

# Outline

Module 1 : 大數據簡介

Module 2 : Hadoop Ecosystem介紹

Module 3 : Hadoop 平台安裝

Module 4 : Hadoop 分散式檔案系統 (HDFS)

Module 5 : Hadoop MapReduce

Module 6 : Apache Hive

Module 7 : Sqoop與Flume

**Module 8 : Apache Spark**

Module 9 : Spark 平台安裝

Module 10 : RDD — Resilient distributed dataset

Module 11 : Scala 程式開發基礎

Module 12 : Spark SQL 及 DataFrame

Module 13 : Spark 機器學習函式庫(MLlib)



# Spark介紹

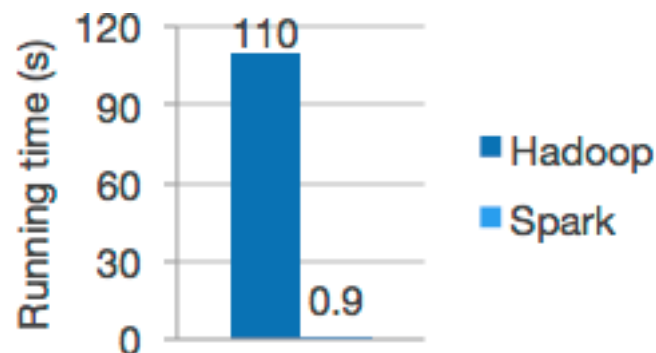
- ▶ 是開放原始碼的**叢集運算**框架，由加州大學柏克萊分校的AMPLab開發、現由Apache基金會維護
- ▶ 以**記憶體**存放**運算資料**，取得更佳運算效能
- ▶ 提供**簡單易用的程式介面**，讓開發人員輕易撰寫**平行處理**應用
- ▶ **便宜且容易擴充**的大數據運算平台
- ▶ 內建**機器學習**、**圖型計算**及**串流處理**模組



# Spark的優勢

## ► Speed

- 以Memory為儲存媒體，Spark的MapReduce比Hadoop快100x倍
- 以Disk為儲存媒體，亦可快10倍以上



Logistic regression in Hadoop and Spark

# Spark的優勢

## ► Ease of Use

- 內建支援Java、Scala、Python、R程式語言，以及超過80組以上的高階操作，開發人員可輕易實作平行處理的應用
- 提供RDD、SQL及DataFrame等介面，使資料操作更為簡單靈活
- 內建shell為極佳之高互動介面

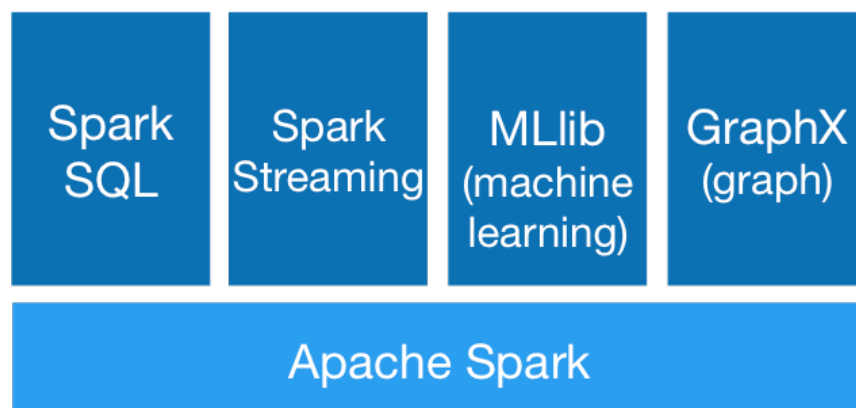
```
val text_file = spark.textFile("hdfs://...")  
  
text_file.flatMap(line => line.split())  
           .map(word => (word, 1))  
           .reduceByKey(case(a, b)=> a+b)
```

Word count in Spark's Scala API

# Spark的優勢

## ► Generality

- 內建機器學習、圖型計算及串流處理等高階模組，彼此之間可相互整合，無縫接軌
- One Stack to rule them all
  - 以RDD結構一之貫之



# Spark的優勢

## ▶ Runs Everywhere

- Spark可獨立執行(**Standalone**)、亦可跑在**Hadoop YARN**、**Mesos**等現有平台
- 可存取不同資料來源如**HDFS**、**Cassandra**、**HBase**及**S3**等



# 世界記錄

## Apache Spark officially sets a new record in large-scale sorting



by Reynold Xin

Posted in **ENGINEERING BLOG** | November 5, 2014

A month ago, we shared with you our entry to the 2014 Gray Sort competition, a 3rd-party benchmark measuring how fast a system can sort 100 TB of data (1 trillion records). Today, we are happy to announce that our entry has been reviewed by the benchmark committee and we have officially won the [Daytona GraySort contest](#)!

