

Spark

Resilient Distributed Datasets (RDD)
wydajność i elementy zaawansowane

Krzysztof Jankiewicz

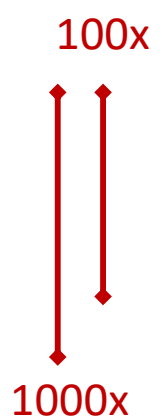
Plan

- Wprowadzenie – czas dostępu do danych (latency)
- *Shuffle operations* – przesyłanie danych
- Partycjonowanie
- Wąskie i szerokie zależności
- Zmienne rozgłoszeniowe i akumulatory

Czas oczekiwania – latency

- Spark (jak również Hadoop) to przetwarzanie rozproszone
- Przy rozproszonym przetwarzaniu (danych) istotnym jest tzw. **lokalność danych**.
- Powodem jest czas oczekiwania (*latency*) na dostęp do danych
- Przykłady:

Czas (ns)	Czynność	Czas	Czynność
0,5	L1 cache	0,5 s	Uderzenie serca
7	L2 cache		
100	pobranie danych z RAM		
3 000	kompresja 1K (Zippy)		
20 000	przesłanie 2K przez sieć	5,5 h	Od rana do obiadu
150 000	pobranie danych z dysku		
250 000	odczyt 1M sekwencji	2,9 dni	Długi weekend
500 000	RTT (round-trip time)		
1 000 000	odczyt 1M sekwencji	11,6 dni	Dwutygodniowe wakacje
20 000 000	odczyt 1M sekwencji	7,8 miesięcy	Od poczęcia do narodzin 😊 no prawie
150 000 000	wysłanie pakietu CA-	4,8 lat	Czas trwania dwóch poziomów studiów



Patrz: <http://research.google.com/people/jeff/>
<http://norvig.com/21-days.html#answers>

A także np.: <https://prezi.com/pdkvgys-r0y6/latency-numbers-for-programmers-web-development/>

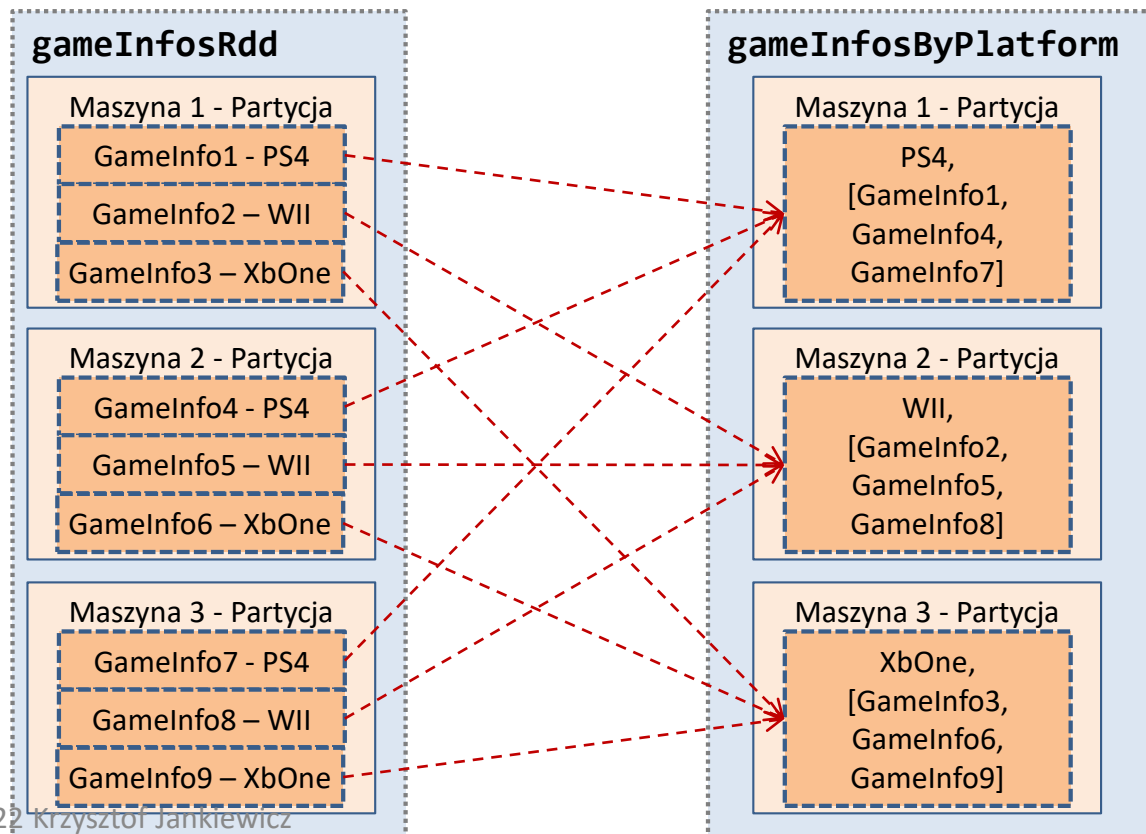
Shuffle operations – przesyłanie

- The shuffle is Spark's mechanism for **re-distributing data** so that it's grouped differently **across partitions**. This typically involves **copying data across executors and machines**, making the shuffle a **complex** and **costly** operation.

Spark Programming Guide

Patrz: <https://spark.apache.org/docs/latest/programming-guide.html#shuffle-operations>

```
scala> val gameInfosByPlatform = gameInfosRdd.groupBy(gi => gi.platform)
gameInfosByPlatform: org.apache.spark.rdd.RDD[(String, Iterable[GameInfo])] = ShuffledRDD[208]
```



Które transformacje wykonane na **gameInfosRdd** spowodują przenoszenie danych:

- groupBy?
- map(gi => gi.platform)?
- map(gi=>(gi.platform,gi)).groupByKey()?

