Técnicas para Big Data

Clase 11 - GraphX

GraphX

¿Cómo podríamos aprovecharnos de una **arquitectura paralela** para resolver problemas de grafos?

- Shortest path
- Contar triángulos
- Computar pagerank de los nodos

Idea 1: Map reduce

- Base de datos en Neo4j con personas, todas con atributo año
- Queremos computar la edad promedio
- ¿Es posible hacer esto con Map-Reduce?

Idea 1: Map reduce

- Base de datos en Neo4j con ciudades y distancias entre ellas
- Queremos computar camino más corto desde A hasta B

¿Que pasa si tratamos de hacer esto con Map Reduce?

No es claro que podamos ganar algo.

Solución: paralelismo sincrónico

Imaginemos que queremos saber quien tiene la edad mayor entre nosotros ¿Cómo lo haríamos?

Solución: paralelismo sincrónico

Ronda 1

- Comunico **mi edad** con todos los que tengo al lado
- Ellos me comunican su edad
- Me quedo con la mayor

Rondas sucesivas

- Comunico **la mayor edad (que tengo)** a los de al lado
- Ellos me comunican su edad mayor
- Me quedo con la mayor

Solución: paralelismo sincrónico

Para Finalizar

- Si nadie me da una edad mayor voto para parar
- Me detengo si todos votamos por parar

Pregel

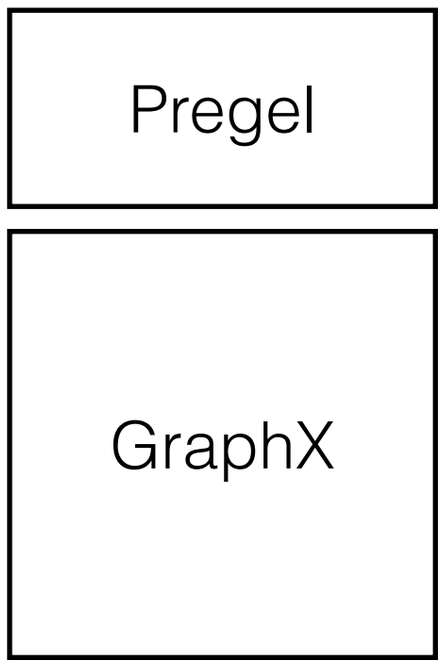
- Cada nodo del grafo es un agente

- Agentes tienen dos componentes principales:
  - Una funcion que dice que mensajes se mandan
  - Otra que dice que hacer con los mensajes que llegan

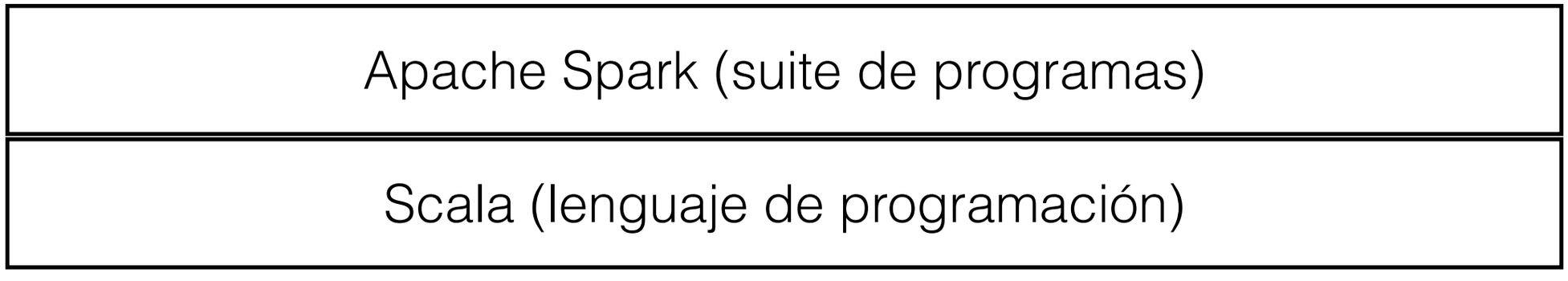
Está todo coordinado, todos mandan mensajes al mismo tiempo

- Nodos vota por parar (cuando se cumple una condición)
- Algoritmo termina cuando todos paran

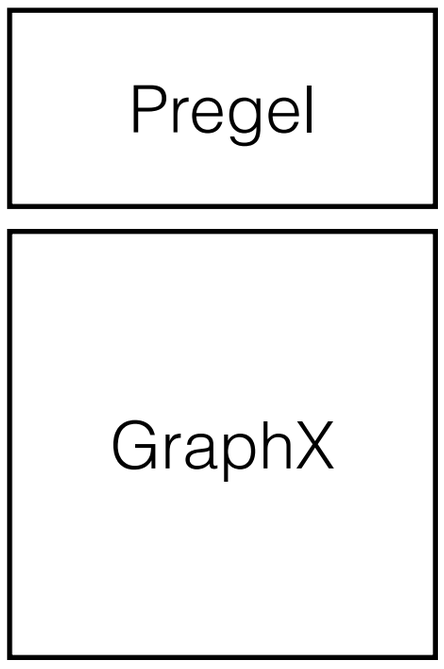
Pregel y GraphX (arquitectura)



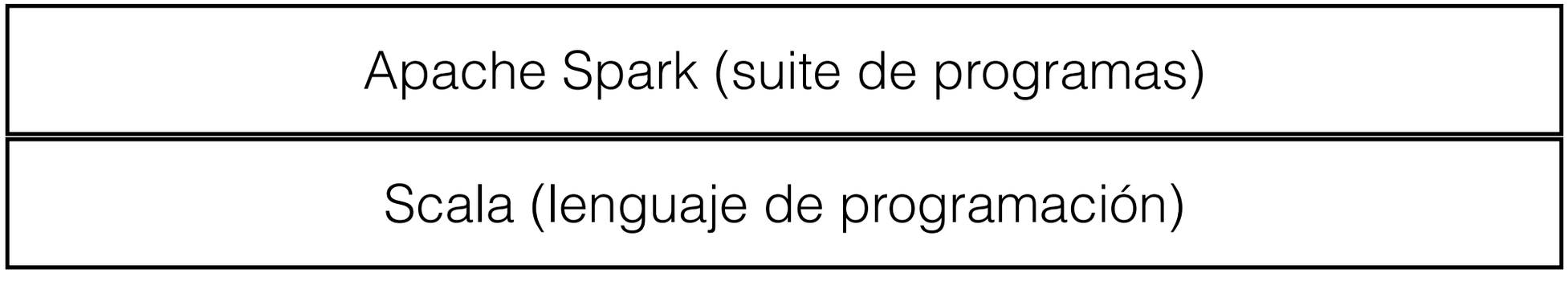
- GraphX es una librería para manejar datos en forma de grafos
- Pregel se implementa como API



Pregel y GraphX (arquitectura)



- GraphX es una librería para manejar datos en forma de grafos
- Pregel se implementa como API



(No hay GraphX para Python)

GraphX

GraphX

- GraphX es una API de Spark
- Sirve para procesar grafos, incluso de forma paralela
- Contiene varios algoritmos ya implementados

GraphX

Definir vértices

GraphX

Definir vértices

Para definir los vértices en GraphX necesitamos un arreglo:

GraphX

Definir vértices

Para definir los vértices en GraphX necesitamos un arreglo:

```
val vertexArray = Array(
  (1L, ("Alicia", 52)),
  (2L, ("Bob", 17)),
  (3L, ("Carlos", 65)),
  (4L, ("David", 24)),
  (5L, ("Eduardo", 42))
)
```

GraphX

Definir vértices

GraphX

Definir vértices

Cada elemento del arreglo es un vértice de la forma `(id, (propiedades))`

GraphX

Definir vértices

Cada elemento del arreglo es un vértice de la forma `(id, (propiedades))`

El `id` es de tipo `long`, mientras que las propiedades se definen como una tupla

GraphX

Definir vértices

Cada elemento del arreglo es un vértice de la forma `(id, (propiedades))`

El `id` es de tipo `long`, mientras que las propiedades se definen como una tupla

En el ejemplo el primer elemento de la tupla representa el nombre y el segundo la edad

GraphX

Definir aristas

GraphX

Definir aristas

Para definir las aristas en GraphX necesitamos un arreglo:

GraphX

Definir aristas

Para definir las aristas en GraphX necesitamos un arreglo:

```
val edgeArray = Array(
  Edge(1L, 2L, "follows"),
  Edge(2L, 1L, "follows"),
  Edge(1L, 4L, "follows"),
  Edge(2L, 4L, "follows"),
  Edge(5L, 4L, "follows"),
  Edge(3L, 2L, "follows"),
  Edge(4L, 3L, "blocks")
)
```

GraphX

Definir aristas

GraphX

Definir aristas

Cada elemento del arreglo es un vértice de la forma `(id_salida, id_entrada, (propiedades))`

GraphX

Definir aristas

Cada elemento del arreglo es un vértice de la forma `(id_salida, id_entrada, (propiedades))`

Los `id` representan los vértices de salida y entrada de la arista respectivamente

GraphX

Definir aristas

Cada elemento del arreglo es un vértice de la forma `(id_salida, id_entrada, (propiedades))`

Los `id` representan los vértices de salida y entrada de la arista respectivamente

Existe una tupla de propiedades para las aristas

GraphX

Definir aristas

GraphX

Definir aristas

En el ejemplo la propiedad es un único elemento que representa un label

GraphX

Definir aristas

En el ejemplo la propiedad es un único elemento que representa un label

Puede haber más de una arista entre dos nodos.