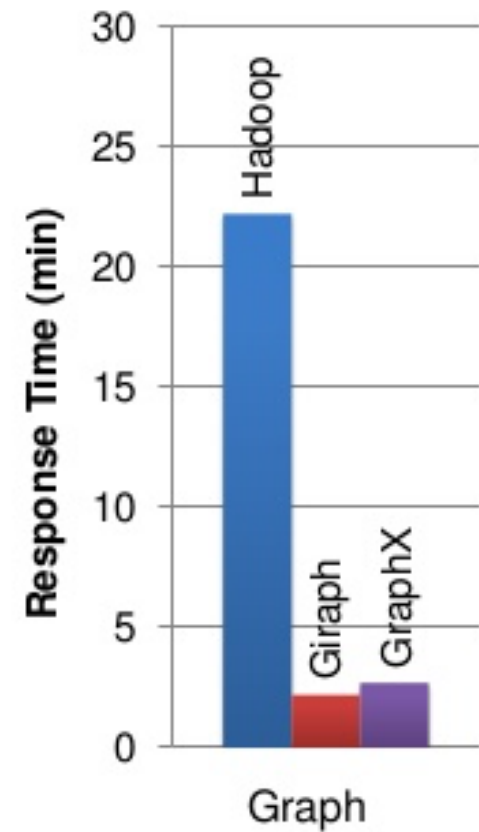
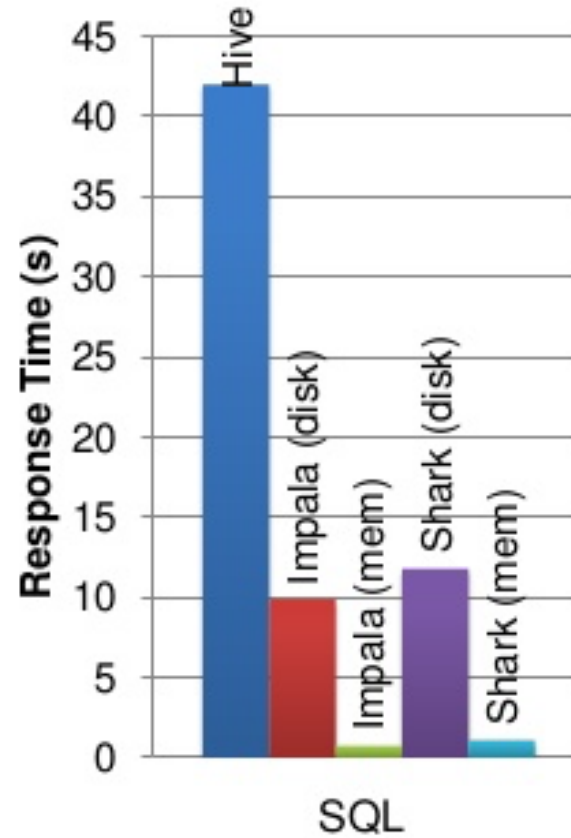
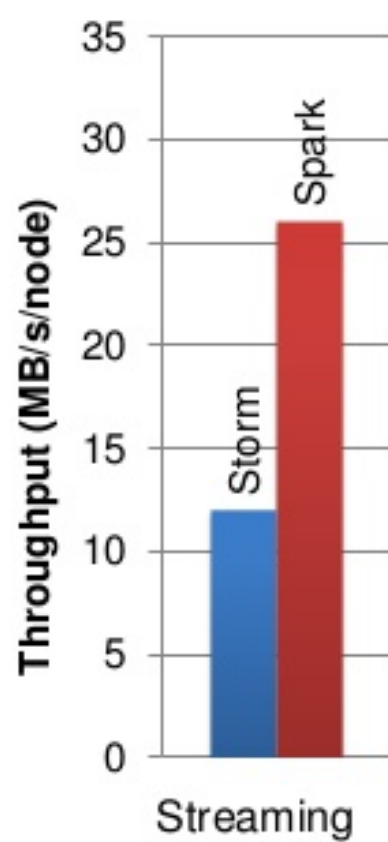
In case you missed our [earlier blog post](#), using Spark on 206 EC2 machines, we sorted 100 TB of data on disk in 23 minutes. In comparison, the previous world record set by Hadoop MapReduce used 2100 machines and took 72 minutes. This means that Apache Spark sorted the same data **3X faster** using **10X fewer machines**. All the sorting took place on disk (HDFS), without using Spark's high performance system memory.

