```

References

- [1] Irene Schmidtmann, Gaël Hammer, Murat Sariyar, Aslıhan Gerhold-Ay, and Körperschaft des öffentlichen Rechts. Evaluation des krebsregisters nrw schwerpunkt record linkage. *Abschlußbericht vom*, 2009.
- [2] Murat Sariyar, Andreas Borg, and Klaus Pommerening. Controlling false match rates in record linkage using extreme value theory. *Journal of biomedical informatics*, 44(4):648–654, 2011.

By Carlos Montemuiño

© Copyright 2022.