Лекция 10. Apache PySpark. User-Defined Function – UDF

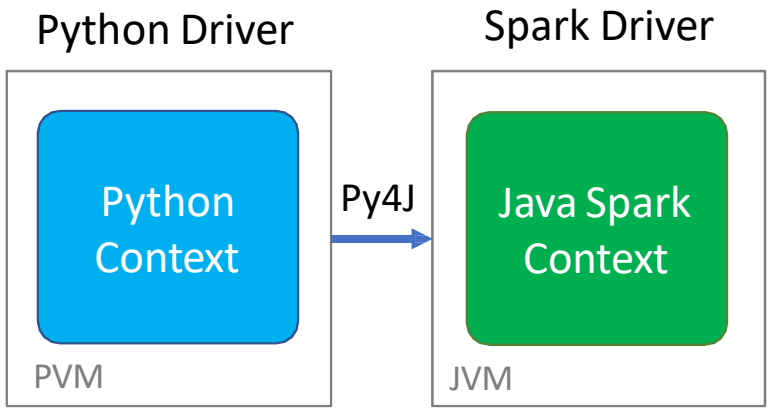
Основные темы

- PySpark
- Py4J
- Python UDF
- Pandas Vectorized UDF

PySpark

PySpark

- PySpark – Python API для Spark
- PySpark позволяет запускать Spark приложения, написанные на Python
- Spark использует JVM для работы основных компонентов и обработки данных



- Python Driver содержит Spark Context, который запускает JavaSparkContext и взаимодействует с ним посредством **Py4J**
- Трансформации над RDD в Python представляются как трансформации над PythonRDD объектами в Java
- PythonRDD в **executor'**ах запускают Python вокеры для обработки данных

Py4J

- Py4J обеспечивает Python программам, запущенным в PVM, динамический доступ к Java объектам в JVM
- Методы вызываются так, как если бы Java объекты находились в PVM
- Java Collections доступны через стандартные методы работы с коллекциями в Python
- Для доступа к JVM используется экземпляр класса GatewayServer, который позволяет взаимодействовать с JVM через сокет
- Java программа с GatewayServer должна быть запущена перед обращением из Python программы
- На стороне Python программы для работы с JVM используется класса JavaGateway

Py4J. Пример

Java App

```
import py4j.GatewayServer;

public static void main(String[] args)
     { GatewayServer gatewayServer =
          new GatewayServer(new
     StackEntryPoint()); gatewayServer.start();
     System.out.println("Gateway Server Started");
}
```

JVM – Java Virtual Machine

Python App

```
from py4j.java_gateway import JavaGateway

gateway = JavaGateway()
java_list =
gateway.jvm.java.util.ArrayList()
java_list.append(5)
```

PVM – Python Virtual Machine

По умолчанию

Адрес: 127.0.0.1

Порт: 25333

Сокет

PySpark и UDF

- Определяемая пользователем функция (User-Defined Function – **UDF**) – анонимная функция (*lambda*), функции для трансформаций *map*, *flatMap* и др.

RDD

```
rdd.map(lambda x: 1.0 if x == "F" else 0.0)
```

Dataframe

```
def convert2num_func(x):
        return 1.0 if x == "F" else 0.0

convert2num_udf = F.udf(convert2num_func, FloatType())
df.select(convert2num_udf(df["Gender"])).alias("GenderNum")
```

PySpark и UDF

Для обработки данных посредством Spark с использованием UDF на Python необходимо:

- Запустить функцию в PVM (т.е. в отдельном от JVM процессе)
- Преобразовать записи RDD из Java в Python контекст
- Результат обработки обратно преобразовать в Java контекст

