The Spark Big Data Analytics Platform

Amir H. Payberah
Swedish Institute of Computer Science

amir@sics.se
June 17, 2014

"THAT'S your Ark for the Big Data flood? Noah, you will need a lot more storage space!"

- Big Data refers to datasets and flows large enough that has outpaced our capability to store, process, analyze, and understand.

Where Does Big Data Come From?

Big Data Market Driving Factors

The number of web pages indexed by Google, which were around one million in 1998, have exceeded one trillion in 2008, and its expansion is accelerated by appearance of the social networks.∗

∗ "Mining big data: current status, and forecast to the future" [Wei Fan et al., 2013]

Big Data Market Driving Factors

The amount of mobile data traffic is expected to grow to 10.8 Exabyte per month by 2016.∗

∗"Worldwide Big Data Technology and Services 2012-2015 Forecast" [Dan Vesset et al., 2013]

Big Data Market Driving Factors

More than 65 billion devices were connected to the Internet by 2010, and this number will go up to 230 billion by 2020.∗

∗ "The Internet of Things Is Coming" [John Mahoney et al., 2013]

Big Data Market Driving Factors

Many companies are moving towards using Cloud services to access Big Data analytical tools.

Big Data Market Driving Factors

Open source communities

How To Process Big Data?

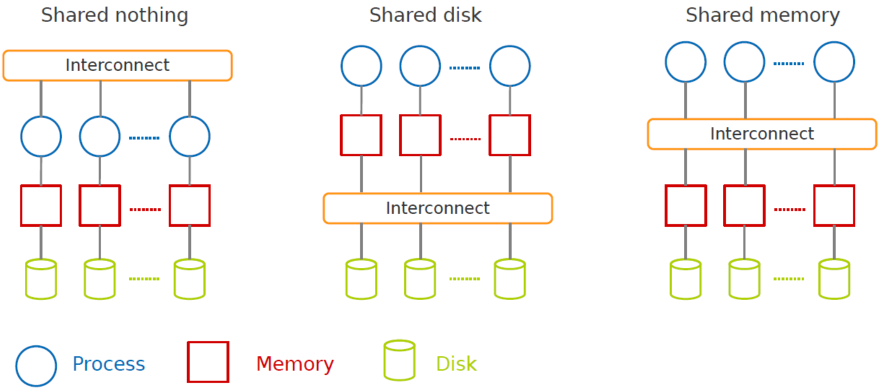
Scale Up vs. Scale Out (1/2)

- Scale up or scale vertically: adding resources to a single node in a system.
- Scale out or scale horizontally: adding more nodes to a system.


Scale Up vs. Scale Out (2/2)

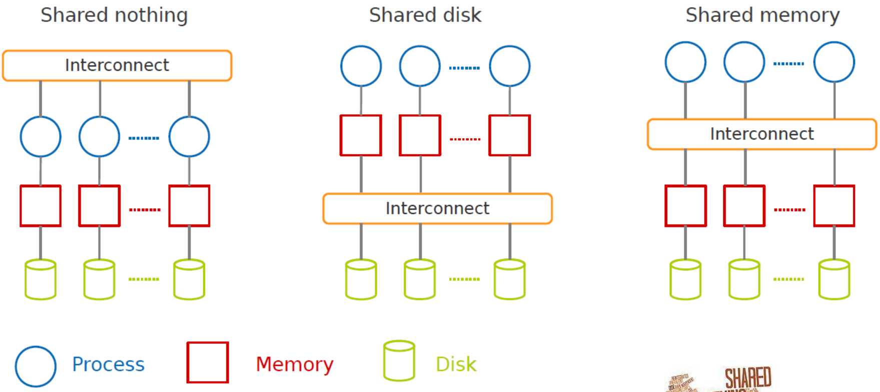
- Scale up: more expensive than scaling out.
- Scale out: more challenging for fault tolerance and software development.

Taxonomy of Parallel Architectures



DeWitt, D. and Gray, J. "Parallel database systems: the future of high performance database systems". ACM Communications, 35(6), 85-98, 1992.

Taxonomy of Parallel Architectures



DeWitt, D. and Gray, J. "Parallel database systems: the future of high performance database systems". ACM Communications, 35(6), 85-98, 1992.

APACHE HBASE

mahout

GraphX

hadoop

StratoSphere
Above the Clouds

Storm

GraphLab

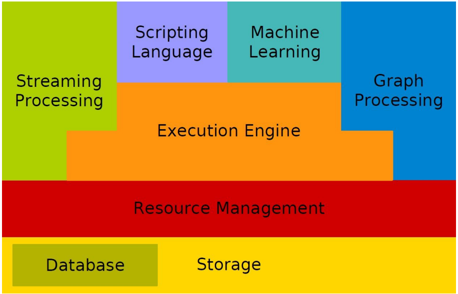
S4 distributed stream computing platform

HIVE

Spark

cassandra

Big Data Analytics Stack



Outline

- Introduction to Scala
- Data exploration using Spark
- Stream processing with Spark Streaming
- Graph analytics with GraphX

Scala

- Scala: scalable language
- A blend of object-oriented and functional programming
- Runs on the Java Virtual Machine
- Designed by Martin Odersky at EPFL

Functional Programming Languages

- In a restricted sense: a language that does not have mutable variables, assignments, or imperative control structures.

- In a wider sense: it enables the construction of programs that focus on functions.

Functional Programming Languages

- In a restricted sense: a language that does not have mutable variables, assignments, or imperative control structures.

- In a wider sense: it enables the construction of programs that focus on functions.

- Functions are first-class citizens:
  - Defined anywhere (including inside other functions).
  - Passed as parameters to functions and returned as results.
  - Operators to compose functions.

Scala Variables

- Values: immutable
- Variables: mutable

```scala
var myVar: Int = 0
val myVal: Int = 1
```

- Scala data types:
  - Boolean, Byte, Short, Char, Int, Long, Float, Double, String

If ... Else

```
var x = 30;

if (x == 10) {
  println("Value of X is 10");
} else if (x == 20) {
  println("Value of X is 20");
} else {
  println("This is else statement");
}
```

Loop

```
var a = 0
var b = 0
for (a <- 1 to 3; b <- 1 until 3) {
  println("Value of a: " + a + ", b: " + b )
}
```

```
// loop with collections
val numList = List(1, 2, 3, 4, 5, 6)
for (a <- numList) {
  println("Value of a: " + a)
}
```

Functions

```
def functionName([list of parameters]): [return type] = {
  function body
  return [expr]
}

def addInt(a: Int, b: Int): Int = {
  var sum: Int = 0
  sum = a + b
  sum
}

println("Returned Value: " + addInt(5, 7))
```

Anonymous Functions

- Lightweight syntax for defining functions.

```
var mul = (x: Int, y: Int) => x * y
println(mul(3, 4))
```

Higher-Order Functions


```scala
def apply(f: Int => String, v: Int) = f(v)

def layout(x: Int) = "[" + x.toString() + "]"

println(apply(layout, 10))
```

Collections (1/2)

- Array: fixed-size sequential collection of elements of the same type

```
val t = Array("zero", "one", "two")
val b = t(0) // b = zero
```

Collections (1/2)

- **Array**: fixed-size sequential collection of elements of the same type

```scala
val t = Array("zero", "one", "two")
val b = t(0) // b = zero
```

- **List**: sequential collection of elements of the same type

```scala
val t = List("zero", "one", "two")
val b = t(0) // b = zero
```

Collections (1/2)

- **Array**: fixed-size sequential collection of elements of the same type

```scala
val t = Array("zero", "one", "two")
val b = t(0) // b = zero
```

- **List**: sequential collection of elements of the same type

```scala
val t = List("zero", "one", "two")
val b = t(0) // b = zero
```

- **Set**: sequential collection of elements of the same type without duplicates

```scala
val t = Set("zero", "one", "two")
val t.contains("zero")
```

Collections (2/2)

- Map: collection of key/value pairs

```
val m = Map(1 -> "sics", 2 -> "kth")
val b = m(1) // b = sics
```

Collections (2/2)

- **Map**: collection of key/value pairs

```
val m = Map(1 -> "sics", 2 -> "kth")
val b = m(1) // b = sics
```

- **Tuple**: A fixed number of items of different types together

```
val t = (1, "hello")
val b = t._1 // b = 1
val c = t._2 // c = hello
```

Functional Combinators

- **map**: applies a function over each element in the list

```
val numbers = List(1, 2, 3, 4)
numbers.map(i => i * 2) // List(2, 4, 6, 8)
```

- **flatten**: it collapses one level of nested structure

```
List(List(1, 2), List(3, 4)).flatten // List(1, 2, 3, 4)
```

- **flatMap**: map + flatten

- **foreach**: it is like map but returns nothing

Classes and Objects

```scala
class Calculator {
  val brand: String = "HP"
  def add(m: Int, n: Int): Int = m + n
}

val calc = new Calculator
calc.add(1, 2)
println(calc.brand)
```

Classes and Objects

```scala
class Calculator {
  val brand: String = "HP"
  def add(m: Int, n: Int): Int = m + n
}

val calc = new Calculator
calc.add(1, 2)
println(calc.brand)
```

- A singleton is a class that can have only one instance.

```scala
object Test {
  def main(args: Array[String]) { ... }
}

Test.main(null)
```

Case Classes and Pattern Matching

- Case classes are used to store and match on the contents of a class.
- They are designed to be used with pattern matching.
- You can construct them without using new.

```
case class Calc(brand: String, model: String)

def calcType(calc: Calc) = calc match {
  case Calc("hp", "20B") => "financial"
  case Calc("hp", "48G") => "scientific"
  case Calc("hp", "30B") => "business"
  case _ => "Calculator of unknown type"
}

calcType(Calc("hp", "20B"))
```

Simple Build Tool (SBT)

- An open source build tool for Scala and Java projects.
- Similar to Java's Maven or Ant.
- It is written in Scala.

