Spark Streaming

Krzysztof Jankiewicz

Plan

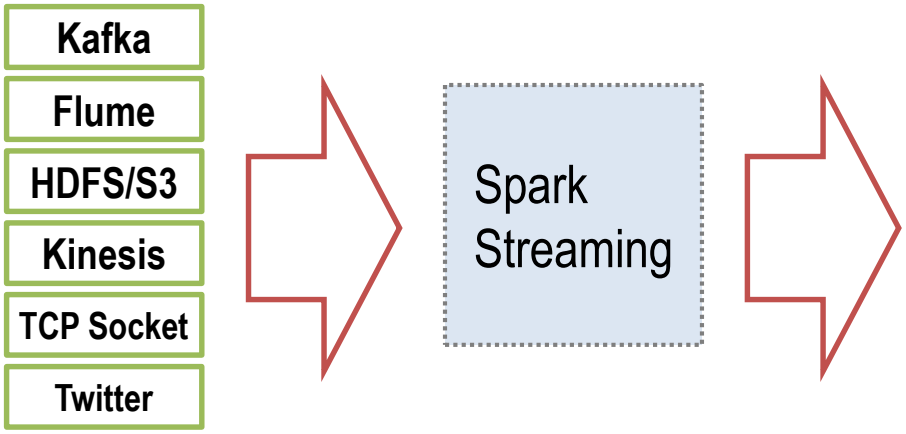
- Wprowadzenie
- Cele
- Podstawy
- Architektura
- Źródła danych, strumienie wejściowe, odbiorniki
- Transformacje
  - bezstanowe
  - stanowe
- Operacje wynikowe
- Odporność na awarie – aplikacje 24/7

Wprowadzenie

- W 2015 roku Databricks, firma założona przez twórców Sparka, przeprowadziła badanie wśród swoich użytkowników.
  Okazało się, że 56% z nich wykorzystuje mechanizmy Spark Streaming, a 48% uważa, że jest to jeden z najważniejszych komponentów Sparka w ich biznesie.
- Do aktywnych użytkowników Spark Streaming zaliczają się dla przykładu:
  - Uber (analizy danych),
  - Netflix (systemy rekomendacji) czy
  - Pinterest (systemy rekomendacji).

Czym jest Spark Streaming?

- Spark Streaming został dodany do rdzenia Spark API w 2013 roku.
- Pozwala na implementację przetwarzania strumieni danych, która charakteryzuje się skalowalnością, dużą przepustowością i odpornością na błędy
- Dane mogą być pobierane z wielu źródeł np.: Kafka, Flume, Kinesis, HDFS, a także gniazd TCP
- Uzyskane dane mogą być przetwarzane przy użyciu złożonych algorytmów zdefiniowanych za pomocą funkcji wysokiego poziomu takich jak: `map`, `reduce`, `join` czy `window`.
- Przetwarzanie strumieni danych można wykorzystać także implementacji przetwarzania grafów czy mechanizmów uczenia maszynowego.
- Ostatecznie przetworzone dane mogą trafiać do systemu plików, baz danych, pulpitów menadżerskich.



Główne języki Spark Streaming to Scala i Java.

Python posiada szereg niezaimplementowanych fragmentów API

Cele

- **ETL czasu rzeczywistego** – dane są w sposób ciągły transformowane i dostarczane do systemów docelowych.
- **Wyzwalanie akcji** – detekcja zdarzeń w czasie rzeczywistym i podejmowanie odpowiednich akcji.
- **Ubogacanie danych** – dane pojawiające się w systemach źródłowych uzupełniane są o dodatkowe informacje pochodzące ze statycznych źródeł pozwalając na bogatszą analizę czasu rzeczywistego.
- Obsługa złożonych sesji oraz **systemów ciągłego uczenia się** – zdarzenia powiązane z bieżącymi sesjami użytkownika (np. w ramach stron WWW lub aplikacji) są grupowane i analizowane. Uzyskiwana w ten sposób informacja może być wykorzystywana do regularnej aktualizacji modeli uczenia maszynowego.

