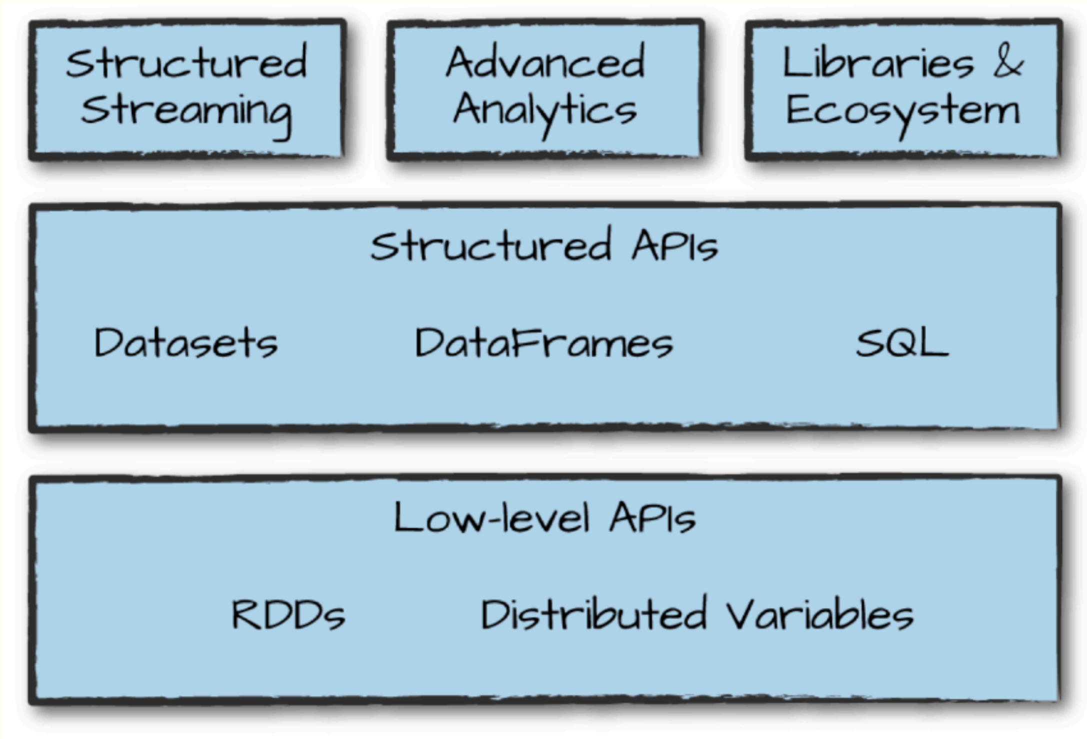
Apache Spark - PySpark

Srini Badri

Spark APIs

- Spark provide three high-level APIs for distributed query and processing:
  - Datasets
  - DataFrames
  - SQL (table)
- In addition, Spark provides low-level APIs in the form of:
  - RDDs
  - Distributed Variables



source: Spark: The Definitive Guide by Bill Chambers, Matei Zaharia

Spark APIs

	Java / Scala	Python / R
Dataset	Yes	No
DataFrame	Yes	Yes
SQL (table)	Yes	Yes
RDD	Yes	Yes

Spark Data Sources

- Files:
  - Text, CSV, JSON, HDFS, Parquet, Avro, etc.

- External Databases:
  - via JDBC:
    - MySQL, Postgres, MongoDB, Cassandra, etc.
  - via connector for specific databases:
    - Cassandra, MongoDB, Neo4J, etc.

