Apache Spark で分散コンピューティング

ファイファー・トビアス
Preferred Infrastructure
2014 年 8 月 21 日

自己紹介

- 名前：ファイファー　トビアス　　（Tobias Pfeiffer）
- 出身：ベルリン（ドイツ）
- 入社：２０１４年４月

- 最近の課題：Spark プログラミング

Spark と Scala 言語の関係

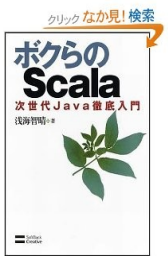
- Apache Spark は、Scala 言語で作った「fast and general engine for large-scale data processing」である。
- Spark と Scala の似ている概念は多い
- 講演の内容：
  - Scala 言語の入門
  - Spark の入門

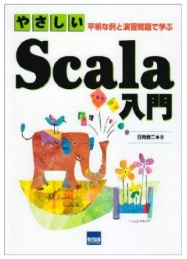
Scala 言語

Image Source: http://www.scalacity.ee

Scala 言語の歴史

- Scala は、２００１年からスイスのＥＰＦＬ大学で開発されている。（最近のバージョン： 2.11 ）
- プロジェクト・リーダーはＥＰＦＬの教授マーティン・オダースキー（ Java Generics の発明者）である。
- 有名なユーザ： Twitter, LinkedIn, Sony Pictures Imageworks, Foursquare





Scala と Java のエコシステム

- Scala コードは Java バイトコードにコンパイルされて、JVM の上で実行される。
- Scala と Java のライブラリは 100 %互換性がある
- Java エコシステムのツール（Maven 等）を使える

Java:

```
$ javac HelloWorldJavaApp.java
$ java HelloWorldJavaApp
Hello World!
```

Scala:

```
$ scalac HelloWorldScalaApp.scala
$ java HelloWorldScalaApp
Hello World!
```

Scala は object-functional な言語

- 全ての Java の OOP コンセプトある：クラス、オブジェクト、メソッド、継承等
- Trait（インタフェース＋機能、Mixin と似ている）もある。

クラス定義）

```
class Complex(val real: Double,
              val imag: Double) {
  // add other and return new
  def add(other: Complex) = {
    val newRe = real + other.real
    val newIm = imag + other.imag
    new Complex(newRe, newIm)
  }
}
```

- セミコロン要らない
- return も要らない
- 型指定も要らない

Scala は object-functional な言語

- 数学では、関数は入力値で出力値を計算する。
  - 入力値を変えない
  - 数学の関数は、「pass by reference」のコンセプトない
- もう１つのコンセプトは、・・・


L'IMMUTABILITÉ, C'EST CHIC.
Je vous souhaite de passer une bonne journée aujourd'hui.

Scala は object-functional な言語

- 数学では、関数は入力値で出力値を計算する。
  - 入力値を変えない
  - 数学の関数は、「pass by reference」のコンセプトがない
- プログラミングでイミュータビリティと副作用がない関数を使うと、
  - プログラムの正確性の証明
  - ユニット・テスト
  - 並列化

  簡単になる

```
val x = 5
x = 3  // does not compile

var y = 5
y = 3  // works, but bad style
```

宣言のとき値アサインする

- イミュータブルなので、宣言のとき値アサインは必要
- Scala では、全てのステートメントは返り値がある
- 時々、返り値のタイプは Unit（= void）である

返り値の例）

```
5                                  // Int
a == 0                             // Boolean
List(1, 2.5, 3L)                   // List[Double]
List(false, 1, "two")              // List[Any]
if (a % 2 == 0) "e" else "o"       // String
println("hello")                   // Unit
```

イニシャライズ）

```
val x =
  if (...) 7  else 8

val list =
  for (i <- 1 to 10)
  yield someFun(i)
```

ファンクションは「First Class Citizens」

- ファンクションも、パラーメターや返り値として使える
- インラインで定義できる：

```
// define function and assign
// to identifier
val f = (x: Int) => x * x
  // => f has type (Int => Int)
f(4)  // 16

// shorthand form for
// one-parameter functions
val g: Int => Int = _ % 3
g(4)  // 1
```

活用：リストの演算は楽しくなる

```
val list = List(2.3, -7.1, 8.9,
-4.0)

list.sortBy(i => i.abs)
// => List(2.3, -4.0, -7.1, 8.9)

list.map(i => i * i)
// => List(5.29, 50.41, 79.21, 16.0)

list.filter(_ > 0)
// => List(2.3, 8.9)

list.partition(_ > 3)
// => (List(8.9),List(2.3, -7.1, -4.0))
```