Запуск UDF

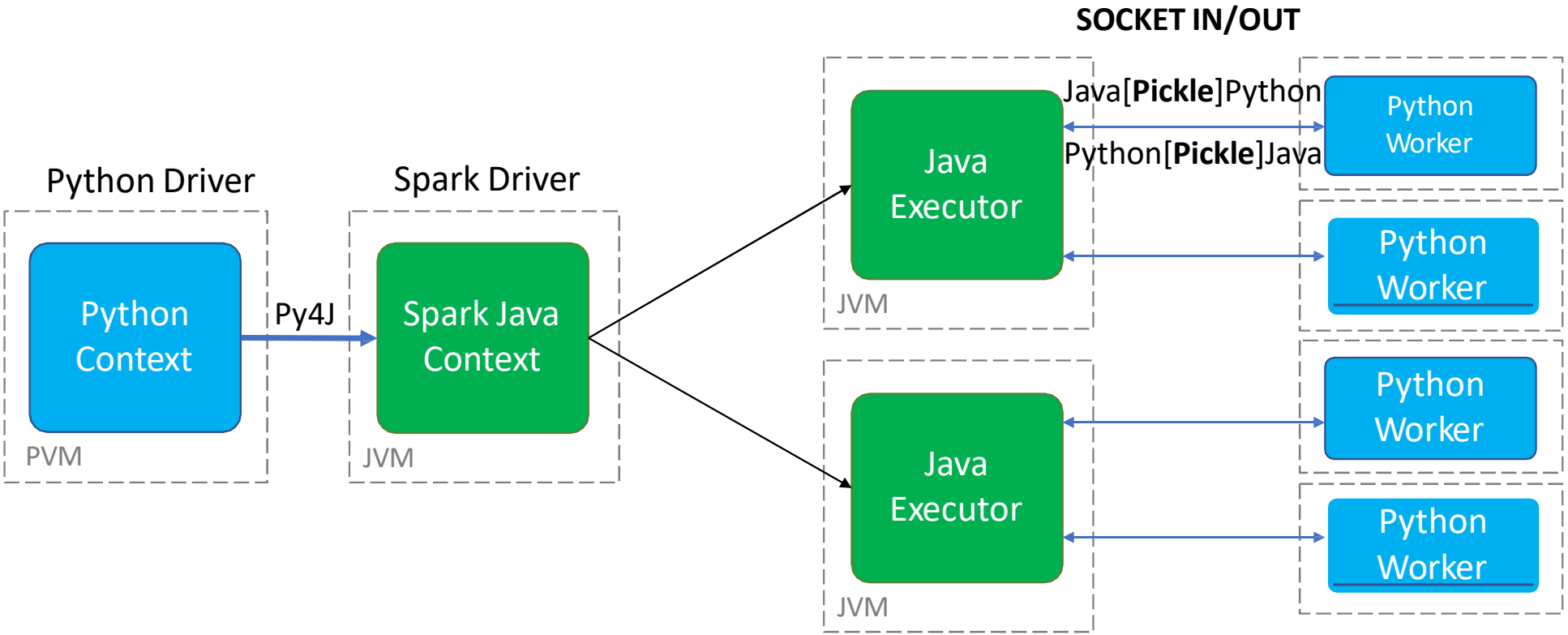
- Сериализация **UDF** (***cloudpickle***) и передача на рабочие узлы
- Десериализация **UDF** на рабочем узле и запуск в PVM процессе

Обработка данных UDF

- Записи **partition** на **executor’**ах необходимо преобразовать в Python контекст
- Поэтому входные данные для **UDF** предварительно сериализуются (***pyrolite***, ***pickle***) на Java Executor’ах
- Перед обработкой данные десериализуются
- После обработки выходные данные подвергаются обратному процессу сериализации/десериализации
- Для оптимизации записи **partition’**ов **RDD** передаются группами (**batchSize**). Соответственно, процесс сериализации/десериализации происходит для группы записей

```
sc = SparkContext('local', 'test', batchSize=2)
```