Inne operacje, które dokonują przenoszenia to:

- połączenia** (cogroup, join)
- zmiany w partycjach** (repartition, coalesce)

Shuffle operations

szczegóły, ograniczanie

- Szczegóły
 - Operacje shuffle są kosztowne – wymagają:
 - operacji I/O na dyskach,
 - serializacji/deserializacji danych,
 - przesyłania danych przez sieć
 - Na potrzeby operacji shuffle Spark generuje zbiór zadań mapowania (do organizacji danych) i redukcji (do agregacji) (to nie te same operacje co w MapReduce)
 - Wyniki zadań mapowania są utrzymywane w pamięci o ile tylko jest to możliwe, następnie są one sortowane na podstawie klucza partycjonowania i zapisywane do pliku.
 - Zadanie redukcji czyta odpowiedni zakres bloków.
- Ograniczanie operacji shuffle
 - Nie ma możliwości całkowitego uniknięcia przesyłania danych pomiędzy węzłami
 - Można natomiast znacząco ograniczyć ilość przesyłanych danych
 - Ograniczanie jest możliwe poprzez agregowanie danych na poziomie węzłów źródłowych (i przesyłanie zagregowanych danych – o mniejszej wielkości)
 - `reduceByKey`
 - `aggregateByKey`

Shuffle przykład

Zadanie: oblicz średnią ocenę gier dla każdej z platform

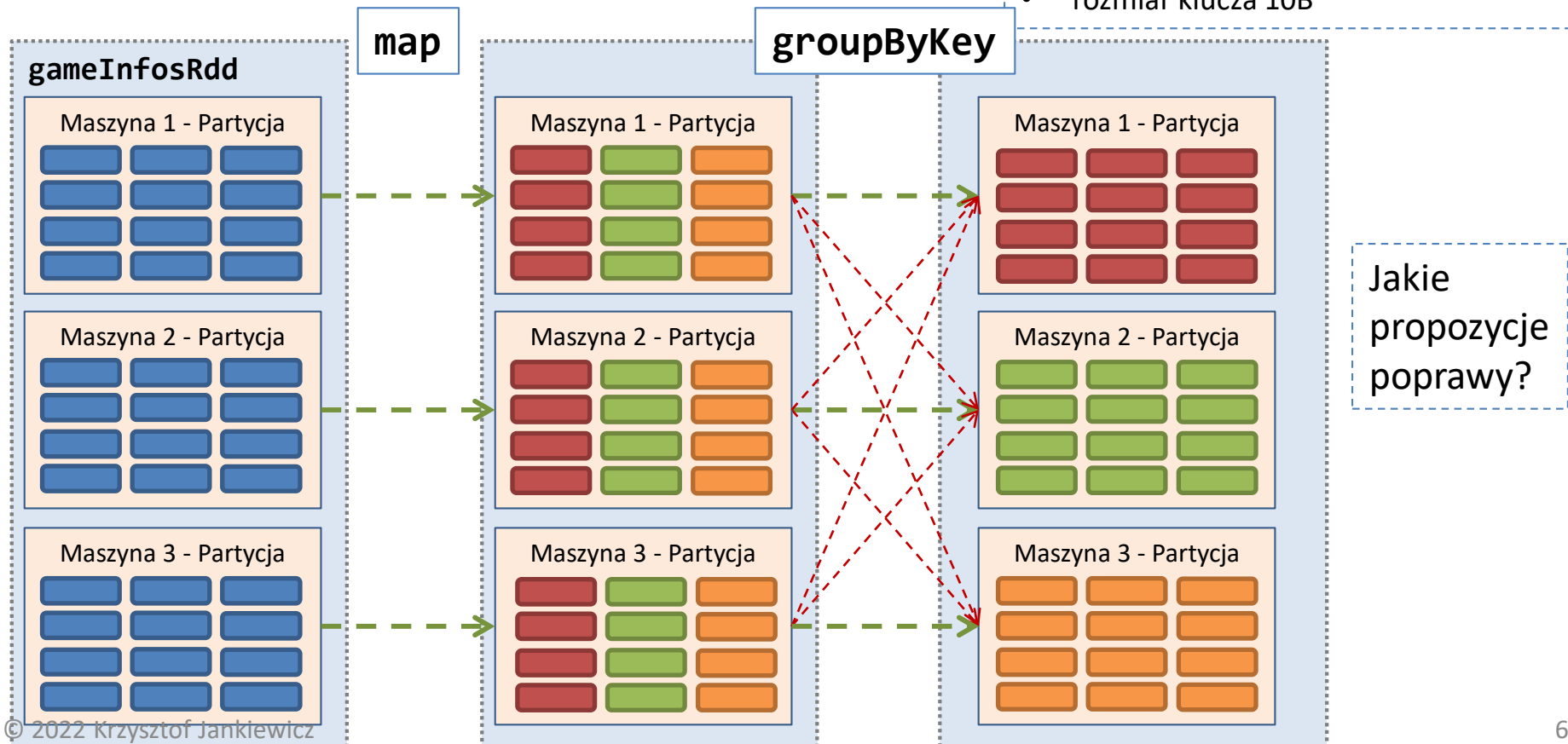
```
gameInfosRdd.map(gi => (gi.platform,gi)).groupByKey().  
  mapValues(gis => gis.map(gi => (gi.score))).  
  mapValues(scores => scores.sum/scores.size)
```

Summary Metrics for 1 Completed

Metric	Min
Duration	0.1 s
GC Time	0 ms
Input Size / Records	99.0 MB / 18589
Shuffle Write Size / Records	104.8 KB / 18589