ナイス・スタッフ

- XML はネイティブデータ型

```
val username = "a>&b"
val xml = <div><a href="/login">Hello {username}!</a></div>
  // <div><a href="/login">Hello a&gt;&amp;b!</a></div>
```

- パターンマッチング

```
val user: User = // ...
user match {
  case NormalUser(age, name) if age >= 20  =>  println(name + " is adult")
  case NormalUser(age, name)               =>  println(name + " is a child")
  case AdminUser(name)                     =>  println(name + " is an admin")
}
```

- 糖衣構文
  - メソッドの引数は 1 つだけの場合、括弧とドットが要らない
  - 約物も使える

```
val request = :/("localhost", 8080) / "api" <<? Map("key" -> "val")
    <:< Map("Accept" -> "text/xml")
```

JVM 上で実現の特徴

- JVM の制約は Scala のバイトコードにも影響を与える
- 例）Type erasure

```
val aString: Any = "abc"
aString match {
  case i: Int =>    println("int")
  case s: String => println("str")
}
// => str

val aList: List[String] = List("a", "b")
aList match {
  case il: List[Int] =>
    println("list of int")
  case sl: List[String] =>
    println("list of str")
}
// => list of int  TADASHIKUNAI!
```

例）ファンクションはクラスになる

```
class Hello {
    def example = {
        val x = List(1, 2, 3, 4)
        val factor = 5
        // anonymous inner function  (クロージャ！)
        val y = x.map(_ * factor)
    }
}
```

```
$ javap Hello.class
Compiled from "Hello.scala"
  public void example();
  public Hello();
}
```

```
$ javap Hello$$anonfun$1.class
public final class Hello$$anonfun$1 extends
    scala.runtime.AbstractFunction1$mcII$sp
    implements scala.Serializable {
  public final int apply(int);
  public Hello$$anonfun$1(Hello, int);
  ...
}
```


Spark

Apache Spark の件

- Spark は、「fast and general engine for large-scale data processing」である。標語は、「lightning-fast cluster computing」。
- 2009 年から、UC Berkeley の AMPLab（Algorithms, Machines and People Lab）の研究プロジェクト
- 2010 年からオープンソース
- 2013 年 Apache Incubator Project になった
- 2014 年 Apache Incubator を卒業した
- 最近のバージョン：1.0.2（8 月 5 日）

Spark アーキテクチャー

- Scala（または、 Java か Python ）プログラムで、計算の内容を宣言：「 Spark Driver 」
- ドライバーは管理やタスク分配の担当、タスクの実行はクラスター上でされる
- 色々なクラターマネージャを使える
  - Standalone Spark Cluster
  - Mesos
  - YARN

