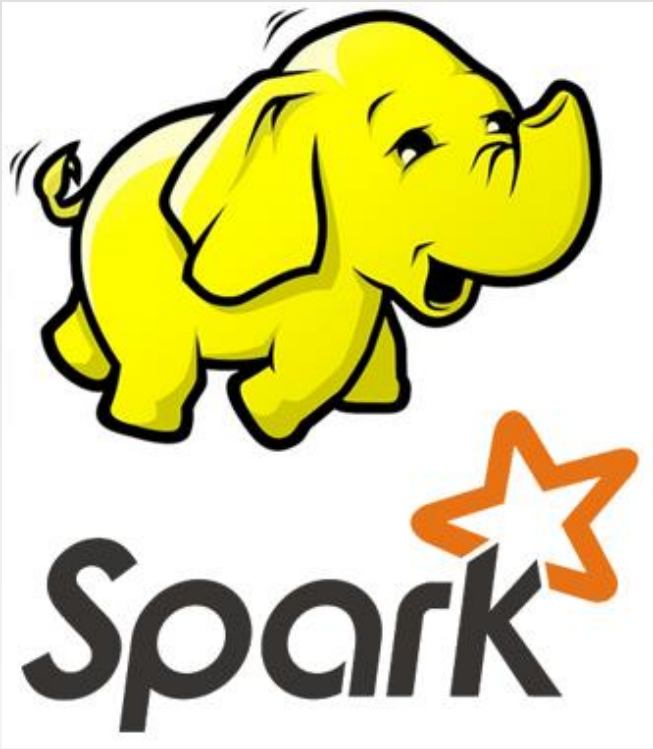
COMP9313: Big Data Management



Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

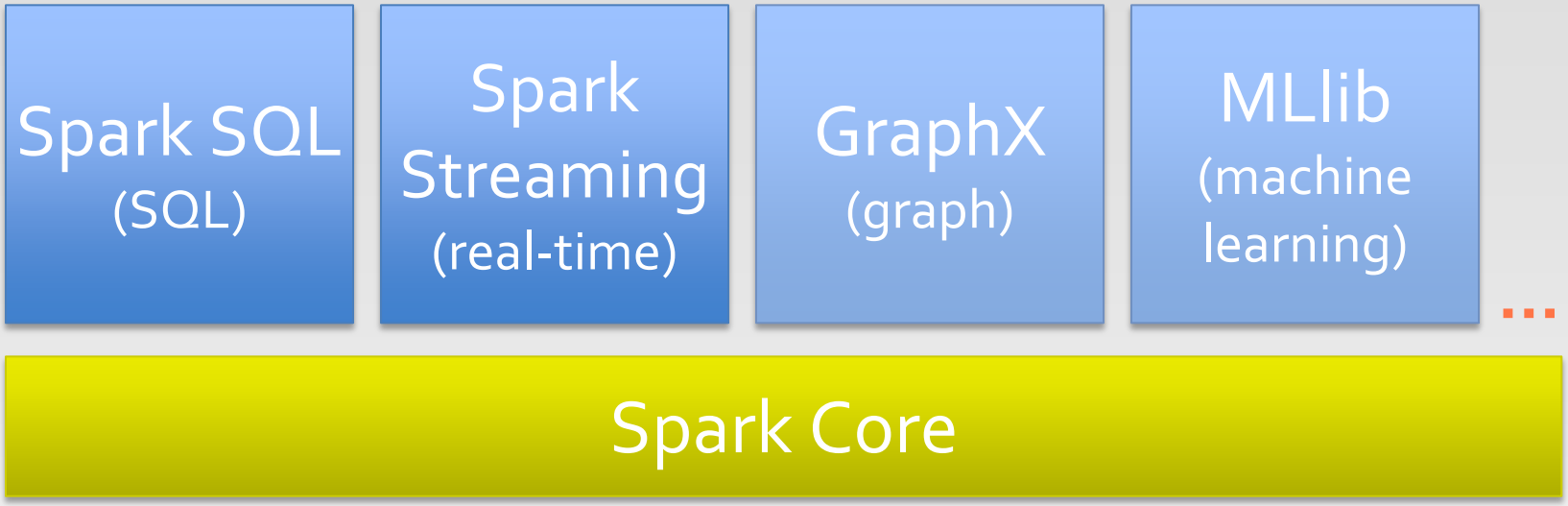
Chapter 6: Spark II

Review of Chapter 5

Spark Ideas

- Expressive computing system, not limited to map-reduce model
- Facilitate system memory
  - avoid saving intermediate results to disk
  - cache data for repetitive queries (e.g. for machine learning)
- Layer an in-memory system on top of Hadoop.
- Achieve fault-tolerance by re-execution instead of replication