Oszacuj

- liczb
- seri
- 3 w
- 3 węzły
- rozmiar score 10B
- rozmiar klucza 10B



Shuffle

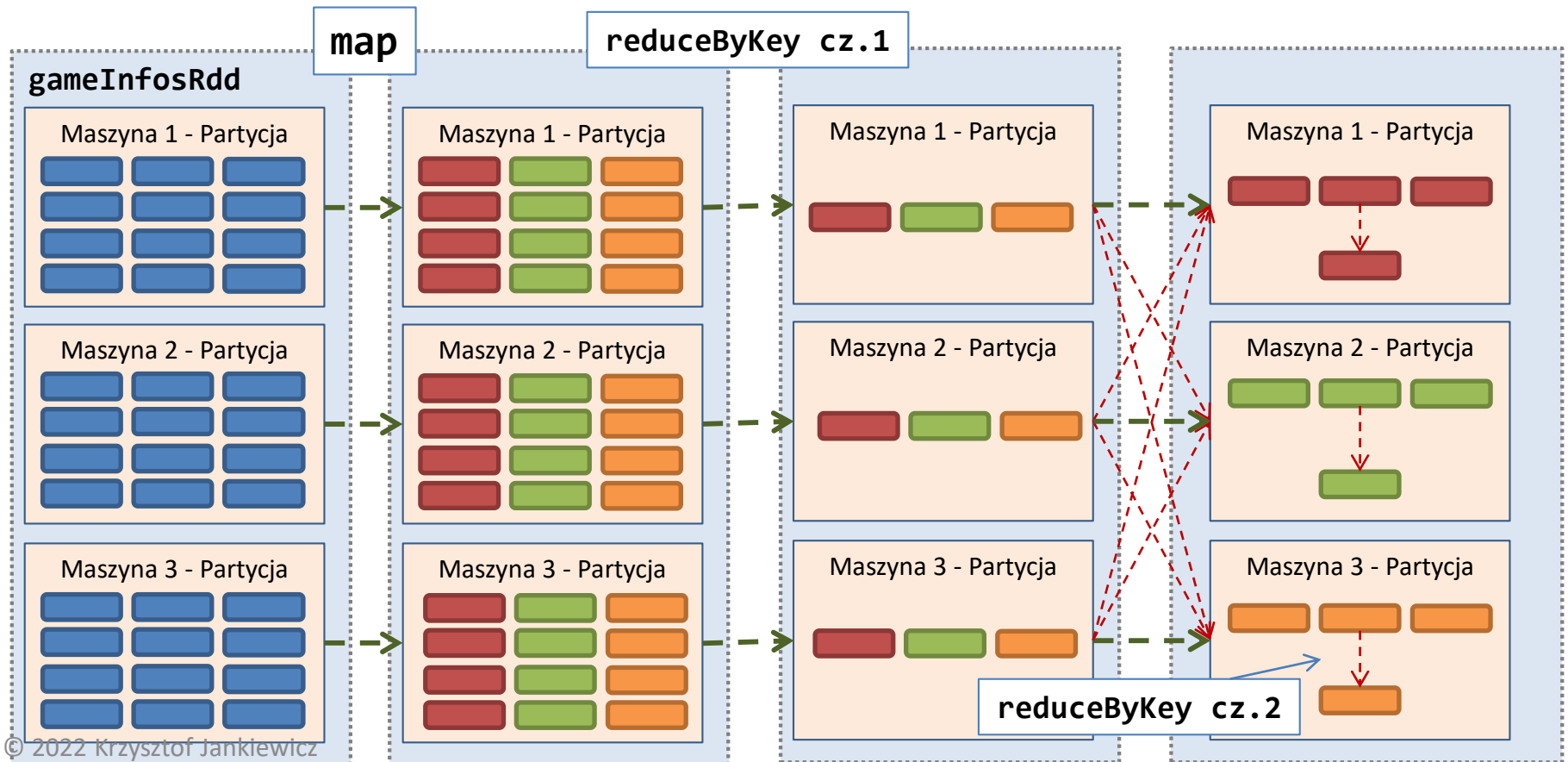
ograniczanie wielkości przesyłanych danych

Zadanie: oblicz średnią ocenę gier dla każdej z platform

```
gameInfosRdd.map(gi => (gi.platform,(1,gi.score))).  
  reduceByKey((p1,p2) => (p1._1 + p2._1, p1._2 + p2._2)).  
  mapValues(p => p._2/p._1)
```

Summary Metrics for 1 Completion

Metric	Min
Duration	42 ms
GC Time	0 ms
Input Size / Records	99.0 MB / 18589
Shuffle Write Size / Records	1322.0 B / 59



Operacje które mogą prowadzić do przesyłania danych

- cogroup, groupWith, groupByKey, reduceByKey
- join, left|right|fullOuterJoin
- distinct, intersection,
- repartition, coalesce

Dlaczego nie union? Narysuj jak wygląda operacja union?

Partycjonowanie

- RDD są dzielone na partycje, każda **partycja** znajduje się na **jednym** określonym **węźle**.
- Pojedynczy **węzeł** może zarządzać jedną lub **wieloma partycjami** tego samego RDD
- Liczba partycji jest konfigurowalna i domyślnie jest **równa** sumarycznej liczby **rdzeni** procesora przeznaczonych dla wszystkich wykonawców
- W pojedynczej **partycji** mogą znajdować się **losowe** węzły, jak również węzły posiadające określoną **wartość klucza** (w przypadku RDD par)
- W przypadku partycjonowania RDD par, Spark określa docelową partycję dla każdego rekordu
 - w oparciu o wartość klucza i
 - na podstawie **partycjonowania haszującego** (*hash partitioning*)dla (k, v) partycja wyznacza jest na podstawie wzoru:
$$p = k.hashCode() \% liczbaPartycji$$
- Alternatywne sposoby partycjonowania to:
 - partycjonowanie **zakresowe** (*range partitioning*)
 - partycjonowanie **użytkownika** (tylko dla RDD par)

Partycjonowanie zakresowe

- Dla RDD par posiadających klucze o typach implementujących cechę (trait) **Ordering** (zdolnych do uporządkowania) np.: Int, Char, String można wykorzystać partycjonowanie **zakresowe**
- Partycjonowanie zakresowe określane jest na podstawie
 - wartości klucza
 - zdefiniowanego **zbioru zakresów** dla kluczy
- Partycjonowanie zakresowe można zdefiniować na dwa sposoby:
 1. wykorzystując metodę **partitionBy** i jawny obiekt partycjonera (Partitioner)
 2. wykorzystując **transformację**, która daje w wyniku RDD z partycjonowaniem zakresowym
- Metoda 1

```
val scoresByPlatfom = gameInfosRdd.map(gi => (gi.platform, (1,gi.score)))
import org.apache.spark._
val myPartitioner = new RangePartitioner(8, scoresByPlatfom)
val scoresByPlatPart = scoresByPlatfom.partitionBy(myPartitioner).persist()
scoresByPlatPart: org.apache.spark.rdd.RDD[(String, (Int, Double))] =
  ShuffledRDD[245] at partitionBy at <console>:37

scoresByPlatPart.partitioner
Option[org.apache.spark.Partitioner] = Some(org.apache.spark.RangePartitioner@6375b62f)
scoresByPlatfom.partitioner
Option[org.apache.spark.Partitioner] = None
```