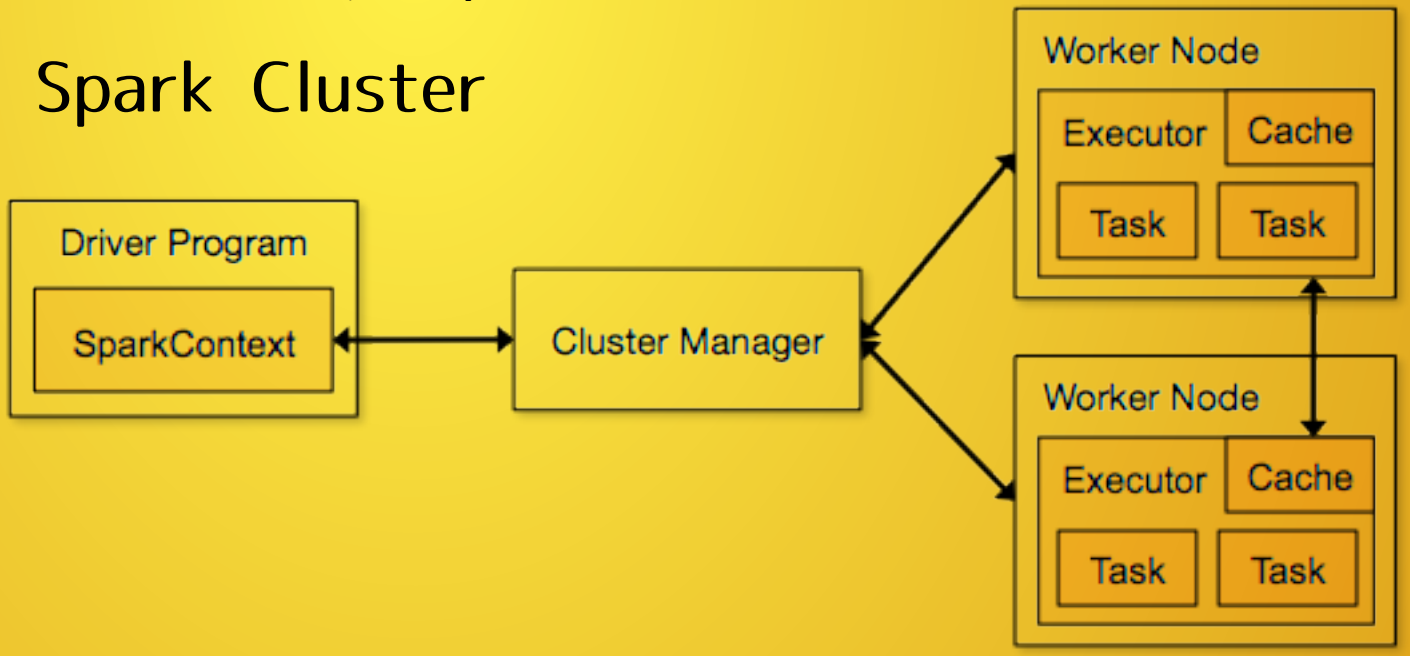


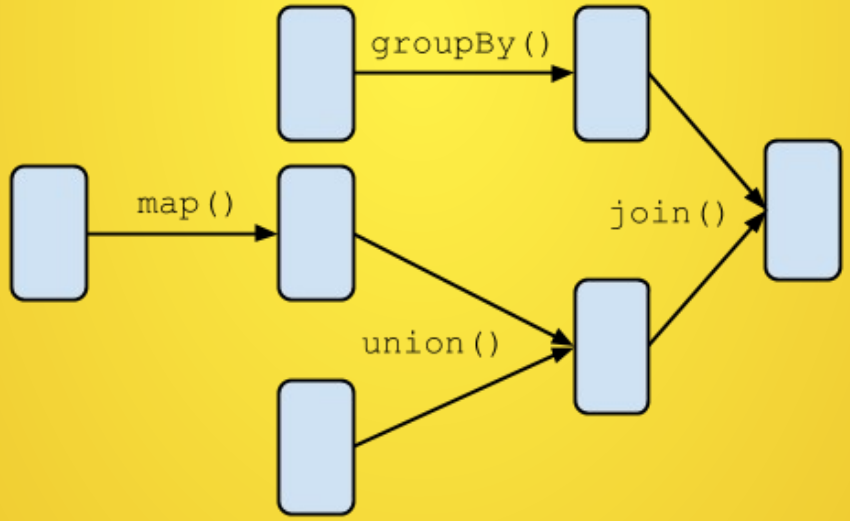
Image Source: https://spark.apache.org/docs/latest/running-on-mesos.html

Spark の概念

- ゴールは、並行で堅固なデータ処理
- 思想：イミュータブルなデータで巾等（アイデムポテント）、副作用ない、再現可能なファンクションを実行
- 結論：
  - 共有の書き込み可能メモリーがない
  - クラスターノードの故障の時、別のノードで再実行できる
- 最も重要なデータ構造は RDD（ Resilient Distributed Dataset ）

RDDの使い方

- Scala開発者にとって、リストみたい： map(), filter(), ...
- リストに対して、RDDの上で `lines.map(someFunction)` を実行時には、`someFunction` は実行されず、先にRDD依存のグラフが作られる：



- 全部の「transformation」の計算は、「output operation」のコールに誘起される

コード

```
object SimpleSpark {
    def main(args: Array[String]) = {
        val sc = new SparkContext("local[2]", "Spark Demo")

        // create RDD from file (input)
        val lines: RDD[String] =
            sc.textFile("...")

        // transform RDDs
        val words: RDD[String] =
            lines.flatMap(_.split("\\W+").
            filterNot(_.isEmpty))
        val wordTuples: RDD[(String, Int)] =
            words.map(word => (word, 1))
        val wordCounts: RDD[(String, Int)] =
            wordTuples.reduceByKey(_ + _)

        // output
        val maxCount =
            wordCounts.max()(Ordering[Int].on(_._2))
        println("most frequent word: " + maxCount)

        sc.stop()
    }
}
```