Запуск Python UDF в PySpark

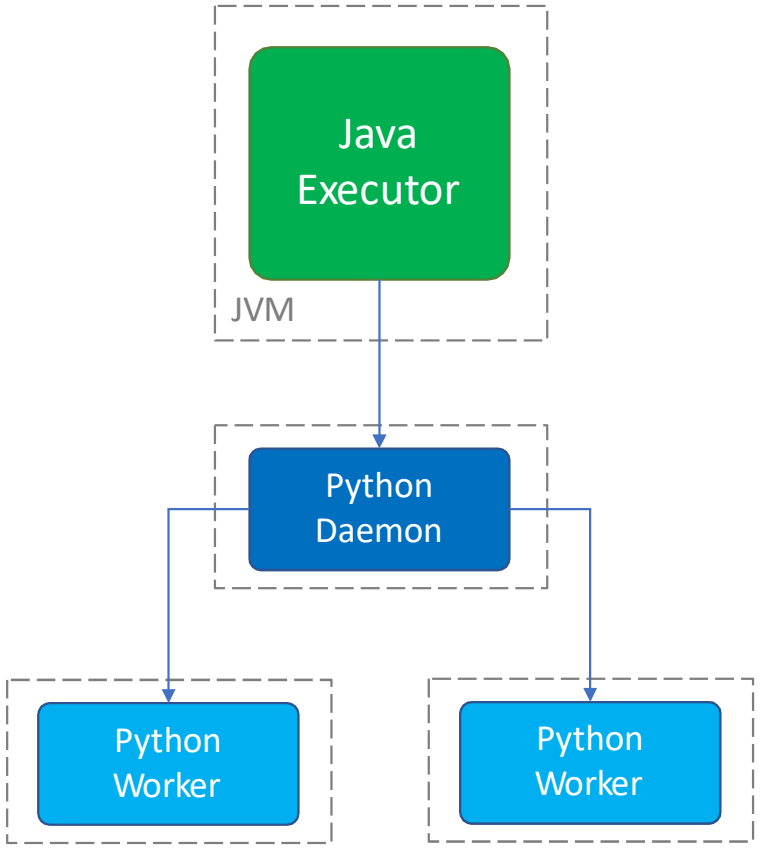


PYTHON_WORKER_MEMORY = PYSPARK_EXECUTOR_MEMORY / EXECUTOR_CORES

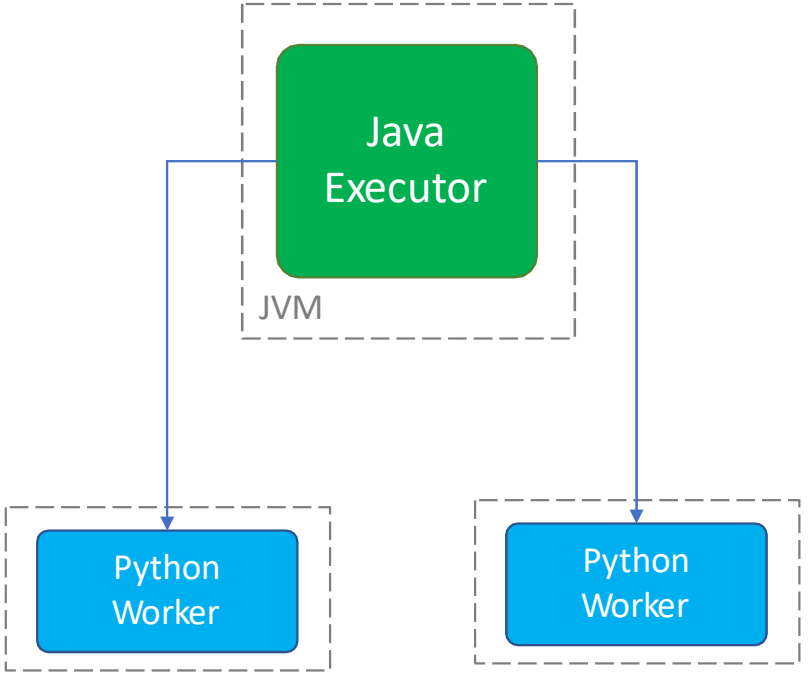
Режимы запуска Python Worker

ДЛЯ UNIX-BASED ОС

Запуск через демон



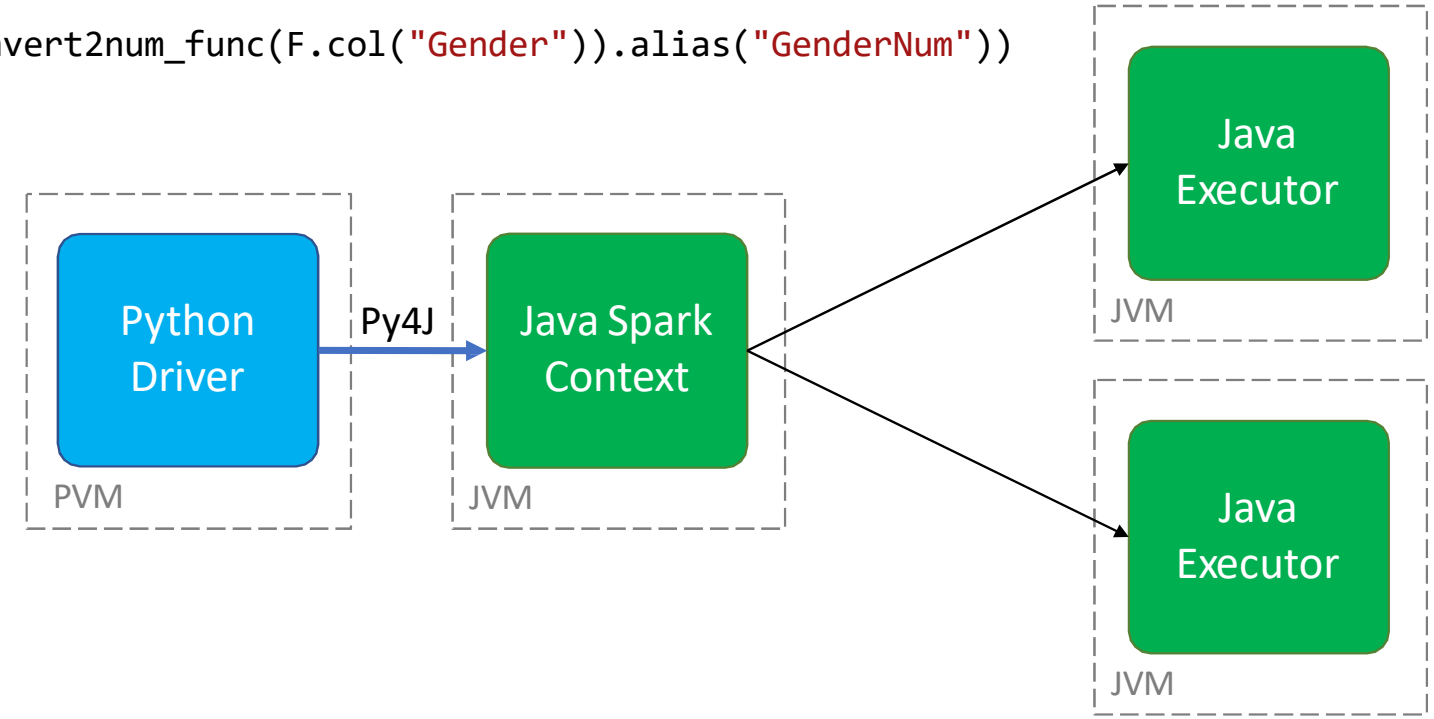
Запуск worker’а напрямую



PySpark и стандартные операции над Dataframe’ами

```
def convert2num_func(col):
        return F.when(col == "f", 1.0).otherwise(0.0)

df.select(convert2num_func(F.col("Gender")).alias("GenderNum"))
```



Пример плана выполнения UDF

```
def convert2num_func(x):
        return 1.0 if x == "F" else 0.0

convert2num_udf = F.udf(convert2num_func, FloatType())
df.select(convert2num_udf(df["Gender"])).alias("GenderNum")
```

```
== Physical Plan ==
*(2) Project [pythonUDF0#175 AS GenderNum#173]
+ BatchEvalPython [convert2num_func(Gender#5)], [Gender#5, pythonUDF0#175]
   + *(1) FileScan csv [Gender#5]
```