SBT - Hello World!

```scala
// make dir hello and edit Hello.scala
object Hello {
  def main(args: Array[String]) {
    println("Hello world.")
  }
}
```

```
$ cd hello
$ sbt compile run
```

Common Commands

- `compile`: compiles the main sources.
- `run <argument>*`: run the main class.
- `package`: creates a jar file.
- `console`: starts the Scala interpreter.
- `clean`: deletes all generated files.
- `help <command>`: displays detailed help for the specified command.

Create a Simple Project

- Create `project` directory.
- Create `src/main/scala` directory.
- Create `build.sbt` in the project root.

build.sbt

- A list of Scala expressions, separated by blank lines.
- Located in the project's base directory.

```
$ cat build.sbt
name := "hello"

version := "1.0"

scalaVersion := "2.10.4"
```

Add Dependencies

- Add in `build.sbt`.
- Module ID format:
  `"groupID" %% "artifact" % "version" % "configuration"`

```
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.0.0"

// multiple dependencies
libraryDependencies ++= Seq(
    "org.apache.spark" %% "spark-core" % "1.0.0",
    "org.apache.spark" %% "spark-streaming" % "1.0.0"
)
```

- sbt uses the standard Maven2 repository by default, but you can add more resolvers.

```
resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

Scala Hands-on Exercises (1/4)

- Declare a list of integers as a variable called `myNumbers`

Scala Hands-on Exercises (1/4)

- Declare a list of integers as a variable called `myNumbers`

```
val myNumbers = List(1, 2, 5, 4, 7, 3)
```

Scala Hands-on Exercises (1/4)

- Declare a list of integers as a variable called `myNumbers`

```
val myNumbers = List(1, 2, 5, 4, 7, 3)
```

- Declare a function, `pow`, that computes the second power of an Int

Scala Hands-on Exercises (1/4)

- Declare a list of integers as a variable called `myNumbers`

```
val myNumbers = List(1, 2, 5, 4, 7, 3)
```

- Declare a function, `pow`, that computes the second power of an Int

```
def pow(a: Int): Int = a * a
```

Scala Hands-on Exercises (2/4)

- Apply the function to `myNumbers` using the `map` function

Scala Hands-on Exercises (2/4)

- Apply the function to `myNumbers` using the `map` function

```
myNumbers.map(x => pow(x))
// or
myNumbers.map(pow(_))
// or
myNumbers.map(pow)
```

Scala Hands-on Exercises (2/4)

- Apply the function to myNumbers using the map function

```
myNumbers.map(x => pow(x))
// or
myNumbers.map(pow(_))
// or
myNumbers.map(pow)
```

- Write the pow function inline in a map call, using closure notation

Scala Hands-on Exercises (2/4)

- Apply the function to `myNumbers` using the `map` function

```
myNumbers.map(x => pow(x))
// or
myNumbers.map(pow(_))
// or
myNumbers.map(pow)
```

- Write the `pow` function inline in a `map` call, using closure notation

```
myNumbers.map(x => x * x)
```

Scala Hands-on Exercises (2/4)

- Apply the function to myNumbers using the map function

```
myNumbers.map(x => pow(x))
// or
myNumbers.map(pow(_))
// or
myNumbers.map(pow)
```

- Write the pow function inline in a map call, using closure notation

```
myNumbers.map(x => x * x)
```

- Iterate through myNumbers and print out its items

Scala Hands-on Exercises (2/4)

- Apply the function to `myNumbers` using the `map` function

```
myNumbers.map(x => pow(x))
// or
myNumbers.map(pow(_))
// or
myNumbers.map(pow)
```

- Write the `pow` function inline in a `map` call, using closure notation

```
myNumbers.map(x => x * x)
```

- Iterate through `myNumbers` and print out its items

```
for (i <- myNumbers)
  println(i)
// or
myNumbers.foreach(println)
```

Scala Hands-on Exercises (3/4)

- Declare a list of pair of string and integers as a variable called `myList`

Scala Hands-on Exercises (3/4)

- Declare a list of pair of string and integers as a variable called `myList`

```
val myList = List[(String, Int)](("a", 1), ("b", 2), ("c", 3))
```

Scala Hands-on Exercises (3/4)

- Declare a list of pair of string and integers as a variable called `myList`

```
val myList = List[(String, Int)](("a", 1), ("b", 2), ("c", 3))
```

- Write an inline function to increment the integer values of the list `myList`

Scala Hands-on Exercises (3/4)

- Declare a list of pair of string and integers as a variable called `myList`

```
val myList = List[(String, Int)](("a", 1), ("b", 2), ("c", 3))
```

- Write an inline function to increment the integer values of the list `myList`

```
val x = v.map { case (name, age) =>  age + 1 }
// or
val x = v.map(i => i._2 + 1)
// or
val x = v.map(_._2 + 1)
```

Scala Hands-on Exercises (4/4)

- Do a word-count of a text file: create a Map with words as keys and counts of the number of occurrences of the word as values

Scala Hands-on Exercises (4/4)

- Do a word-count of a text file: create a Map with words as keys and counts of the number of occurrences of the word as values

- You can load a text file as an array of lines as shown below:

```
import scala.io.Source
val lines = Source.fromFile("/root/spark/README.md").getLines.toArray
```

Scala Hands-on Exercises (4/4)

- Do a word-count of a text file: create a Map with words as keys and counts of the number of occurrences of the word as values

- You can load a text file as an array of lines as shown below:

```
import scala.io.Source
val lines = Source.fromFile("/root/spark/README.md").getLines.toArray
```

- Then, instantiate a `HashMap[String, Int]` and use functional methods to populate it with word-counts

Scala Hands-on Exercises (4/4)

- Do a word-count of a text file: create a Map with words as keys and counts of the number of occurrences of the word as values

- You can load a text file as an array of lines as shown below:

```
import scala.io.Source
val lines = Source.fromFile("/root/spark/README.md").getLines.toArray
```

- Then, instantiate a `HashMap[String, Int]` and use functional methods to populate it with word-counts

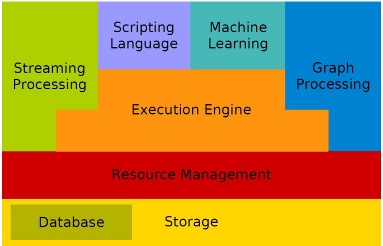
```
val counts = new collection.mutable.HashMap[String, Int].withDefaultValue(0)
lines.flatMap(_.split("""\W+""")).foreach(word => counts(word) += 1)
counts.foreach(println)
```

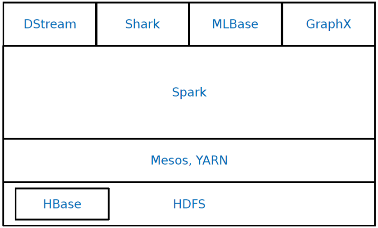
What is Spark?

- An efficient distributed general-purpose data analysis platform.

- Focusing on ease of programming and high performance.

Spark Big Data Analytics Stack





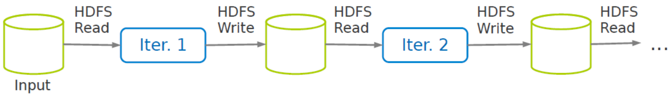
Motivation

- MapReduce programming model has not been designed for complex operations, e.g., data mining.
- Very expensive, i.e., always goes to disk and HDFS.

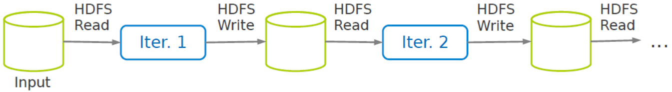
Solution

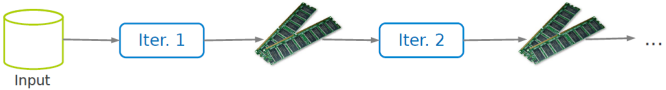
- Extends MapReduce with more operators.
- Support for advanced data flow graphs.
- In-memory and out-of-core processing.

Spark vs. Hadoop

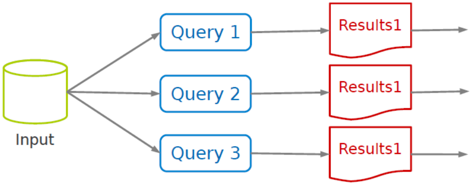


Spark vs. Hadoop

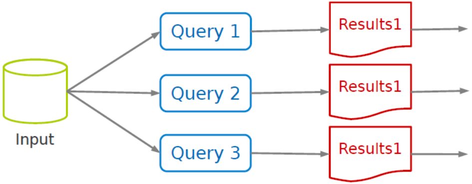


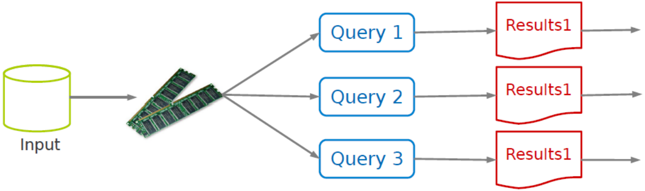


Spark vs. Hadoop



Spark vs. Hadoop





Resilient Distributed Datasets (RDD) (1/2)

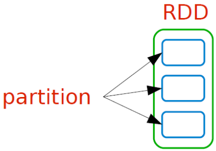
- A distributed memory abstraction.

Resilient Distributed Datasets (RDD) (1/2)

- A distributed memory abstraction.
- Immutable collections of objects spread across a cluster.

Resilient Distributed Datasets (RDD) (2/2)

- An RDD is divided into a number of partitions, which are atomic pieces of information.



- Partitions of an RDD can be stored on different nodes of a cluster.

RDD Operators

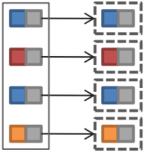
- Higher-order functions: transformations and actions.
- Transformations: lazy operators that create new RDDs.
- Actions: launch a computation and return a value to the program or write data to the external storage.

Transformations vs. Actions

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V},\mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T],RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)],RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)],RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T],RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.,* HDFS

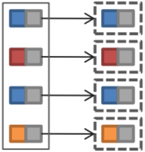
RDD Transformations - Map

- All pairs are independently processed.



RDD Transformations - Map

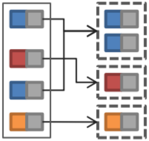
- All pairs are independently processed.



```scala
// passing each element through a function.
val nums = sc.parallelize(Array(1, 2, 3))
val squares = nums.map(x => x * x) // {1, 4, 9}
```

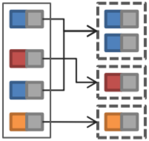
RDD Transformations - GroupBy

- Pairs with identical key are grouped.
- Groups are independently processed.



RDD Transformations - GroupBy

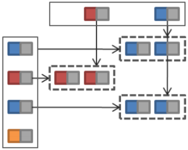
- Pairs with identical key are grouped.
- Groups are independently processed.



```
val schools = sc.parallelize(Seq(("sics", 1), ("kth", 1), ("sics", 2)))

schools.groupByKey()
// {("sics", (1, 2)), ("kth", (1))}

schools.reduceByKey((x, y) => x + y)
// {("sics", 3), ("kth", 1)}
```

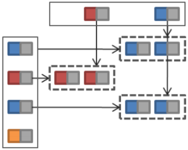
RDD Transformations - Join

- Performs an equi-join on the key.
- Join candidates are independently processed.



RDD Transformations - Join

- Performs an equi-join on the key.
- Join candidates are independently processed.



```
val list1 = sc.parallelize(Seq(("sics", "10"),
                               ("kth", "50"),
                               ("sics", "20")))

val list2 = sc.parallelize(Seq(("sics", "upsala"),
                               ("kth", "stockholm")))

list1.join(list2)
  // ("sics", ("10", "upsala"))
  // ("sics", ("20", "upsala"))
  // ("kth", ("50", "stockholm"))
```

Basic RDD Actions

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

Basic RDD Actions

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

Basic RDD Actions

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

- Return the number of elements in the RDD.

```
nums.count() // 3
```

Basic RDD Actions

- Return all the elements of the RDD as an array.

```
val nums = sc.parallelize(Array(1, 2, 3))
nums.collect() // Array(1, 2, 3)
```

- Return an array with the first n elements of the RDD.

```
nums.take(2) // Array(1, 2)
```

- Return the number of elements in the RDD.

```
nums.count() // 3
```

- Aggregate the elements of the RDD using the given function.

```
nums.reduce((x, y) => x + y) // 6
```

Creating RDDs

- Turn a collection into an RDD.

```
val a = sc.parallelize(Array(1, 2, 3))
```

- Load text file from local FS, HDFS, or S3.

```
val a = sc.textFile("file.txt")
val b = sc.textFile("directory/*.txt")
val c = sc.textFile("hdfs://namenode:9000/path/file")
```

SparkContext

- Main entry point to Spark functionality.
- Available in shell as variable sc.
- In standalone programs, you should make your own.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

val sc = new SparkContext(master, appName, [sparkHome], [jars])
```

Spark Hands-on Exercises (1/3)

- Read data from the given file hamlet and create an RDD named `pagecounts`

Spark Hands-on Exercises (1/3)

- Read data from the given file hamlet and create an RDD named `pagecounts`

```
val pagecounts = sc.textFile("hamlet")
```

Spark Hands-on Exercises (1/3)

- Read data from the given file hamlet and create an RDD named `pagecounts`

```
val pagecounts = sc.textFile("hamlet")
```

- Get the first 10 lines of hamlet

Spark Hands-on Exercises (1/3)

- Read data from the given file hamlet and create an RDD named `pagecounts`

```
val pagecounts = sc.textFile("hamlet")
```

- Get the first 10 lines of hamlet

```
pagecounts.take(10).foreach(println)
```

Spark Hands-on Exercises (1/3)

- Read data from the given file hamlet and create an RDD named `pagecounts`

```
val pagecounts = sc.textFile("hamlet")
```

- Get the first 10 lines of hamlet

```
pagecounts.take(10).foreach(println)
```

- Count the total records in the data set `pagecounts`

Spark Hands-on Exercises (1/3)

- Read data from the given file hamlet and create an RDD named `pagecounts`

```
val pagecounts = sc.textFile("hamlet")
```

- Get the first 10 lines of hamlet

```
pagecounts.take(10).foreach(println)
```

- Count the total records in the data set `pagecounts`

```
pagecounts.count
```

Spark Hands-on Exercises (2/3)

- Filter the data set `pagecounts` and return the items that have the word `this`, and cache in the memory

Spark Hands-on Exercises (2/3)

- Filter the data set `pagecounts` and return the items that have the word `this`, and cache in the memory

```
val linesWithThis = pagecounts.filter(line => line.contains("this")).cache
\\ or
val linesWithThis = pagecounts.filter(_.contains("this")).cache
```

Spark Hands-on Exercises (2/3)

- Filter the data set `pagecounts` and return the items that have the word `this`, and cache in the memory

```
val linesWithThis = pagecounts.filter(line => line.contains("this")).cache
\\ or
val linesWithThis = pagecounts.filter(_.contains("this")).cache
```

- Find the lines with the most number of words.

Spark Hands-on Exercises (2/3)

- Filter the data set `pagecounts` and return the items that have the word `this`, and cache in the memory

```
val linesWithThis = pagecounts.filter(line => line.contains("this")).cache
\\ or
val linesWithThis = pagecounts.filter(_.contains("this")).cache
```

- Find the lines with the most number of words.

```
linesWithThis.map(line => line.split(" ").size)
  .reduce((a, b) => if (a > b) a else b)
```

Spark Hands-on Exercises (2/3)

- Filter the data set `pagecounts` and return the items that have the word `this`, and cache in the memory

```
val linesWithThis = pagecounts.filter(line => line.contains("this")).cache
\\ or
val linesWithThis = pagecounts.filter(_.contains("this")).cache
```

- Find the lines with the most number of words.

```
linesWithThis.map(line => line.split(" ").size)
  .reduce((a, b) => if (a > b) a else b)
```

- Count the total number of words

Spark Hands-on Exercises (2/3)

- Filter the data set pagecounts and return the items that have the word this, and cache in the memory

```
val linesWithThis = pagecounts.filter(line => line.contains("this")).cache
\\ or
val linesWithThis = pagecounts.filter(_.contains("this")).cache
```

- Find the lines with the most number of words.

```
linesWithThis.map(line => line.split(" ").size)
  .reduce((a, b) => if (a > b) a else b)
```

- Count the total number of words

```
val wordCounts = linesWithThis.flatMap(line => line.split(" ")).count
\\ or
val wordCounts = linesWithThis.flatMap(_.split(" ")).count
```

Spark Hands-on Exercises (3/3)

- Count the number of distinct words

Spark Hands-on Exercises (3/3)

- Count the number of distinct words

```
val uniqueWordCounts = linesWithThis.flatMap(_.split(" ")).distinct.count
```

Spark Hands-on Exercises (3/3)

- Count the number of distinct words

```
val uniqueWordCounts = linesWithThis.flatMap(_.split(" ")).distinct.count
```

- Count the number of each word

Spark Hands-on Exercises (3/3)

- Count the number of distinct words

```
val uniqueWordCounts = linesWithThis.flatMap(_.split(" ")).distinct.count
```

- Count the number of each word

```
val eachWordCounts = linesWithThis.flatMap(_.split(" "))
  .map(word => (word, 1))
  .reduceByKey((a, b) => a + b)
```

Spark Streaming

Motivation

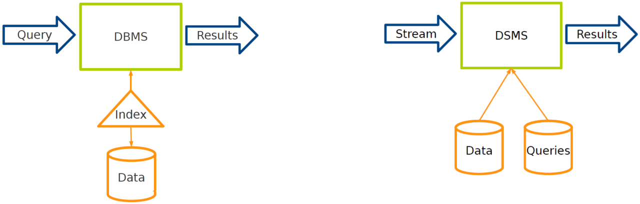
- Many applications must process large streams of live data and provide results in real-time.
- Processing information as it flows, without storing them persistently.