クラスタータイプ

データソース

トランスフォーメーション

依存の DAG：

```
MapPartitionsRDD[6] at reduceByKey
  ShuffledRDD[5] at reduceByKey
    MapPartitionsRDD[4] at reduceByKey
      MappedRDD[3] at map
        FlatMappedRDD[2] at flatMap
          MappedRDD[1] at textFile
            HadoopRDD[0] at textFile
```

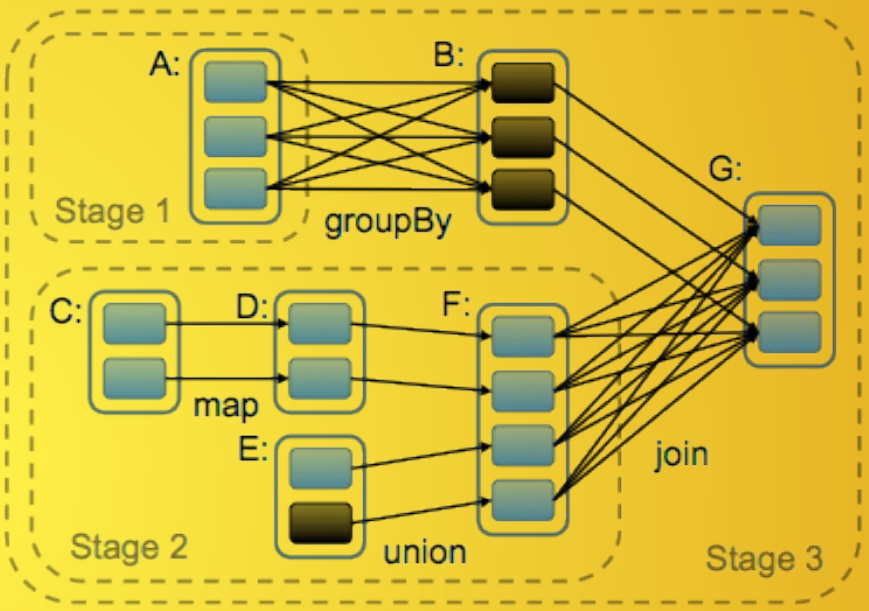
アウトプット・オペレーション

RDDの詳細

- 「Resilient」は、ノード故障の時、いつでも原初のデータで再計算できる。
- 「Distributed」は、クラスターで分散のパーティションの中で保存する。計算はデータの近くで実行される。
- 途中結果はメモリーに保存されるので、反復のアルゴリズムも速い