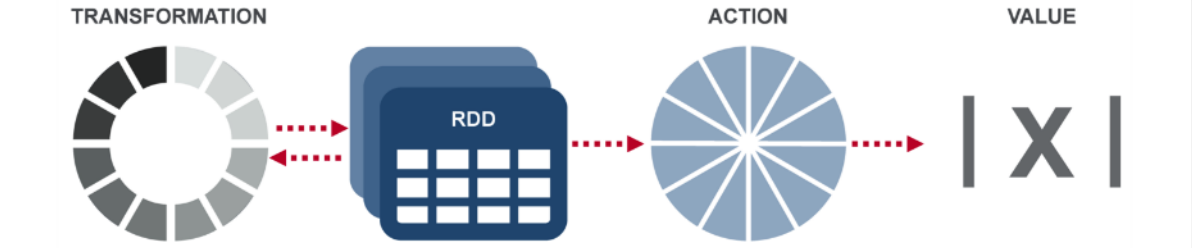
Spark Components



- Spark SQL(SQL on Spark)
- Spark Streaming (stream processing)
- GraphX (graph processing)
- MLlib (machine learning library)

RDD

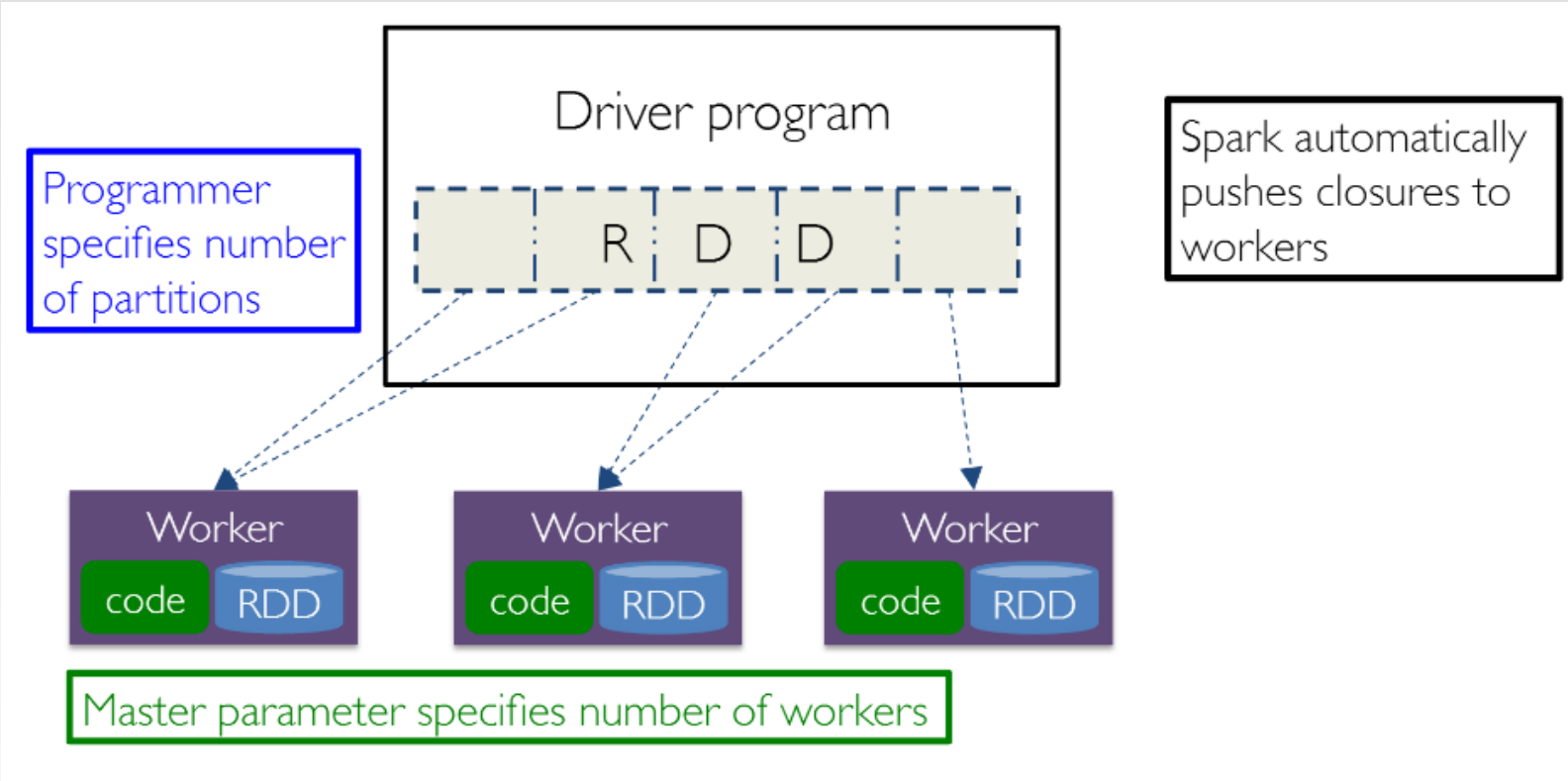
- Resilient Distributed Datasets (RDDs):
  - A **distributed** memory abstraction that lets programmers perform **in-memory** computations on large clusters in a **fault-tolerant** manner.
- RDD operations:
  - **Transformation:** returns a new RDD
  - **Action:** evaluates and returns a new value