## \$1.44 per terabyte: setting a new world record with Apache Spark

November 14, 2016 | by Reynold Xin in **ENGINEERING BLOG**

We are excited to share with you that a joint effort by Nanjing University, Alibaba Group, and Databricks set a new world record in CloudSort, a well-established third-party benchmark. We together architected the most efficient way to sort 100 TB of data, using only \$144.22 USD worth of cloud resources, 3X more cost-efficient than the previous...

## Spark Performance



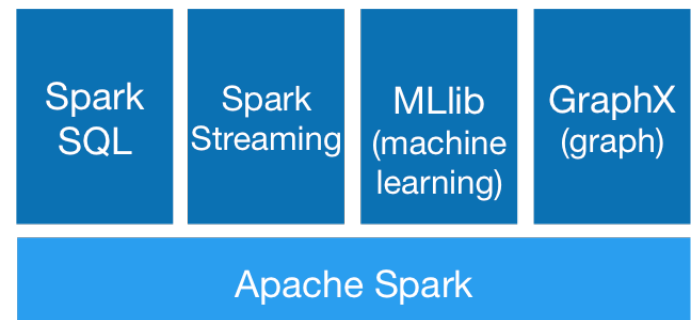


# But ...

- ▶ Spark的出現不是要消滅Hadoop Ecosystem
  - 僅針對平行運算架構作強化
  - Spark + Hadoop仍為最常見之組合
    - Spark建置於Hadoop叢集上，以YARN作為Manager
    - Spark負責平行處理及運算
    - 運算來源及結果資料存放於HDFS

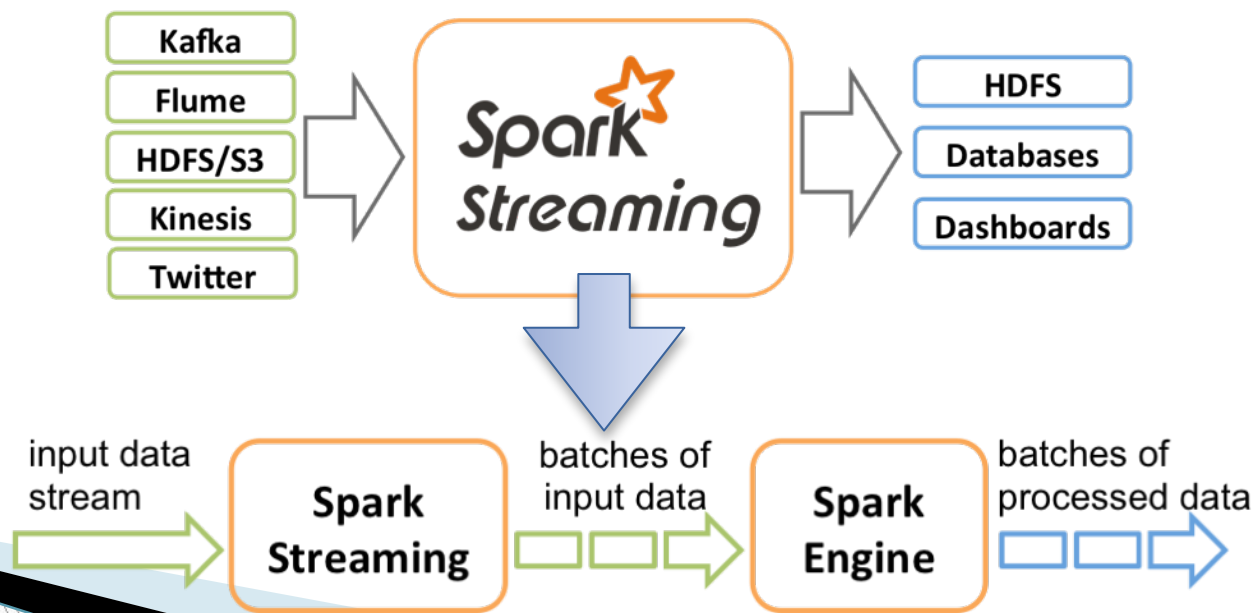
# Spark Library Stacks

- ▶ **Spark SQL**
  - module for working with structured data
- ▶ **Spark Streaming**
  - makes it easy to build scalable fault-tolerant streaming applications
- ▶ **MLlib(machine learning)**
  - Spark's scalable machine learning library
- ▶ **GraphX(graph)**
  - Spark's API for graphs and graph-parallel computation