クラスターの上でタスク実行

- Spark ドライバーは RDD 依存グラフの トランスフォーメーション を「ステージ」と言うこと にグループする

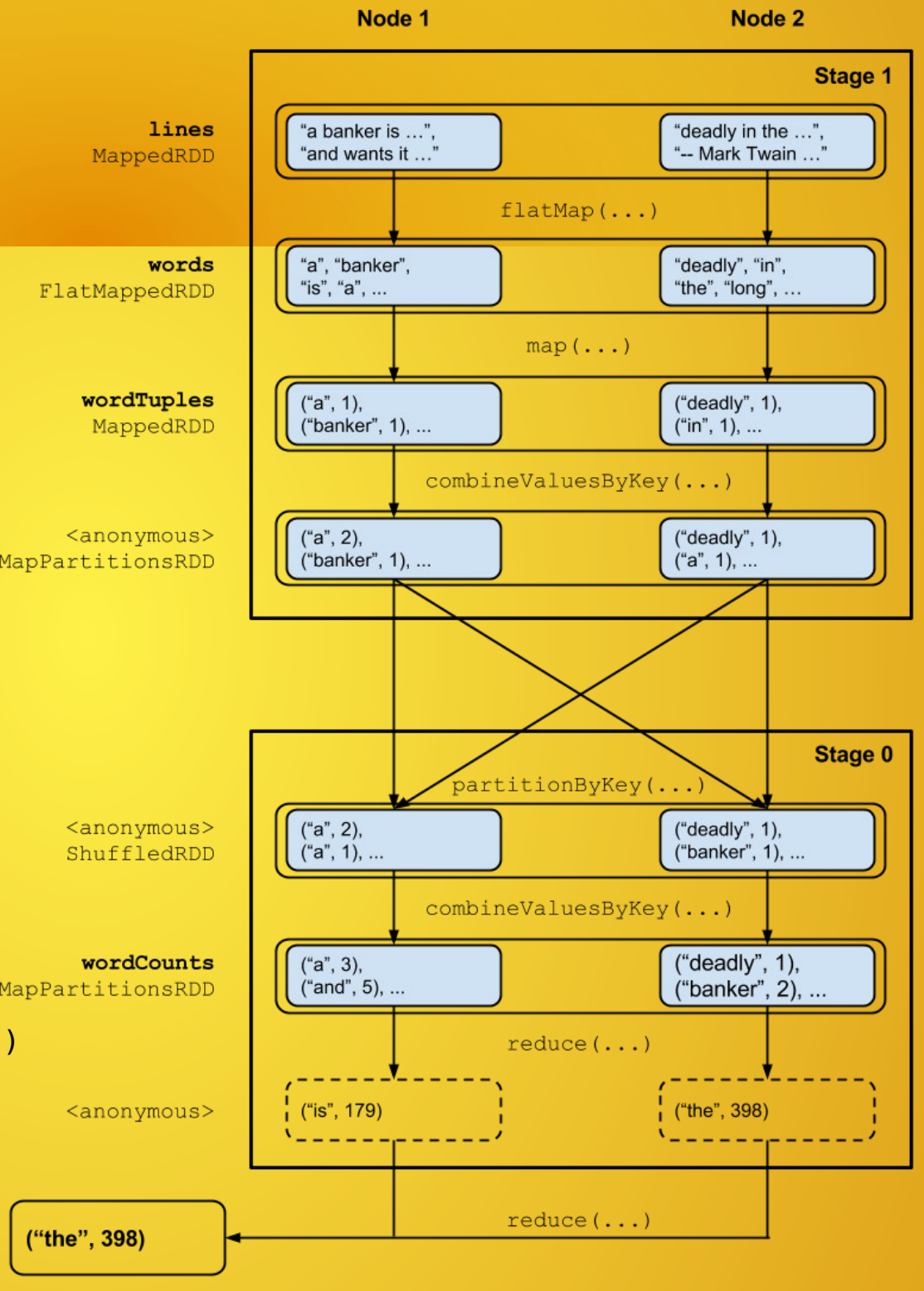


- Spark ドライバーはタスクをクラスターノードに送る。格ノードはローカルのパーティションのデータを処理する。
  - 「タスクをノードに送る」の実現は、 Scala ファンクション から作ったクラスのオブジェクトをシリアライズして送る
- 処理の結果はアウトプットの為にドライバーに返信

Image Source: Resilient Distributed Datasets - A Fault-Tolerant Abstraction for In-Memory Cluster Computing

コード

```
...
val sc = new SparkContext("local[2]",
    "Spark Demo")

// create RDD from file (input)
val lines: RDD[String] =
    sc.textFile("...")

// transform RDDs
val words: RDD[String] =
    lines.flatMap(_.split("\\W+").
    filterNot(_.isEmpty))
val wordTuples: RDD[(String, Int)] =
    words.map(word => (word, 1))
val wordCounts: RDD[(String, Int)] =
    wordTuples.reduceByKey(_ + _)

// output
val maxCount =
    wordCounts.max()(Ordering[Int].on(_._2))
println("most frequent word: " + maxCount)

sc.stop()
...
```



Spark Streaming

Spark Streaming の概念

- Spark Streaming は、 Spark のストリーム処理ライブラリである
- 前提：「 one-record-at-a-time 」処理のシステムでは、クラスターノードの故障も整合性は難しい
- 結論：固定間隔（例えば：５秒）の間データ集めて、後でバッチ処理。「 real-time 」ではない。