Motivation

- Many applications must process large streams of live data and provide results in real-time.
- Processing information as it flows, without storing them persistently.
- Traditional DBMSs:
  - Store and index data before processing it.
  - Process data only when explicitly asked by the users.
  - Both aspects contrast with our requirements.

DBMS vs. DSMS (1/3)

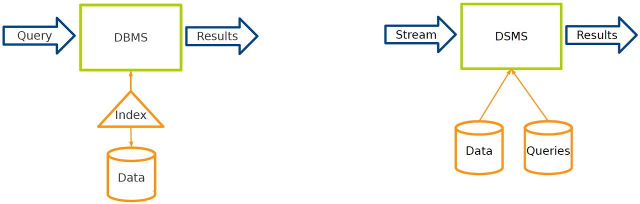
- DBMS: persistent data where updates are relatively infrequent.
- DSMS: transient data that is continuously updated.



DBMS vs. DSMS (2/3)

- DBMS: runs queries just once to return a complete answer.

- DSMS: executes standing queries, which run continuously and provide updated answers as new data arrives.



DBMS vs. DSMS (3/3)

- Despite these differences, DSMSs resemble DBMSs: both process incoming data through a sequence of transformations based on SQL operators, e.g., selections, aggregates, joins.

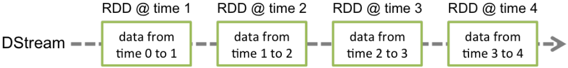
Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.

Spark Streaming

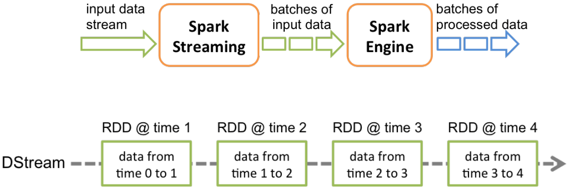
- Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chop up the live stream into batches of X seconds.
  - Spark treats each batch of data as RDDs and processes them using RDD operations.
  - Finally, the processed results of the RDD operations are returned in batches.

- DStream: sequence of RDDs representing a stream of data.
    - TCP sockets, Twitter, HDFS, Kafka, ...



Spark Streaming API (1/4)

- DStream: sequence of RDDs representing a stream of data.
  - TCP sockets, Twitter, HDFS, Kafka, ...

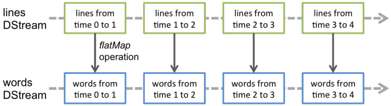


- Initializing Spark streaming

```
val scc = new StreamingContext(master, appName, batchDuration,
[sparkHome], [jars])
```

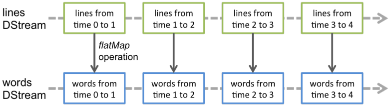
Spark Streaming API (2/4)

- Transformations: modify data from on DStream to a new DStream.
  - Standard RDD operations (stateless operations): map, join, ...

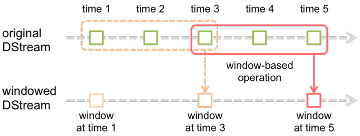


Spark Streaming API (2/4)

- Transformations: modify data from on DStream to a new DStream.
  - Standard RDD operations (stateless operations): map, join, ...



  - Stateful operations: group all the records from a sliding window of the past time intervals into one RDD: window, reduceByAndWindow, ...



Window length: the duration of the window.
Slide interval: the interval at which the operation is performed.

Spark Streaming API (3/4)

- Output operations: send data to external entity
  - saveAsHadoopFiles, foreach, print, ...

Spark Streaming API (3/4)

- Output operations: send data to external entity
  - saveAsHadoopFiles, foreach, print, ...

- Attaching input sources

```
ssc.textFileStream(directory)
ssc.socketStream(hostname, port)
```

Spark Streaming API (4/4)

- Stream + Batch: It can be used to apply any RDD operation that is not exposed in the DStream API.

```
val spamInfoRDD = sparkContext.hadoopFile(...)
// join data stream with spam information to do data cleaning
val cleanedDStream = inputDStream.transform(_.join(spamInfoRDD).filter(...))
```

Spark Streaming API (4/4)

- Stream + Batch: It can be used to apply any RDD operation that is not exposed in the DStream API.

```
val spamInfoRDD = sparkContext.hadoopFile(...)
// join data stream with spam information to do data cleaning
val cleanedDStream = inputDStream.transform(_.join(spamInfoRDD).filter(...))
```

- Stream + Interactive: Interactive queries on stream state from the Spark interpreter

```
freqs.slice("21:00", "21:05").topK(10)
```

Spark Streaming API (4/4)

- Stream + Batch: It can be used to apply any RDD operation that is not exposed in the DStream API.

```
val spamInfoRDD = sparkContext.hadoopFile(...)
// join data stream with spam information to do data cleaning
val cleanedDStream = inputDStream.transform(_.join(spamInfoRDD).filter(...))
```

- Stream + Interactive: Interactive queries on stream state from the Spark interpreter

```
freqs.slice("21:00", "21:05").topK(10)
```

- Starting/stopping the streaming computation

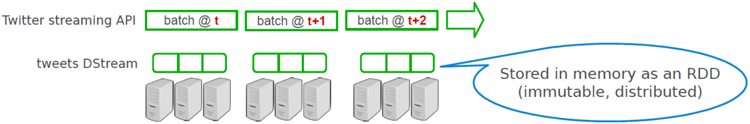
```
ssc.start()
ssc.stop()
ssc.awaitTermination()
```

Example 1 (1/3)

- Get hash-tags from Twitter.

```
val ssc = new StreamingContext("local[2]", "Tweets", Seconds(1))
val tweets = TwitterUtils.createStream(ssc, None)
```

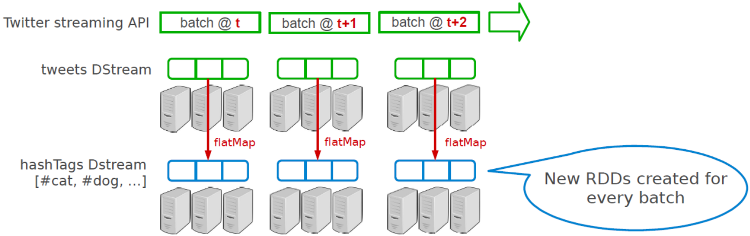
DStream: a sequence of RDD representing a stream of data



- Get hash-tags from Twitter.

```
val ssc = new StreamingContext("local[2]", "Tweets", Seconds(1))
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
```

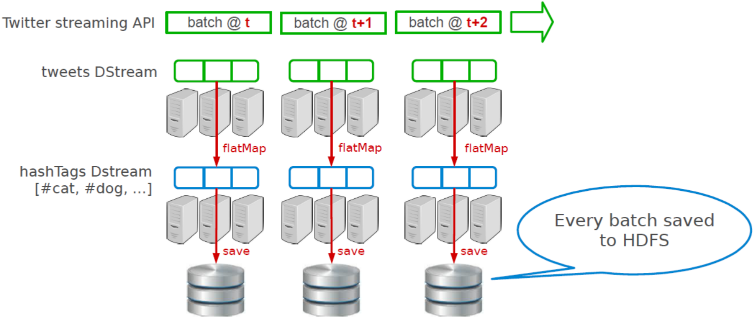
transformation: modify data in one DStream to create another DStream



Example 1 (3/3)

- Get hash-tags from Twitter.

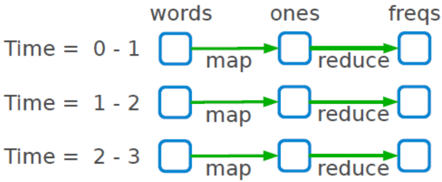
```
val ssc = new StreamingContext("local[2]", "Tweets", Seconds(1))
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```



Example 2

- Count frequency of words received every second.

