Spark
wprowadzenie

Krzysztof Jankiewicz

Wprowadzenie

- **Spark** to szereg komponentów posiadających API dla szeregu języków programowania: Scala, Java, Python i R.
- Wszystkie komponenty razem tworzą ze Sparka środowisko do różnego typu przetwarzania:
  - ogólne przetwarzanie danych,
  - ETL,
  - interaktywnych zapytań (Spark SQL),
  - zaawansowanej analizy danych (Machine Learning – MLib),
  - przetwarzania danych grafowych (GraphX),
  - przetwarzania strumieni danych (Structured Streaming).

Wszystko to jest dostępne w ramach jednego silnika.

- Spark pozwala na dostęp do różnego typu źródeł danych: HDFS, Apache Cassandra, Apache Hbase, S3

Historia

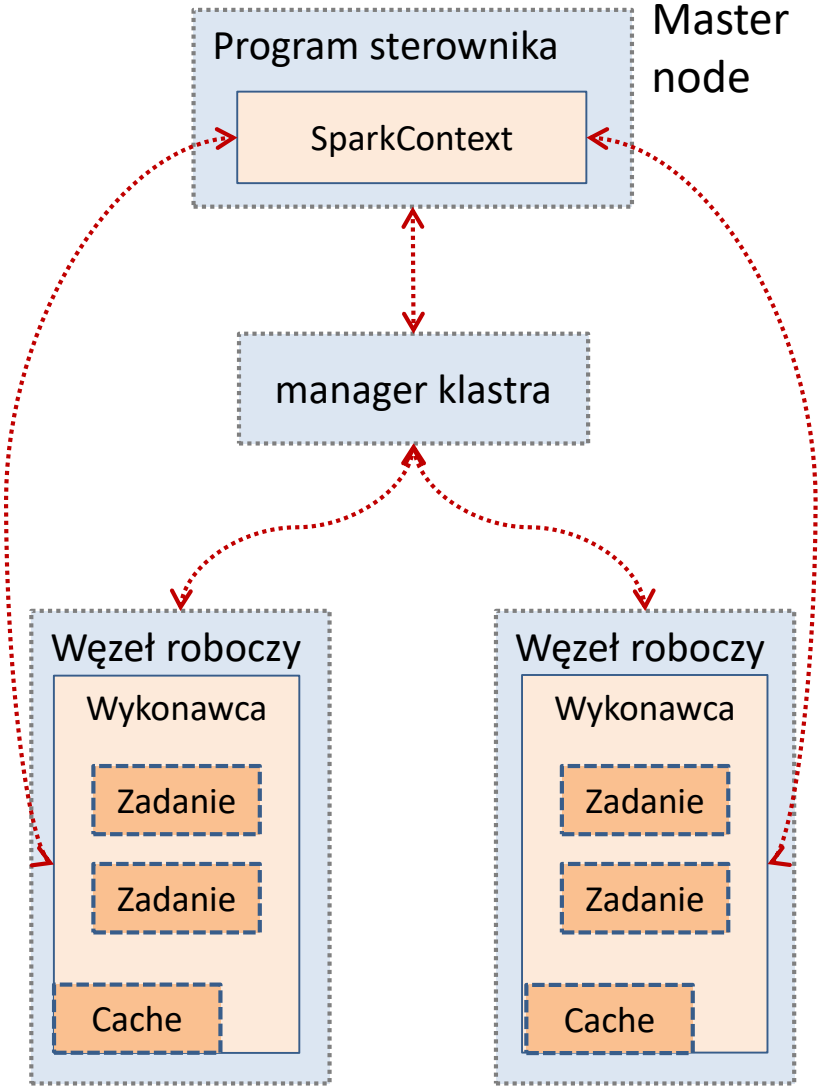
- 2009 – Pierwsza wersja napisana w ramach doktoratu Matei Zaharia w UC Berkeley AMPLab
- 2010 – upubliczniony kod źródłowy – licencja BSD
- 2013 – projekt w inkubatorze Apache
- 2014 – oficjalny projekt Apache; rekord Databricks w sortowaniu
- 2014.05-2016.11 – wersje 1.0.0-1.6.4
- 2016.07-2021.05 – wersje 2.0.0-2.4.8
- 2020.06-2023.09 – wersje 3.0.0-3.5.0