Partition

- The data inside a RDD is partitioned (split into partitions) and then distributed across nodes in a cluster.
  - val inputfile = sc.textFile("...", 4)
- Typically you want 2-4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster.
  - So if you have a cluster with 50 cores, you want your RDDs to at least have 50 partitions (and probably 2-4x times that).
- The maximum size of a partition is ultimately limited by the available memory of an executor.
- Smaller/more numerous partitions allow work to be distributed among more workers, but larger/fewer partitions allow work to be done in larger chunks, which may result in the work getting done more quickly as long as all workers are kept busy, due to reduced overhead.

Summary


Driver program
R D D
Programmer specifies number of partitions
Spark automatically pushes closures to workers
Worker
code
RDD
Worker
code
RDD
Worker
code
RDD
Master parameter specifies number of workers

Part 1: Shared Variables

Using Local Variables

- Any external variables you use in a closure will automatically be shipped to the cluster:

```
query = sys.stdin.readline()
pages.filter(x => x.contains(query)).count()
```

- Some caveats:
  - Each task gets a new copy (updates aren't sent back)
  - Variable must be Serializable

Shared Variables

- When you perform transformations and actions that use functions (e.g., map(f: T=>U)), Spark will automatically push a closure containing that function to the workers so that it can run at the workers.

- Any variable or data within a closure or data structure will be distributed to the worker nodes along with the closure

- When a function (such as map or reduce) is executed on a cluster node, it works on **separate** copies of all the variables used in it.

- Usually these variables are just constants but they cannot be shared across workers efficiently.

Shared Variables

- Consider These Use Cases
  - Iterative or single jobs with large global variables
    - Sending large read-only lookup table to workers
    - Sending large feature vector in a ML algorithm to workers
    - Problems? Inefficient to send large data to each worker with each iteration
    - Solution: Broadcast variables
  - Counting events that occur during job execution
    - How many input lines were blank?
    - How many input records were corrupt?
    - Problems? Closures are one way: driver -> worker
    - Solution: Accumulators

Broadcast Variables

- Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.
  - For example, to give every node a copy of a large input dataset efficiently
- Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost
- Broadcast variables are created from a variable **v** by calling **SparkContext.broadcast(v)**. Its value can be accessed by calling the **value** method.

```
scala > val broadcastVar =sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
scala > broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

- The broadcast variable should be used instead of the value **v** in any functions run on the cluster, so that **v** is not shipped to the nodes more than once.

Accumulators

- Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.
- They can be used to implement counters (as in MapReduce) or sums.
- Spark natively supports accumulators of numeric types, and programmers can add support for new types.
- Only driver can read an accumulator’s value, not tasks
- An accumulator is created from an initial value **v** by calling **SparkContext.accumulator(v)**.

```
scala> val accum = sc.longAccumulator("My Accumulator")
accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some(My Accumulator), value: 0)
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
... 10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
scala> accum.value
res2: Long = 10
```

Accumulators Example (Python)

- Counting empty lines

```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
        global blankLines # Make the global variable accessible
        if (line == ""):
                blankLines += 1
        return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
print "Blank lines: %d" % blankLines.value
```

  - blankLines is created in the driver, and shared among workers
  - Each worker can access this variable

Does this counter work?

- One of the harder things about Spark is understanding the scope and life cycle of variables and methods when executing code across a cluster.