```
val ssc = new StreamingContext("local[2]", "NetworkWordCount", Seconds(1))
val lines = ssc.socketTextStream(ip, port)
val words = lines.flatMap(_.split(" "))
val ones = words.map(x => (x, 1))
val freqs = ones.reduceByKey(_ + _)
```



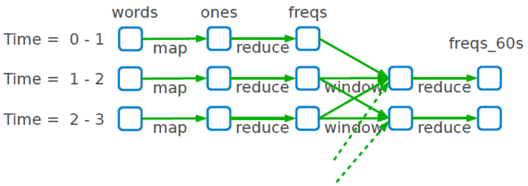
Example 3

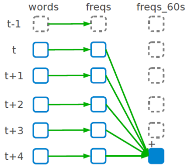
- Count frequency of words received in last minute.

```
val ssc = new StreamingContext("local[2]", "NetworkWordCount", Seconds(1))
val lines = ssc.socketTextStream(ip, port)
val words = lines.flatMap(_.split(" "))
val ones = words.map(x => (x, 1))
val freqs = ones.reduceByKey(_ + _)
val freqs_60s = freqs.window(Seconds(60), Second(1)).reduceByKey(_ + _)
```

window length

window movement

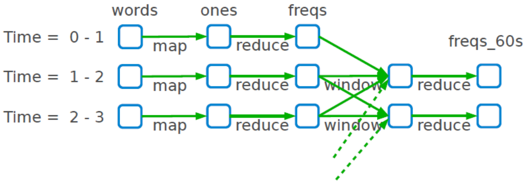


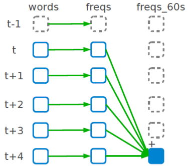


Example 3 - Simpler Model

- Count frequency of words received in last minute.

```
val ssc = new StreamingContext("local[2]", "NetworkWordCount", Seconds(1))
val lines = ssc.socketTextStream(ip, port)
val words = lines.flatMap(_.split(" "))
val ones = words.map(x => (x, 1))
val freqs_60s = ones.reduceByKeyAndWindow(_ + _, Seconds(60), Seconds(1))
```

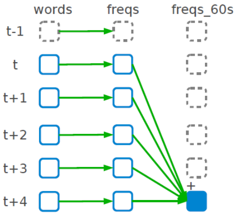




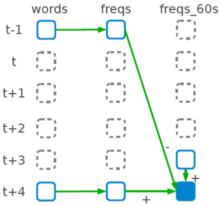
Example 3 - Incremental Window Operators

- Count frequency of words received in last minute.

```
// Associative only
freqs_60s = ones.reduceByKeyAndWindow(_ + _, Seconds(60), Seconds(1))

// Associative and invertible
freqs_60s = ones.reduceByKeyAndWindow(_ + _, _ - _, Seconds(60), Seconds(1))
```



Associative only



Associative and invertible

Spark Streaming Hands-on Exercises (1/2)

- Stream data through a TCP connection and port 9999

```
nc -lk 9999
```

Spark Streaming Hands-on Exercises (1/2)

- Stream data through a TCP connection and port 9999

```
nc -lk 9999
```

- import the streaming libraries

Spark Streaming Hands-on Exercises (1/2)

- Stream data through a TCP connection and port 9999

```
nc -lk 9999
```

- import the streaming libraries

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
```

Spark Streaming Hands-on Exercises (1/2)

- Stream data through a TCP connection and port 9999

```
nc -lk 9999
```

- import the streaming libraries

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
```

- Print out the incoming stream every five seconds at port 9999

Spark Streaming Hands-on Exercises (1/2)

- Stream data through a TCP connection and port 9999

```
nc -lk 9999
```

- import the streaming libraries

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
```

- Print out the incoming stream every five seconds at port 9999

```
val ssc = new StreamingContext("local[2]", "NetworkWordCount", Seconds(5))
val lines = ssc.socketTextStream("127.0.0.1", 9999)
lines.print()
```

Spark Streaming Hands-on Exercises (1/2)

- Count the number of each word in the incoming stream every five seconds at port 9999

Spark Streaming Hands-on Exercises (1/2)

- Count the number of each word in the incoming stream every five seconds at port 9999

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._

val ssc = new StreamingContext("local[2]", "NetworkWordCount", Seconds(1))
val lines = ssc.socketTextStream("127.0.0.1", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(x => (x, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()
```

Spark Streaming Hands-on Exercises (2/2)

- Extend the code to generate word count over last 30 seconds of data, and repeat the computation every 10 seconds

Spark Streaming Hands-on Exercises (2/2)

- Extend the code to generate word count over last 30 seconds of data, and repeat the computation every 10 seconds

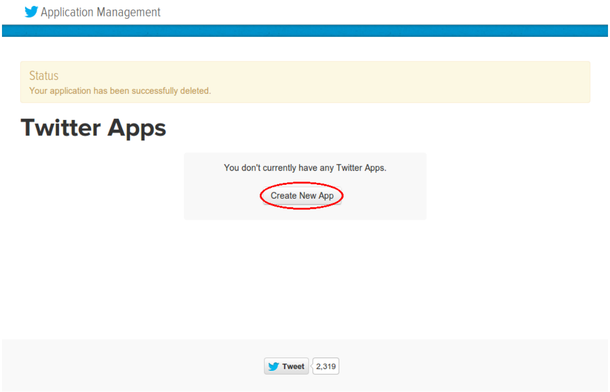
```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._

val ssc = new StreamingContext("local[2]", "NetworkWordCount", Seconds(5))
val lines = ssc.socketTextStream("127.0.0.1", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val windowedWordCounts = pairs
  .reduceByKeyAndWindow(_ + _, _ - _, Seconds(30), Seconds(10))
windowedWordCounts.print()
wordCounts.print()
```

Spark Streaming Hands-on Exercises - Twitter (1/7)

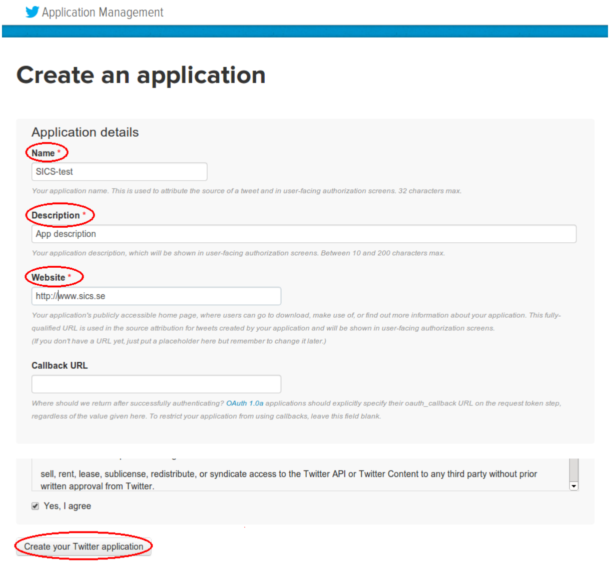
- Twitter credential setup: to access Twitter's sample tweet stream

Spark Streaming Hands-on Exercises - Twitter (1/7)

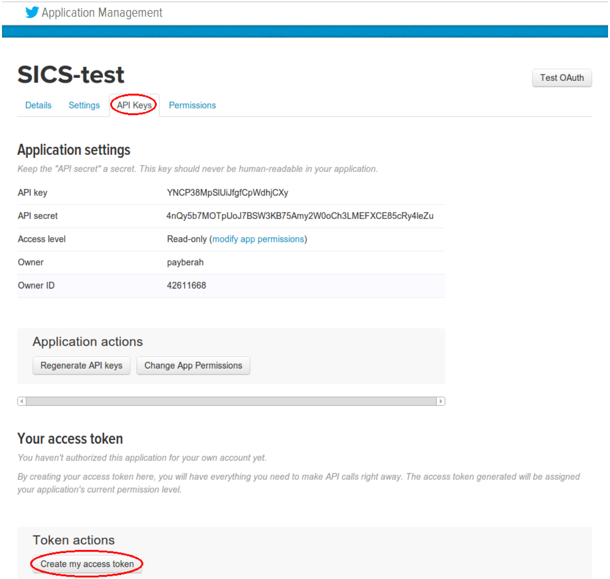
- Twitter credential setup: to access Twitter's sample tweet stream
- Open the link: https://apps.twitter.com/

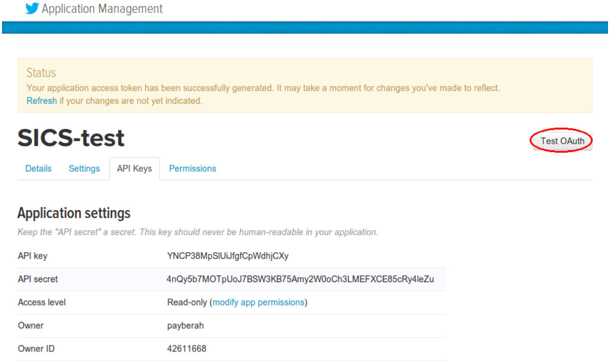


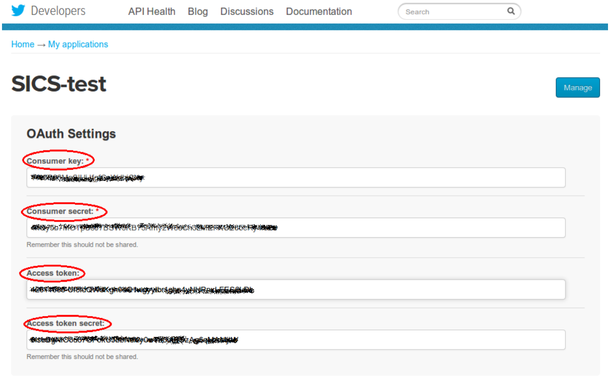
Spark Streaming Hands-on Exercises - Twitter (2/7)



Spark Streaming Hands-on Exercises - Twitter (3/7)




Application Management
Status
Your application access token has been successfully generated. It may take a moment for changes you've made to reflect.
Refresh if your changes are not yet indicated.
SICS-test
Test OAuth
Details
Settings
API Keys
Permissions
Application settings
Keep the "API secret" a secret. This key should never be human-readable in your application.
API key YNCP38MpSlUiJfgfCpWdhjCXy
API secret 4nQy5b7MOTpUoJ7BSW3KB75Amy2W0oCh3LMEFXCE85cRy4leZu
Access level Read-only (modify app permissions)
Owner payberah
Owner ID 42611668


Developers
API Health
Blog
Discussions
Documentation
Search
Home → My applications
SICS-test
Manage
OAuth Settings
Consumer key: *
Consumer secret: *
Remember this should not be shared.
Access token:
Access token secret:
Remember this should not be shared.

- Create an `StreamingContext` for a batch duration of 5 seconds and use this context to create a stream of tweets

Spark Streaming Hands-on Exercises - Twitter (6/7)

- Create an StreamingContext for a batch duration of 5 seconds and use this context to create a stream of tweets