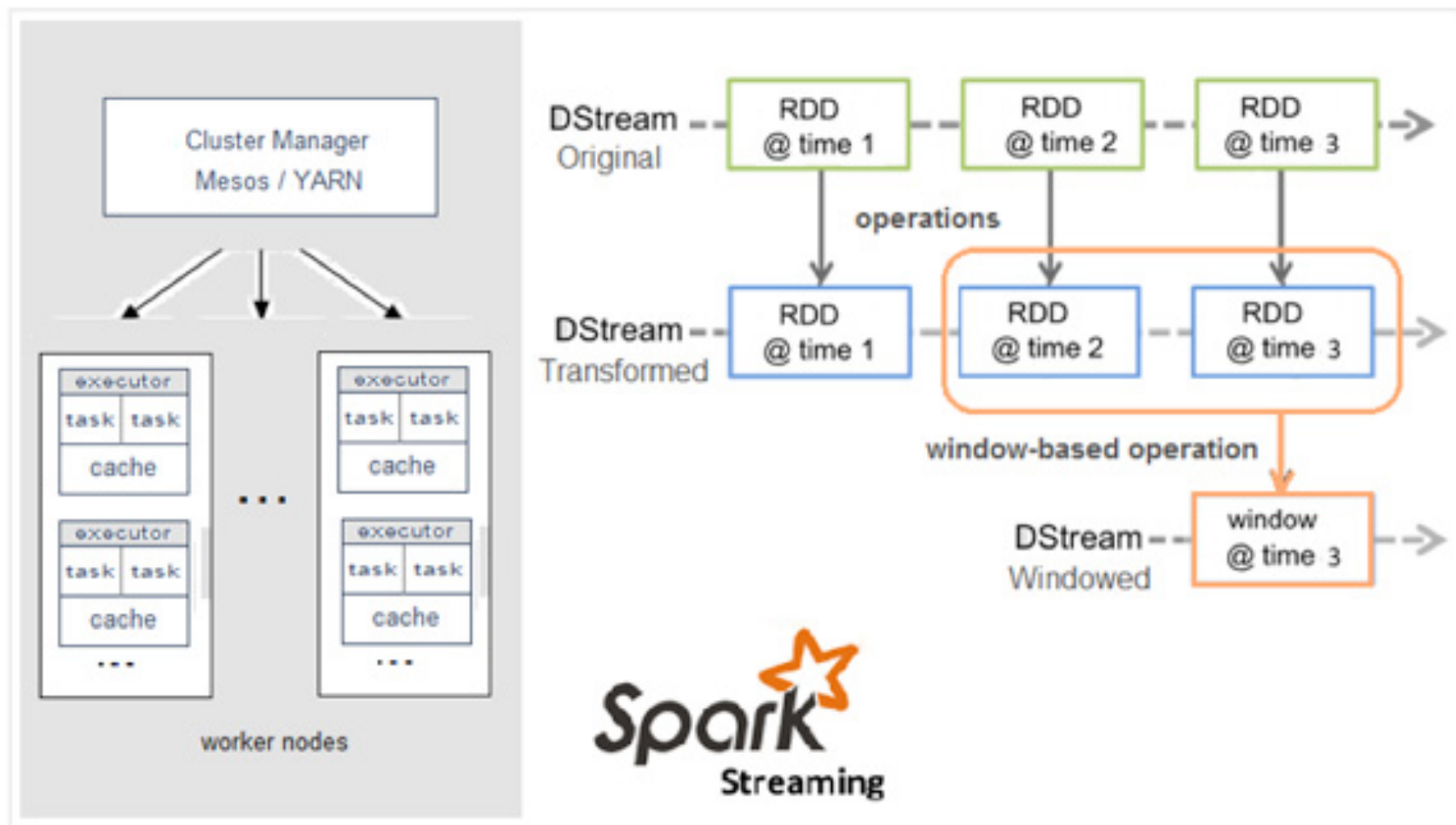


# Spark Streaming

- ▶ 用於處理即時化需求，如即時推薦、即時效能分析等
- ▶ 將串流依時間間隔切分為數個資料片段(DStream)作處理
- ▶ 以DStream (DiscretizedStream) 結構作為批次處理單位(準即時處理)
- ▶ 能與SparkSQL、MLlib及GraphX無縫整合



# DStream的運作



# Streaming Word Count

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3

def main(args: Array[String]) {
  // Create a local StreamingContext with two working thread and batch interval of 1 second.
  // The master requires 2 cores to prevent from a starvation scenario.

  val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
  val ssc = new StreamingContext(conf, Seconds(1))

  // Create a DStream that will connect to hostname:port, like localhost:9999
  val lines = ssc.socketTextStream("localhost", 9999)