GraphX

Crear el grafo

Luego debemos crear el grafo:

```scala
// Suponemos el Spark Context (sc) inicializado
val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
val edgeRDD: RDD[Edge[String]] = sc.parallelize(edgeArray)

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

GraphX

Crear el grafo

Luego debemos crear el grafo:

```
// Suponemos el Spark Context (sc) inicializado
val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
val edgeRDD: RDD[Edge[String]] = sc.parallelize(edgeArray)

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

Notar que estamos indicando el tipo de las propiedades de los vértices y aristas

GraphX

Crear el grafo

GraphX

Crear el grafo

Cuando inicializamos el grafo:

GraphX

Crear el grafo

Cuando inicializamos el grafo:

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

GraphX

Crear el grafo

Cuando inicializamos el grafo:

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

Estamos indicando las propiedades de los vértices (en este caso `(String, Int)`) y la de las aristas (en este caso `String`)

GraphX

Acceder al grafo entero

GraphX

Acceder al grafo entero

Podemos consultar todo el grafo en forma de triples

GraphX

Acceder al grafo entero

Podemos consultar todo el grafo en forma de triples

```
for (triplet <- graph.triplets.collect) {
  println(s"${triplet.srcAttr} -" +
          s" ${triplet.attr} -> " + "" +
          s"${triplet.dstAttr}")
}
```

GraphX

Acceder al grafo entero

Podemos consultar todo el grafo en forma de triples

```
for (triplet <- graph.triplets.collect) {
  println(s"${triplet.srcAttr} -" +
          s" ${triplet.attr} -> " + "" +
          s"${triplet.dstAttr}")
}
```

```
(Alicia,52) - follows -> (Bob,17)
(Alicia,52) - follows -> (David,24)
(Bob,17) - follows -> (Alicia,52)
(Bob,17) - follows -> (David,24)
(Eduardo,42) - follows -> (David,24)
(Carlos,65) - follows -> (Bob,17)
(David,24) - blocks -> (Carlos,65)
```

GraphX

Src - Dst

GraphX

Src - Dst

`srcAttr`, `attr` y `dstAttr` son atributos especiales de GraphX para obtener los elementos de un triple

GraphX

Src - Dst

`srcAttr`, `attr` y `dstAttr` son atributos especiales de GraphX para obtener los elementos de un triple

`srcAttr` o `src` es el nodo fuente (desde donde sale la arista)

GraphX

Src - Dst

`srcAttr`, `attr` y `dstAttr` son atributos especiales de GraphX para obtener los elementos de un triple

`srcAttr` o `src` es el nodo fuente (desde donde sale la arista)

`dstAttr` o `dst` es el nodo de destino (donde entra la arista)

GraphX

Consultas a los vértices

GraphX

Consultas a los vértices

Podemos filtrar según las propiedades de los vértices

GraphX

Consultas a los vértices

Podemos filtrar según las propiedades de los vértices

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

GraphX

Consultas a los vértices

Podemos filtrar según las propiedades de los vértices

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

```
Alicia is 52
Eduardo is 42
Carlos is 65
```

GraphX

Consultas a las aristas

GraphX

Consultas a las aristas

Podemos filtrar según las propiedades de las aristas

GraphX

Consultas a las aristas

Podemos filtrar según las propiedades de las aristas

```
graph.edges.filter{ case Edge(src, dst, prop) => prop == "blocks" }.
  collect.foreach(e => println(s"Edge: ${e}"))
```

GraphX

Construir subgrafos

GraphX

Construir subgrafos

Podemos construir un subgrafo que satisfaga una condición

GraphX

Construir subgrafos

Podemos construir un subgrafo que satisfaga una condición

```scala
val oldPeople = graph.subgraph(vpred = (id, attr) => attr._2 >= 18)
```

GraphX

Construir subgrafos

Podemos construir un subgrafo que satisfaga una condición

```scala
val oldPeople = graph.subgraph(vpred = (id, attr) => attr._2 >= 18)

val nonBlocked = graph.subgraph(epred = e => e.attr != "blocks")
```

GraphX

Map - Reduce

GraphX

Map - Reduce

Podemos hacer Map - Reduce sobre el grafo

GraphX

Map - Reduce

Podemos hacer Map - Reduce sobre el grafo

El siguiente ejemplo calcula el seguidor más viejo para cada usuario

GraphX

Map - Reduce

Podemos hacer Map - Reduce sobre el grafo

El siguiente ejemplo calcula el seguidor más viejo para cada usuario

```
val oldestFollowers: VertexRDD[Int] = graph.aggregateMessages(
  triplet => {
    triplet.sendToDst(triplet.srcAttr._2)
  },
  (a, b) => max(a, b)
)
oldestFollowers.collect().foreach(v => println(s"${v}"))
```

GraphX

Map - Reduce

GraphX

Map - Reduce

Primero declaramos el tipo del *value* del par `(key, value)` en el map (en este caso es `int`)

GraphX

Map - Reduce

Primero declaramos el tipo del *value* del par `(key, value)` en el map (en este caso es `int`)

```
val oldestFollowers: VertexRDD[Int] = graph.aggregateMessages(
 …
)
```

GraphX

Map - Reduce

GraphX

Map - Reduce

Luego declaramos la función reduce

GraphX

Map - Reduce

Luego declaramos la función reduce

```
(a, b) => max(a, b)
```

GraphX

Map - Reduce

Luego declaramos la función reduce

```
(a, b) => max(a, b)
```

Donde `a` y `b` son **edades** para un `key` (i.e. no son un par `(key, value)`)

GraphX

Map - Reduce

Luego declaramos la función reduce

```
(a, b) => max(a, b)
```

Donde `a` y `b` son **edades** para un `key` (i.e. no son un par `(key, value)`)

Obs. La función `max` debe ser importada

```
import static org.apache.spark.sql.functions.*
```

Pregel

Pregel

Modelo computacional para hacer consultas distribuidas sobre grafos

Pregel

Modelo computacional para hacer consultas distribuidas sobre grafos

Consta de varias iteraciones (*supersteps*) en que cada vértice emite un mensaje a sus vecinos, y ejecutan acciones según el mensaje recibido

Pregel

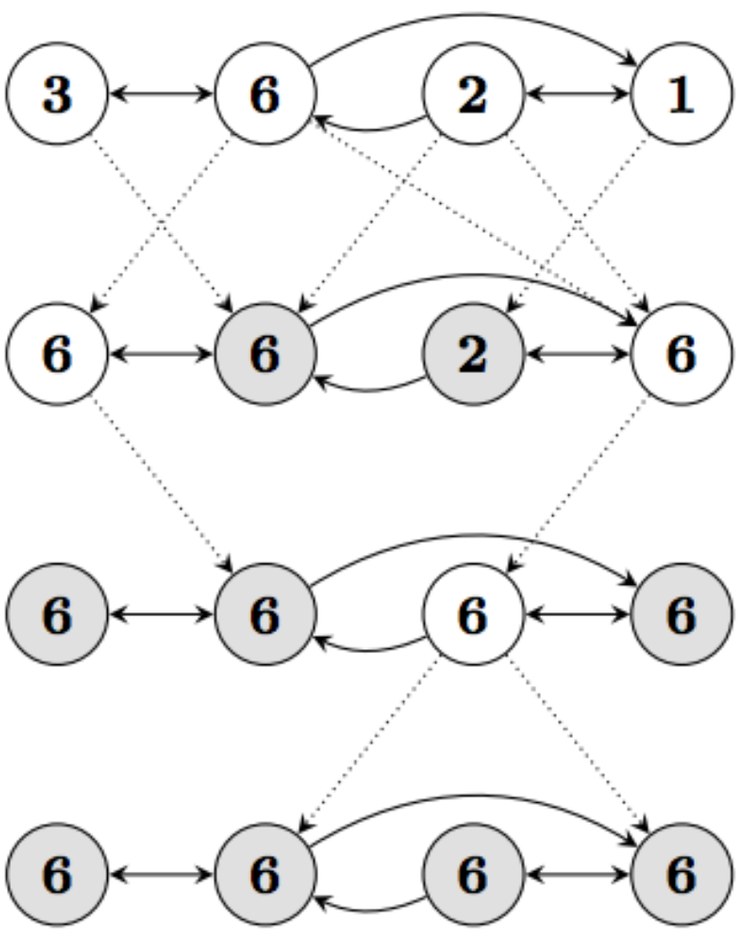
Modelo computacional para hacer consultas distribuidas sobre grafos

Consta de varias iteraciones (*supersteps*) en que cada vértice emite un mensaje a sus vecinos, y ejecutan acciones según el mensaje recibido