```
var counter = 0
var rdd = sc.parallelize(data)

rdd.foreach(x => counter += x)

println("Counter value: " + counter)
```

  - The behavior of the above code is undefined, and may not work as intended.
  - Spark sends the closure to each task containing variables must be visible to the executors. Thus “counter” in the executor is only a copy of the “counter” in the driver!

Part 2: Self-Contained Applications

WordCount (Scala)

- Standalone code

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

object WordCount {
    def main(args: Array[String]) {
      val inputFile = args(0)
      val outputFolder = args(1)
      val conf = new SparkConf().setAppName("wordCount").setMaster("local")
      // Create a Scala Spark Context.
      val sc = new SparkContext(conf)
      // Load our input data.
      val input =  sc.textFile(inputFile)
      // Split up into words.
      val words = input.flatMap(line => line.split(" "))
      // Transform into word and count.
      val counts = words.map(word => (word, 1)).reduceByKey(_+_)
      counts.saveAsTextFile(outputFolder)
    }
}
```

WordCount (Scala)

- Linking with Apache Spark
  - The first step is to explicitly import the required spark classes into your Spark program

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

- Initializing Spark
  - Create a Spark context object with the desired spark configuration that tells Apache Spark on how to access a cluster

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

  - SparkConf: Spark configuration class
  - setAppName: set the name for your application
  - setMaster: set the cluster master URL

setMaster

- Set the cluster master URL to connect to
- Parameters for setMaster:
  - local(default) - run locally with only one worker thread (no parallel)
  - local[k] - run locally with k worker threads
  - spark://HOST:PORT - connect to Spark standalone cluster URL
  - mesos://HOST:PORT - connect to Mesos cluster URL
  - yarn - connect to Yarn cluster URL
    - Specified in SPARK_HOME/conf/yarn-site.xml
- setMaster parameters configurations:
  - In source code
    - SparkConf().setAppName("wordCount").setMaster("local")
  - spark-submit
    - spark-submit --master local
  - In SPARK_HOME/conf/spark-default.conf
    - Set value for spark.master

WordCount (Scala)

- Creating a Spark RDD
  - Create an input Spark RDD that reads the text file input.txt using the Spark Context created in the previous step

```
val input =  sc.textFile(inputFile)
```

- Spark RDD Transformations in Wordcount Example
  - flatMap() is used to tokenize the lines from input text file into words
  - map() method counts the frequency of each word
  - reduceByKey() method counts the repetitions of word in the text file
- Save the results to disk

```
counts.saveAsTextFile(outputFolder)
```