```
import org.apache.spark.streaming.twitter._

val ssc = new StreamingContext("local[2]", "Tweets", Seconds(5))
val tweets = TwitterUtils.createStream(ssc, None)
```

Spark Streaming Hands-on Exercises - Twitter (6/7)

- Create an StreamingContext for a batch duration of 5 seconds and use this context to create a stream of tweets

```
import org.apache.spark.streaming.twitter._

val ssc = new StreamingContext("local[2]", "Tweets", Seconds(5))
val tweets = TwitterUtils.createStream(ssc, None)
```

- Print the status text of the some of the tweets

Spark Streaming Hands-on Exercises - Twitter (6/7)

- Create an StreamingContext for a batch duration of 5 seconds and use this context to create a stream of tweets

```
import org.apache.spark.streaming.twitter._

val ssc = new StreamingContext("local[2]", "Tweets", Seconds(5))
val tweets = TwitterUtils.createStream(ssc, None)
```

- Print the status text of the some of the tweets

```
val statuses = tweets.map(status => status.getText())
statuses.print()
```

Spark Streaming Hands-on Exercises - Twitter (6/7)

- Create an StreamingContext for a batch duration of 5 seconds and use this context to create a stream of tweets

```
import org.apache.spark.streaming.twitter._

val ssc = new StreamingContext("local[2]", "Tweets", Seconds(5))
val tweets = TwitterUtils.createStream(ssc, None)
```

- Print the status text of the some of the tweets

```
val statuses = tweets.map(status => status.getText())
statuses.print()
```

- Get the stream of hashtags from the stream of tweets

Spark Streaming Hands-on Exercises - Twitter (6/7)

- Create an `StreamingContext` for a batch duration of 5 seconds and use this context to create a stream of tweets

```
import org.apache.spark.streaming.twitter._

val ssc = new StreamingContext("local[2]", "Tweets", Seconds(5))
val tweets = TwitterUtils.createStream(ssc, None)
```

- Print the status text of the some of the tweets

```
val statuses = tweets.map(status => status.getText())
statuses.print()
```

- Get the stream of hashtags from the stream of tweets

```
val words = statuses.flatMap(status => status.split(" "))
val hashtags = words.filter(word => word.startsWith("#"))
hashtags.print()
```

Spark Streaming Hands-on Exercises - Twitter (7/7)

- Set a path for periodic checkpointing of the intermediate data, and then count the hashtags over a one minute window

Spark Streaming Hands-on Exercises - Twitter (7/7)

- Set a path for periodic checkpointing of the intermediate data, and then count the hashtags over a one minute window

```
ssc.checkpoint("/home/sics/temp")
val counts = hashtags.map(tag => (tag, 1))
  .reduceByKeyAndWindow(_ + _, _ - _, Seconds(60), Seconds(5))
counts.print()
```

Spark Streaming Hands-on Exercises - Twitter (7/7)

- Set a path for periodic checkpointing of the intermediate data, and then count the hashtags over a one minute window

```
ssc.checkpoint("/home/sics/temp")
val counts = hashtags.map(tag => (tag, 1))
  .reduceByKeyAndWindow(_ + _, _ - _, Seconds(60), Seconds(5))
counts.print()
```

- Find the top 10 hashtags based on their counts

Spark Streaming Hands-on Exercises - Twitter (7/7)

- Set a path for periodic checkpointing of the intermediate data, and then count the hashtags over a one minute window

```
ssc.checkpoint("/home/sics/temp")
val counts = hashtags.map(tag => (tag, 1))
  .reduceByKeyAndWindow(_ + _, _ - _, Seconds(60), Seconds(5))
counts.print()
```

- Find the top 10 hashtags based on their counts