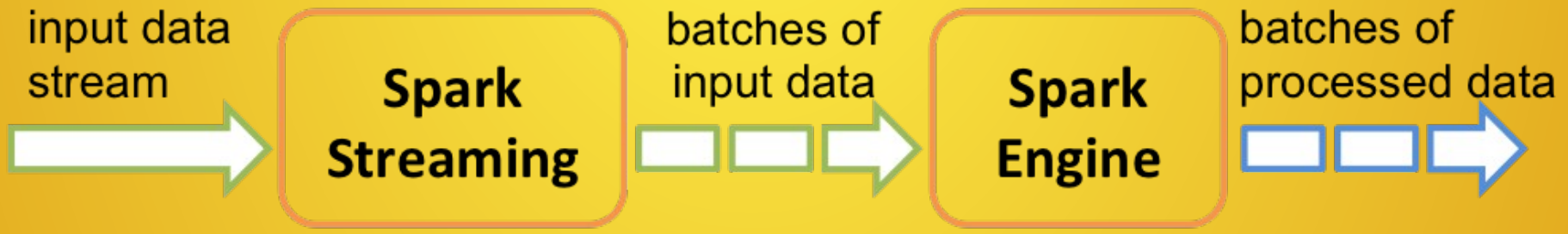
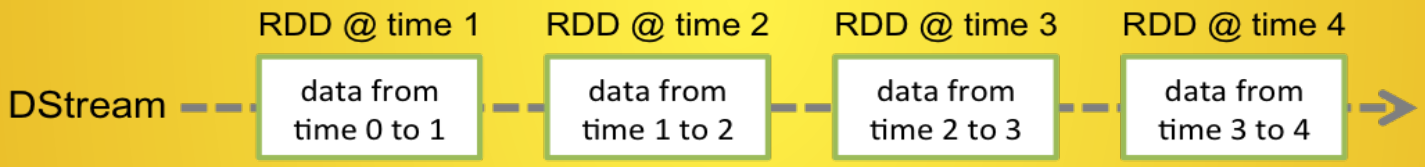


Image Source: https://spark.apache.org/docs/1.0.2/streaming-programming-guide.html

Spark Streaming の詳細

- Spark Streaming の最も重要なデータ構造は DStream（Discretized Stream）である
- DStream は RDD のシークエンスである。間隔の１つのデータは RDD の１つになる。



- インターフェースや使い方は似ている：

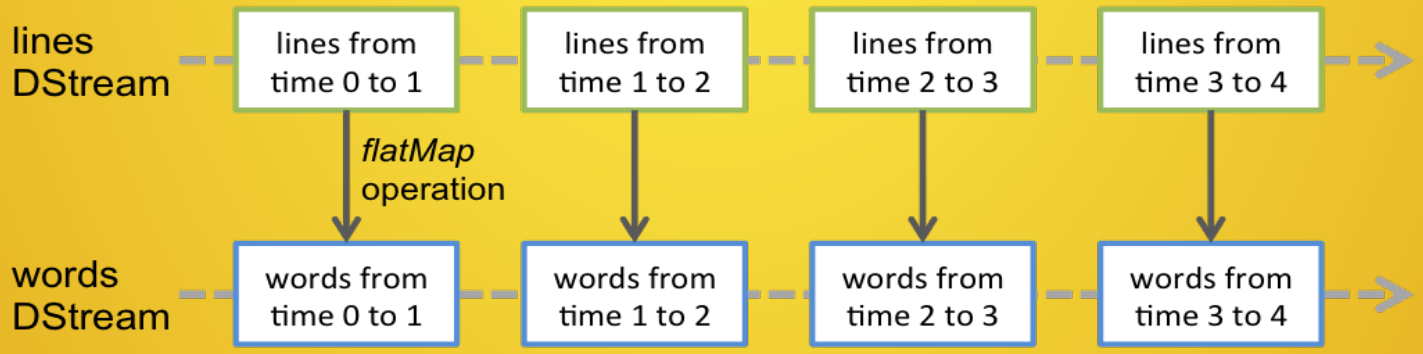


Image Source: https://spark.apache.org/docs/1.0.2/streaming-programming-guide.html