GraphX tiene implementada la API de Pregel

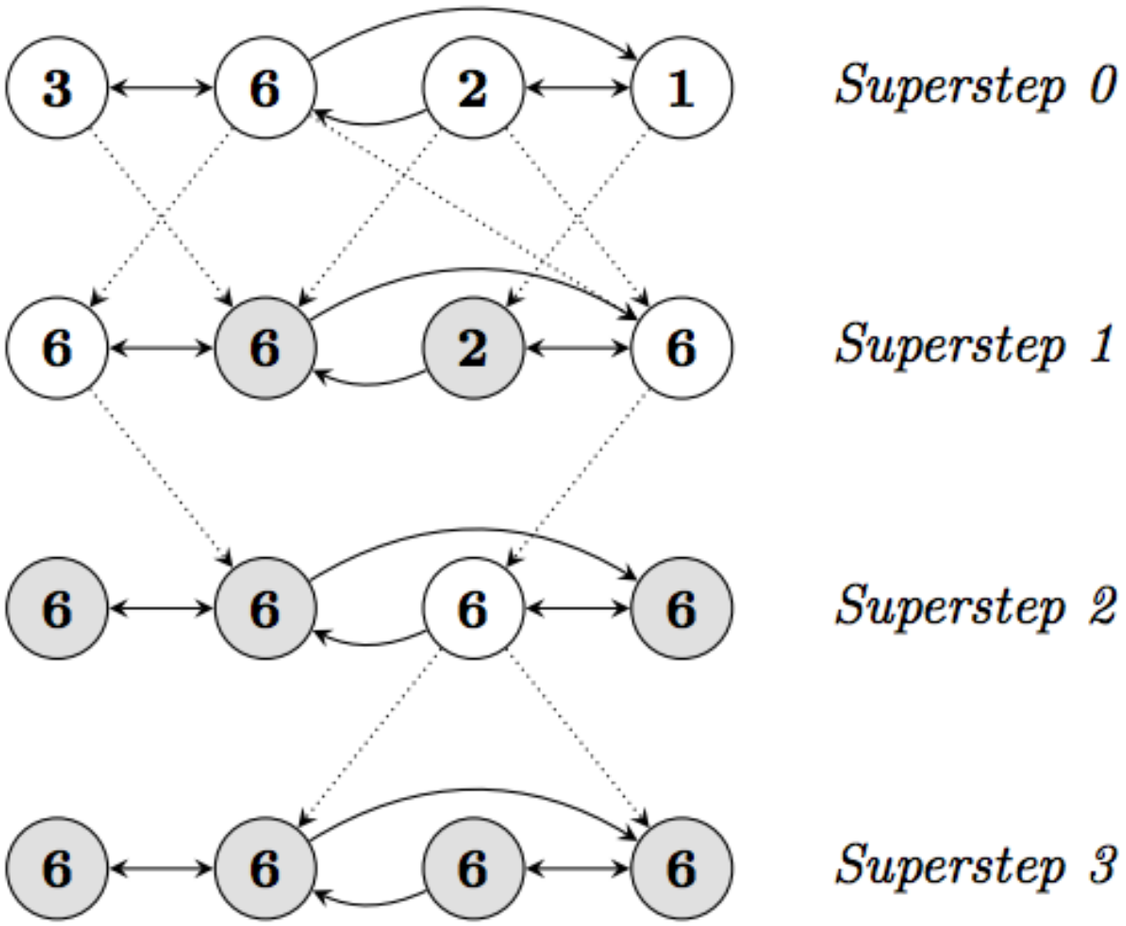
Pregel

Consultar valor máximo de un grafo



Pregel

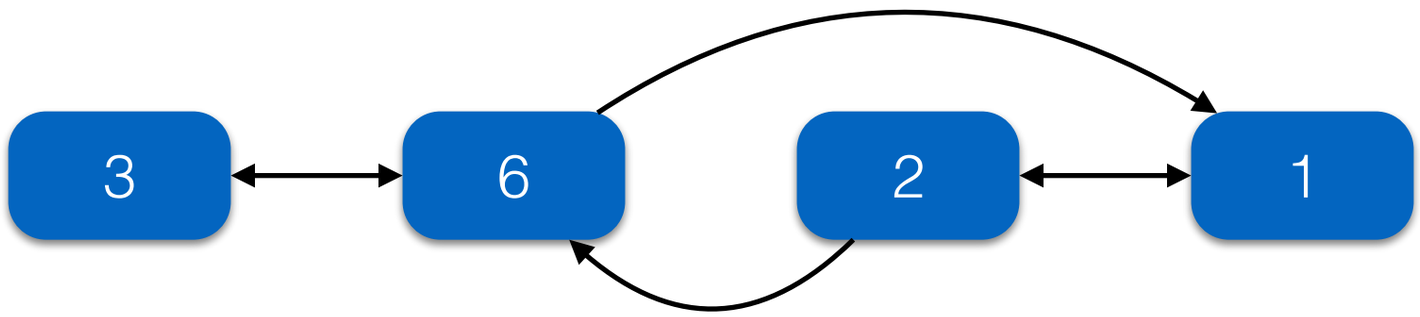
Consultar valor máximo de un grafo



- Se propaga el valor a los vecinos si es mayor
- Un vértice vota por parar cuando no cambia su valor
- El programa para cuando todos paran