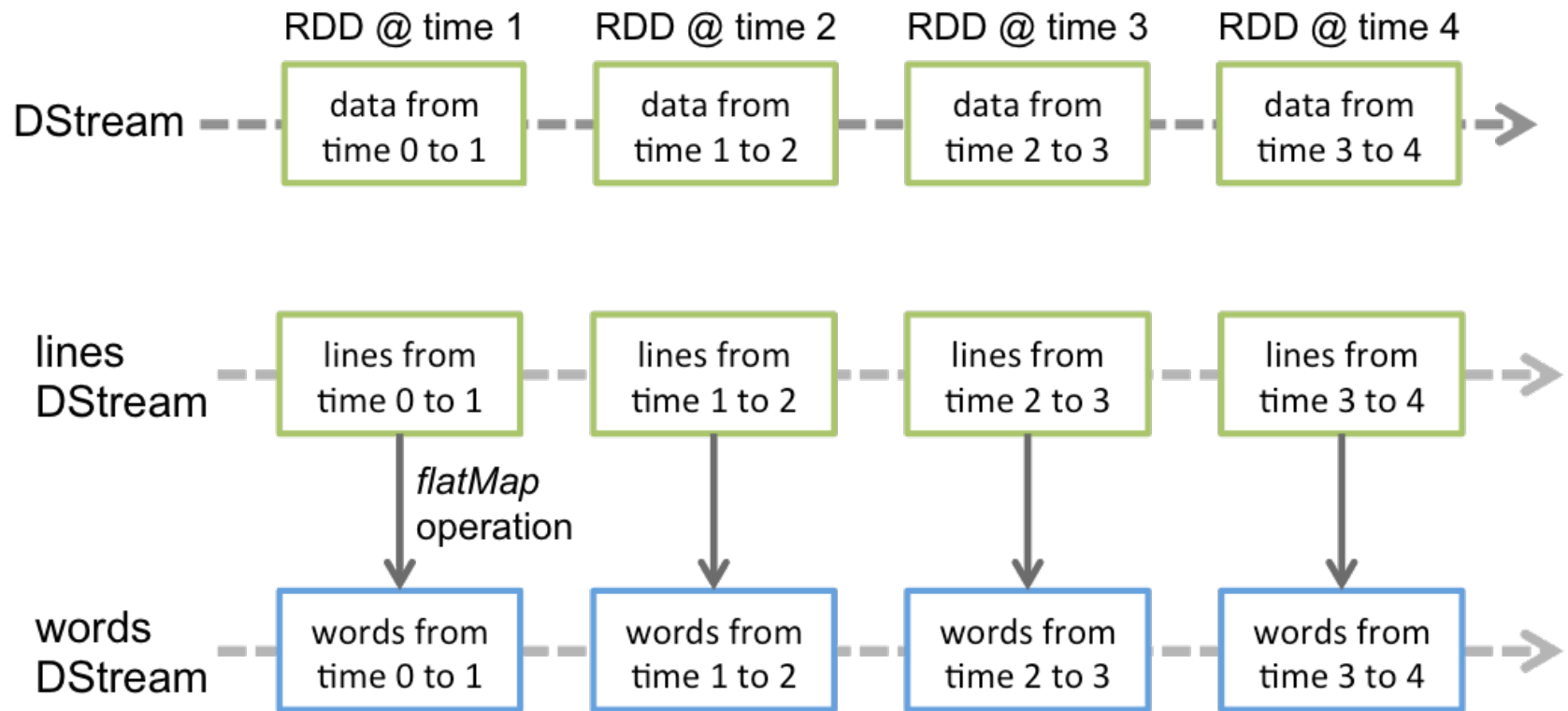
  // Split each line into words
  val words = lines.flatMap(_.split(" "))

  // Count each word in each batch
  val pairs = words.map(word => (word, 1))
  val wordCounts = pairs.reduceByKey(_ + _)