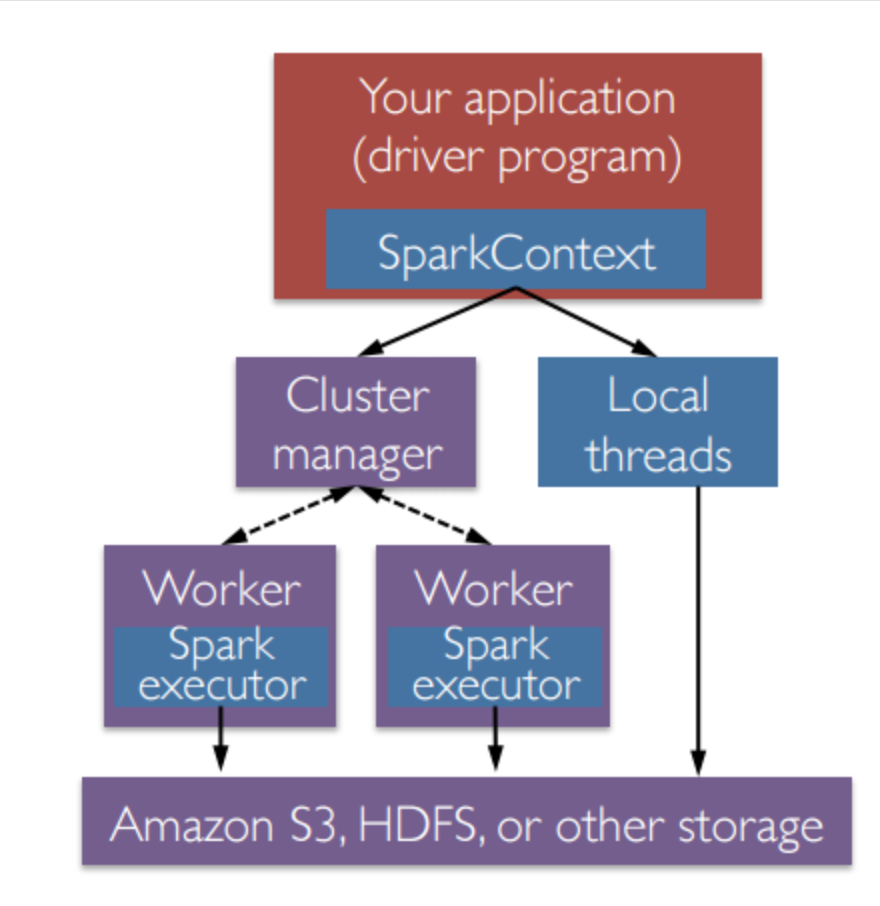
Passing Functions to RDD

- Spark's API relies heavily on passing functions in the driver program to run on the cluster.
  - Anonymous function. E.g.,
    - val words = input.flatMap(line => line.split(" "))
  - Static methods in a global singleton object. E.g,
    - object MyFunctions { def func1(s: String): String = { ... } }

      myRdd.map(MyFunctions.func1)

Run the Application on a Cluster

- A Spark application is launched on a set of machines using an external service called a cluster manager
  - Local threads
  - Standalone
  - Mesos
  - Yarn
- Driver
- Executor



Launching a Program

- Spark provides a single script you can use to submit your program to it called spark-submit
  - The user submits an application using spark-submit
  - spark-submit launches the driver program and invokes the main() method specified by the user
  - The driver program contacts the cluster manager to ask for resources to launch executors
  - The cluster manager launches executors on behalf of the driver program
  - The driver process runs through the user application. Based on the RDD actions and transformations in the program, the driver sends work to executors in the form of tasks
  - Tasks are run on executor processes to compute and save results
  - If the driver's main() method exits or it calls SparkContext.stop(), it will terminate the executors and release resources from the cluster manager

Package Your Code and Dependencies

- Ensure that all your dependencies are present at the runtime of your Spark application
- Java Application (Maven)
- Scala Application (sbt)
  - a newer build tool most often used for Scala projects