Spark - Database Connectors

- MongoDB:
  - https://www.mongodb.com/docs/spark-connector/v10.2/getting-started/

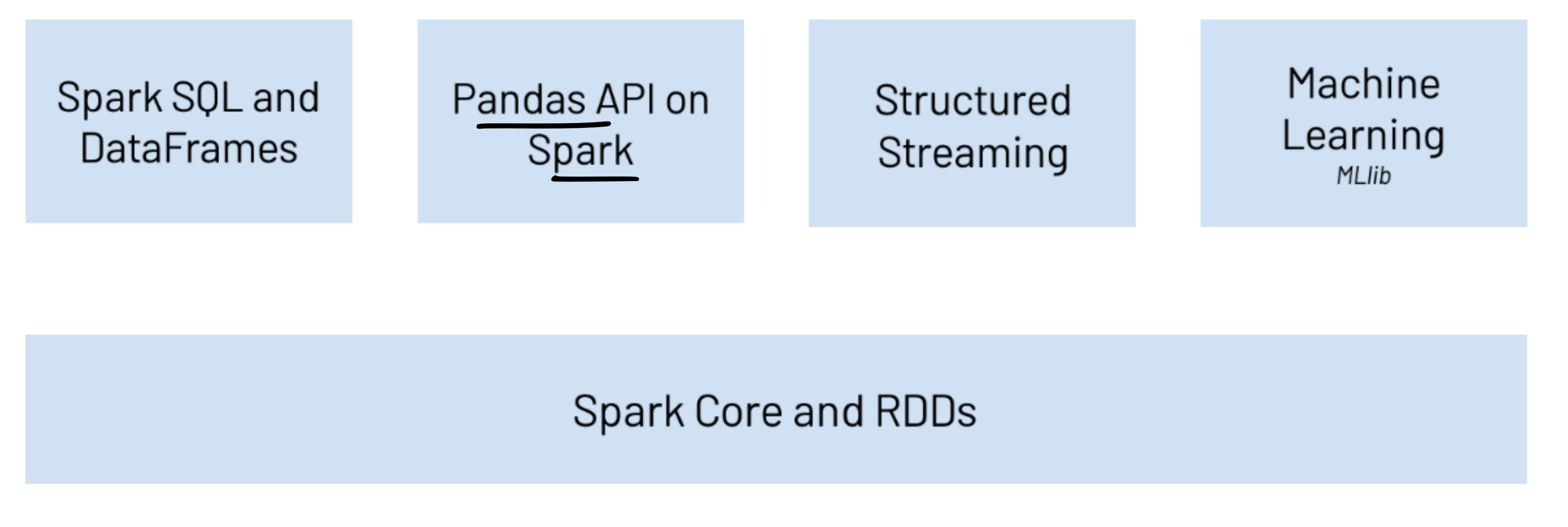
- Neo4J:
  - https://neo4j.com/docs/spark/current/quickstart/

- Cassandra:
  - https://github.com/datastax/spark-cassandra-connector/blob/master/doc/0_quick_start.md

- JDBC to other Databases:
  - https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html#data-source-option

pyspark

- Python API for Apache Spark
- Provides support for Spark SQL and DataFrames
- Also supports Structured Streaming, Machine Learning (MLlib), Pandas API on Spark, and Spark Core (including RDD)



source: Apache Spark Documentation; https://spark.apache.org/docs/latest/api/python/index.html

pyspark SparkSession

- Entry point for all APIs
- SparkSession supports different operations, including: creating DataFrames, registering DataFrames as tables, and executing SQL commands on tables
- Create SparkSession using spark builder:
  - *from pyspark.sql import SparkSession*
  - *spark = SparkSession *

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

source: Apache Spark Documentation; https://spark.apache.org/docs/latest/api/python/index.html

pyspark DataFrame - Reading Data

- CSV:
  - *path = "examples/src/main/resources/people.csv"*
  - *df = spark.read.csv(path)*
  - *df2 = spark.read.option("delimiter", ";").csv(path)*

- JDBC Connector:
  - ```
    df = spark.read \
             .format("jdbc") \
             .option("url", "jdbc:postgresql:dbserver") \
             .option("dbtable", "schema.tablename") \
             .option("user", "username") \
             .option("password", "password") \
             .load()
    ```