```
val sortedCounts = counts.map { case (tag, count) => (count, tag) }
  .transform(rdd => rdd.sortByKey(false))
sortedCounts.foreachRDD(rdd =>
  println("\nTop 10 hashtags:\n" + rdd.take(10).mkString("\n")))
```

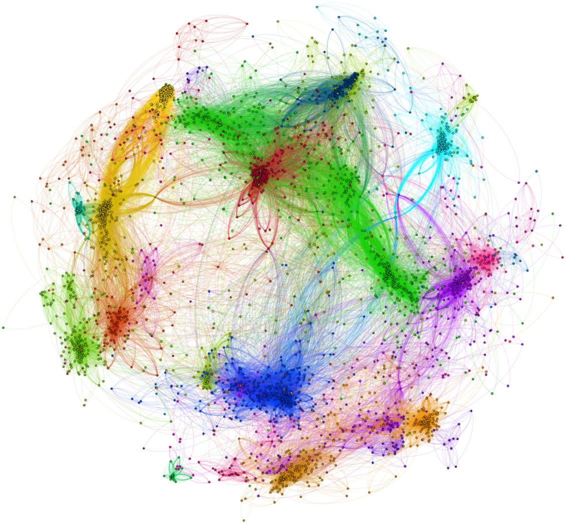
GraphX

Introduction

- Graphs provide a flexible abstraction for describing relationships between discrete objects.

- Many problems can be modeled by graphs and solved with appropriate graph algorithms.

Large Graph

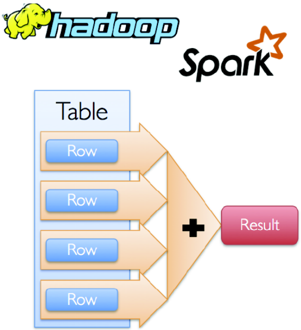


Large-Scale Graph Processing

- Large graphs need large-scale processing.

- A large graph either cannot fit into memory of single computer or it fits with huge cost.

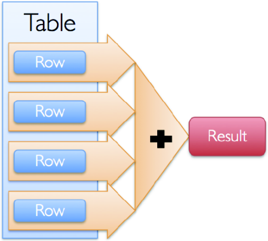
Can we use platforms like MapReduce or Spark, which are based on data-parallel model, for large-scale graph proceeding?



Data-Parallel Model for Large-Scale Graph Processing

- The platforms that have worked well for developing parallel applications are not necessarily effective for large-scale graph problems.

- Why?



Graph Algorithms Characteristics (1/2)

- Unstructured problems
  - Difficult to extract parallelism based on partitioning of the data: the irregular structure of graphs.
  - Limited scalability: unbalanced computational loads resulting from poorly partitioned data.

Graph Algorithms Characteristics (1/2)

- Unstructured problems
  - Difficult to extract parallelism based on partitioning of the data: the irregular structure of graphs.
  - Limited scalability: unbalanced computational loads resulting from poorly partitioned data.

- Data-driven computations
  - Difficult to express parallelism based on partitioning of computation: the structure of computations in the algorithm is not known a priori.
  - The computations are dictated by nodes and links of the graph.

Graph Algorithms Characteristics (2/2)

- Poor data locality
  - The computations and data access patterns do not have much locality: the irregular structure of graphs.

Graph Algorithms Characteristics (2/2)

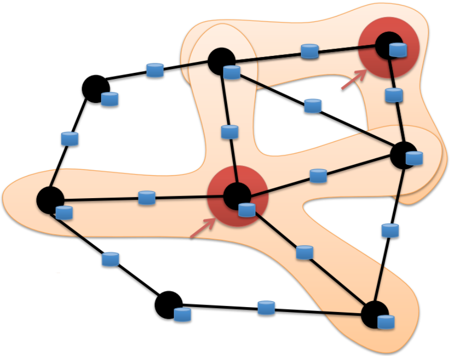
- Poor data locality
  - The computations and data access patterns do not have much locality: the irregular structure of graphs.
- High data access to computation ratio
  - Graph algorithms are often based on exploring the structure of a graph to perform computations on the graph data.
  - Runtime can be dominated by waiting memory fetches: low locality.

Proposed Solution

Graph-Parallel Processing

Proposed Solution

Graph-Parallel Processing

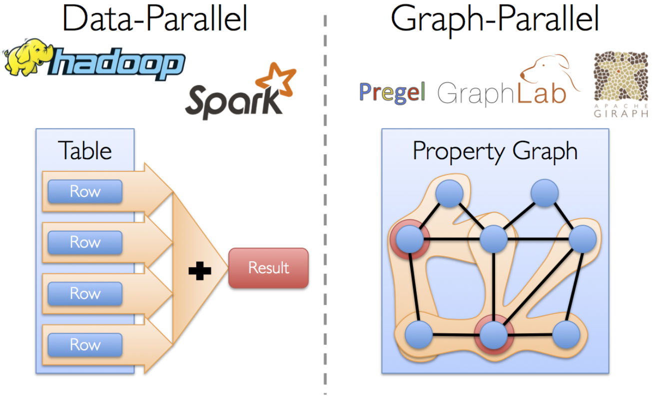


- Computation typically depends on the neighbors.

Graph-Parallel Processing

- Restricts the types of computation.
- New techniques to partition and distribute graphs.
- Exploit graph structure.
- Executes graph algorithms orders-of-magnitude faster than more general data-parallel systems.

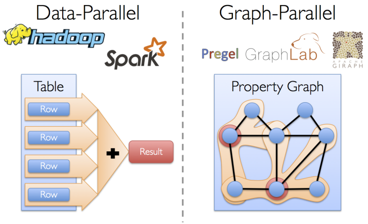
Data-Parallel vs. Graph-Parallel Computation



Data-Parallel vs. Graph-Parallel Computation

- Data-parallel computation
  - Record-centric view of data.
  - Parallelism: processing independent data on separate resources.

- Graph-parallel computation
  - Vertex-centric view of graphs.
  - Parallelism: partitioning graph (dependent) data across processing resources, and resolving dependencies (along edges) through iterative computation and communication.



Graph-Parallel Computation Frameworks

APACHE GIRAPH

GraphLab

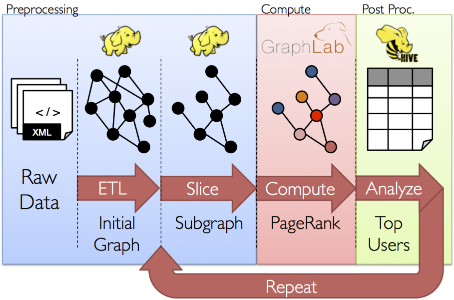
Pregel oogle

Data-Parallel vs. Graph-Parallel Computation

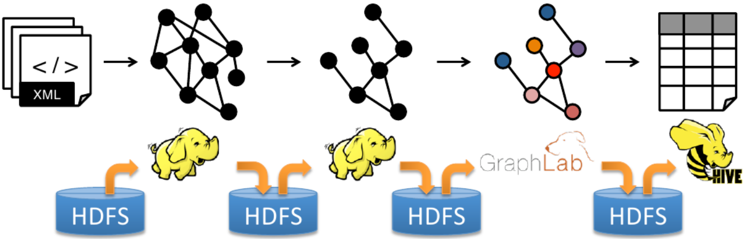
- Graph-parallel computation: restricting the types of computation to achieve performance.

Data-Parallel vs. Graph-Parallel Computation

- Graph-parallel computation: restricting the types of computation to achieve performance.
- But, the same restrictions make it difficult and inefficient to express many stages in a typical graph-analytics pipeline.

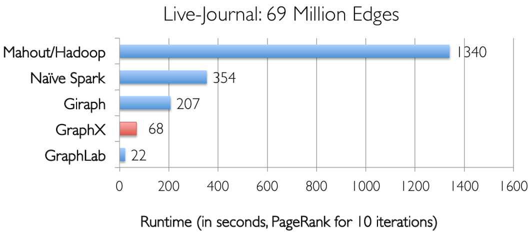


Data-Parallel and Graph-Parallel Pipeline

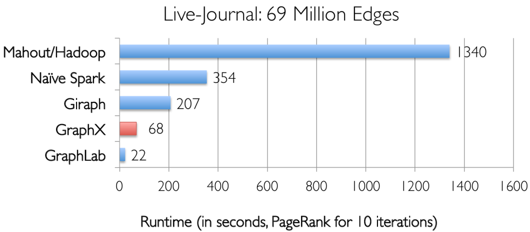


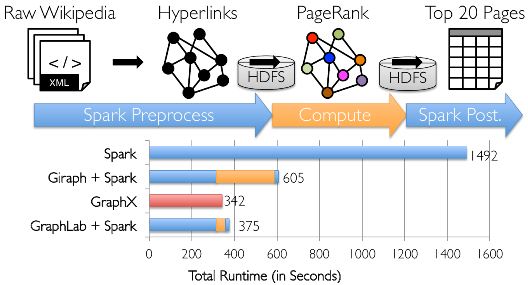
- Moving between table and graph views of the same physical data.
- Inefficient: extensive data movement and duplication across the network and file system.

GraphX vs. Data-Parallel/Graph-Parallel Systems



GraphX vs. Data-Parallel/Graph-Parallel Systems



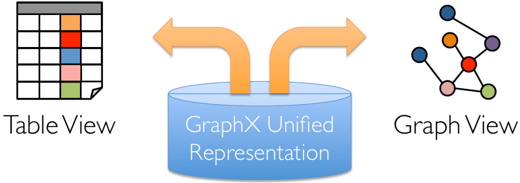


GraphX

- New API that blurs the distinction between Tables and Graphs.
- New system that unifies Data-Parallel and Graph-Parallel systems.
- It is implemented on top of Spark.

Unifying Data-Parallel and Graph-Parallel Analytics

- 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.

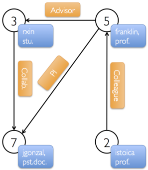


Data Model

- Property Graph: represented using two Spark RDDs:
  - Edge collection: VertexRDD
  - Vertex collection: EdgeRDD

```scala
// VD: the type of the vertex attribute
// ED: the type of the edge attribute
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED, VD]
}
```

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

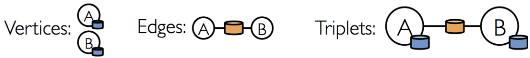
Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Primitive Data Types

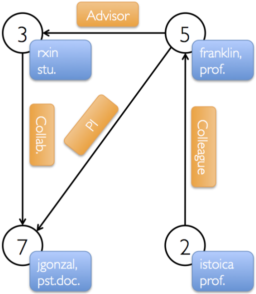
```
// Vertex collection
class VertexRDD[VD] extends RDD[(VertexId, VD)]

// Edge collection
class EdgeRDD[ED] extends RDD[Edge[ED]]
case class Edge[ED, VD](srcId: VertexId = 0, dstId: VertexId = 0,
                        attr: ED = null.asInstanceOf[ED])

// Edge Triple
class EdgeTriplet[VD, ED] extends Edge[ED]
```

- EdgeTriplet represents an edge along with the vertex attributes of its neighboring vertices.



Example (1/3)

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Example (2/3)

```
val sc: SparkContext

// Create an RDD for the vertices
val users: RDD[(Long, (String, String))] = sc.parallelize(
    Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
          (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

// Create an RDD for edges
val relationships: RDD[Edge[String]] = sc.parallelize(
    Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
          Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))

// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")

// Build the initial Graph
val userGraph: Graph[(String, String), String] =
    Graph(users, relationships, defaultUser)
```

Example (3/3)

```
// Constructed from above
val userGraph: Graph[(String, String), String]

// Count all users which are postdocs
userGraph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count

// Count all the edges where src > dst
userGraph.edges.filter(e => e.srcId > e.dstId).count

// Use the triplets view to create an RDD of facts
val facts: RDD[String] = graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " +
    triplet.attr + " of " + triplet.dstAttr._1)

// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")

facts.collect.foreach(println)
```

Property Operators (1/2)

```
class Graph[VD, ED] {
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]

  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]

  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

- They yield new graphs with the vertex or edge properties modified by the map function.
- The graph structure is unaffected.