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.3.1"
```

  - libraryDependencies: list all dependent libraries (including third party libraries)
  - A jar file simple-project_2.11-1.0.jar will be created after compilation

Deploying Applications in Spark

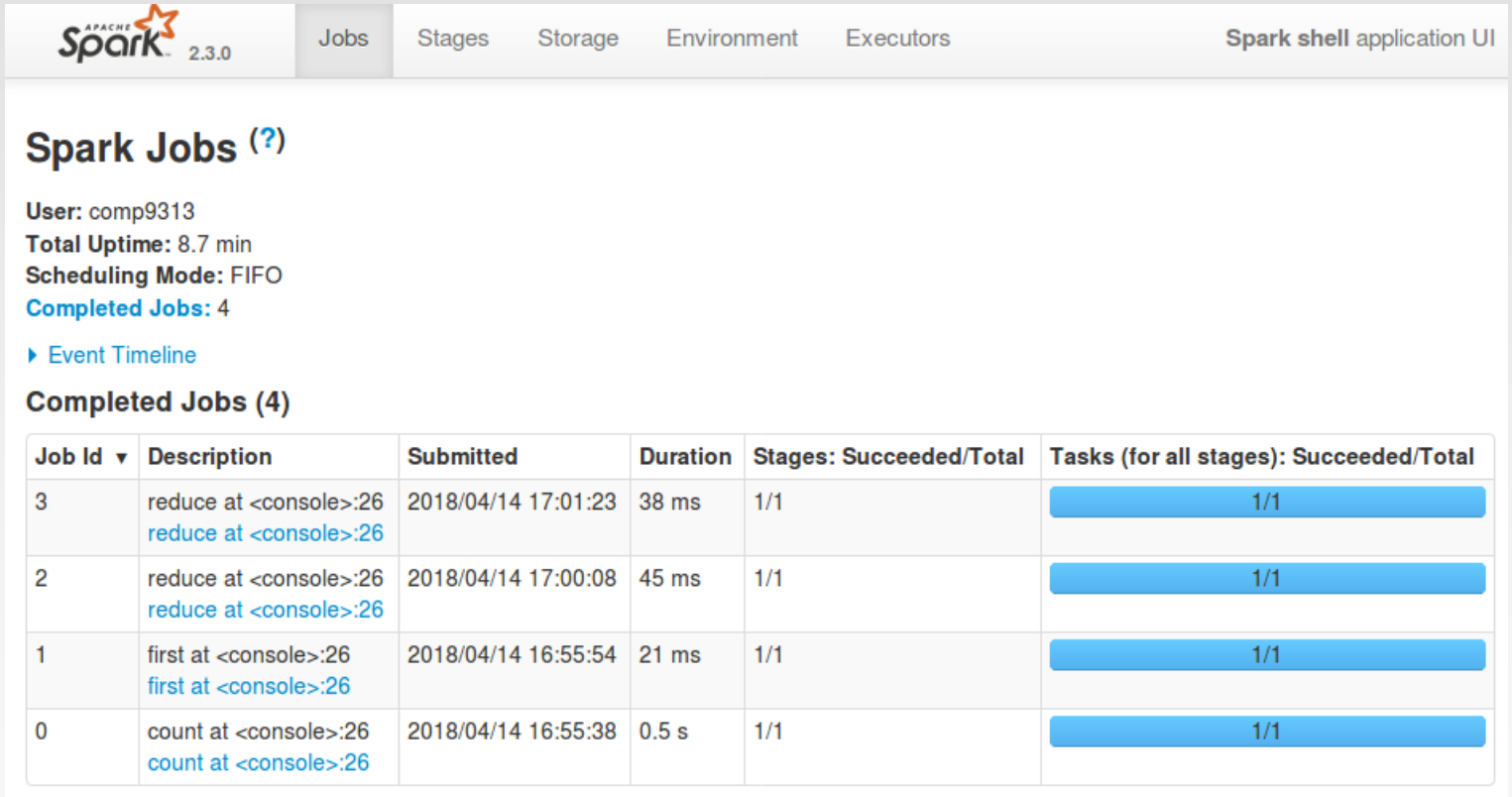
- spark-submit

Common flags	Explanation
--master	Indicates the cluster manager to connect to
--class	The “main” class of your application if you’re running a Java or Scala program
--name	A human-readable name for your application. This will be displayed in Spark’s web UI.
--executor-memory	The amount of memory to use for executors, in bytes. Suffixes can be used to specify larger quantities such as “512m” (512 megabytes) or “15g” (15 gigabytes)
--driver-memory	The amount of memory to use for the driver process, in bytes.

- spark-submit --master spark://hostname:7077 \
  --class YOURCLASS \
  --executor-memory 2g \
  YOURJAR "options" "to your application" "go here"

Spark Web Console

- You can browse the web interface for the information of Spark Jobs, storage, etc. at: **http://localhost:4040**



In-Memory Can Make a Big Difference

- Two iterative Machine Learning algorithms:

Spark Core Programming Practice

Output?

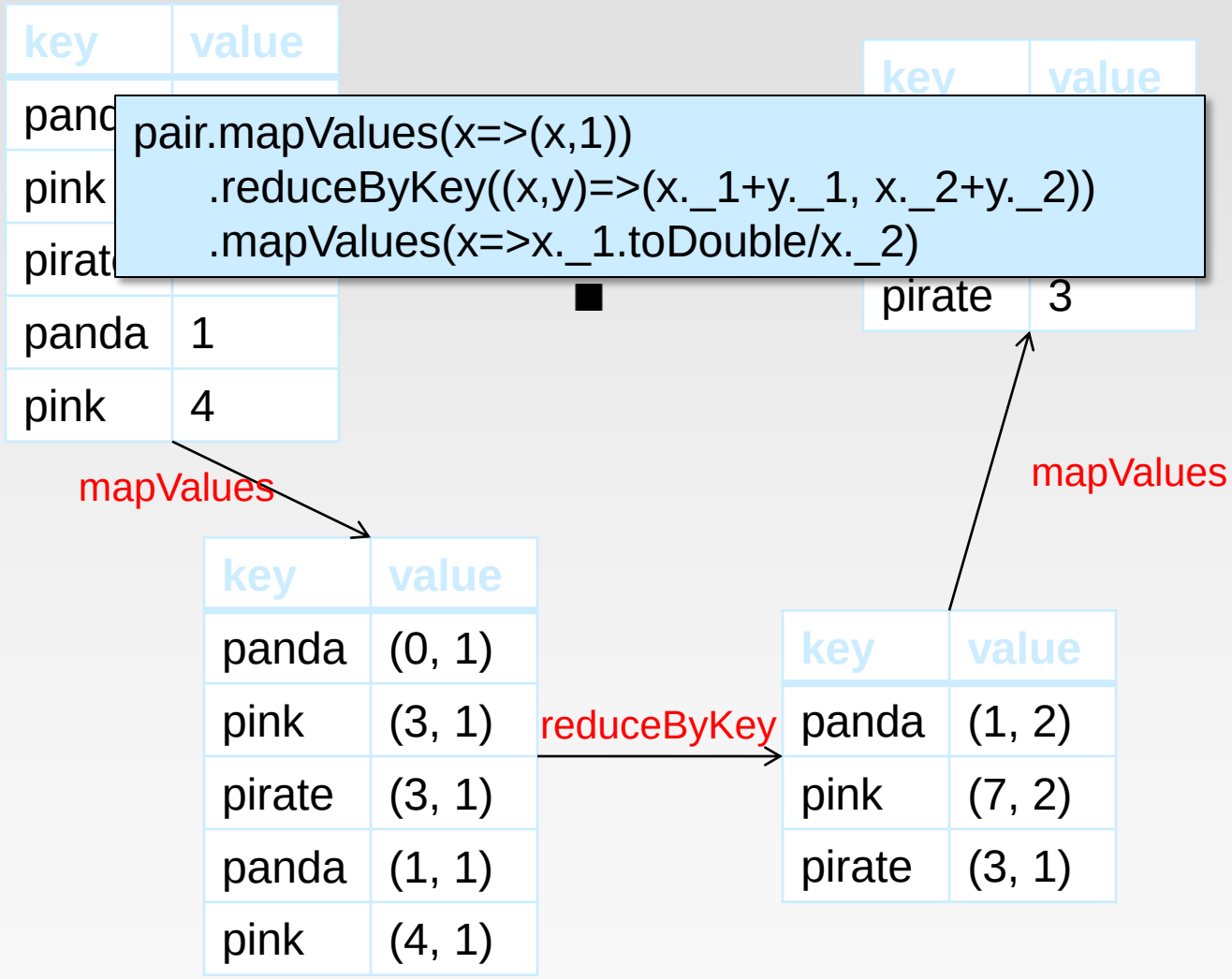
```
val lines = sc.parallelize(List("hello world", "this is a scala program", "to create a pair RDD", "in spark"))
val pairs = lines.map(x => (x.split(" ")(0), x))
pairs.filter {case (key, value) => key.length <3}.foreach(println)
```