Transformacje a partycjonowanie

- Metoda 2
 - **partycjoner źródłowego RDD** – RDD wynikowej transformacji na źródłowym RDD przejmie jego sposób partycjonowania (jego partycjoner)
 - niektóre **transformacje** generują RDD z **określonym partycjonerem**
 - sortByKey – RangePartitioner
 - groupByKey – HashPartitioner
- Transformacje zachowujące lub propagujące partycjonery:
 - cogroup, groupWith, groupByKey, sort, partitionBy,
 - foldByKey, reduceByKey,
 - combineByKey, join, left|right|fullOuterJoin
 - mapValues, flatMapValues, filter – o ile źródłowy RDD posiada określonego partycjonera
- Pozostałe operacje generują wynikowy RDD bez określonego partycjonera np.: map
- To która operacja zachowa informacje dot. partycjonera wynika z tego czy ma możliwość dokonania zmiany w wartości klucza
- Ale po co nam te wszystkie informacje? Co nam daje ta wiedza?
Po co zmieniać sposób partycjonowania lub go ustalać?

Optymalizacja dzięki partycjonowaniu

- Powody:
 - Aby równomiernie rozmieszczać dane pomiędzy węzłami
 - Aby optymalizować kolejne operacje wykonywane na RDD: łączenie, redukcję itp.
- Przykład optymalizacji redukcji

```
def timeOf[A](f: => A) = {  
  val s = System.nanoTime  
  val r = f  
  println("time: " +  
    (System.nanoTime - s)  
    / 1e9 + " sec.")  
  r  
}
```

```
def fun1 = gameInfosRdd.map(gi => (gi.platform,gi)).groupByKey().  
  mapValues(gis => gis.map(gi => (gi.score))).  
  mapValues(scores => scores.sum/scores.size).collect()
```

```
val test = timeOf(fun1)  
time: 1.294257336 sec.
```

```
def fun2 = gameInfosRdd.map(gi => (gi.platform,gi.score)).  
  groupByKey().  
  mapValues(scores => scores.sum/scores.size).collect()
```

```
val test = timeOf(fun2)  
time: 0.183672659 sec.
```

```
def fun3 = gameInfosRdd.map(gi => (gi.platform,(1,gi.score))).  
  reduceByKey((p1,p2) => (p1._1 + p2._1, p1._2 + p2._2)).  
  mapValues(p => p._2/p._1).collect()
```

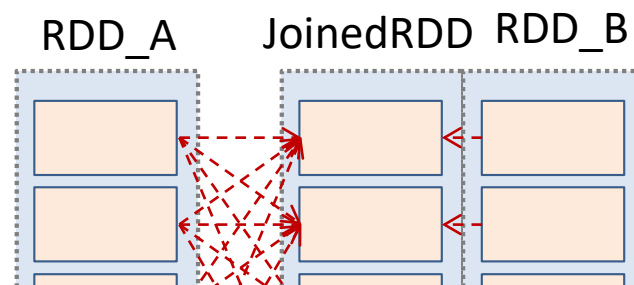
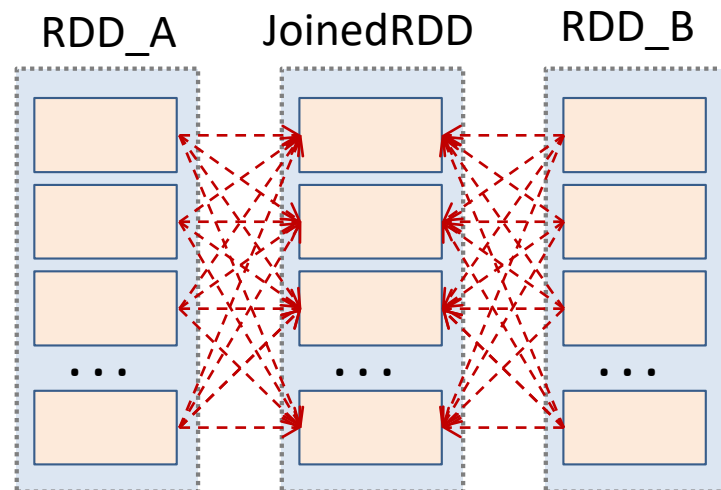
```
val test = timeOf(fun3)  
time: 0.094308428 sec.
```

```
def fun4 = scoresByPlatPart.  
  reduceByKey((p1,p2) => (p1._1 + p2._1, p1._2 + p2._2)).  
  mapValues(p => p._2/p._1).collect()
```

```
val test = timeOf(fun4)  
time: 0.05534141 sec.
```

Optymalizacja połączenia

- Bez partycjonowania danych źródłowych powtarzalne połączenia określonych zbiorów danych są bardzo mało wydajne
- Partycjonowanie jednego (większego) zbioru danych znacząco poprawia wydajność połączenia eliminując przesyłanie danych - tylko z niepartycjonowanego zbioru dane są przesyłane do odpowiednich węzłów zbioru partycjonowanego
- W przypadku partycjonowania obu zbiorów danych, rekordy obu zbiorów znajdują się już na właściwych węzłach?



```
tripRDD.count()
stationRDD.count()
res11: Long = 286858
res12: Long = 58
```

```
val tripPartRDD = tripRDD.map(t => (t.getString(8), t)).partitionBy(new HashPartitioner(29))
val tripNoPartRDD = tripPartRDD.map(t => (t.getString(8), t)).persist()
val stationNoPartRDD = stationRDD.map(s => (s.getString(0), s)).persist()
stationNoPartRDD.join(tripPartRDD).count()
stationNoPartRDD.join(tripNoPartRDD).count()
```

```
1 tripPartRDD.join(stationNoPartRDD).count()
```

► (1) Spark Jobs

```
res11: Long = 286834
```

```
Command took 0.63 seconds -- by krzysztof.jankiewicz@put.
```