Architektura

- **Klaster (Spark Cluster) –** kolekcja maszyn lub węzłów w chmurze, na których Spark jest zainstalowany. W skład klastra wchodzą:
  - Węzły robocze (Spark workers)
  - Węzeł główny (Spark Master) (lub manager klastra w trybie Standalone),
  - Węzeł sterownika (Spark Driver).
- **Spark Master**
  - Pełni rolę managera klastra w przypadku konfiguracji Standalone.
  - Węzły robocze (spark workers) rejestrują się w ramach managera klastra jako część infrastruktury.
  - W zależności od typu konfiguracji może pełnić rolę także managera zasobów określając ile wykonawców (*executors*) uruchomić i na których węzłach roboczych w klastrze.
- **Spark Worker (węzeł roboczy)**
  - Aplikacje Sparka dekomponowane są na etapy, które składają się z atomowych jednostek zadań
  - Węzły robocze (Spark worker JVM), otrzymują jednostki zadań od managera i uruchamiają je w ramach wykonawców na rzecz węzła sterownika (Spark driver).

Architektura

- **Spark Executor**
  - Wykonawca to kontener JVM posiadający zaalokowaną określoną liczbę rdzeni, oraz pamięci, w ramach którego uruchamiane są jednostki zadania Sparka.
  - Każdy węzeł roboczy uruchamia własnego wykonawcę z ilością zasobów wynikającą z parametrów konfiguracji.
  - Oprócz wykonywania zadań, wykonawca przechowuje w pamięci wszystkie partycje danych.
- **Spark Driver**
  - Węzeł sterownika pobiera informacje od managera klastra o węzłach roboczych w klastrze i ich lokalizacji.
  - Program sterownika rozdziela zadania pomiędzy wykonawcami węzłów roboczych, a także odbiera od nich rezultaty tych zadań.



SparkSession

- Przed wersją 2.0 w zależności od wykorzystywanej funkcjonalności używane były trzy obiekty połączeniowe
  - **SparkContext** – środowisko przetwarzania danych (np. RDD)
  - **SQLContext** – wykorzystanie SparkSQL
  - **HiveContext** – wykorzystanie danych składowanych w Hive
- W wersji 2.0 wprowadzony został obiekt **SparkSession**, nowy punkt wejścia, który obejmuje SQLContext i HiveContext.
- Program sterownika używa obiektu kontekstu, aby połączyć się z managerem klastrów w celu komunikowania się i przesyłania zadań.

Aplikacje, zadania, etapy, jednostki

- Aplikacje Sparka zwykle składają się z szeregu operacji, które (jak się szczegółowo dowiemy) są albo **transformacjami** albo **akcjami**
- Operacje te przetwarzają zbiory danych o typach dostępnych w Sparku (np.: RDD, `DataFrames` lub `Datasets`) za pomocą API udostępnianego przez te typy.
- W momencie, w którym w aplikacji Spark program napotka **akcję** wówczas tworzy on **zadanie** (*job*), obejmujące wszystkie **transformacje** zbioru danych "uwieńczone" akcją.

```
./bin/spark-submit --master yarn --deploy-mode cluster [options] <application-file> [app options]
```

Aplikacje, zadania, etapy, jednostki

- *Zadanie Sparka można porównać do zadań MapReduce czy zadań Tez*
- Zadanie jest dekomponowane na jeden lub wiele **etapów** (*stages*), a etapy na pojedyncze **jednostki** zadań (tasks).
- Jednostki zadań są tym, co program sterownika wysyła do wykonawców funkcjonujących w węzłach roboczych w celu ich wykonania w ramach klastra.
- Wiele jednostek zadań może być uruchamianych w ramach tego samego wykonawcy, każda z nich może przetwarzać określone fragmenty partycjonowanych zbiorów danych przechowywanych w pamięci.


```
result = (taxiTripsRDD
    .map(lambda tt: (tt.puLocationID, (1, tt.trip_distance)))
    .reduceByKey(lambda p1, p2: (p1[0] + p2[0], p1[1] + p2[1]))
    .mapValues(lambda p: p[1] / p[0])
    .sortBy(lambda x: x[1], ascending=False)
    .take(5))
```