Podstawy

- Podstawą Spark Streaming są dyskretne strumienie danych – `DStreams`.
- `DStreams`
  - to sekwencja danych (RDD), których porcje pojawiają się w określonym momencie czasu,
  - mogą pochodzić z bardzo wielu źródeł,
  - dostarczają dwa typy operacji – *transformacje* oraz *operacje wynikowe* (które zapisują dane do zewnętrznego systemu)
  - dostarczają operacje, które w dużej mierze przypominają operacje dostępne są dla RDD
  - jest tym dla Spark Streaming czym RDD jest dla Sparka
- Aplikacje Spark Streaming w odróżnieniu od aplikacji Sparka **funkcjonują w sposób ciągły** (24/7)

Architektura

- Spark Streaming opiera się na architekturze "micro-batch"
- Obliczenia strumieniowe traktowane są jako seria obliczeń realizowana na kolejnych porcjach danych
- Dane spływające z różnych systemów są zbierane w ramach określonych interwałów (*batch interwal*), co pozwala utworzyć porcje danych (*batches*) podlegające przetwarzaniu.

by Spark Examples

```
import org.apache.spark.SparkConf
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext}
object NetworkWordCount {
  def main(args: Array[String]) {
    if (args.length < 2) {
      System.err.println("Usage: NetworkWordCount <hostname> <port>")
      System.exit(1)
    }

    val sparkConf = new SparkConf().setAppName("NetworkWordCount")
    val ssc = new StreamingContext(sparkConf, Seconds(1))