Пример плана выполнения для стандартных функций

```
def convert2num_func(col):
        return F.when(col == "f", 1.0).otherwise(0.0)

df.select(convert2num_func(F.col("Gender")).alias("GenderNum"))
```

```
== Physical Plan ==
*(1) Project [CASE WHEN (Gender#5 = f) THEN 1.0 ELSE 0.0 END AS CASE WHEN
(Gender = f) THEN 1.0 ELSE 0.0 END#176]
+ *(1) FileScan csv [Gender#5]
```

Работа с пользовательскими функциями в Spark

```
def square(x):
    return x**2


from pyspark.sql.types import IntegerType
square_udf_int = udf(lambda z: square(z), IntegerType())



df_pd = pd.DataFrame(
data={'integers': [1, 2, 3],
'floats': [-1.0, 0.5, 2.7],
'integer_arrays': [[1, 2], [3, 4, 5], [6, 7, 8, 9]]})

df = spark.createDataFrame(df_pd)

df.select('integers','floats',
square_udf_int('integers').alias('int_squared'),
square_udf_int('floats').alias('float_squared')).show()
```

Работа с пользовательскими функциями в Spark

```
+--------+------+-----------+-------------+
|integers|floats|int_squared|float_squared|
+--------+------+-----------+-------------+
|       1|  -1.0|          1|         null|
|       2|   0.5|          4|         null|
|       3|   2.7|          9|         null|
+--------+------+-----------+-------------+
```

```
from pyspark.sql.types import FloatType
square_udf_float = udf(lambda z: square(z), FloatType())

df.select('integers',
'floats',square_udf_float('integers').alias('int_squared'),
square_udf_float('floats').alias('float_squared')).show()
```

```
+--------+------+-----------+-------------+
|integers|floats|int_squared|float_squared|
+--------+------+-----------+-------------+
|       1|  -1.0|       null|          1.0|
|       2|   0.5|       null|         0.25|
|       3|   2.7|       null|         7.29|
+--------+------+-----------+-------------+
```

Работа с пользовательскими функциями в Spark

```
def square(x):
    return float(x**2)
```

```
square_udf_float = udf(lambda z: square_float(z), FloatType())
```

integers	floats	int_squared	float_squared
1	-1.0	1.0	1.0
2	0.5	4.0	0.25
3	2.7	9.0	7.29

Выводы

Проблемы

- Процесс сериализации/десериализации замедляет процесс обработки
- Требует большего объема оперативной памяти и других вычислительных ресурсов

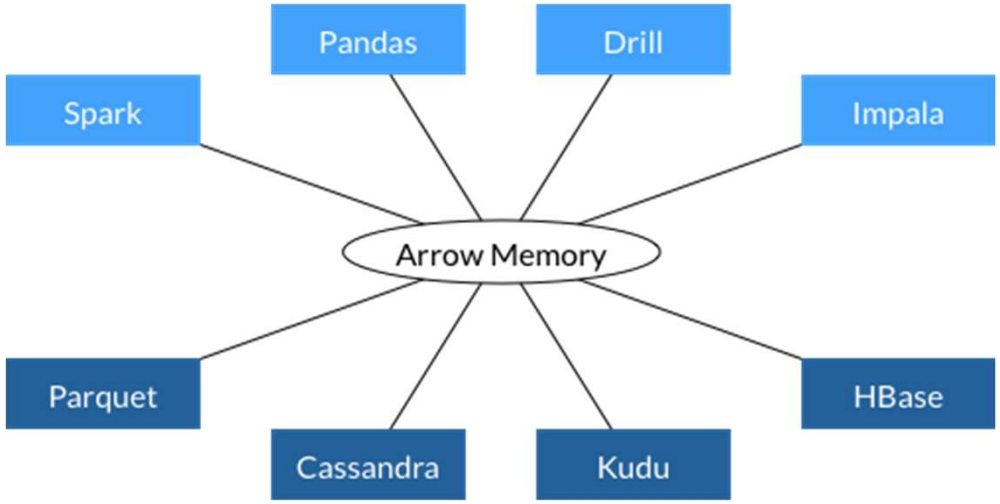
Необходимо

- Минимизировать использование Python UDF
- Использовать **dataframe'**ы со стандартными операциями
- Реализовать UDF на Java/Scala и вызывать из Python программы

Pandas Vectorized UDF

Apache Arrow

- Apache Arrow – платформа для работы со столбчатыми данными в оперативной памяти
- Поддерживаются следующие языки: C, C++, C#, Go, Java, JavaScript, MATLAB, Python, R, Ruby и Rust.