Informacje na temat przenoszenia danych

- Programista korzystając ze środowiska REPL uzyskuje na bieżąco informacje na temat efektów definiowanych transformacji.

```
import org.apache.spark.HashPartitioner
val stationPartRDD = stationRDD.map(s=>(s.getString(0),s)).partitionBy(new HashPartitioner(8)).persist()
val tripPartRDD = tripRDD.map(t => (t.getString(8), t)).partitionBy(new HashPartitioner(8)).persist()

stationPartRDD: org.apache.spark.rdd.RDD[(String, org.apache.spark.sql.Row)] = ShuffledRDD[111] . . .
tripPartRDD: org.apache.spark.rdd.RDD[(String, org.apache.spark.sql.Row)] = ShuffledRDD[113] . . .
```

- Oprócz tego, za pomocą metody `toDebugString` może on uzyskać informacje na temat operacji wymaganych do uzyskania wyników transformacji

```
stationPartRDD.join(tripPartRDD).toDebugString
```

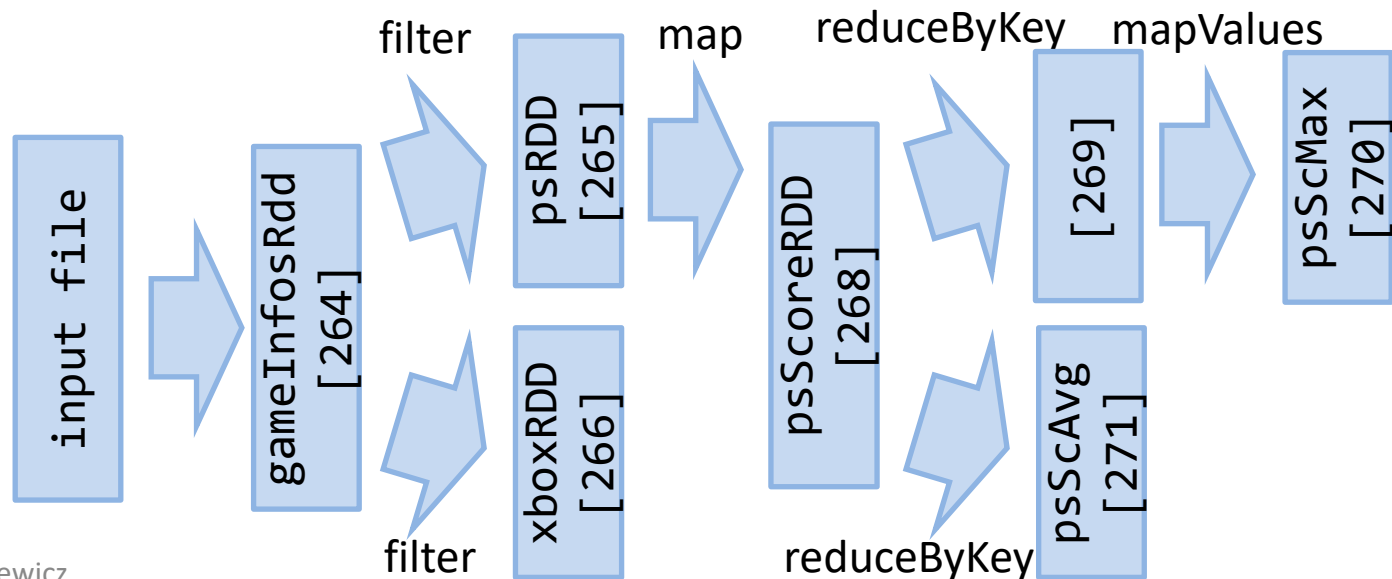
```
res17: String =
(8) MapPartitionsRDD[166] at join at <console>:55 []
| MapPartitionsRDD[165] at join at <console>:55 []
| CoGroupedRDD[164] at join at <console>:55 []
| ShuffledRDD[111] at partitionBy at <console>:50 []
|   CachedPartitions: 8; MemorySize: 33.1 KB; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
+-(1) MapPartitionsRDD[110] at map at <console>:50 []
| MapPartitionsRDD[109] at rdd at <console>:58 []
|   CachedPartitions: 1; MemorySize: 22.8 KB; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
| MapPartitionsRDD[108] at rdd at <console>:58 []
| MapPartitionsRDD[107] at rdd at <console>:58 []
| FileScanRDD[106] at rdd at <console>:58 []
| ShuffledRDD[113] at partitionBy at <console>:51 []
|   CachedPartitions: 8; MemorySize: 499.8 MB; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
+-(8) MapPartitionsRDD[112] at map at <console>:51 []
| MapPartitionsRDD[97] at rdd at <console>:52 []
|   CachedPartitions: 8; MemorySize: 211.6 MB; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
```

Wąskie i szerokie zależności

DAG

- Zależności pomiędzy postaciami danych uzyskiwanymi na poszczególnych etapach przetwarzania RDD reprezentowane są przez DAG (*Directed Acyclic Graph*)

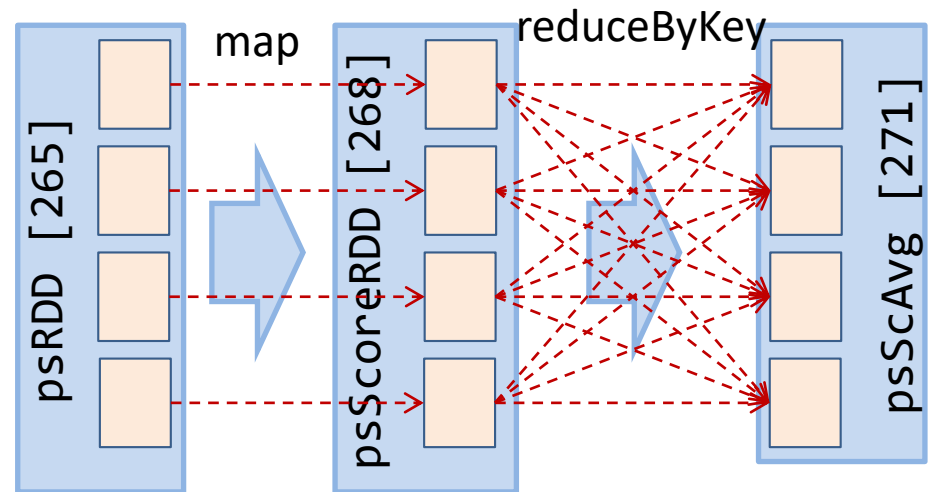
```
val gameInfosRdd=spark.read . . . // MapPartitionsRDD[264]
val psRDD = gameInfosRdd.filter(gi => gi.platform.startsWith("PlayStation")) // MapPartitionsRDD[265]
val xboxRDD = gameInfosRdd.filter(gi => gi.platform.startsWith("Xbox")) //266]
val psScoreRDD = psRDD.map(gi => (gi.platform,(1,gi.score))).persist() //268]
val psScoreAvgRDD = psScoreRDD.reduceByKey((p1,p2) => (p1._1 + p2._1, p1._2 + p2._2)). // MapPartitionsRDD[270]
    mapValues(p => p._2/p._1)
val psScoreMaxRDD = psScoreRDD.reduceByKey((p1,p2) => (1, math.max(p1._2,p2._2))) // ShuffledRDD[271]
```