    val lines = ssc.socketTextStream(args(0), args(1).toInt, StorageLevel.MEMORY_AND_DISK_SER)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
    wordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

- Skąd będzie pochodziło źródło?
- Jak długo dane będą buforowane?
- Na czym będą polegały operacje dokonywane na porcjach danych?

```
val lines – ReceiverInputDStream[String]
val words - DStream[String]
val wordCounts: DStream[(String, Int)]
```

Architektura – przykład

Terminal 1	Terminal 2
`./bin/run-example streaming.NetworkWordCount localhost 9999`	
	`nc -lk 9999`

`-------------------------------------------`
`Time: 1506602018000 ms`
`-------------------------------------------`	

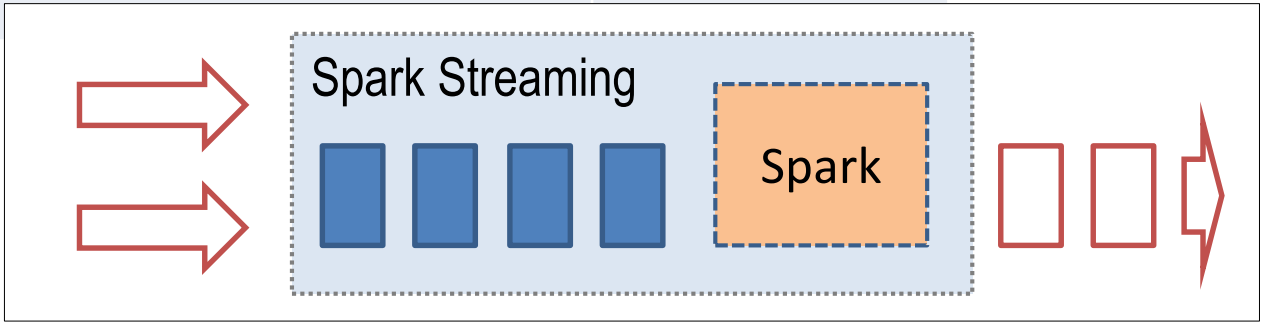
	`hello hello again`

`-------------------------------------------`
`Time: 1506602019000 ms`
`-------------------------------------------`
`(again,1)`
`(hello,2)`	

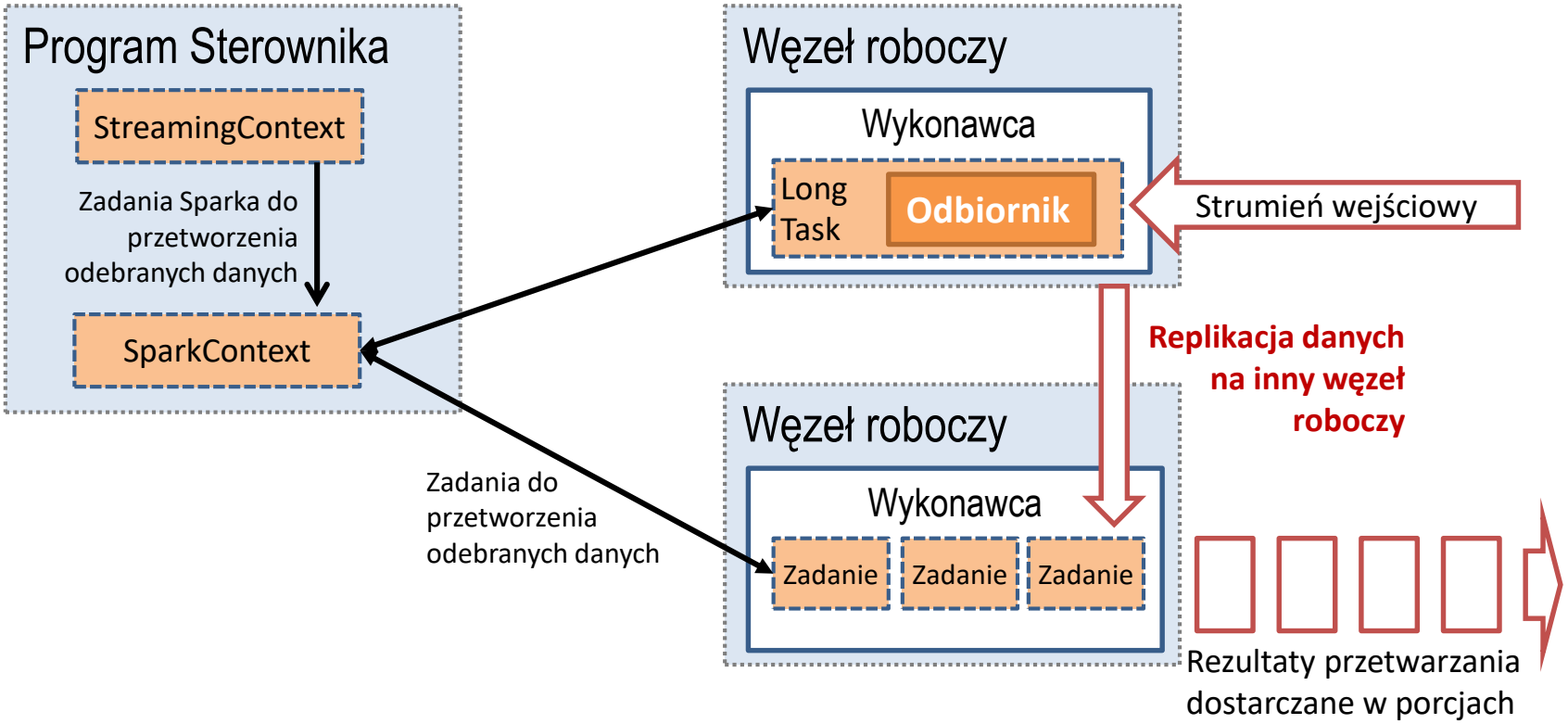
`-------------------------------------------`
`Time: 1506602020000 ms`
`-------------------------------------------`	

	`hello`

`-------------------------------------------`
`Time: 1506602021000 ms`
`-------------------------------------------`
`(hello,1)`	



Architektura



StreamingContext

- Aby uruchomić program Spark Streaming konieczne jest utworzenie obiektu `StreamingContext`
- Podczas tworzenia obiektu `StreamingContext` określany jest interwał, którego wartość zależna jest charakteru aplikacji oraz dostępnych zasobów.