Typy konfiguracji

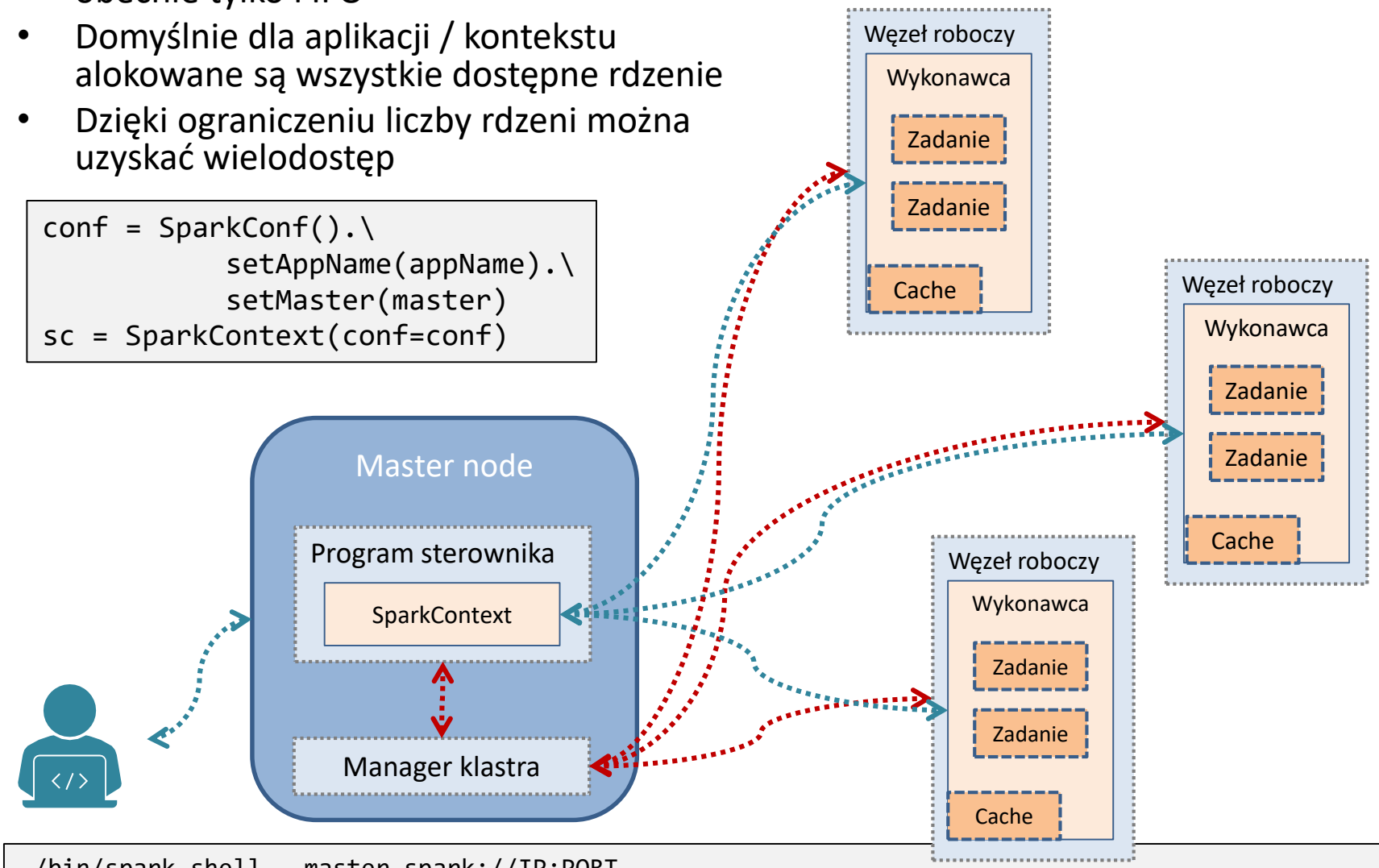
- **Standalone** – prosta konfiguracja bez wykorzystania managerów klastrów YARN lub Apache Mesos
- **YARN** – z wykorzystaniem managera zadań na platformie Hadoop – YARN
- **Mesos** – z wykorzystaniem ogólnego managera zasobów klastra – Apache Mesos
- **Kubernetes** – system do automatyzacji instalacji, skalowania oraz zarządzania aplikacjami wykorzystującymi mechanizmy konterów

Standalone

```
./sbin/start-master.sh
./sbin/start-slave.sh <master-spark-URL>
```

- Mechanizm szeregowania zadań – obecnie tylko FIFO
- Domyślnie dla aplikacji / kontekstu alokowane są wszystkie dostępne rdzenie
- Dzięki ograniczeniu liczby rdzeni można uzyskać wielodostęp