Pregel

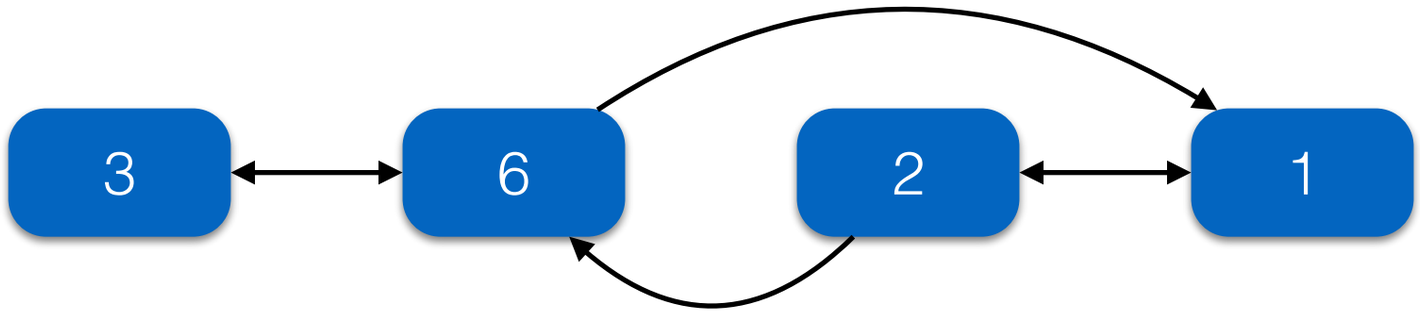
Ejemplo



Pregel

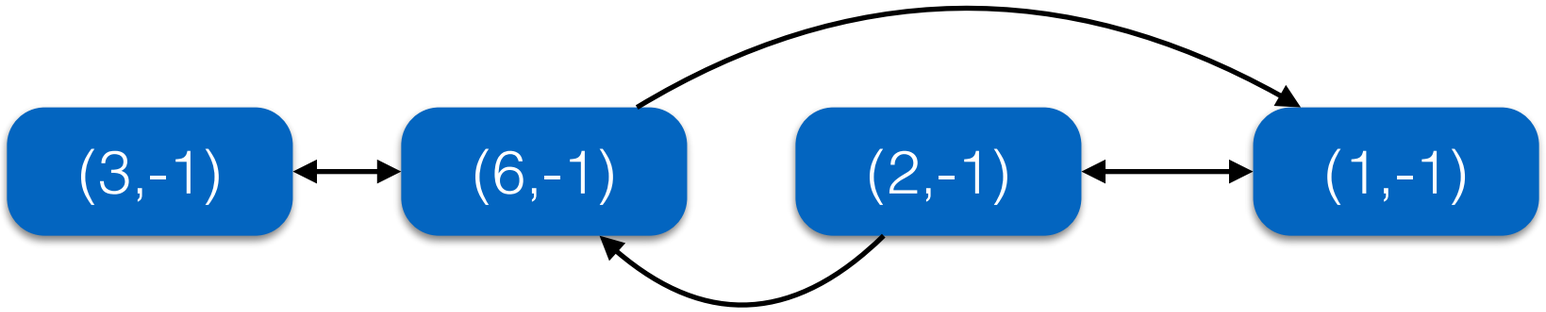
Ejemplo

Consultaremos el valor máximo para el siguiente grafo



Pregel

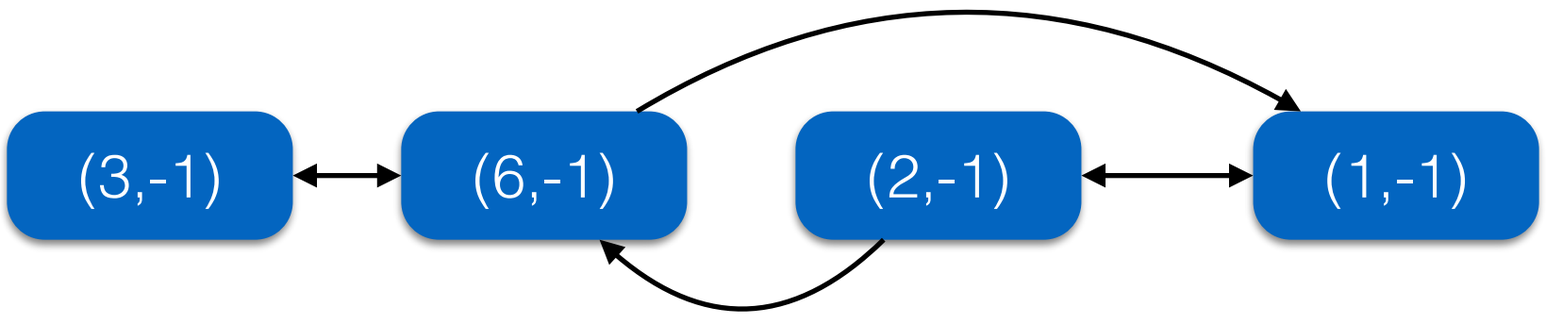
Ejemplo



Pregel

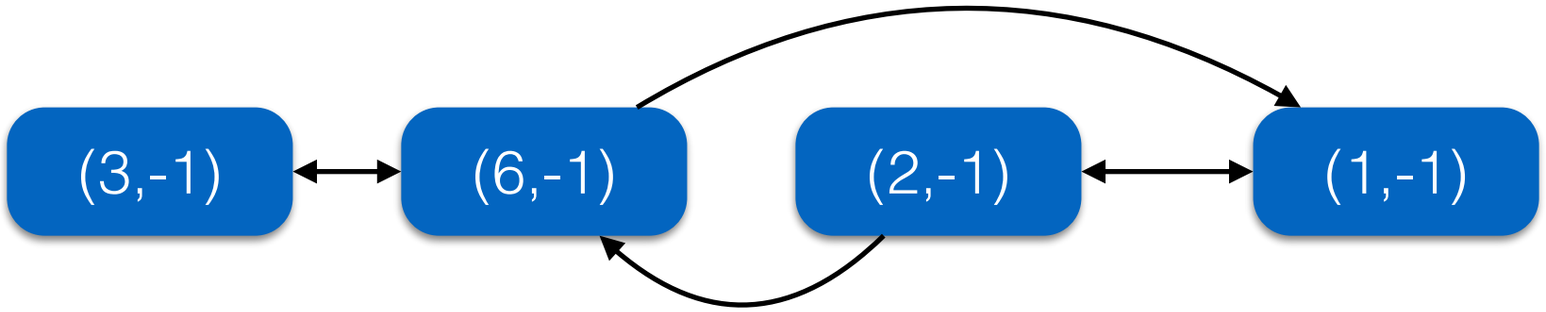
Ejemplo

Inicializamos el grafo en GraphX de la siguiente forma (el segundo elemento de la tupla representará el valor en la iteración anterior para quienes estaban activos):



Pregel

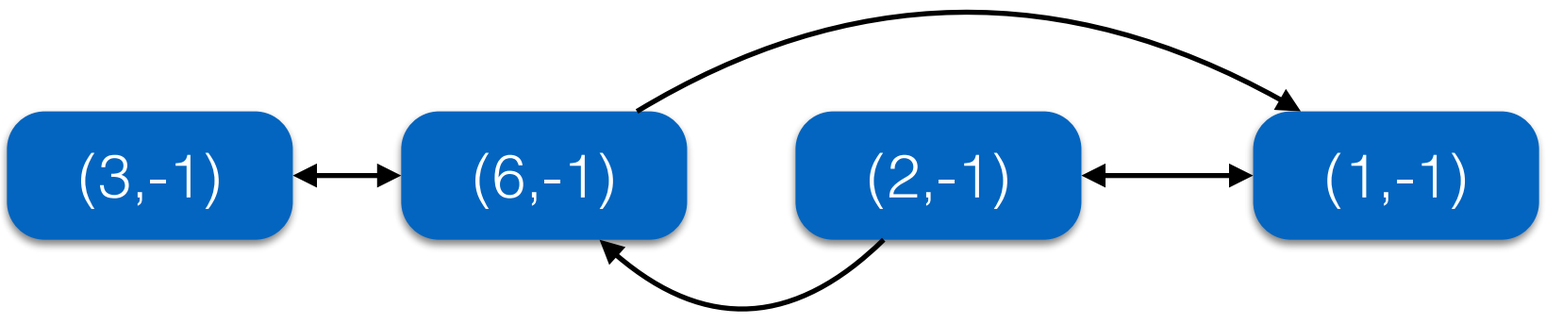
Ejemplo



Pregel

Ejemplo

Inicializamos el grafo en GraphX de la siguiente forma (el segundo elemento de la tupla representará el valor en la iteración anterior para quienes estaban activos):



Pregel

Ejemplo

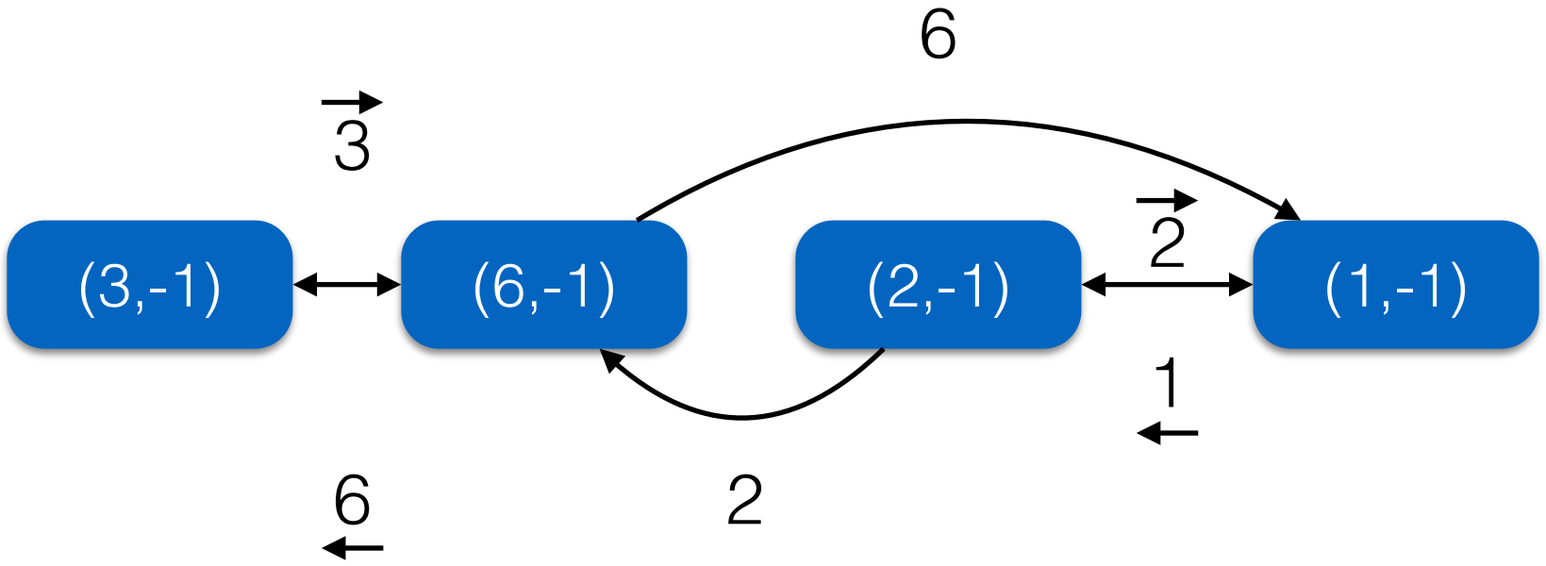
Pregel

Ejemplo

```
val vertices: RDD[(VertexId, (Int, Int))] =
  sc.parallelize(Array(
      (1L, (3,-1)),
      (2L, (6,-1)),
      (3L, (2,-1)),
      (4L, (1,-1))
    )
  )
val relationships: RDD[Edge[Boolean]] =
  sc.parallelize(Array(
      Edge(1L, 2L, true),
      Edge(2L, 1L, true),
      Edge(2L, 4L, true),
      Edge(3L, 2L, true),
      Edge(3L, 4L, true),
      Edge(4L, 3L, true)
    )
  )
val graph = Graph(vertices, relationships)
```