```
val sparkConf = new SparkConf().setAppName("NetworkWordCount")
val ssc = new StreamingContext(sparkConf, Seconds(1))
```

- Po utworzeniu kontekstu programista musi:
  - utworzyć źródła danych wejściowych tworząc wejściowe DStreams,

```
val lines = ssc.socketTextStream(args(0), args(1).toInt, StorageLevel.MEMORY_AND_DISK_SER)
```

  - zdefiniować obliczenia na strumieniach danych obejmujące transformacje

```
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
```

    i operacje wynikowe

```
wordCounts.print()
```

  - rozpocząć odbiór danych i przetwarzanie za pomocą `streamingContext.start()`

```
ssc.start()
```

  - oczekiwać na zakończenie przetwarzania (ręcznie za pomocą lub w wyniku błędu) za pomocą `streamingContext.awaitTermination()`

```
ssc.awaitTermination()
```

  - zatrzymać ręcznie przetwarzanie można za pomocą `streamingContext.stop()`

Patrz:
https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.StreamingContext

InputDStreams i Receiver

```
val lines = ssc.socketTextStream(args(0), args(1).toInt, StorageLevel.MEMORY_AND_DISK_SER)
```

- Wejściowe DStreams reprezentują dane odbierane z systemów zewnętrznych
- Każdy z wejściowych strumieni jest powiązany z obiektem odbiornika (`Receiver`). Wyjątkiem są strumienie plików.
- Uruchamiając aplikację Spark Streaming lokalnie, należy zapewnić możliwość odbioru i jednoczesnego przetwarzania strumieni alokując odpowiednią liczbę rdzeni (większą niż liczba odbiorników)

- Odbiornik odpowiedzialny jest za odbieranie danych ze źródeł i składowanie ich w pamięci Sparka
- Istnieją dwa wbudowane typy strumieni wejściowych
  - Oparte na **prostych źródłach** – źródła dostępne są wówczas bezpośrednio za pomocą *StreamingContext API*: strumienie plików oraz strumienie z gniazd TCP
  - Oparte na **zaawansowanych źródłach** np.: Kafka, Flume, Kinesis itp. – dostępne za pomocą dodatkowych zewnętrznych klas, z koniecznością uwzględnienia zależności pomiędzy klasami
- Istnieją dwa typy źródeł, zależne od poziomu wiarygodności (*reliability*).
  Dla przykładu Kafka i Flume mogą oczekiwać potwierdzania odbioru danych.
  W zależności od sytuacji konieczne jest użycie wiarygodnego lub nie odbiornika

Custom Receivers oraz Reliable Receiver vs Unreliable Receiver – patrz:
`https://spark.apache.org/docs/latest/streaming-custom-receivers.html`

Plikowe źródła danych

- Dla plikowych źródeł danych (HDFS, S3, NFS itp.) strumień wejściowy może być utworzony za pomocą:

```
streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)
```

lub w przypadku prostych plików tekstowych

```
streamingContext.textFileStream(dataDirectory)
```

- Spark Streaming monitoruje w takim przypadku wskazany katalog i przetwarza pliki, które bezpośrednio w tym katalogu się pojawiają
- Istotne uwagi:
  - wszystkie pliki muszą mieć ten sam format,
  - pliki w katalogu muszą być utworzone w sposób atomowy – np. w wyniku przeniesienia pliku (lub zmiany nazwy)
  - pliki nie mogą zmieniać swojej zawartości – dodawane dane nie będą przetwarzane

by Spark Examples

```scala
object HdfsWordCount {
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: HdfsWordCount <directory>")
      System.exit(1)
    }

    StreamingExamples.setStreamingLogLevels()
    val sparkConf = new SparkConf().setAppName("HdfsWordCount")

    val ssc = new StreamingContext(sparkConf, Seconds(2))

    val lines = ssc.textFileStream(args(0))
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
    wordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