Spark Streaming の使い方

例）

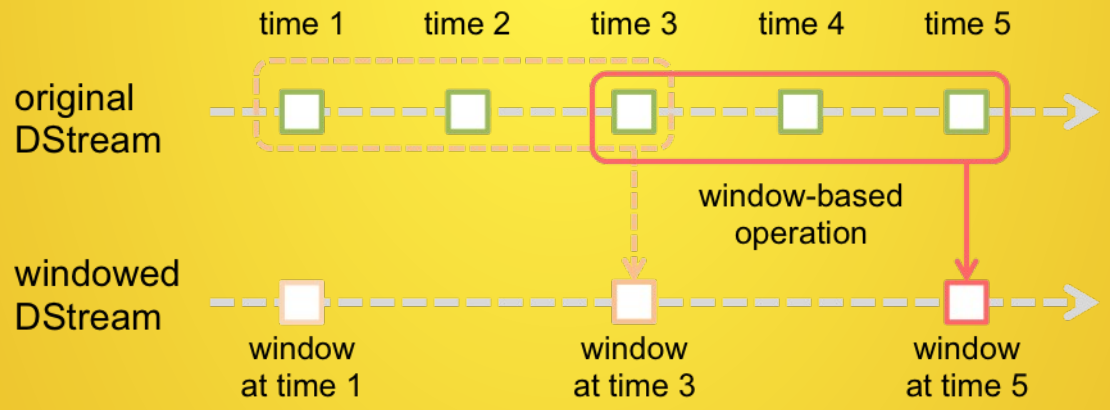
```
object SimpleSparkStream {

    def main(args: Array[String]) = {
        val sc = new SparkContext("local[3]", "Spark")
        val ssc = new StreamingContext(sc, Seconds(5))

        // create DStream from stream (input)
        val lines: DStream[String] =
            ssc.socketTextStream("localhost", 4444)

        // transform RDDs
        val words: DStream[String] =
            lines.flatMap(_.split("\\W+").
            filterNot(_.isEmpty))
        val wordTuples: DStream[(String, Int)] =
            words.map(word => (word, 1))
        val wordCounts: DStream[(String, Int)] =
            wordTuples.reduceByKey(_ + _)

        // output
        val maxCount =
            wordCounts.reduce((a, b) =>
              if (a._2 >= b._2) a else b)
        // this will print something every 5 seconds
        maxCount.print()

        ssc.start()
    }
}
```

- 色々な入力可能性ある：
  - TCP sockets
  - HDFS ファイル
  - Apache Kafka
  - Twitter
  - ・・・

時間帯オペレーション

- ストリームの「sliding window」も使える：
  「２秒間置き、作３秒間のデータで、・・・」

```
val wordsWindow: DStream[String] =
  words.window(windowDuration=Seconds(3),
               slideDuration=Seconds(2))
```



- トレンド認識に便利

Image Source: https://spark.apache.org/docs/1.0.2/streaming-programming-guide.html

他の Spark のモジュール

Spark SQL

- SQL ストリングから RDD オペレーションに変換
- 自動的でクエリの実行プラン作る
- アイディア：「プログラマではなくても、データ分析できる」
- 問題： Scala の型安全性は無くなる

```
// Spark SQL requires RDD of some "case class"
case class WordCount(word: String, num: Int)

object SimpleSparkSQL extends Logging {
    def main(args: Array[String]) = {
        // (init Spark)
        // transform RDDs
        val words: RDD[String] = ...
        val wordTuples: RDD[(String, Int)] = ...
        val wordCounts: RDD[(String, Int)] = ...

        // wrap in class WordCount for column names
        val wordCountObjs: RDD[WordCount] =
            wordCounts.map(wc => WordCount(wc._1, wc._2))
        // register with a certain table name
        wordCountObjs.registerAsTable("wordcounts")
        // compute RDD from SQL statement
        val sqlResult = sqlc.sql("SELECT * FROM wordcounts
            WHERE num > 200 ORDER BY num DESC")

        // output
        sqlResult.foreach(println)
    }
}

---------------------------------

== Query Plan ==
Sort [num#1:1 DESC], true
 Exchange (RangePartitioning [num#1 DESC], 200)
  Filter (num#1:1 > 200)
   ExistingRdd [word#0,num#1], MapPartitionsRDD[8] at ...
```

他のモジュール

- MLlib
  - 機械学習のライブラリー
  - 次のバージョンでは、オンライン学習ある
- Bagel
  - グラフ処理のライブラリー（Pregel on Spark）
- GraphX
  - グラフ処理のライブラリー
  - Bagel の後継機

概括

- Spark では、普通の Scala プログラムみたいでクラスター上で実行されるデータ処理プログラム作れる。
- 「関数型プログラミング使うと並列化は簡単」のキャッチフレーズの実現
- イミュータブルなデータのおかげで故障の回復できる
- 同じコンセプトでストリーム処理できる
- 色々なモジュールは特別なユースケースに役に立つ

ありがとうございます！