Pregel

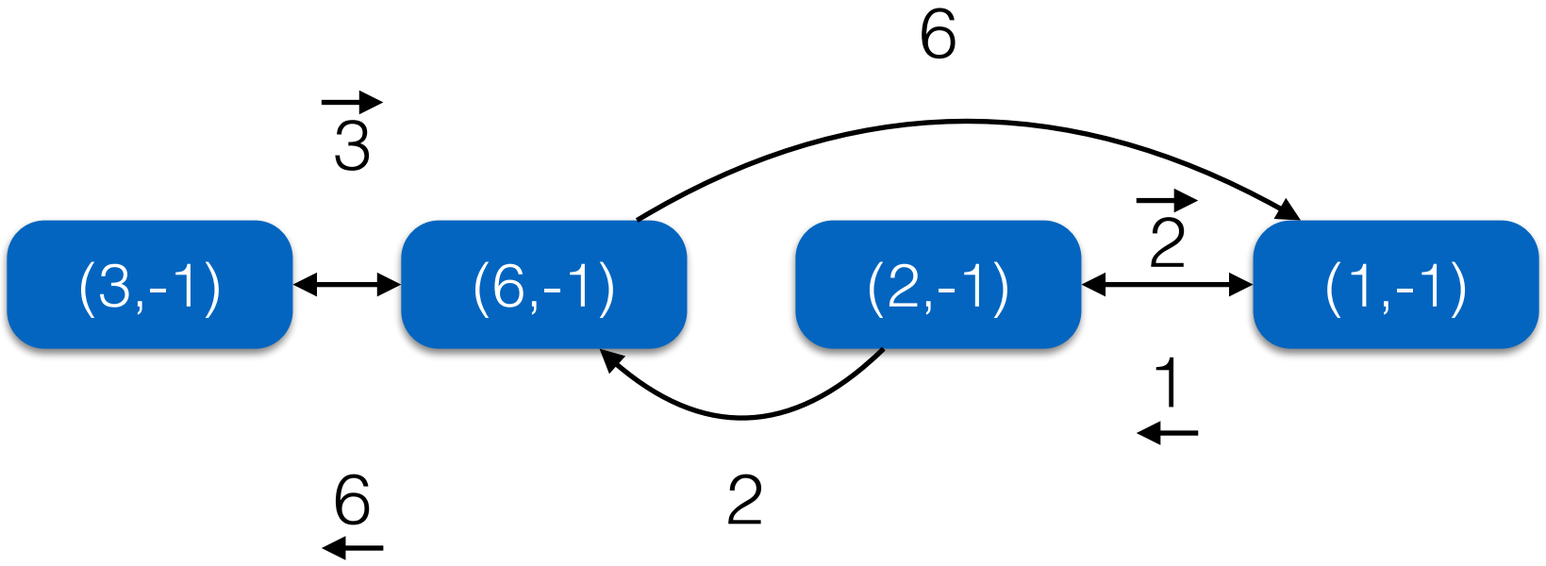
Ejemplo



Pregel

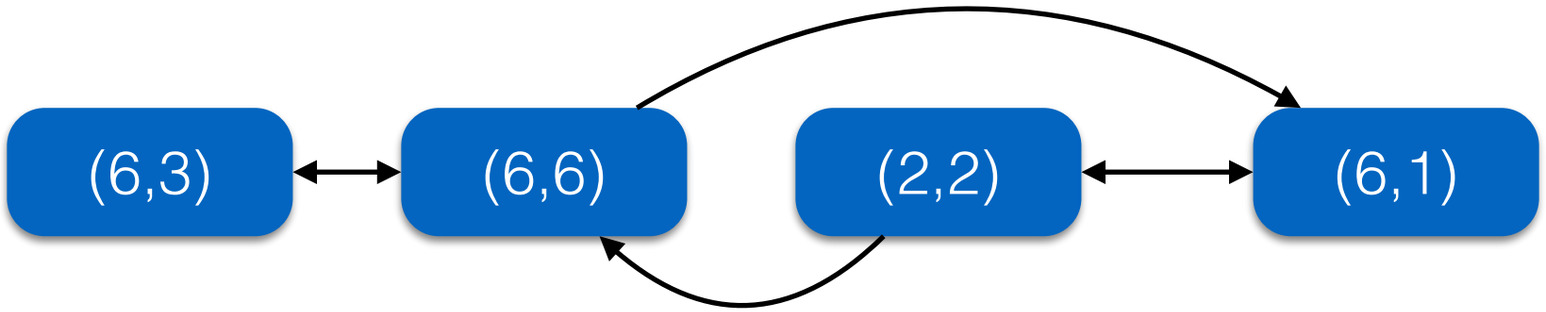
Ejemplo

En el *superstep* 0 enviamos los mensajes con los valores



Pregel

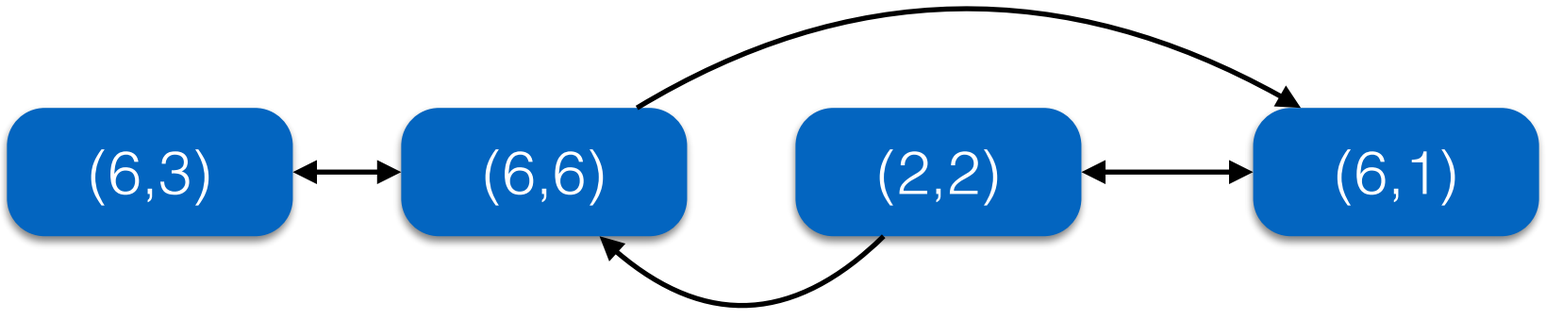
Ejemplo



Pregel

Ejemplo

En el *superstep* 0 enviamos los mensajes con los valores



Pregel

Ejemplo

Pregel

Ejemplo

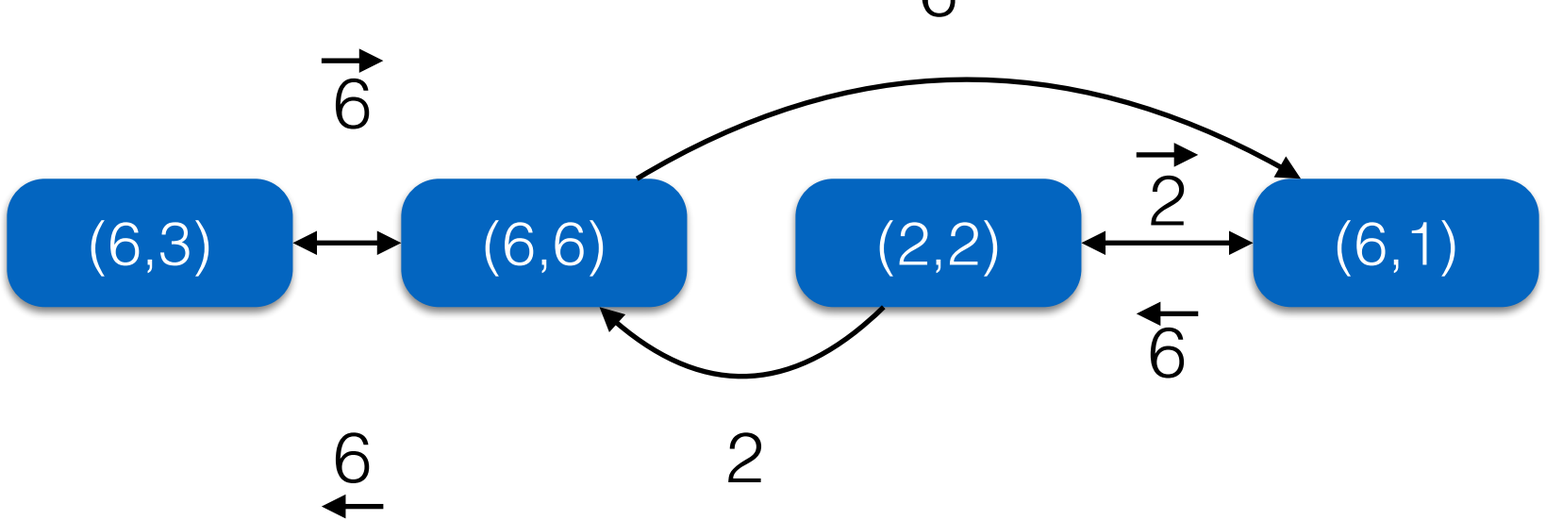
- En cada *superstep* siguiente los nodos activos envían un mensaje con su valor

- El valor nuevo del nodo corresponde al máximo entre su valor actual y todos los valores recibidos

- Cada vez que se recibe un mensaje se debe actualizar el segundo elemento de la tupla con el valor en la iteración anterior