Wąskie i szerokie zależności

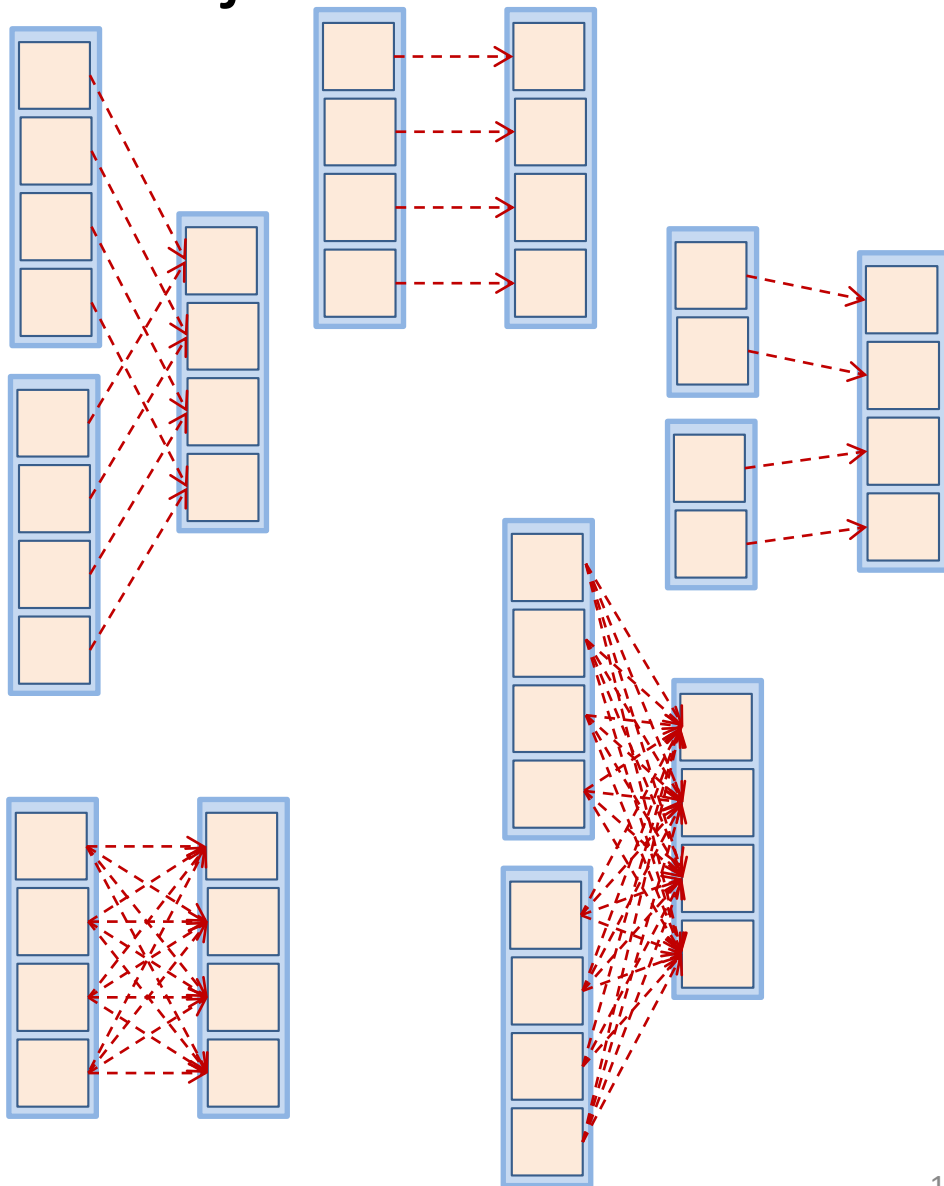
Reprezentacja RDD

- RDD reprezentowane są przez:
 - **Partycje**
 - **Zależności**
 - pomiędzy RDD
 - pomiędzy poszczególnymi partycjami
 - **Funkcje** wykorzystane do transformacji RDD
 - **Metadane** dotyczące tego gdzie RDD się znajdują i jaki schemat partycjonowania wykorzystują
- Zależności na poziomie partycji mogą być w dwóch formach
 - **wąskie** zależności – gdy dane z jednej partycji źródłowej wykorzystywane są przez jedną partycję docelową
 - **szerokie** zależności – gdy dane z jednej partycji źródłowej wykorzystywane są przez wiele partycji docelowych
- Wąskie zależności
 - nie wymagają przesyłania danych pomiędzy węzłami
 - możliwe potokowe przetwarzanie
- Szerokie zależności
 - wolniejsze
 - wymagają przesyłania danych