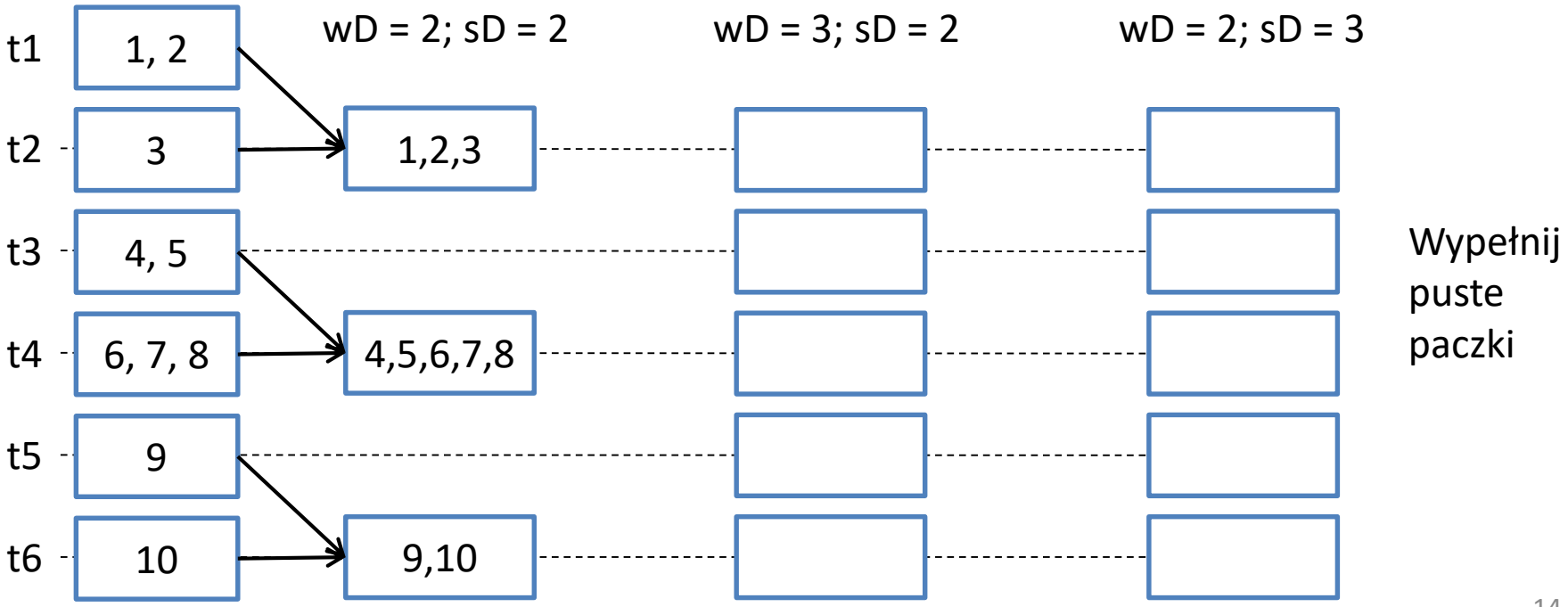
Transformacje

- Transformacje `DStreams` dzielą się na:
  - **Bezstanowe** (*stateless*) – przetwarzają porcje danych bez uwzględniania porcji wcześniejszych
  - **Stanowe** (*stateful*) – przetwarzają porcje danych z wykorzystaniem danych w porcjach wcześniejszych
- Transformacje bezstanowe
  - Są prostymi transformacjami RDD przetwarzającymi kolejne porcje danych. Przykłady: `map()`, `flatMap()`, `filter()`, `repartition()`, `reduce()`, `reduceByKey()`, `groupByKey()`, `count()`, `countByValue()`, `join()`, `union()`
  - Jeżeli zestaw dostępnych transformacji jest niewystarczający, wówczas należy wykorzystać operator `transform()`, który jako argument przyjmuje dowolną funkcję przekształcającą RDD w inny RDD
- Transformacje stanowe
  - Wykorzystują poprzednie porcje danych do utworzenia nowych porcji
  - Wymagają większej dbałości o zapewnienie odporności na awarie (patrz: punkty kontrolne)
  - Dwa podstawowe typy operacji stanowych:
    - operacje, które działają na zasadzie **okna przesuwnego**
    - **`updateStateByKey()`**, która aktualizuje stan przetwarzania na podstawie kolejnych porcji danych.

Transformacje stanowe

Operacje okna przesuwnego

- Pozwalają na dokonywanie obliczeń obejmujących wiele paczek danych
- Wymagają dwóch argumentów, które muszą być wielokrotnością interwału tworzenia paczek (*batchInterval*)
  - długości okna (*windowDuration*) – decyduje o tym ile poprzednich paczek będzie branych pod uwagę (*windowDuration/batchInterval)*
  - długości poślizgu (*slidingDuration*) – określa jak często wynikowa paczka (nowy `DStream`) będzie tworzona – domyślną wartością jest *batchInterval*



Transformacje stanowe

Operacje okna przesuwnego

- Najprostszą operacją okna przesuwnego jest

```
window(wD: Duration, sD: Duration): DStream[T]
```

- Na jej wyniku można tworzyć kolejne poziomy przetwarzania

```
ints.window(Seconds(30), Seconds(10)).reduce(_ + _)
```

- Ponadto Spark Streaming dostarcza szereg specjalizowanych operacji np.:

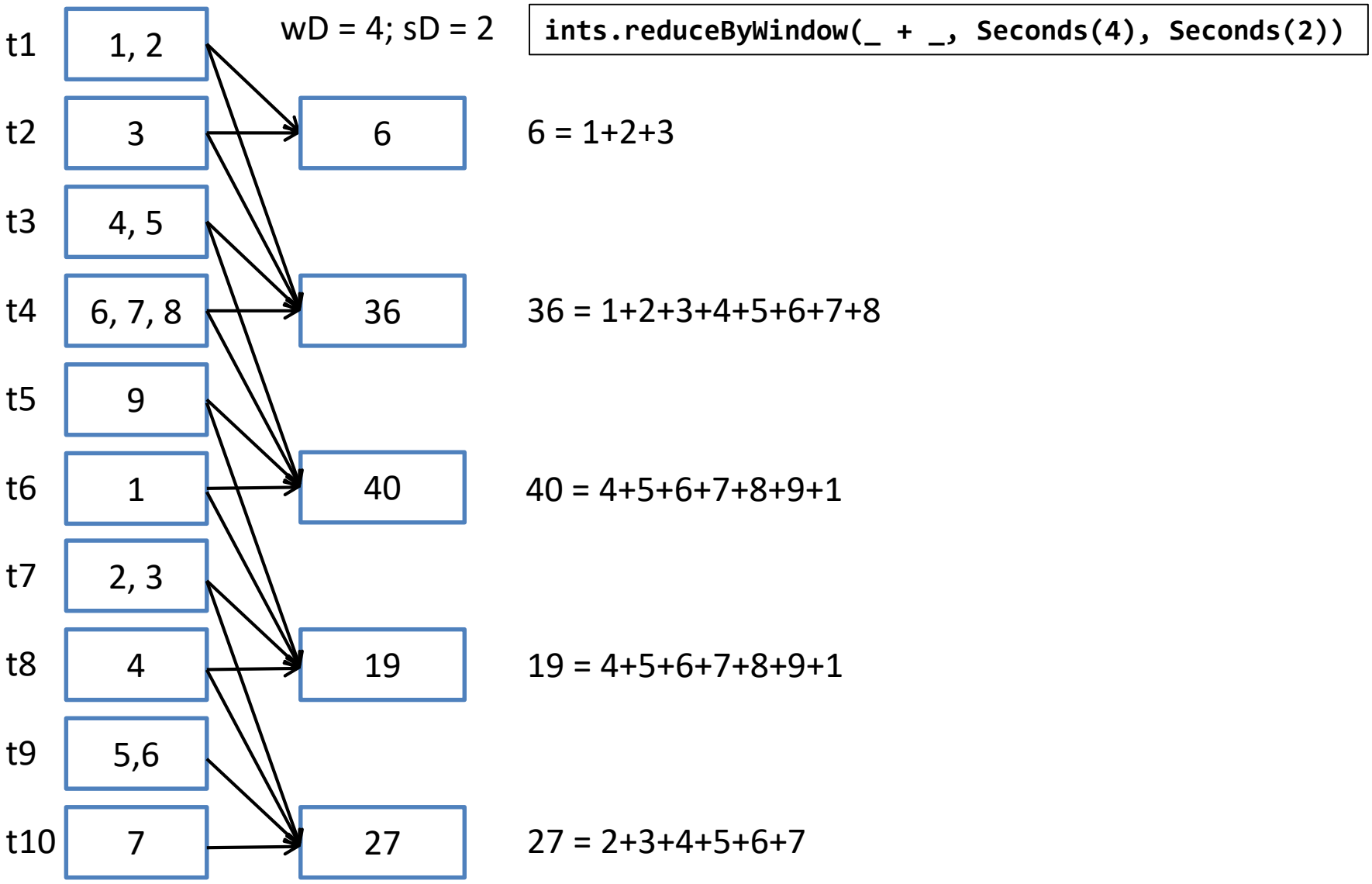
```
reduceByWindow(reduceFunc: (T, T) ⇒ T,
                        wD: Duration, sD: Duration): DStream[T]
reduceByKeyAndWindow(reduceFunc: (V, V) ⇒ V,
                        wD: Duration, sD: Duration): DStream[(K, V)]
```

```
ints.reduceByWindow(_ + _, Seconds(30), Seconds(10))
```

- W przypadku dużych okien warto rozważyć **przyrostowe** uzyskiwanie wyników powyższych funkcji przez zastosowanie odwrotnych (*inverse*) funkcji redukujących