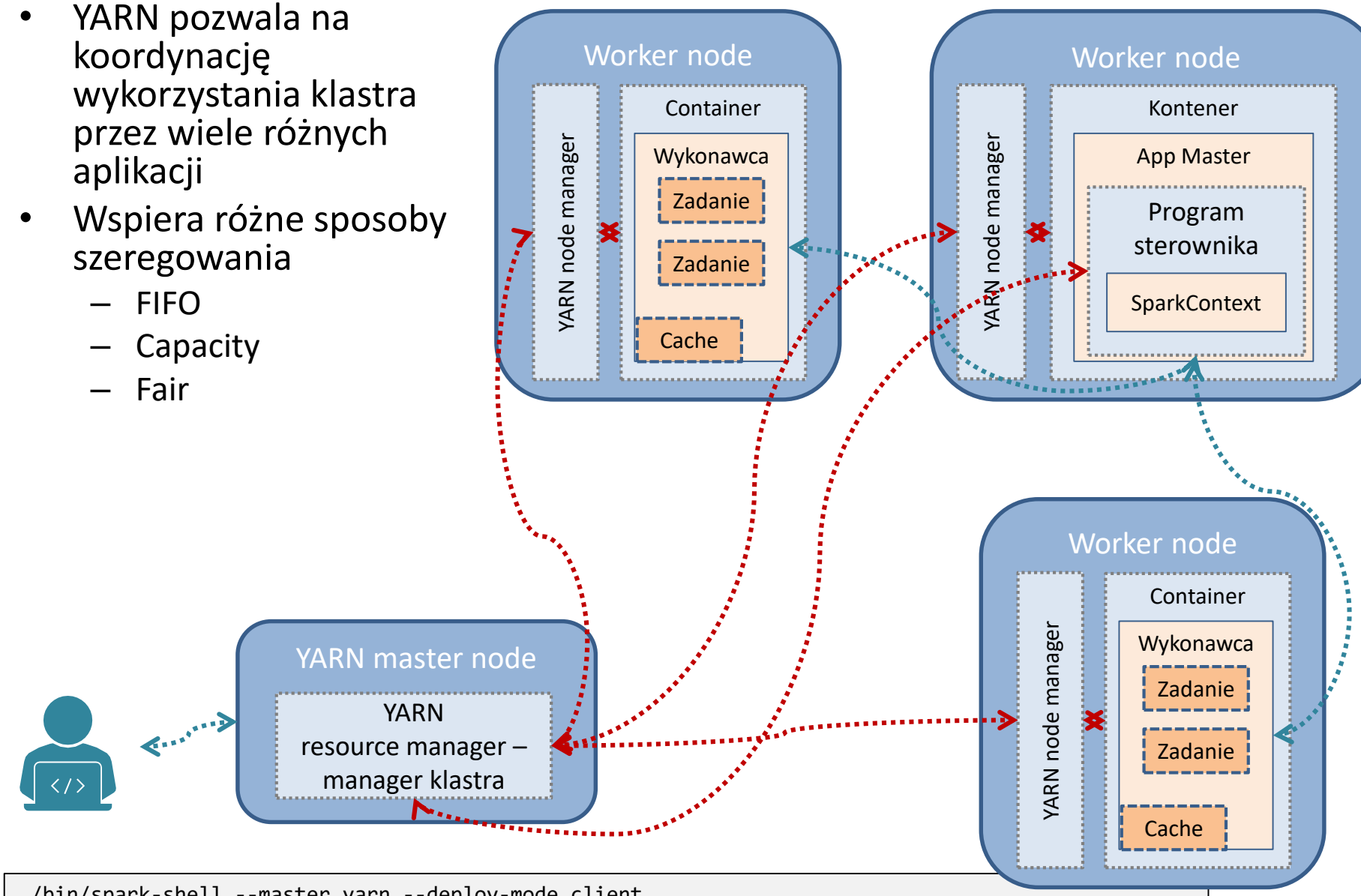
```
conf = SparkConf().\
           setAppName(appName).\
           setMaster(master)
sc = SparkContext(conf=conf)
```




```
./bin/spark-shell --master spark://IP:PORT
./bin/spark-submit --master spark://IP:PORT --deploy-mode <deploy-mode> --conf <key>=<value> \
  <application-file> [application-params]
```

YARN

- YARN pozwala na koordynację wykorzystania klastra przez wiele różnych aplikacji
- Wspiera różne sposoby szeregowania
  - FIFO
  - Capacity
  - Fair




```
./bin/spark-shell --master yarn --deploy-mode client
./bin/spark-submit --master yarn --deploy-mode cluster [options] <application-file> [app options]
```

PySpark shell

- Podczas pracy ze Pythonem korzystaliśmy ze środowiska REPL (*read-eval-print loop*), która nadaje się idealnie do programowania sterowanego eksperymentami
- Spark posiada analogiczne narzędzie – PySpark shell
- PySpark shell to interaktywna powłoka, która pozwala w pełni wykorzystać możliwości Sparka.
- Uruchamianie:
  - `./bin/pyspark --master local[4]`
  - `./bin/pyspark --master yarn`
  - `./bin/pyspark --master local[4] --py-files code.py`
- Podstawowe komendy:
  - `quit()` – wyjście z aplikacji
  - `type(wyrażenie)` – sprawdzenie typu wyrażenia