Apache Arrow в Spark

- Используется в Spark’е для передачи данных между Java и Python процессами
- Оптимизирует преобразование Spark Dataframe в Pandas Dataframe и обратно
- Все Spark SQL типы данных поддерживаются Arrow, кроме MapType, ArrayType для TimestampType и вложенные StructType
- Partition’ы преобразуются в пакеты записей (record batches), что временно ведет к повышенному использованию памяти в JVM

Pandas UDF в Spark

- Pandas UDF в Spark’е использует Arrow для передачи данных и Pandas для обработки данных

- Два типа Pandas UDF:
  - Scalar
  - Grouped

Scalar Pandas UDF

- **Scalar Pandas UDF** используется для поэлементных скалярных операций над векторами
- Входом и результатом Python функции должны быть **pandas**.**Series** одного размера
- Может быть использован с методами **select** и **withColumn**
- Spark разбивает столбцы на пакеты (**batch**) и вызывает UDF для каждого пакета, затем объединяет результаты обработки
- По умолчанию 10000 записей на один пакет

Пример Scalar Pandas UDF

```
# Функция перемножения значений
def multiply_func(a, b):
        return a * b

# Создание Pandas UDF
multiply = pandas_udf(multiply_func, returnType=LongType())

# Исходные Pandas данные
x = pd.Series([1, 2, 3])

# Создание Spark Dataframe'а из Pandas Series
df = spark.createDataFrame(pd.DataFrame(x, columns=["x"]))

# Выполнение
df.select(multiply(col("x"), col("x"))).show()
```

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

Grouped Map Pandas UDF

- Соответствует паттерну «split-apply-combine»

```
groupBy().apply()
```

- Три стадии
  - Разбиение данных на группы (DataFrame.groupBy)
  - Применение функции для каждой группы (вход и выход – pandas.DataFrame)
  - Объединение результатов обработки в DataFrame

- Все данные группы загружаются в оперативную память перед применением функции

Пример Grouped Map Pandas UDF

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

@pandas_udf("id long, v double", PandasUDFType.GROUPED_MAP)
def subtract_mean(pdf):
        # pdf - pandas.DataFrame
        v = pdf.v
        return pdf.assign(v=v - v.mean())

df.groupby("id").apply(subtract_mean).show()
```

```
# +---+----+
# | id| v|
# +-----+------+
# | 1|-0.5|
# | 1| 0.5|
# | 2|-3.0|
# | 2|-1.0|
# | 2| 4.0|
# +-----+----+
```

Grouped Aggregate Pandas UDF

- Соответствует паттерну «split-aggregate-combine»

```
groupBy().agg()
```

- Определяет агрегацию одного или нескольких **pandas.Series** (столбцы или окно) в скалярное значение

- Все данные группы загружаются в оперативную память перед применением функции

Пример Grouped Aggregate Pandas UDF

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

@pandas_udf("double", PandasUDFType.GROUPED_AGG)
def mean_udf(v):
        return v.mean()

df.groupby("id").agg(mean_udf(df['v'])).show()
```

```
# +---+-----------+
# | id|mean_udf(v)|
# +---+-----------+
# | 1| 1.5|
# | 2| 6.0|
# +-----+-----------+
```

Источники

Spark (github source code)

Py4J (official site)

PySpark Internals (wiki)

Apache Arrow (official site)

PySpark Usage Guide for Pandas with Apache Arrow (doc)

Introducing Pandas UDF for PySpark (blog)