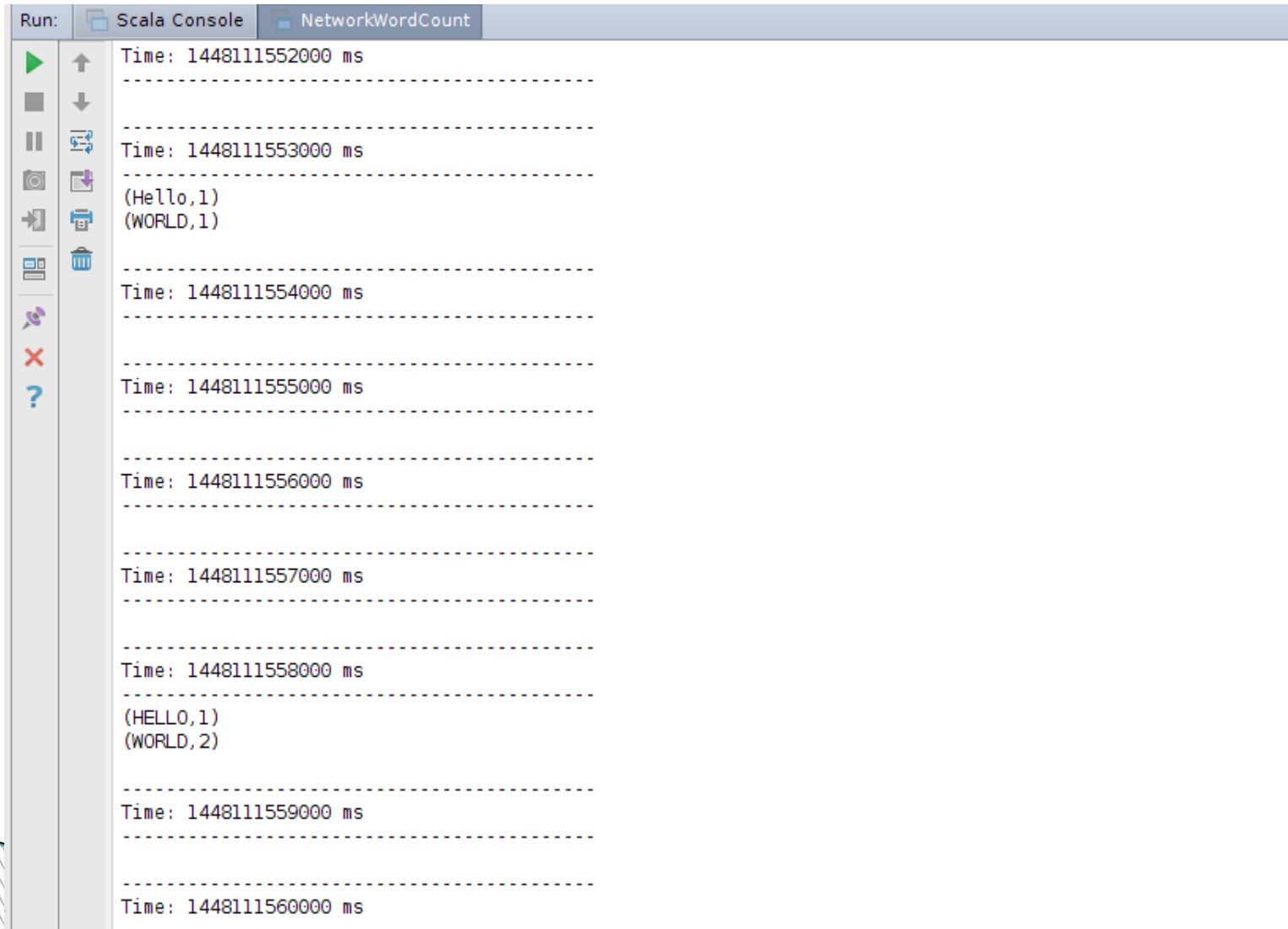
  // Print the first ten elements of each RDD generated in this DStream to the console
  wordCounts.print()