Pregel

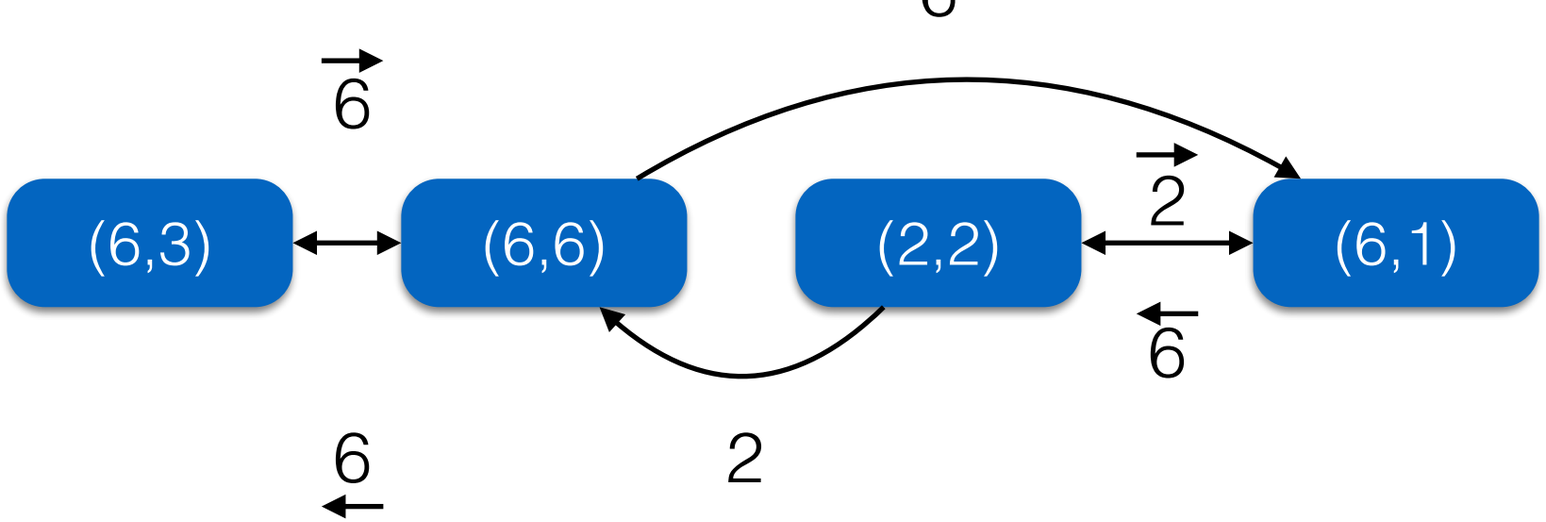
Ejemplo



Pregel

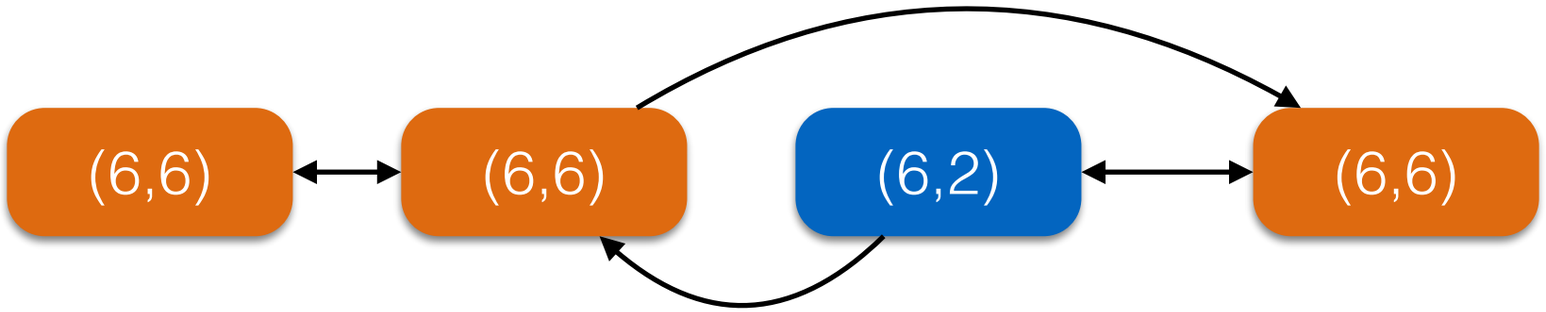
Ejemplo

En el *superstep* 1 se reciben los mensajes y actualizan los valores



Pregel

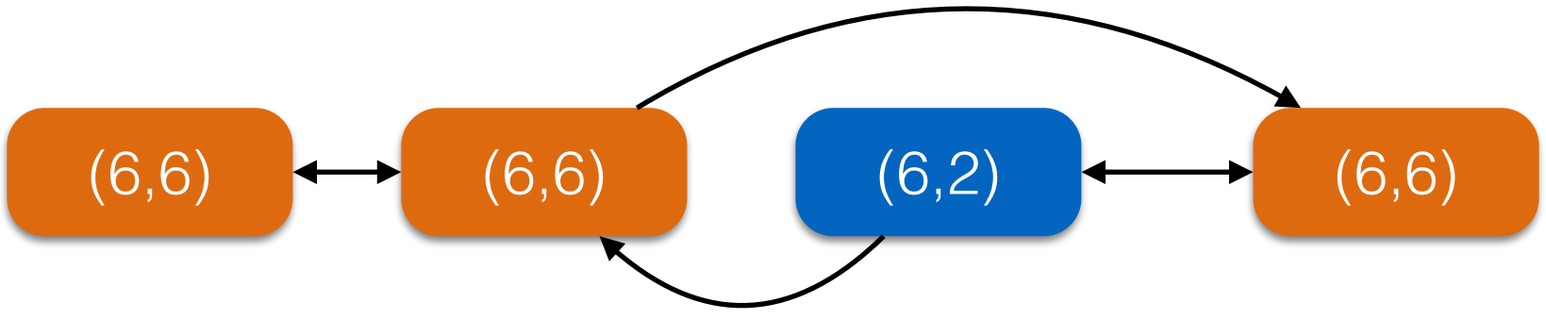
Ejemplo



Pregel

Ejemplo

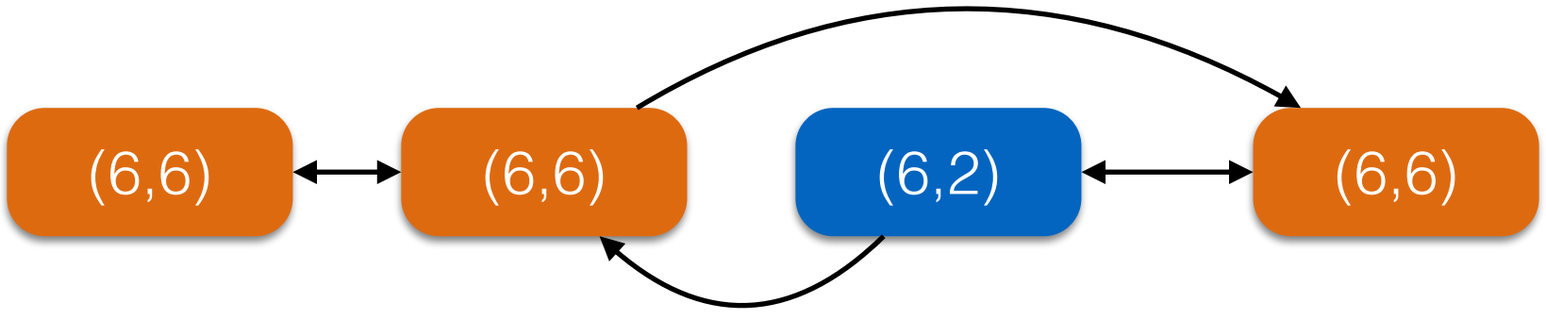
Votan por parar quienes no envían mensajes o no recibieron mensajes



Pregel

Ejemplo

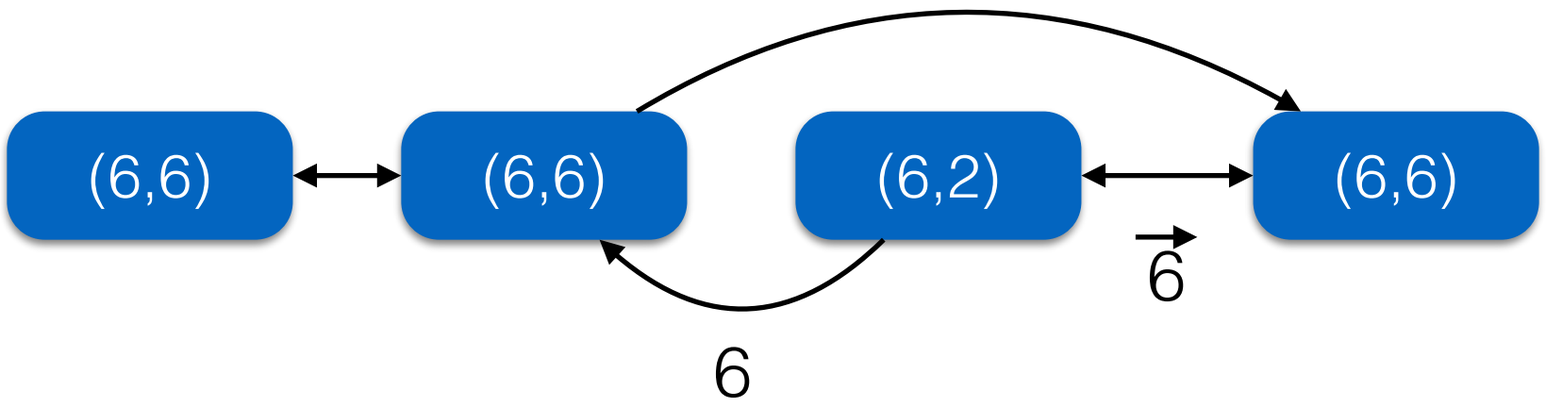
Votan por parar quienes no envían mensajes o no recibieron mensajes