  - ```
    df2 = spark.read \
             .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
             properties={"user": "username", "password": "password"})
    ```

source: Apache Spark Documentation; https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html

pyspark DataFrame - Writing Data

- CSV:
  - *df.write.csv("output")*

- JDBC Connector:
  - *jdbcDF.write *
    *.format("jdbc") *
    *.option("url", "jdbc:postgresql:dbserver") *
    *.option("dbtable", "schema.tablename") *
    *.option("user", "username") *
    *.option("password", "password") *
    .save()

- jdbcDF2.write \\
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
  properties={"user": "username", "password": "password"})

source: Apache Spark Documentation; https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html

pyspark DataFrame - Column Selection

- . (dot) operator - invokes apply() method and returns the selected column:
  - *people = spark.createDataFrame([*
    {"deptId": 1, "age": 40, "name": "Hyukjin Kwon", "gender": "M", "salary": 50},
    {"deptId": 1, "age": 50, "name": "Takuya Ueshin", "gender": "M", "salary": 100},
    {"deptId": 2, "age": 60, "name": "Xinrong Meng", "gender": "F", "salary": 150},
    {"deptId": 3, "age": 20, "name": "Haejoon Lee", "gender": "M", "salary": 200}])
  - *age_col = people.age*

- select() - returns a new DataFrame with the specified columns or expression:
  - *people.select(people.deptId, people.name)*

- drop() - returns a new DataFrame without the specified columns:
  - *df = spark.createDataFrame([(14, "Tom"), (23, "Alice"), (16, "Bob")], ["age", "name"])*
  - *df.drop('age')*

- toDF() - returns a new DataFrame with new specified column names:
  - *df = spark.createDataFrame([(14, "Tom"), (23, "Alice"), (16, "Bob")], ["age", "name"])*
  - *df.toDF('f1', 'f2')*

source: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html

pyspark DataFrame - Row Selection

- filter() - filters rows with the specified condition:
  - *df = spark.createDataFrame([(2, "Alice"), (5, "Bob")], schema=["age", "name"])*
  - *df.filter(df.age > 3)*

- distinct() - returns a new DataFrame with the distinct rows:
  - *df = spark.createDataFrame([(14, "Tom"), (23, "Alice"), (23, "Alice")], ["age", "name"])*
  - *df.distinct()*

- head() - returns the first (n) rows as a list of Row:
  - *df = spark.createDataFrame([(2, "Alice"), (5, "Bob")], schema=["age", "name"])*
  - *df.head(2)*

- tail() - returns the last n) rows as a list of Row:
  - *df = spark.createDataFrame([(14, "Tom"), (23, "Alice"), (16, "Bob")], ["age", "name"])*
  - *df.tail(2)*

source: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html

pyspark DataFrame - Join Operations

- join() - joins with another DataFrame per the given expression:
  - *df = spark.createDataFrame([(2, "Alice"), (5, "Bob")]).toDF("age", "name")*
  - *df2 = spark.createDataFrame([Row(height=80, name="Tom"), Row(height=85, name="Bob")])*
  - *df.join(df2, 'name').select(df.name, df2.height)*

source: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html

pyspark DataFrame - Miscellaneous Methods

- show() - prints the first (n) rows to the console:
  - *df = spark.createDataFrame([(2, "Alice"), (5, "Bob")], schema=["age", "name"])*
  - *df.show(2)*

- count() - returns the number of rows in the DataFrame:
  - *df = spark.createDataFrame([(14, "Tom"), (23, "Alice"), (16, "Bob")], ["age", "name"])*
  - *df.count()*