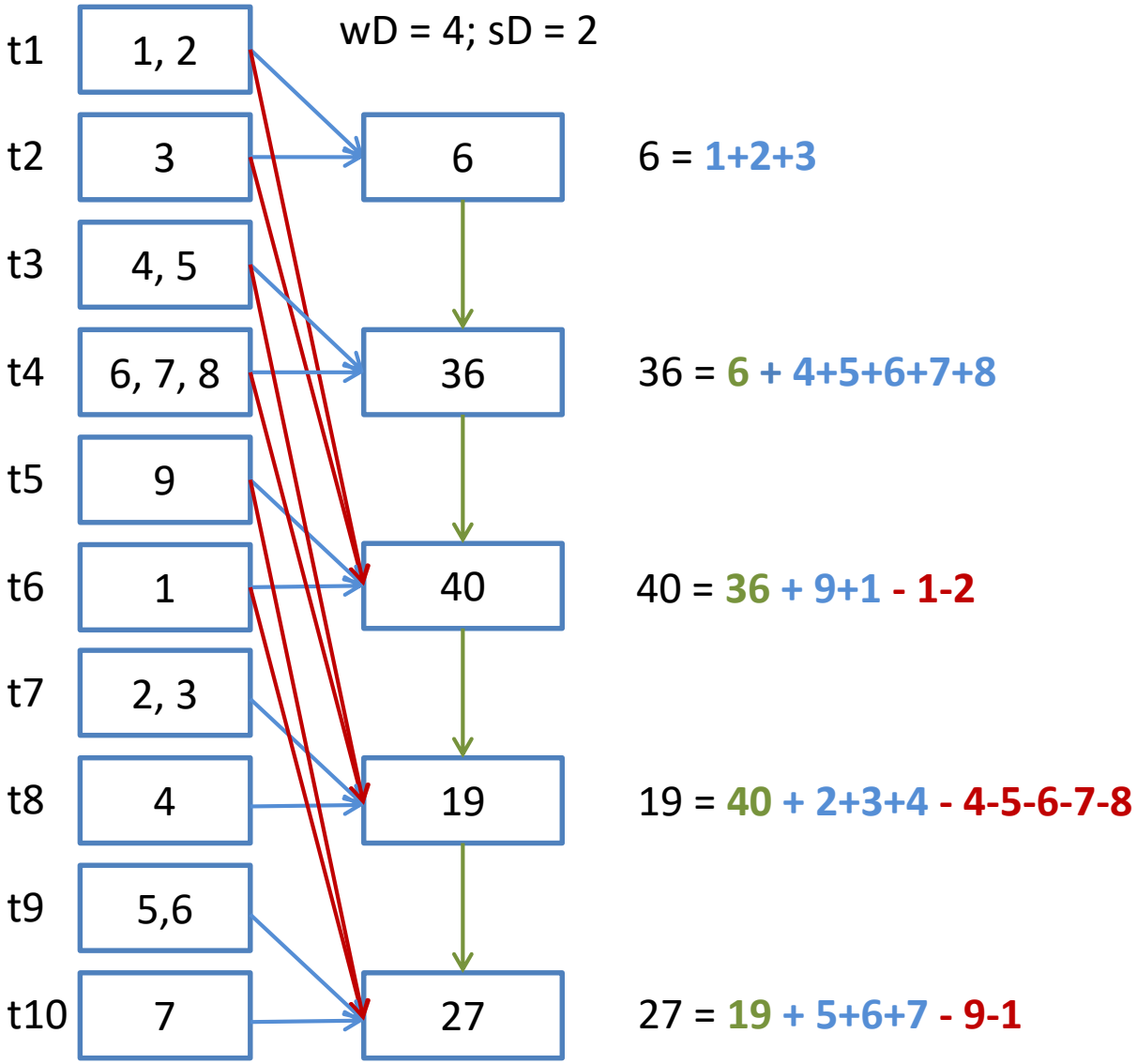
```
reduceByWindow(reduceFunc: (T, T) ⇒ T, invReduceFunc: (T, T) ⇒ T,
                        wD: Duration, sD: Duration): DStream[T]
reduceByKeyAndWindow(reduceFunc: (V, V) ⇒ V, invReduceFunc: (V, V) ⇒ V,
                        wD: Duration, sD: Duration): DStream[(K, V)]
```

Pełne uzyskiwanie wyników



Przyrostowe uzyskiwanie wyników

```
ints.reduceByWindow(_ + _, _ - _, Seconds(4), Seconds(2))
```



updateStateByKey

```
import org.apache.spark.SparkConf
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext}

object NetworkWordCount {
  def updateFunc(values: Seq[Int], state: Option[Int]): Option[Int] = {
     val currentCount = values.sum
     val previousCount = state.getOrElse(0)
     Some(currentCount + previousCount)
   }

  def main(args: Array[String]) {
    if (args.length < 2) {
      System.err.println("Usage: NetworkWordCount <hostname> <port>")
      System.exit(1)
    }

    val sparkConf = new SparkConf().setAppName("NetworkWordCount")
    val ssc = new StreamingContext(sparkConf, Seconds(1))

    val lines = ssc.socketTextStream(args(0), args(1).toInt, StorageLevel.MEMORY_AND_DISK_SER)
    val words = lines.flatMap(_.split(" "))
    val pairs = words.map(word => (word, 1))
     val globalCountStream = pairs.updateStateByKey(updateFunc)
    globalCountStream.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

Drugą z podstawowych typów operacji stanowych jest **updateStateByKey()**, która aktualizuje stan przetwarzania na podstawie kolejnych porcji danych.

Operacje wynikowe

- Głównym celem operacji wynikowych jest dostarczanie wynikowych danych do systemów docelowych
- Operacje wynikowe wyzwalają rzeczywiste uruchomienie przetwarzania za pomocą transformacji (leniwych) – analogicznie jak akcje w RDD
- Lista operacji wynikowych:
  - `print()` – głównie w celach testowych i projektowych
  - `saveAsTextFiles(prefix: String, suffix: String = ""): Unit`
  - `saveAsObjectFiles(prefix: String, suffix: String = ""): Unit`
  - `saveAsHadoopFiles[F <: OutputFormat[K, V]](prefix: String, suffix: String) (implicit fm: ClassTag[F]): Unit`
  - `foreachRDD(foreachFunc: (RDD[T]) ⇒ Unit): Unit` – najbardziej ogólny i wszechstronny operator, który uruchamia `foreachFunc` dla każdego RDD pojawiającego się w strumieniu. Uwaga! funkcja `foreachFunc` wykonywana jest po stronie węzła roboczego!