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.3.0
      /_/

Using Python version 3.10.6 (main, Aug 22 2022 20:35:26)
Spark context Web UI available at http://a3e171571718:4040
Spark context available as 'sc' (master = local[*], app id = local-1662370359602).
SparkSession available as 'spark'.
>>> type(sc)
<class 'pyspark.context.SparkContext'>
>>> type(spark)
<class 'pyspark.sql.session.SparkSession'>
>>> quit()
```

Dlaczego Python?

- Zalety

Sep 2022	Sep 2021	Change	Programming Language	Ratings	Change
1	2	▲	Python	15.74%	+4.07%
2	1	▼	C	13.96%	+2.13%
3	**3**		**Java**	**11.72%**	**+0.60%**

https://www.tiobe.com/tiobe-index/

Rank	Change	Language	Share	Trend
1		Python	28.29 %	-1.8 %
2		Java	17.31 %	-0.7 %
3		JavaScript	9.44 %	-0.1 %

https://pypl.github.io/PYPL.html

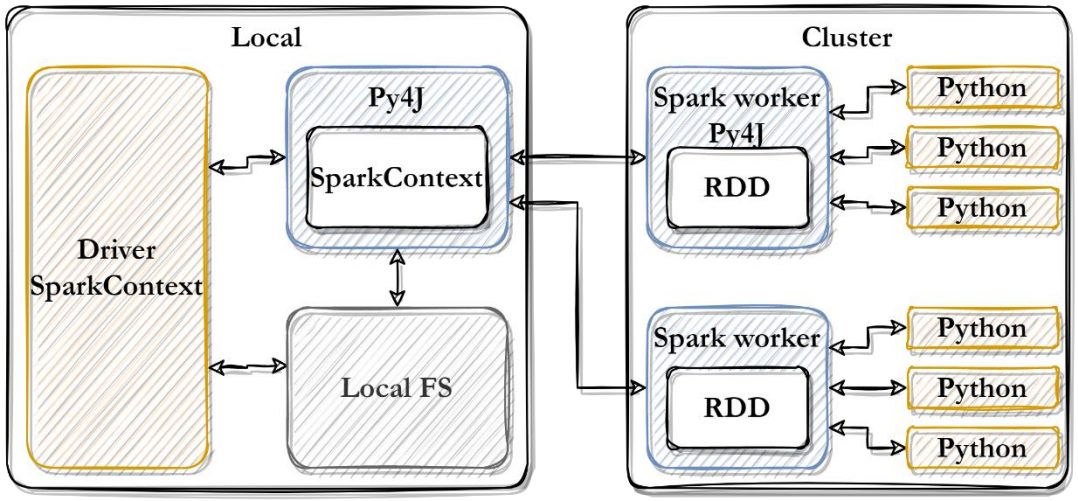
  - Mnogość bibliotek, funkcjonalności
  - Coraz większe wsparcie wśród platform Big Data
    - więcej narzędzi
    - większy zakres funkcjonalności

- Wady
  - Język interpretowany