```
val pairs = sc.parallelize(List((1, 2), (3, 1), (3, 6), (4,2)))
val pairs1 = pairs.mapValues(x=>(x, 1))
val pairs2 = pairs1.reduceByKey((x,y) => (x._1 + y._1, x._2+y._2))
pairs2.foreach(println)
```

```
val pairs = sc.parallelize(List((1, 2), (3, 4), (3, 9), (4,2)))
val pairs1 = pairs.mapValues(x=>(x, 1)).reduceByKey((x,y) => (x._1 + y._1,
x._2+y._2)).mapValues(x=>x._2/x._1)
pairs1.foreach(println)
```

Practice

- Problem 1: Given a pair RDD of type [(String, Int)], compute the per-key average



Practice

- Problem 2: Given the data in format of key-value pairs <Int, Int>, find the maximum value for each key across all values associated with that key.

```
val pairs = sc. Parallelize(List((1, 2), (3, 4),… …))

val resMax = pairs.reduceByKey( (a, b) => if(a > b) a else b )

resMax.foreach(x => println(x._1, x._2))
```

Practice

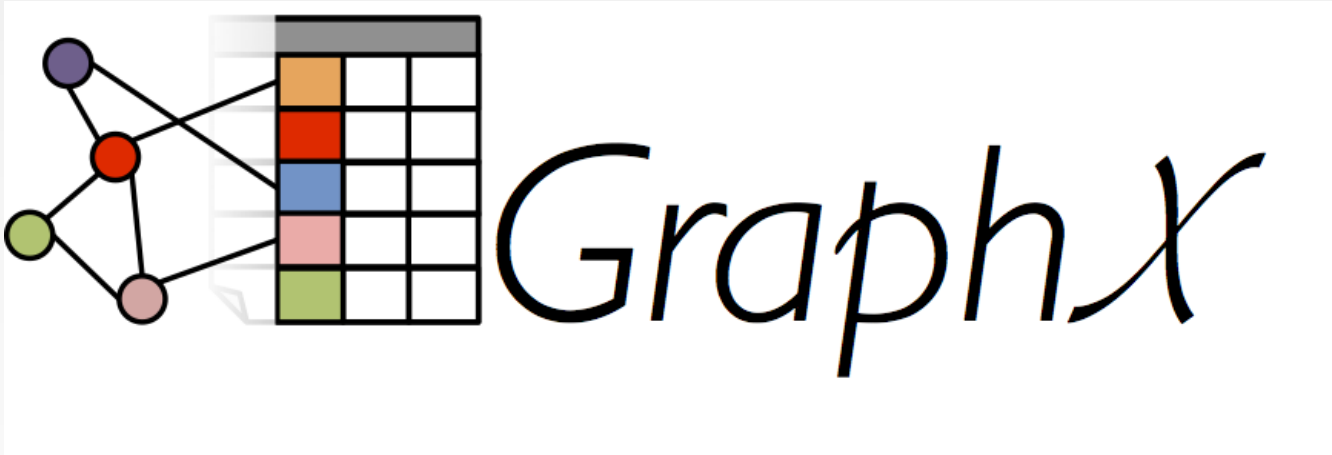
- Problem 3: Given a collection of documents, compute the average length of words starting with each letter.