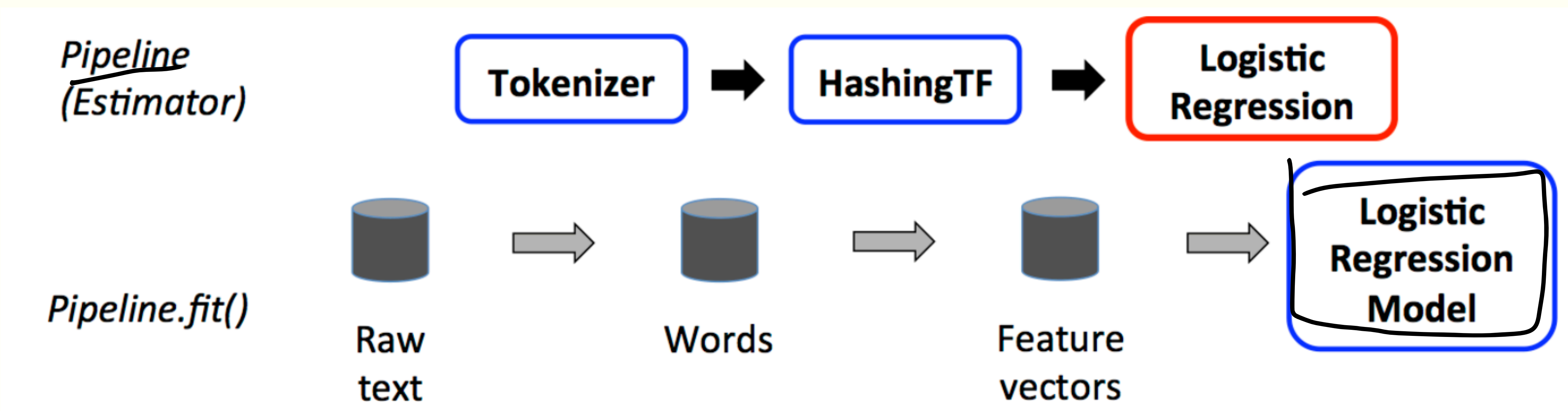
source: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html

Spark MLlib

- Scalable machine learning library for Spark
- Supports common ML algorithms including classification, regression, clustering, and collaborative filtering
- Integrates well with Spark SQL, DataFrames and Structures Streaming APIs
- Supports Scala, Java, Python and R programing languages
- Guide:
  - https://spark.apache.org/docs/latest/ml-guide.html

Spark MLlib - Pipeline

- Pipeline consists of a sequence of algorithms (Transformers and Estimators)
- Transformer transforms one DataFrame to another
- Estimator can be fit on a DataFrame to produce a Transformer