  - Brak kompilacji do kodu bajtowego Javy
  - Konieczność wymiany danych pomiędzy procesami JVM a procesami obsługującymi kod Pythona
- Kierunki zmniejszania konsekwencji wad lub ich eliminacji
  - Narzędzia do wydajnego (kolumnowego) transferu danych pomiędzy procesami (PyArrow)
  - Implementacja API wzorowana na takich bibliotek jak Pandas działająca bezpośrednio na danych obsługiwanych przez narzędzia Big Data

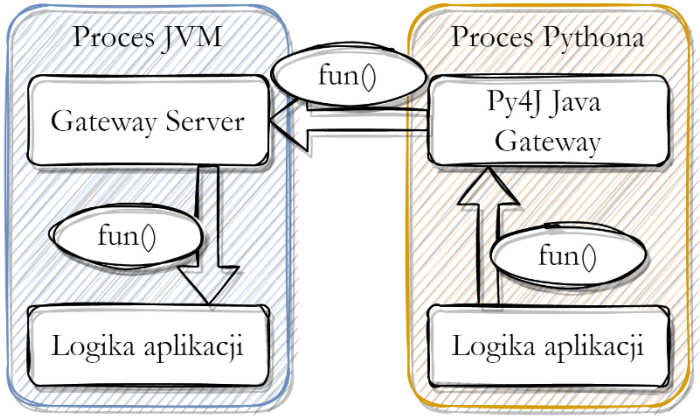
Ale jak to działa?

Py4J

- PySpark może być traktowany jako warstwa, pod którą działają mechanizmy i procesy JVM – fundament klastra
- Do komunikacji pomiędzy procesami **Javy** a procesami **Pythona** wykorzystywany jest **Py4J**



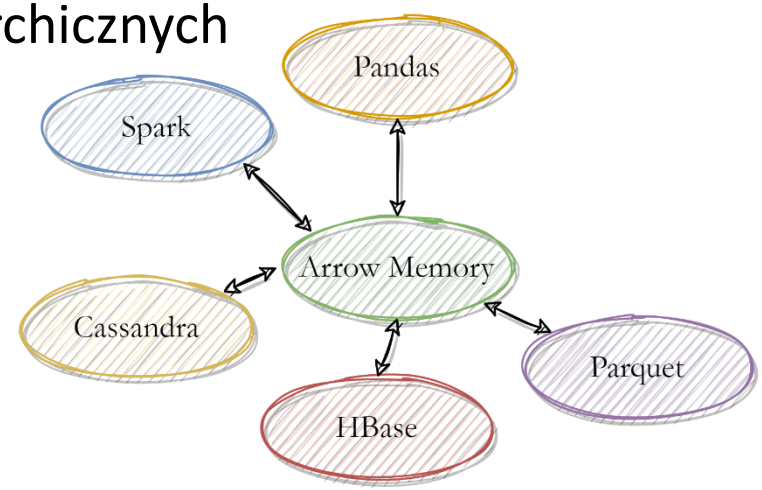
- **Py4J** pozwala programom Pythona uruchomionym w interpreterze Pythona na dynamiczny dostęp do obiektów Javy w JVM. Metody są wywoływane tak, jakby obiekty Javy znajdowały się w interpreterze Pythona, a dostęp do kolekcji Javy można uzyskać za pomocą standardowych metod kolekcji Pythona.