No enviarán mensaje quienes hayan permanecido igual

Pregel

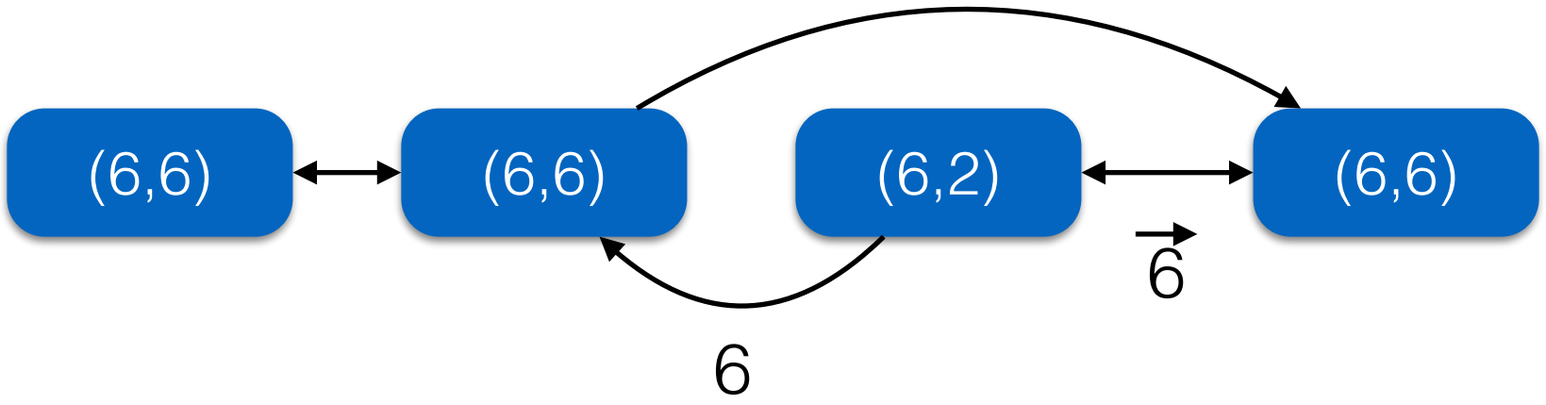
Ejemplo



Pregel

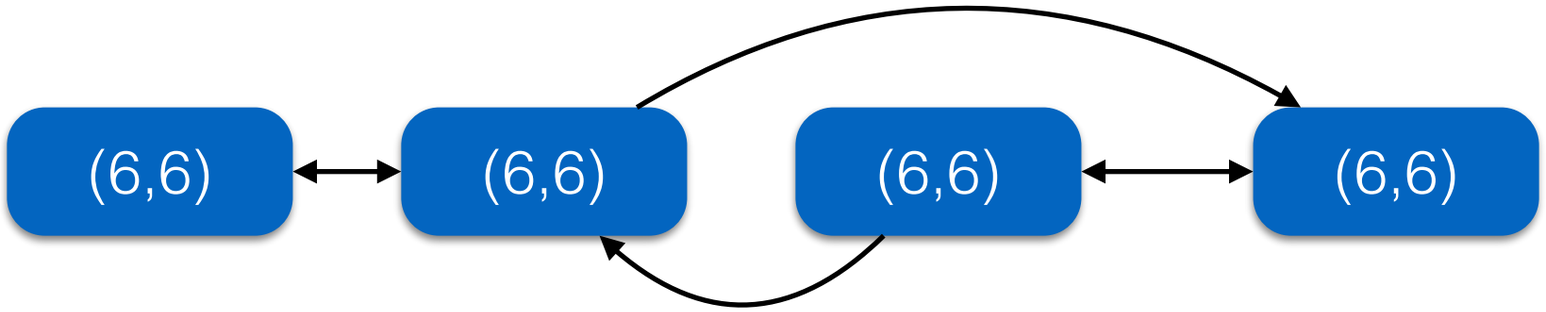
Ejemplo

En el *superstep 2* quienes cambiaron emiten un mensaje



Pregel

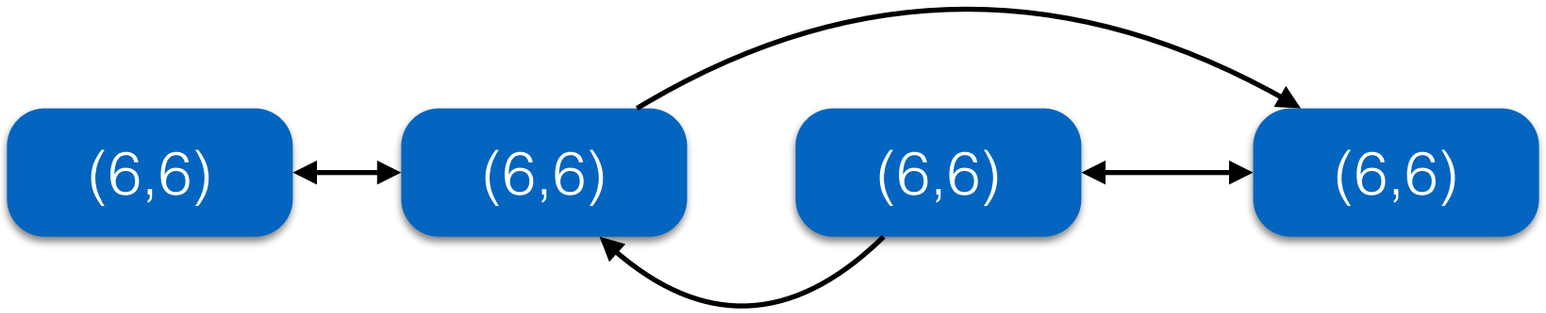
Ejemplo



Pregel

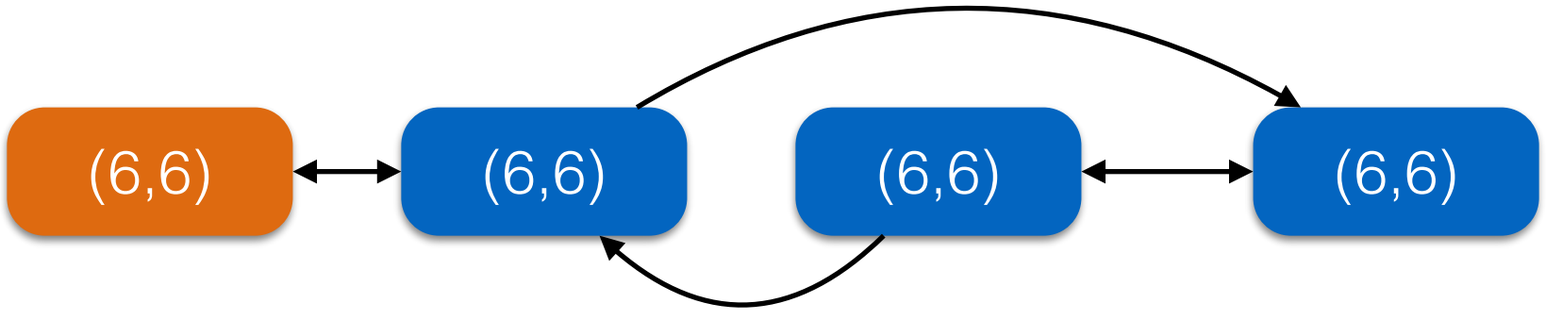
Ejemplo

En el *superstep 2* se actualizan los valores (si no recibió mensaje, entonces se deja el valor actual)