  ssc.start()           // Start the computation
  ssc.awaitTermination() // Wait for the computation to terminate
}
```

# Streaming Word Count運作示意



# Streaming Word Count結果示意

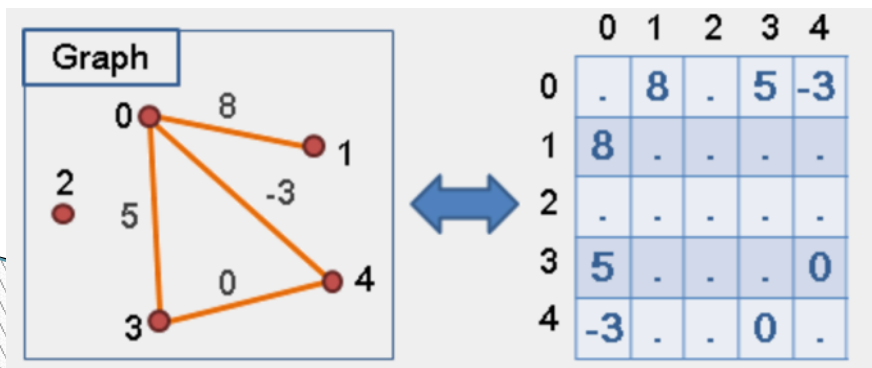


The screenshot shows a Scala IDE console window with two tabs: "Scala Console" and "NetworkWordCount". The "NetworkWordCount" tab is active, displaying a stream of word count results. The results are formatted as "Time: [timestamp] ms" followed by a dashed line and then the word count pairs. The word count pairs are shown in a sequence that demonstrates the streaming nature of the application, with words being counted as they arrive in the stream.

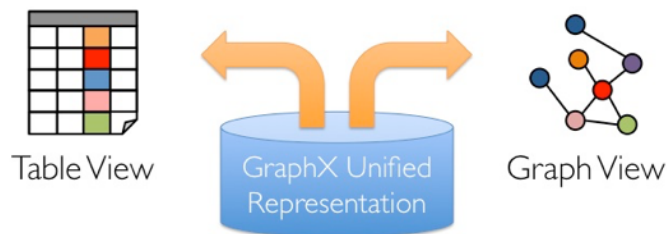
```
Run: Scala Console NetworkWordCount
Time: 1448111552000 ms
-----
Time: 1448111553000 ms
-----
(Hello,1)
(WORLD,1)
-----
Time: 1448111554000 ms
-----
Time: 1448111555000 ms
-----
Time: 1448111556000 ms
-----
Time: 1448111557000 ms
-----
Time: 1448111558000 ms
-----
(HELLO,1)
(WORLD,2)
-----
Time: 1448111559000 ms
-----
Time: 1448111560000 ms
-----
```

# GraphX

- ▶ Spark內建處理graph computation的功能模組
- ▶ 特化的結構支援Table及Graph間的轉換
- ▶ 可對Graph進行運算、查詢、篩選、排序、pageRank等操作
- ▶ 可與SparkSQL、Streaming及MLlib搭配運用



Tables and Graphs are *composable views* of the same *physical data*



Each view has its own *operators* that *exploit the semantics* of the view to achieve *efficient execution*

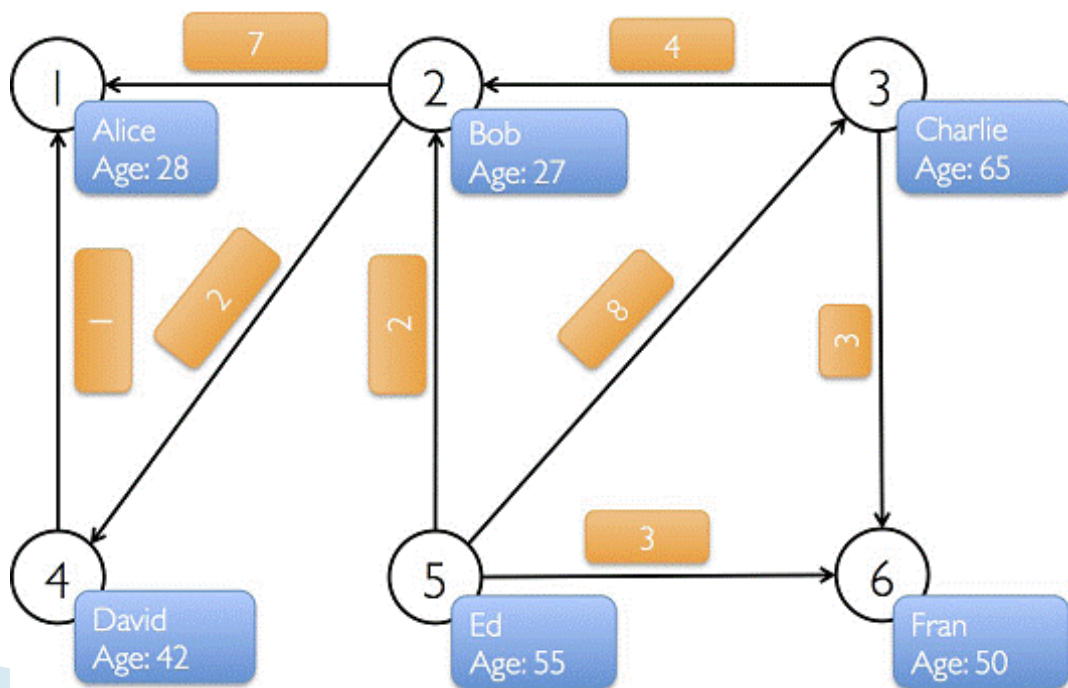


# GraphX的應用情境

- ▶ 在社群網站中推薦新朋友、評估社群及使用者影響力等
- ▶ 搜尋結果的排序及推薦(PageRank)

ex：Charile與Fran、Bob為好友

- ▶ 系統是否要建議Charile加Ed為好友？
- ▶ 系統是否要建議Charile加David為好友？



# Graph的建立

```
//设置顶点和边，注意顶点和边都是用元组定义的Array
```

```
//顶点的数据类型是VD:(String,Int)
```

```
val vertexArray = Array(  
  (1L, ("Alice", 28)), (2L, ("Bob", 27)), (3L, ("Charlie", 65)),  
  (4L, ("David", 42)), (5L, ("Ed", 55)), (6L, ("Fran", 50))  
)
```