Wąskie i szerokie zależności transformacje

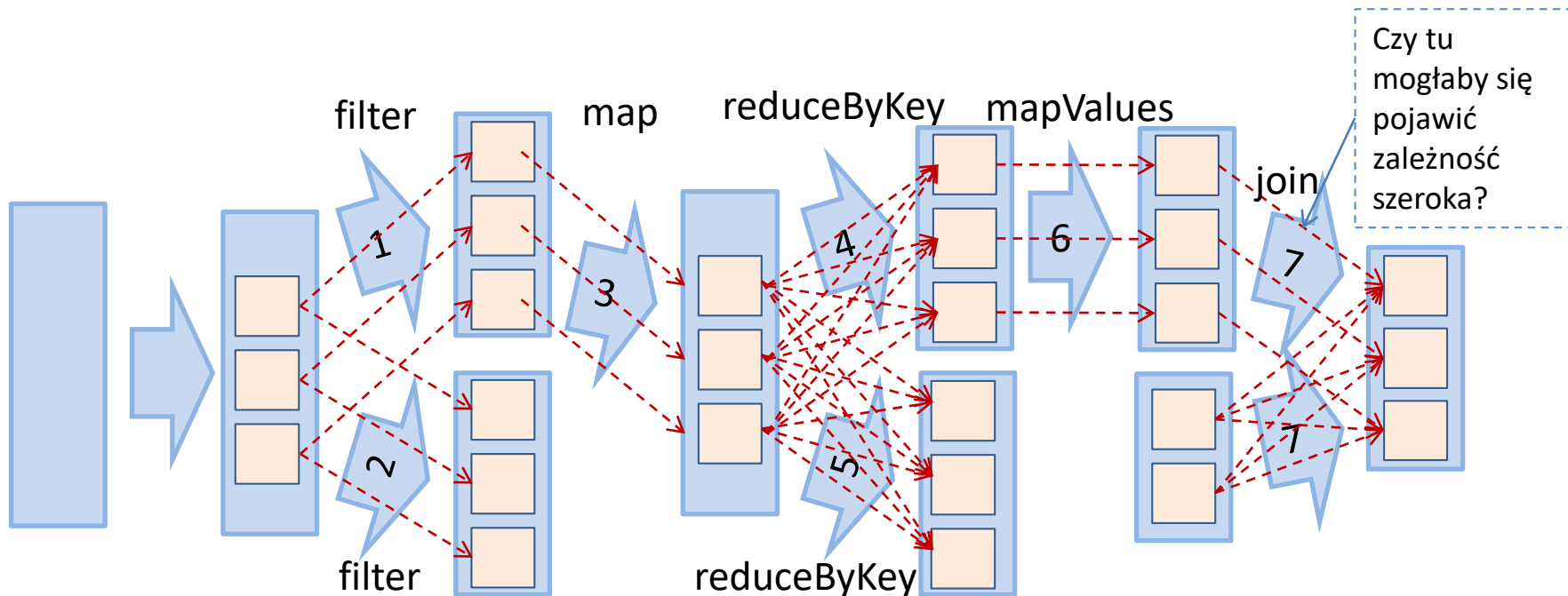
- Przykłady transformacji generujących wąskie zależności
 - join (w przypadku takiego samego partycjonowania obu źródeł danych)
 - map, flatMap, mapValues, flatMapValues
 - filter
 - union
 - mapPartitions
- Przykłady transformacji generujących szerokie zależności
 - groupByKey
 - join (w przypadku gdy jedno lub wiele źródeł nie jest partycjonowane zgodnie z partycjonowaniem wynikowym)
 - reduceByKey
 - distinct
 - intersection
 - repartition
 - coalesce
 - groupWith
 - cogroup



Wąskie i szerokie zależności

DAG

```
val gameInfosRdd=spark.read . . . // MapPartitionsRDD[264]
val psRDD = gameInfosRdd.filter(gi => gi.platform.startsWith("PlayStation")) // MapPartitionsRDD[265]
val xboxRDD = gameInfosRdd.filter(gi => gi.platform.startsWith("Xbox")) //266]
val psScoreRDD = psRDD.map(gi => (gi.platform,(1,gi.score))).persist() //268]
val psScoreAvgRDD = psScoreRDD.reduceByKey((p1,p2) => (p1._1 + p2._1, p1._2 + p2._2)). // MapPartitionsRDD[270]
    mapValues(p => p._2/p._1)
val psScoreMaxRDD = psScoreRDD.reduceByKey((p1,p2) => (1, math.max(p1._2,p2._2))) // ShuffledRDD[271]
val psScoreMaxByCountryRDD = psScoreMaxRDD.join(platformInfoWithCountryRDD)
```



Wąskie i szerokie zależności

Uzyskiwanie informacji

- RDD dostarcza metodę `dependencies`, za pomocą której można uzyskać informacje na temat zależności partycji obiektu od partycji obiektów źródłowych
- Typy wąskich zależności (`NarrowDependency`)
 - `OneToOneDependency`
 - `PruneDependency`
 - `RangeDependency`
- Typ szerokich zależności – `ShuffleDependency`

```
val psRDD = gameInfosRdd.filter(. . .  
psRDD.dependencies  
List(org.apache.spark.OneToOneDependency@10d90b6a)
```

```
val psScoreRDD = psRDD.map(. . .  
psScoreRDD.dependencies  
List(org.apache.spark.OneToOneDependency@3a9f27d0)
```

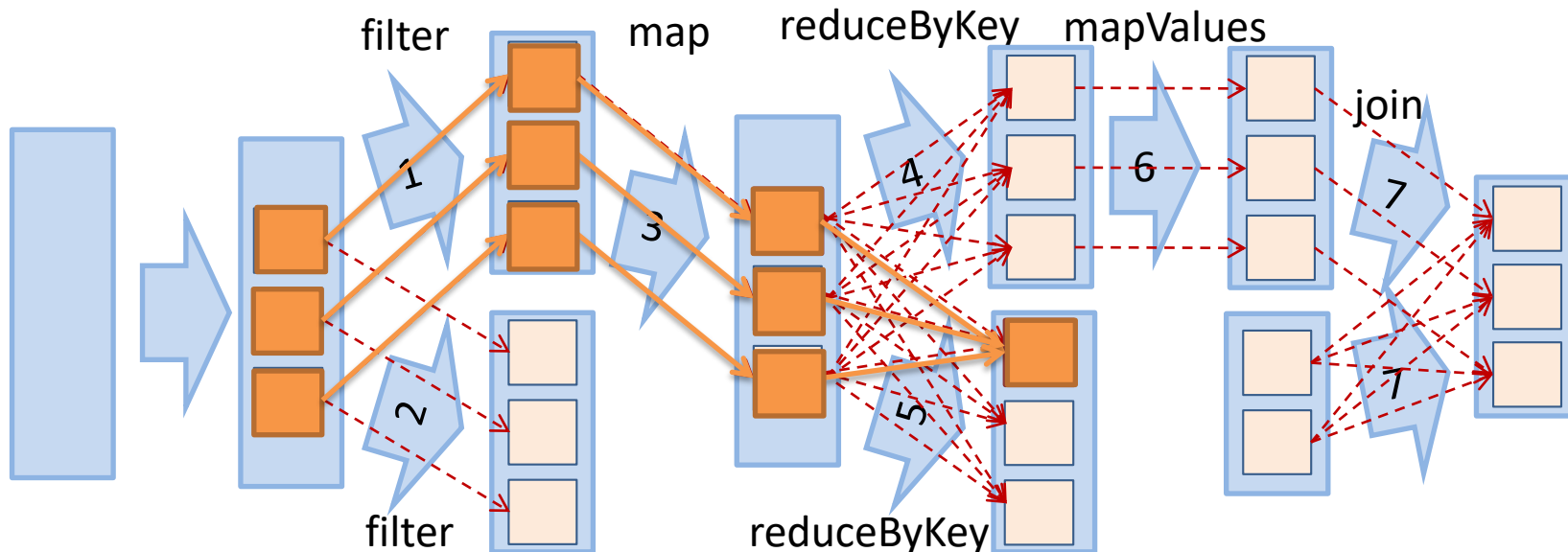
```
val psScoreMaxRDD = psScoreRDD.reduceByKey(. . .  
psScoreMaxRDD.dependencies  
List(org.apache.spark.ShuffleDependency@56f7fbca)
```