```
. . .
import org.apache.spark.sql.SparkSession
object NetworkWordCount {
  def main(args: Array[String]) {
    . . .
    val words = lines.flatMap(_.split(" "))
    words.foreachRDD { rdd =>
      val spark = SparkSession.builder.config(rdd.sparkContext.getConf).getOrCreate()
      import spark.implicits._
      val wordsDF = rdd.toDF("word")
      wordsDF.createOrReplaceTempView("words")
      val wordCountsDF =
        spark.sql("select word, count(*) as total from words group by word")
      wordCountsDF.show()
    }
    . . .
  }
}
```

Odporność na awarie – aplikacje 24/7

- Aplikacje oparte na strumieniach danych działają w trybie 24/7, w związku z czym muszą być one wyjątkowo odporne na błędy (systemowe, JVM itp.)
- Aby to uzyskać aplikacje Spark Streaming **zapisuje** pewne informacje na **nośnikach** zdolnych przetrwać awarie. Taki zapis nosi nazwę **punktu kontrolnego**.
- Punkty kontrolne wykorzystywane są w szczególności z dwóch powodów:
  - aby **materializować stan obliczeń** w celu uniknięcia czasochłonnego przeliczania tego stanu z danych źródłowych
  - umożliwić ponowne uruchomienie programu sterownika. Po załamaniu aplikacji, można ją zrestartować wskazując miejsce zapisu punktów kontrolnych skąd Spark Streaming będzie wiedział w jakim momencie przetwarzanie zostało przerwane
- W związku z powyższym rozróżnia się dwa typy punktów kontrolnych:
  - punkty kontrolne **metadanych** (*metadata checkpointing*), zawierające:
    - dane dotyczące konfiguracji – wymagane do restartu aplikacji
    - operacje DStream – zbiór operacji definiujących aplikację przetwarzającą DStreams
    - "niedokończone" porcje danych – porcje danych, których przetwarzanie nie zostało zakończone
  - punkty kontrolne **danych** (*data checkpointing*), zawierające pośrednie etapy przetwarzania transformacji stanowych, dzięki czemu znacząco może zostać ograniczona:
    - liczba źródłowych porcji danych, oraz
    - czas i zasoby

    Punkty kontrolne danych wymagane są do odzyskania stanu przetwarzania

Odporność na awarie – aplikacje 24/7 szczegóły

- Wprowadzenie możliwości tworzenia punktów kontrolnych jest wymagane gdy:
  - wykorzystywane są transformacje stanowe
  - konieczna jest obsługa awarii programu sterownika
- Możliwość tworzenia punktów kontrolnych wymaga wskazania trwałego miejsca ich składowania

```
streamingContext.checkpoint(checkpointDirectory)
```

- Obsługa awarii programu sterownika wymaga podczas uruchamiania aplikacji sprawdzenia istnienia punktów kontrolnych

```
def main(args: Array[String]) {
  if (args.length < 3) {
    System.err.println("Usage: NetworkWordCount <hostname> <port> <checkpointDirectory>")
    System.exit(1)
  }
  def functionToCreateContext(): StreamingContext = {
    val sparkConf = new SparkConf().setAppName("NetworkWordCount")
    val ssc = new StreamingContext(sparkConf, Seconds(1))
    val lines = ssc.socketTextStream(args(0), args(1).toInt, StorageLevel.MEMORY_AND_DISK_SER)
    . . .
    ssc.checkpoint(args(2))
    ssc
  }
  val ssc = StreamingContext.getOrCreate(args(2), functionToCreateContext)
  ssc.start()
  ssc.awaitTermination()
}
```

Instalacja aplikacji 24/7 – wymagania

- Klaster
- Pakiet aplikacji Spark Streaming w JAR
- Wystarczająca ilość pamięci dla wykonawców
- Skonfigurowane punkty kontrolne
- Skonfigurowany mechanizm automatycznego restartowania aplikacji (możliwe są do wykorzystania różne mechanizmy: Spark, YARN, Mesos)
- Wykorzystywane wiarygodne źródła danych (mające możliwość ponownego wysłania danych w przypadku awarii)
- Wykorzystywane wiarygodne odbiorniki
- Skonfigurowany mechanizm *write ahead logs* – dzięki czemu odebrane dane są zapisywane do logów zanim odbiornik wyśle potwierdzenie odebrania danych
- Ustawiona maksymalna częstotliwość obsługi paczek danych

Patrz także:
`https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html`

Podsumowanie

- Cele
- Podstawy
- Architektura
- Źródła danych, strumienie wejściowe, odbiorniki
- Transformacje
  - bezstanowe
  - stanowe
- Operacje wynikowe
- Odporność na awarie – aplikacje 24/7