```
//边的数据类型ED:Int
```

```
val edgeArray = Array(  
  Edge(2L, 1L, 7), Edge(2L, 4L, 2), Edge(3L, 2L, 4), Edge(3L, 6L, 3),  
  Edge(4L, 1L, 1), Edge(5L, 2L, 2), Edge(5L, 3L, 8), Edge(5L, 6L, 3)  
)
```

```
//构造vertexRDD和edgeRDD
```

```
val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)  
val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
```

```
//构造图Graph[VD,ED]
```

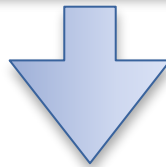
```
val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)
```

# Graph的操作：Filter

```
println("找出图中年龄大于30的顶点：") //filter操作
graph.vertices.filter { case (id, (name, age)) => age > 30 }.collect.foreach {
  case (id, (name, age)) => println(s"$name is $age")
}
```

//边操作：找出图中属性大于5的边

```
println("找出图中属性大于5的边：")
graph.edges.filter(e => e.attr > 5).collect.foreach(e => println(s"${e.srcId} to $
{e.dstId} att ${e.attr}"))
```



找出图中年龄大于30的顶点：

David is 42

Fran is 50

Charlie is 65

Ed is 55

找出图中属性大于5的边：

2 to 1 att 7

5 to 3 att 8

# Graph的操作：Join

```
println("找出年纪最大的追求者：")
val oldestFollower: VertexRDD[(String, Int)] = userGraph.mapReduceTriplets[(String,
Int)](
  // 将源顶点的属性发送给目标顶点，map过程
  edge => Iterator((edge.dstId, (edge.srcAttr.name, edge.srcAttr.age))),
  // 得到最大追求者，reduce过程
  (a, b) => if (a._2 > b._2) a else b
)

userGraph.vertices.leftJoin(oldestFollower) { (id, user, optOldestFollower) =>
  optOldestFollower match {
    case None => s"${user.name} does not have any followers."
    case Some((name, age)) => s"${name} is the oldest follower of ${user.name}."
  }
}.collect.foreach { case (id, str) => println(str)}
```



找出年纪最大的追求者：

- Bob is the oldest follower of David.
- David is the oldest follower of Alice.
- Charlie is the oldest follower of Fran.
- Ed is the oldest follower of Charlie.
- Ed does not have any followers.
- Charlie is the oldest follower of Bob.

# PageRank

- ▶ 網頁排名，又稱網頁級別、Google左側排名或佩奇排名，是一種由搜尋引擎根據網頁之間相互的超連結計算的技術
- ▶ 在搜尋引擎最佳化操作中是經常被用來評估網頁最佳化的成效因素之一
- ▶ 把從A頁面到B頁面的連結解釋為A頁面給B頁面投票，根據投票來源（即連結到A頁面的頁面）和投票目標的等級來決定目標頁面的等級



# PageRank範例

## ► Berkeley關鍵字搜尋的PageRank排名

page-title資訊

```
138 3932908833397101503 St. Michael's College School
139 2708418598117237725 Metropolitan Junior Hockey League
140 5090956282167233219 List of mountain ranges of California
141 4102223989096646779 Java (programming language)
```

page-page連結資訊

```
98 1746517089350976281 443952852637640503
99 1746517089350976281 497048819723143676
100 1746517089350976281 533544275833168761
```



### PageRank計算最佳的搜尋結果排名

\*\*\*\*\*

University of California, Berkeley: 1321.111754312097

Berkeley, California: 664.8841977233583

Uc berkeley: 162.50132743397873

Berkeley Software Distribution: 90.4786038848606

Lawrence Berkeley National Laboratory: 81.90404939641944

Link Table

Title	Link



Hyperlinks



PageRank



Title	PR



//讀取Vertices及Edges資訊

```
val articles: RDD[String] = sc.textFile("page-title-vertices.txt")
val links: RDD[String] = sc.textFile("page-page-edges.txt")
```

//建立RDD

```
val vertices = articles.map { line =>
    val fields = line.split('\t')
    (fields(0).toLong, fields(1))
}
```

```
val edges = links.map { line =>
    val fields = line.split('\t')
    Edge(fields(0).toLong, fields(1).toLong, 0)
}
```

//graph操作

```
val graph = Graph(vertices, edges, "").persist()
```

//pageRank演算法呼叫

```
println("*****")
println("PageRank計算最佳的搜尋結果排名")
println("*****")
val prGraph = graph.pageRank(0.001).cache()
val titleAndPrGraph = graph.outerJoinVertices(prGraph.vertices) {
    (v, title, rank) => (rank.getOrElse(0.0), title)
}
```

//印出排名前10筆結果

```
titleAndPrGraph.vertices.top(10) {
    Ordering.by((entry: (VertexId, (Double, String))) => entry._2._1)
}.foreach(t => println(t._2._2 + ": " + t._2._1))
```