`PruneDependency` – wynikowe RDD (`PartitionPruningRDD`) jest podzbiorem partycji źródłowych

`RangeDependency` – wynikowe RDD (np.: `UnionRDD`) posiada partycje oparte na źródłowych partycjach w relacji 1 do 1

Typy zależności a awarie

- Graf zależności i ich typy określa sposób rozwiązywania sytuacji awaryjnych
- Złożone przetwarzanie *Hadoop* oparte o MapReduce zabezpiecza się przed sytuacjami awaryjnymi zapisując do HDFS wyniki pośrednich zadań
- Spark rozwiązuje za pomocą przeliczania odpowiednich partycji RDD na podstawie odpowiednich partycji źródłowych
- Ponowne przeliczanie partycji jest proste (szybkie) dla zależności wąskich, podczas gdy dla zależności szerokich są one złożone i wolniejsze



Domknięcia funkcji (*closures*)

- W Scali funkcje mogą korzystać ze zmiennych zdefiniowanych poza ich definicją
- W przypadku Sparka ta własność staje się bardziej skomplikowana, wynika to z dwóch powodów
 - funkcje delegowane są do wykonawców i tam wykonywane
 - zmienne definiowane są po stronie programu sterownika

```
def parse(line: String, subs: String): (String,String) = {  
    val i = line.indexOf(subs)  
    val title = line.substring(13, i) // <page><title>  
    val text  = line.substring(i + subs.length, line.length-14) // </text></page>  
    (title, text)  
}  
val subs = "</title><text>"  
val wikiTitleTextRdd = wikiLinesRdd.map(line => parse(line, subs))
```

- Spark rozwiązuje ten problem następująco:
 - każde zadanie uruchomione w klastrze otrzymuje kopię zmiennej dla każdego uruchomienia kodu, w którym następuje odwołanie do zmiennej (rozmiar!)
 - aktualizacje takich zmiennych są aktualizacjami kopii i nie wracają do sterownika
- Rozwiązanie takie jest w wielu przypadkach niewystarczające dlatego Spark udostępnia dodatkowo dwa specjalne typy zmiennych
 - zmienne rozgłoszeniowe
 - akumulatory

Zmienne rozgłoszeniowe

- Tworzone są za pomocą metody broadcast obiektu SparkContext
- Są obiektem `spark.broadcast.Broadcast[T]`
- Ich wartość możemy uzyskać za pomocą metody `value`

```
import org.apache.spark.broadcast._

def parse(line: String, subs: Broadcast[String]): (String,String) = {
  val i = line.indexOf(subs.value)
  val title = line.substring(13, i)
  val text = line.substring(i + subs.value.length, line.length-14)
  (title, text)
}

val subs = sc.broadcast("</title><text>")
val wikiTitleTextRdd = wikiLinesRdd.map(line => parse(line, subs))
```

- Wysyłane są do węzła tylko jeden raz
- Powinny być traktowane jako zmienne tylko do odczytu
- Zmienne muszą być `Serializable`, ze względów optymalizacyjnych serializacja jak i deserializacja powinna być maksymalnie wydajna

Akumulatory

- Podstawowym celem jest agregacja wartości z węzłów roboczych w programie sterownika
- Częstym przypadkiem użycia jest zliczanie zdarzeń jakie pojawiają się podczas wykonywania zadań w węzłach roboczych (np. w celu debugowania)
- Można korzystać z akumulatorów
 - wbudowanych dla typów `long` (`LongAccumulator`), `double` (`DoubleAccumulator`) lub kolekcji (`CollectionAccumulator`)
 - własnych, rejestrowanych za pomocą metod `SparkContext.register`

```
val ints = sc.parallelize(List(1,2,3,4,5,6,7,"B",9,0))
val notInts = spark.sparkContext.longAccumulator

val sumOfInts = ints.reduce((acc,n) => {
  val y = acc match {
    case x: Int => x
    case _ => {notInts.add(1); 0 }
  }
  n match {
    case x: Int => y + x
    case _ => {notInts.add(1); y }
  })
sumOfInts: Any = 37

notInts.value
res28: Long = 1
```

Od wersji Sparka 2.0 akumulatory posiadają bogate API, pozwalające np. na przechowywanie metadanych

```
scala> val notInts =
spark.sparkContext.longAccumulator("
Counter of not Ints")
notInts:
org.apache.spark.util.LongAccumulator = LongAccumulator(id: 6089, name:
Some(Counter of not Ints), value: 0)
```

Podsumowanie

- Czas dostępu do danych
- *Shuffle operations* – przesyłanie danych
- Partycjonowanie
- Wąskie i szerokie zależności
- Domknięcia
 - zmienne rozgłoszeniowe
 - akumulatory
- Pamiętajmy także o:
 - `cache()`, `persistent()`
 - zmniejszaniu wolumenu przetwarzanych danych przez
 - filtrowanie jak najwcześniejszym etapie (*Filter Pushdown*)
 - agregację na jak najwcześniejszym etapie