  Py4J umożliwia również programom Javy wywoływanie obiektów Pythona.
- Domyślnie dane są serializowane za pomocą formatu *Pickle*



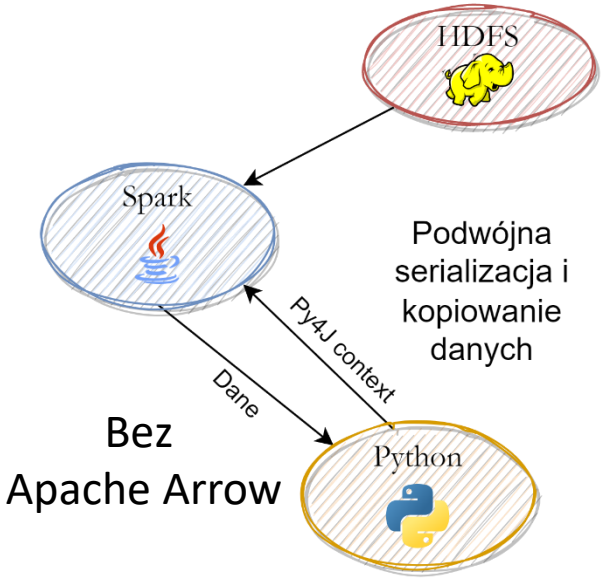
Ale jak to działa?

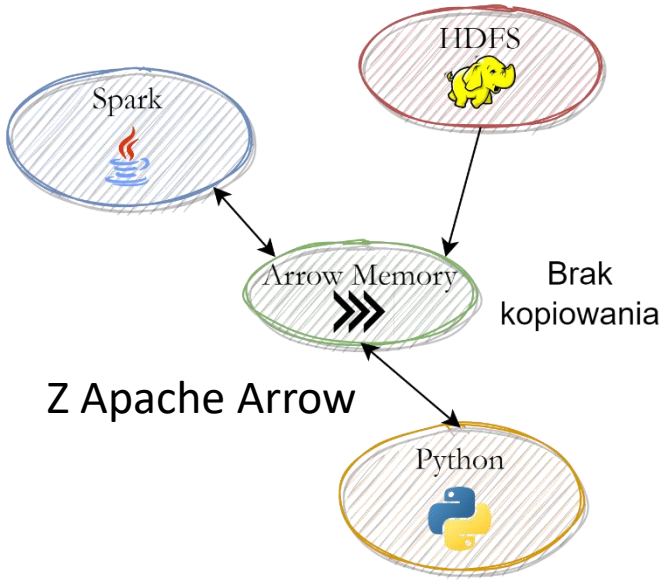
Apache Arrow

- Platforma programistyczna zarządzająca danymi przechowywanymi w pamięci.
- Definiuje znormalizowany, niezależny od języka, format danych w formacie kolumnowym zarówno dla danych płaskich jak i hierarchicznych



- Dostępna dla Spark SQL (DataFrame)
- Eliminuje użycie serializacji z użyciem formatu Pickle
- Pozwala na wykorzystanie Pandas UDFs, dzięki którym możliwe jest wykonywanie operacji wektorowych



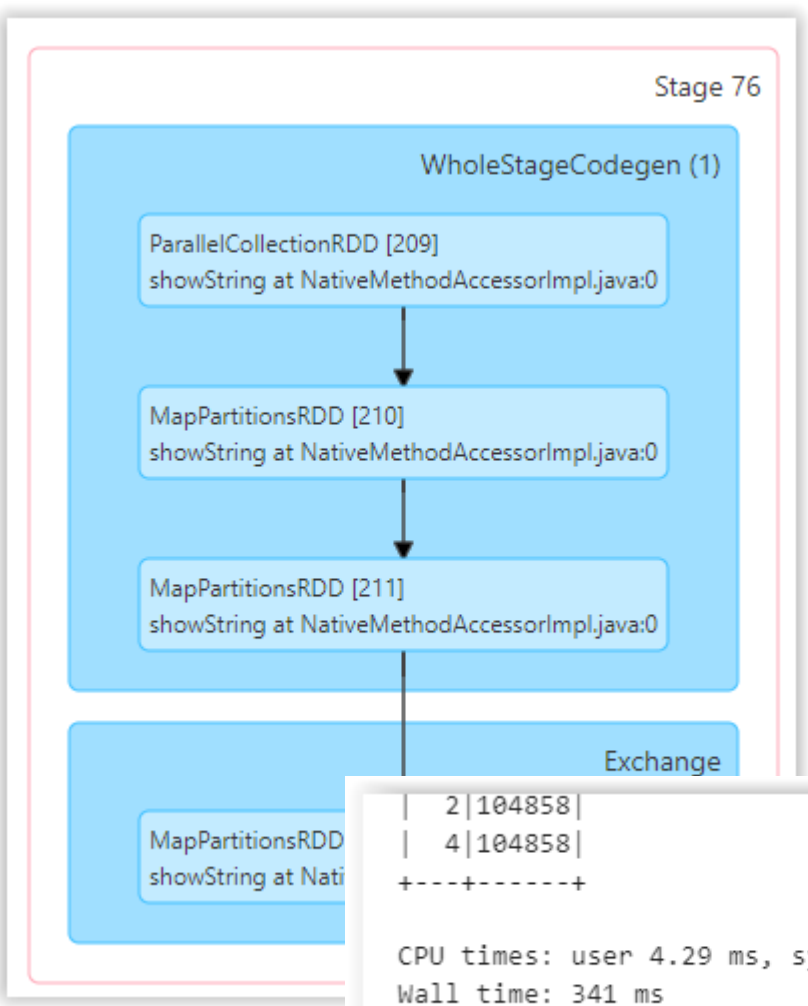


Przykład – PySpark == Python?

[80]:
```
from pyspark.sql.functions import rand, col
df = spark.range(1 << 20).toDF("id").withColumn("y", col("id") % 10).withColumn("x", rand())
df.printSchema()
```

[82]

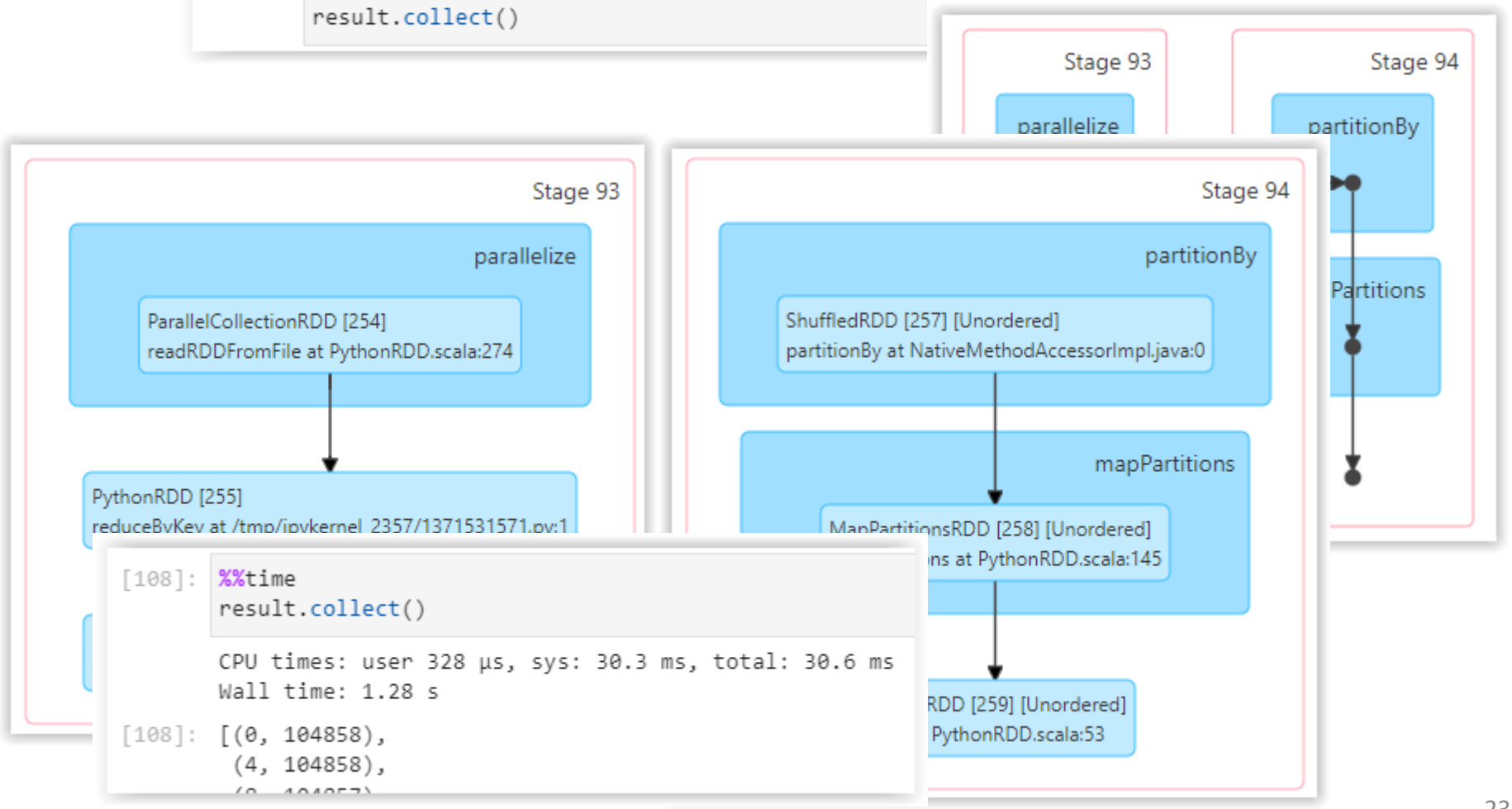
[83]