Pregel

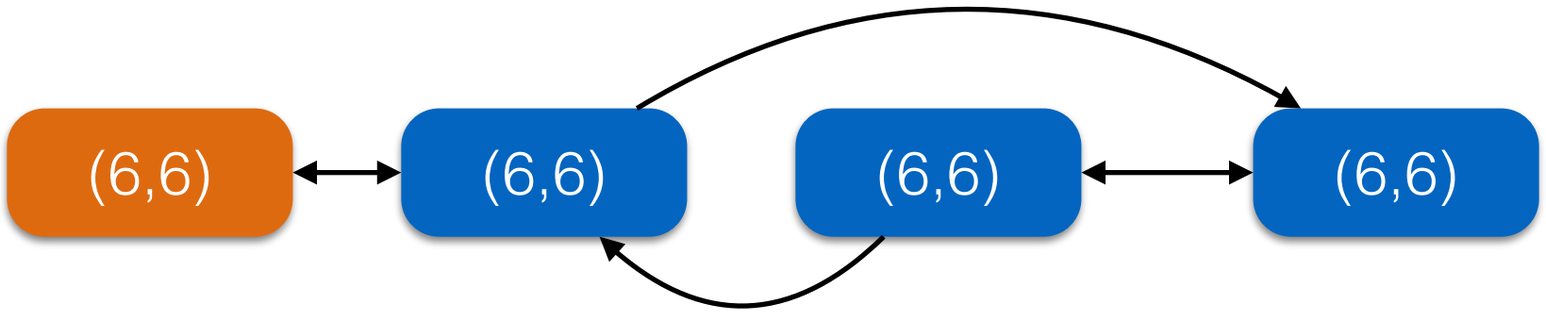
Ejemplo



Pregel

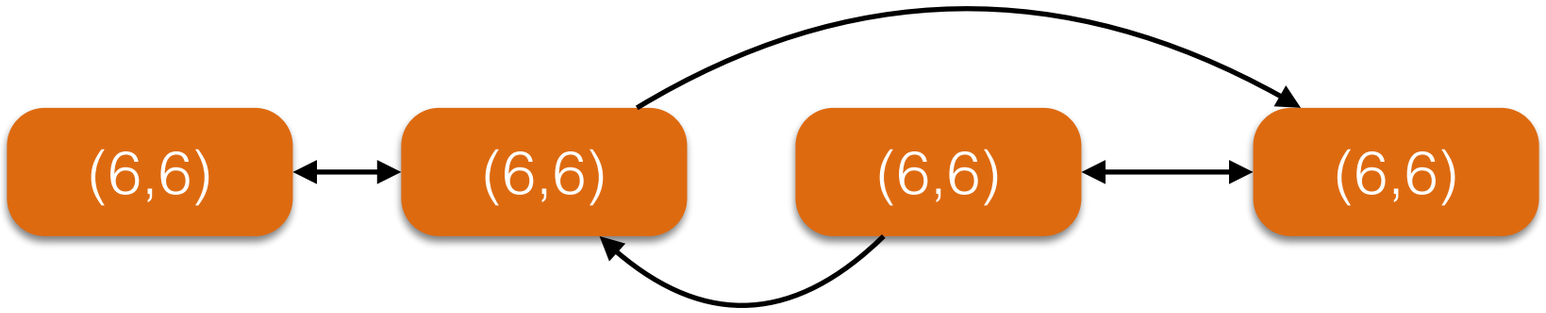
Ejemplo

Votan por parar quienes no envían mensajes ni recibieron mensajes



Pregel

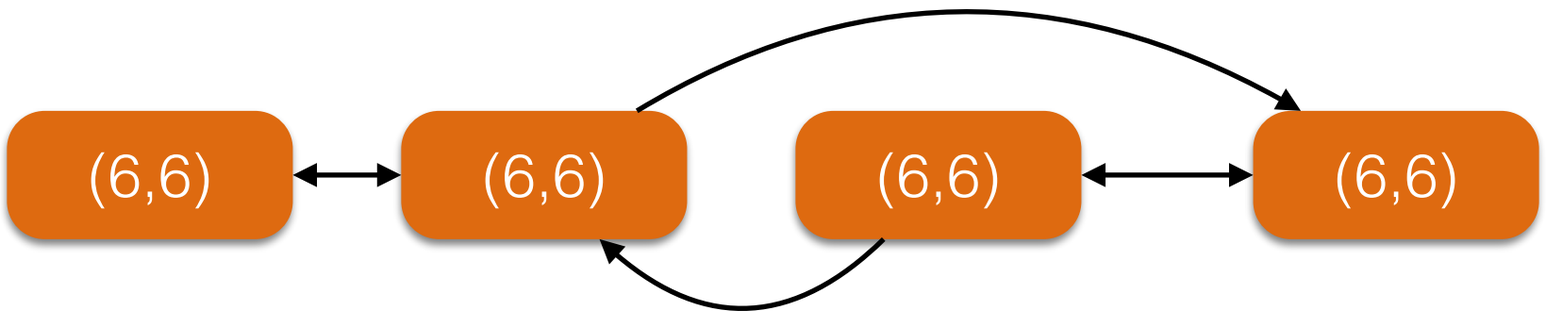
Ejemplo



Pregel

Ejemplo

Todos votan por parar y el programa termina



Pregel

Ejemplo

Pregel

Ejemplo

Para ocupar la API de Pregel necesitamos lo siguiente:

Pregel

Ejemplo

Para ocupar la API de Pregel necesitamos lo siguiente:

- `initialMsg`: mensaje que será enviado a todos los nodos al iniciar
- `maxIter`: número máximo de iteraciones
- `activeDir`: dirección hacia la que van los mensajes (entrate, saliente o bidireccional)

Pregel

Ejemplo

Pregel

Ejemplo

Para ocupar la API de Pregel necesitamos lo siguiente:

Pregel

Ejemplo

Para ocupar la API de Pregel necesitamos lo siguiente:

- `vprog`: función definida para la recepción de mensajes
- `sendMsg`: función definida para enviar mensajes
- `mergeMsg`: función definida en caso de que lleguen múltiples mensajes a un nodo en una iteración

Pregel

Ejemplo

Pregel

Ejemplo

En nuestro ejemplo:

Pregel

Ejemplo

En nuestro ejemplo:

- `initialMsg = 0` (suponemos valores mayores que 0 en el grafo)
- `maxIter = Int.MaxValue`
- `activeDir = EdgeDirection.Out`

Pregel

Ejemplo

Pregel

Ejemplo

```
def vprog(vertexId: VertexId, value: (Int, Int), message: Int): (Int,
Int) = {
  if (message == initialMsg) {
    value
  }
  else {
    (max(message, value._1), value._1)
  }
}
```

Pregel

Ejemplo

```
def vprog(vertexId: VertexId, value: (Int, Int), message: Int): (Int,
Int) = {
  if (message == initialMsg) {
    value
  }
  else {
    (max(message, value._1), value._1)
  }
}
```

Cada vez que recibo un mensaje: si es el inicial, los nodos permanecen igual (con valor `value`); en otro caso cambian según lo indicado en el `else`

Pregel

Ejemplo

Pregel

Ejemplo

```
def sendMsg(triplet: EdgeTriplet[(Int, Int), Boolean]):
Iterator[(VertexId, Int)] = {
  val sourceVertex = triplet.srcAttr
  if (sourceVertex._1 == sourceVertex._2) {
    Iterator.empty
  }
  else {
    Iterator((triplet.dstId, sourceVertex._1))
  }
}
```

Pregel

Ejemplo

```
def sendMsg(triplet: EdgeTriplet[(Int, Int), Boolean]):
Iterator[(VertexId, Int)] = {
  val sourceVertex = triplet.srcAttr
  if (sourceVertex._1 == sourceVertex._2) {
    Iterator.empty
  }
  else {
    Iterator((triplet.dstId, sourceVertex._1))
  }
}
```

Si en alguna iteración el `srcAttr` los valores anterior y actuales son iguales no envío nada, sino envío mi valor.

Pregel

Ejemplo

Pregel

Ejemplo

```
def mergeMsg(msg1: Int, msg2: Int): Int = max(msg1, msg2)
```

Pregel

Ejemplo

```
def mergeMsg(msg1: Int, msg2: Int): Int = max(msg1, msg2)
```

La función **`mergeMsg`** acepta el máximo de los valores (notar que funciona como un reduce)

Pregel

Ejemplo

Pregel

Ejemplo

Finalmente llamamos la función:

Pregel

Ejemplo

Finalmente llamamos la función:

```
val maxGraph = graph.pregel(initialMsg,
  Int.MaxValue,
  EdgeDirection.Out)(
  vprog,
  sendMsg,
  mergeMsg)

maxGraph.vertices.collect.foreach(v => println(s"${v}"))
```

Pregel

PageRank

Pregel

PageRank

Veamos una implementación de PageRank en Pregel

Pregel

PageRank

Veamos una implementación de PageRank en Pregel

```
val vertexArray = Array(
  (1L, ("Alicia", 52)),
  (2L, ("Bob", 17)),
  (3L, ("Carlos", 65)),
  (4L, ("David", 24)),
  (5L, ("Eduardo", 42))
)
val edgeArray = Array(
  Edge(1L, 2L, "follows"),
  Edge(2L, 1L, "follows"),
  Edge(1L, 4L, "follows"),
  Edge(2L, 4L, "follows"),
  Edge(5L, 4L, "follows"),
  Edge(3L, 2L, "follows"),
  Edge(2L, 3L, "follows")
)
```

Pregel

PageRank

Pregel

PageRank

```
val conf = new SparkConf().setAppName("TestApp").setMaster("local")
val sc = new SparkContext(conf)

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

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

Pregel

PageRank

Pregel

PageRank

Creamos nuestro grafo inicial:

Pregel

PageRank

Creamos nuestro grafo inicial:

```
val initialGraph: Graph[Double, Double] = graph.
  outerJoinVertices(graph.outDegrees) {
    (vid, vdata, deg) => deg.getOrElse(0)
  }.
  mapTriplets(e => 1.0 / e.srcAttr).
  mapVertices((id, attr) => 1.0)

initialGraph.triplets.take(10).foreach(t => println(s"${t}"))
```

Pregel

PageRank

Pregel

PageRank

Cuyo resultado es (formato `(vértice origen, vértice destino, peso de la arista)`):

Pregel

PageRank

Cuyo resultado es (formato `(vértice origen, vértice destino, peso de la arista)`):

```
((1,1.0),(2,1.0),0.5)
((1,1.0),(4,1.0),0.5)
((2,1.0),(1,1.0),0.3333333333333333)
((2,1.0),(3,1.0),0.3333333333333333)
((2,1.0),(4,1.0),0.3333333333333333)
((3,1.0),(2,1.0),1.0)
((5,1.0),(4,1.0),1.0)
```

Pregel

PageRank

Pregel

PageRank

Definimos las funciones y variables necesarias:

Pregel

PageRank

Definimos las funciones y variables necesarias:

```
val resetProb = 0.15
val initialMsg = 0.0
val numIter = 20

def vprog(id: VertexId, attr: Double, msgSum: Double): Double = {
  resetProb + (1.0 - resetProb) * msgSum
}

def sendMsg(edge: EdgeTriplet[Double, Double]): Iterator[(VertexId,
Double)] = {
  Iterator((edge.dstId, edge.srcAttr * edge.attr))
}

def mergeMsg(a: Double, b: Double): Double = a + b
```

Pregel

PageRank

Pregel

PageRank

Llamamos a la función Pregel e imprimimos el resultado:

Pregel

PageRank

Llamamos a la función Pregel e imprimimos el resultado:

```
val pagerankGraph = initialGraph.pregel(initialMsg, numIter,
EdgeDirection.Either)(
  vprog, sendMsg, mergeMsg)

pagerankGraph.triplets.collect().foreach(t => println(s"${t}"))
```

Pregel

PageRank

Pregel

PageRank

El resultado es:

Pregel

PageRank

El resultado es:

```
((1,0.3004385864006998),(2,0.5318825654786997),0.5)
((1,0.3004385864006998),(4,0.5552327748935998),0.5)
((2,0.5318825654786997),(1,0.3004385864006998),0.3333333333333333)
((2,0.5318825654786997),(3,0.3004385864006998),0.3333333333333333)
((2,0.5318825654786997),(4,0.5552327748935998),0.3333333333333333)
((3,0.3004385864006998),(2,0.5318825654786997),1.0)
((5,0.15),(4,0.5552327748935998),1.0)
```

Pregel

PageRank

Pregel

PageRank

Notemos que existe una función PageRank ya implementada:

Pregel

PageRank

Notemos que existe una función PageRank ya implementada:

```
val errorTolerance = 0.0001
val pagerankGraph2 = graph.pageRank(errorTolerance).
  triplets.collect().foreach(t => println(s"${t}"))
```

Técnicas para Big Data

Clase 11 - GraphX