Property Operators (2/2)

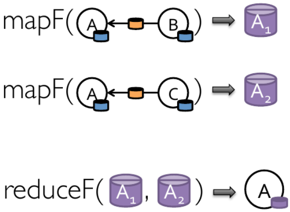
```
val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```

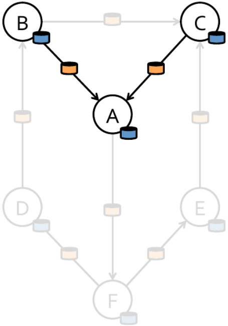
```
val newVertices = graph.vertices.map((id, attr) => (id, mapUdf(id, attr)))
val newGraph = Graph(newVertices, graph.edges)
```

- Both are logically equivalent, but the second one does not preserve the structural indices and would not benefit from the GraphX system optimizations.

Map Reduce Triplets

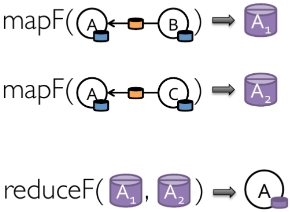
- Map-Reduce for each vertex

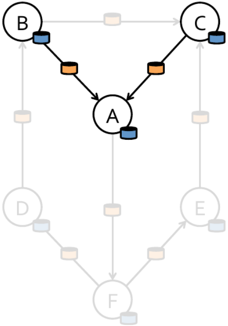




Map Reduce Triplets

- Map-Reduce for each vertex






```
// what is the age of the oldest follower for each user?
val oldestFollowerAge = graph.mapReduceTriplets(
  e => Iterator((e.dstAttr, e.srcAttr)), // Map
  (a, b) => max(a, b) // Reduce
).vertices
```

Structural Operators

```
class Graph[VD, ED] {
  // returns a new graph with all the edge directions reversed
  def reverse: Graph[VD, ED]

  // returns the graph containing only the vertices and edges that satisfy
  // the vertex predicate
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]

  // a subgraph by returning a graph that contains the vertices and edges
  // that are also found in the input graph
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
}
```

Structural Operators Example

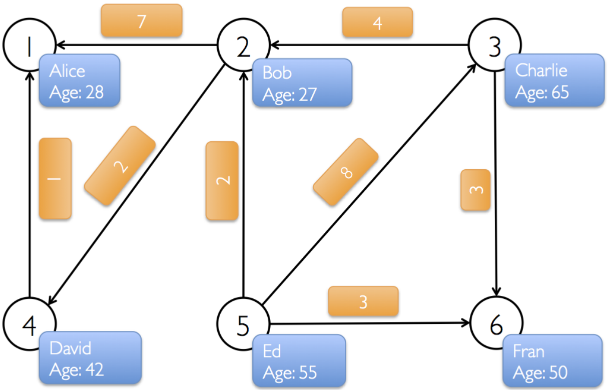
```
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)

// Run Connected Components
val ccGraph = graph.connectedComponents()

// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")

// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

Join Operators

- To join data from external collections (RDDs) with graphs.

```
class Graph[VD, ED] {
  // joins the vertices with the input RDD and returns a new graph
  // by applying the map function to the result of the joined vertices
  def joinVertices[U](table: RDD[(VertexId, U)])
      (map: (VertexId, VD, U) => VD): Graph[VD, ED]

  // similarly to joinVertices, but the map function is applied to
  // all vertices and can change the vertex property type
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])
      (map: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]
}
```

GraphX Hands-on Exercises (1/7)



GraphX Hands-on Exercises (2/7)

- import the streaming libraries

GraphX Hands-on Exercises (2/7)

- import the streaming libraries

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
```

GraphX Hands-on Exercises (2/7)

- import the streaming libraries

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
```

- Build the property graph shown in the last page

GraphX Hands-on Exercises (2/7)

- import the streaming libraries

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
```

- Build the property graph shown in the last page

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

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))

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

GraphX Hands-on Exercises (3/7)

- Display the name of the users older than 30 years old

GraphX Hands-on Exercises (3/7)

- Display the name of the users older than 30 years old

```
graph.vertices.filter { case (id, (name, age)) => age > 30 }.foreach {
  case (id, (name, age)) => println(s"$name is $age")
}
```

GraphX Hands-on Exercises (3/7)

- Display the name of the users older than 30 years old

```
graph.vertices.filter { case (id, (name, age)) => age > 30 }.foreach {
  case (id, (name, age)) => println(s"$name is $age")
}
```

- Display who follows who (through the edges direction).

```
/**
  * Triplet has the following Fields:
  *   triplet.srcAttr: (String, Int)
  *   triplet.dstAttr: (String, Int)
  *   triplet.attr: Int
  *   triplet.srcId: VertexId
  *   triplet.dstId: VertexId
  */
```

GraphX Hands-on Exercises (3/7)

- Display the name of the users older than 30 years old

```
graph.vertices.filter { case (id, (name, age)) => age > 30 }.foreach {
  case (id, (name, age)) => println(s"$name is $age")
}
```

- Display who follows who (through the edges direction).

```
/**
  * Triplet has the following Fields:
  *   triplet.srcAttr: (String, Int)
  *   triplet.dstAttr: (String, Int)
  *   triplet.attr: Int
  *   triplet.srcId: VertexId
  *   triplet.dstId: VertexId
  */
```

```
graph.triplets.foreach(t =>
  println(s"${t.srcAttr._1} follows ${t.dstAttr._1}"))
```

GraphX Hands-on Exercises (4/7)

- Compute the total age of followers of each user and print them out

- Compute the total age of followers of each user and print them out

```
val followers: VertexRDD[Int] = graph.mapReduceTriplets[Int](
  triplet => Iterator(...), // map
  (a, b) => ... // reduce
)
```

GraphX Hands-on Exercises (4/7)

- Compute the total age of followers of each user and print them out

```
val followers: VertexRDD[Int] = graph.mapReduceTriplets[Int](
  triplet => Iterator(...), // map
  (a, b) => ... // reduce
)
```

```
val followers: VertexRDD[Int] = graph.mapReduceTriplets[Int](
  triplet => Iterator((triplet.dstId, triplet.srcAttr._2)),
  (a, b) => a + b)

followers.collect.foreach(print)
```

GraphX Hands-on Exercises (5/7)

- Compute the average age of followers of each user and print them out

GraphX Hands-on Exercises (5/7)

- Compute the average age of followers of each user and print them out

```
val followers: VertexRDD[(Int, Double)] = graph
  .mapReduceTriplets[(Int, Double)](
    triplet => Iterator(...), // map
    (a, b) => (...) // reduce
)

val avgAgeOfFollowers: VertexRDD[Double] = followers.mapValues(...)
```

GraphX Hands-on Exercises (5/7)

- Compute the average age of followers of each user and print them out

```
val followers: VertexRDD[(Int, Double)] = graph
  .mapReduceTriplets[(Int, Double)](
    triplet => Iterator(...), // map
    (a, b) => (...) // reduce
)

val avgAgeOfFollowers: VertexRDD[Double] = followers.mapValues(...)
```

```
val followers: VertexRDD[(Int, Double)] = graph
  .mapReduceTriplets[(Int, Double)](
    triplet => Iterator((triplet.dstId, (1, triplet.srcAttr._2))),
    (a, b) => (a._1 + b._1, a._2 + b._2))

val avgAgeOfFollowers: VertexRDD[Double] =
  followers.mapValues((id, value) => value match {
    case (count, totalAge) => totalAge / count
  })

avgAgeOfFollowers.collect.foreach(print)
```

GraphX Hands-on Exercises (6/7)

- Make a subgraph of the users that are 30 or older

GraphX Hands-on Exercises (6/7)

- Make a subgraph of the users that are 30 or older

```
val olderGraph = graph.subgraph(vpred = ...)
```

GraphX Hands-on Exercises (6/7)

- Make a subgraph of the users that are 30 or older

```
val olderGraph = graph.subgraph(vpred = ...)
```

```
val olderGraph = graph.subgraph(vpred = (id, u) => u._2 >= 30)
```

GraphX Hands-on Exercises (7/7)

- Compute the connected components and display the component id of each user in oldGraph

GraphX Hands-on Exercises (7/7)

- Compute the connected components and display the component id of each user in oldGraph

```
val cc = olderGraph...

olderGraph.vertices.leftJoin(cc.vertices) {
...
}.foreach{...}
```

GraphX Hands-on Exercises (7/7)

- Compute the connected components and display the component id of each user in oldGraph

```
val cc = olderGraph...

olderGraph.vertices.leftJoin(cc.vertices) {
...
}.foreach{...}
```

```
val cc = olderGraph.connectedComponents

olderGraph.vertices.leftJoin(cc.vertices) {
  case (id, u, comp) => s"${u._1} is in component ${comp.get}"
}.foreach{ case (id, str) => println(str) }
```

Questions?