```
|  2|104858|
|  4|104858|
+---+------+

CPU times: user 4.29 ms, sys: 299 µs, total: 4.59 ms
Wall time: 341 ms
```

Przykład – PySpark == Python?

```
[106]: t = spark.sparkContext.parallelize(range(1 << 20))

[107]: result = t.map(lambda x: (x % 10, 1)).reduceByKey(lambda a, b: a + b)

[108]: %%time
       result.collect()
```



Spark, Python vs Scala

```
scala> def time[R](block: => R): R = {
     |     val t0 = System.nanoTime()
     |     val result = block
     |     val t1 = System.nanoTime()
     |     println("Elapsed time: " + (t1 - t0)/1000000 + "ms")
     |     result
     | }
time: [R](block: => R)R

scala> val t = spark.sparkContext.parallelize(Range(1,scala.math.pow(2,22).toInt))
t: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[7] at parallelize at <console>:23

scala> val r = scala.util.Random
r: util.Random.type = scala.util.Random$@70207dcb
```

```
>>> import time
>>> t = spark.sparkContext.parallelize(range(1 << 22))
>>> result = t.map(lambda x: (x % 10, random.random())).reduceByKey(lambda a, b: a + b)
>>> def fun():
...     t0 = time.time_ns()
...     res = result.collect()
...     t1 = time.time_ns()
...     print("Elapsed time: ",(t1 - t0)/1000000,"ms")
...     return res
...
>>> fun()
Elapsed time:  4505.1201 ms
[(0, 209887.710279801), (4, 209771.2495371836), (8, 209960.43794246003), (1, 209194.1201
, 209937.7263769612), (6, 209439.77415714157), (3, 210083.57630583923), (7, 209778.73166
```

Podsumowanie

- Co to jest Spark
- Architektura
- Czym jest SparkSession
- Aplikacje Sparka, zadania, jednostki
- Typy konfiguracji
- PySpark shell
- Spark i Python