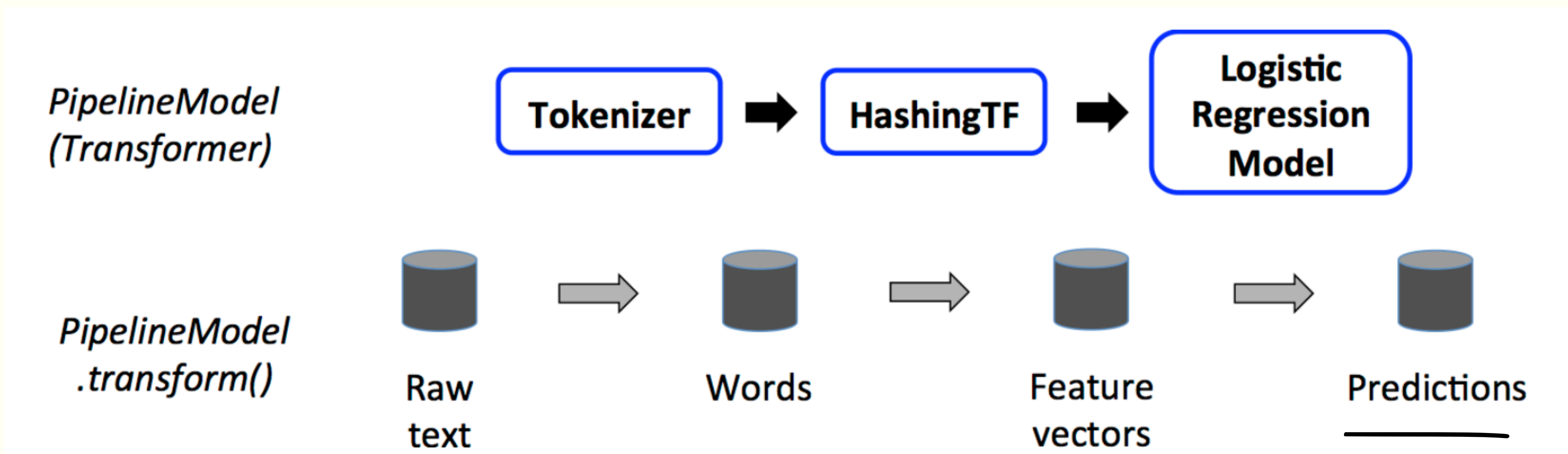


source: https://spark.apache.org/docs/latest/ml-pipeline.html

Spark MLlib - Pipeline (cont.)

- Since Pipeline is an Estimator, Pipeline.fit() returns a PipelineModel transformer

- The Transformer can then be applied to test data set (DataFrame) to generate the predictions (DataFrame)



source: https://spark.apache.org/docs/latest/ml-pipeline.html

GraphFrames

GraphX

- Provide DataFrame based Graphs
- Separate package than Apache Spark:
- Provides high-level APIs for Scala, Java and Python
- Download:
  - http://spark-packages.org/package/graphframes/graphframes
- Documentation:
  - https://graphframes.github.io/graphframes/docs/_site/index.html

GraphFrames

- Steps:
  - Create DataFrames for Vertices and Edges
  - Create GraphFrame from the DataFrames

- Example:

  *from graphframes import **

  v = spark.createDataFrame([("a", "Alice", 34), ("b", "Bob", 36), ("c", "Charlie", 30),], ["id", "name", "age"])

  e = spark.createDataFrame([("a", "b", "friend"), ("b", "c", "follow"), ("c", "b", "follow"),], ["src", "dst", "relationship"])

  g = GraphFrame(v, e)

GraphFrames - Algorithms

- Built-In Algorithms:
  - Breadth First Search
  - Shortest Path
  - Page Rank
  - and more ...

- Example:

```
results = g.pageRank(resetProbability=0.01, maxIter=20)
results.vertices.select("id", "pagerank").show()
```

Spark - Custom Functions

- Custom function support:
  - Spark RDD (Resilient Distributed Dataset) operations
  - Spark UDF (User Defined Functions)

pyspark - UDF

- Custom standalone Python functions can be converted to Spark UDFs.
- pyspark supports UDF operation on DataFrames on column-basis
- Supported data types for pyspark UDFs are the types defined in *pyspark.sql.types*. For example: StringType, IntegerType, FloatType, etc.
- Steps involved:
  - Define standalone Python function
  - Convert the function to a UDF (using *@udf* annotation or *pyspark.sql.functions.udf()* constructor)
  - (Optional) register the Python function or UDF as SQL function using *pyspark.sql.UDFRegistration.register()* method

Spark - UDF

- Example 1:

```
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import udf
@udf
def to_upper(s):
    if s is not None:
        return s.upper()

df = spark.createDataFrame([(1, "John Doe", 21)], ("id", "name", "age"))
df.select("name", to_upper("name")).show()
```

source: https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.functions.udf.html

Spark - UDF

- Example 1:

```
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import udf
@udf(returnType=IntegerType())
def add_one(x):
    if x is not None:
        return x + 1

df = spark.createDataFrame([(1, "John Doe", 21)], ("id", "name", "age"))
df.select("age", add_one("age")).show()
```

source: https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.functions.udf.html

Summary

- Spark provides low-level and high-level APIs for distributed data processing:
  - SQL, DataFrames and DataSets
  - RDDs and Distributed Variables
- DataFrame provide rich library of methods:
  - column-based operations, row-based operations and miscellaneous operations
- Spark MLlib module provides machine learning libraries with DataFrames
- GraphFrame (external) package provides Graph support with DataFrames
- Spark RDD and UDF can be used for implementation custom algorithms