```
val textFile = sc.textFile(inputFile)
val words = textFile.flatMap(_.split(“ “).toLowerCase)

val counts = words.filter(x=> x.length >=1 && x.charAt(0)<=’z’ &&
                x.charAt(0)>=’a’).map(x=>(x.charAt(0), (x.length, 1)))

val avgLen = counts.reduceByKey((a, b)=>(a._1+b._1, a._2+b._2)).foreach(x=>(x._1,
                x._2._1.toDouble/x._2._2))

avgLen.foreach(x => println(x._1, x._2))
```

Part 3: Spark GraphX

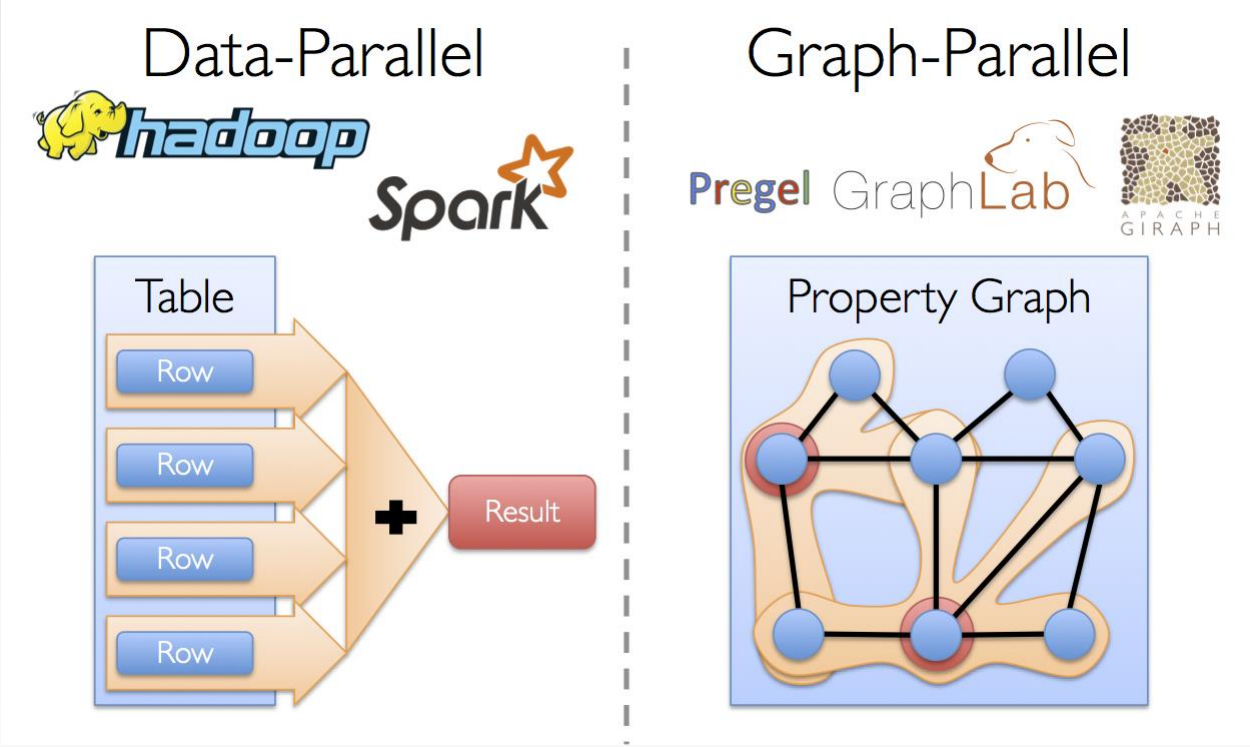
Spark GraphX

- **GraphX** is Apache Spark's API for graphs and graph-parallel computation.
- At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge
- To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices) as well as an optimized variant of the Pregel API
- GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.



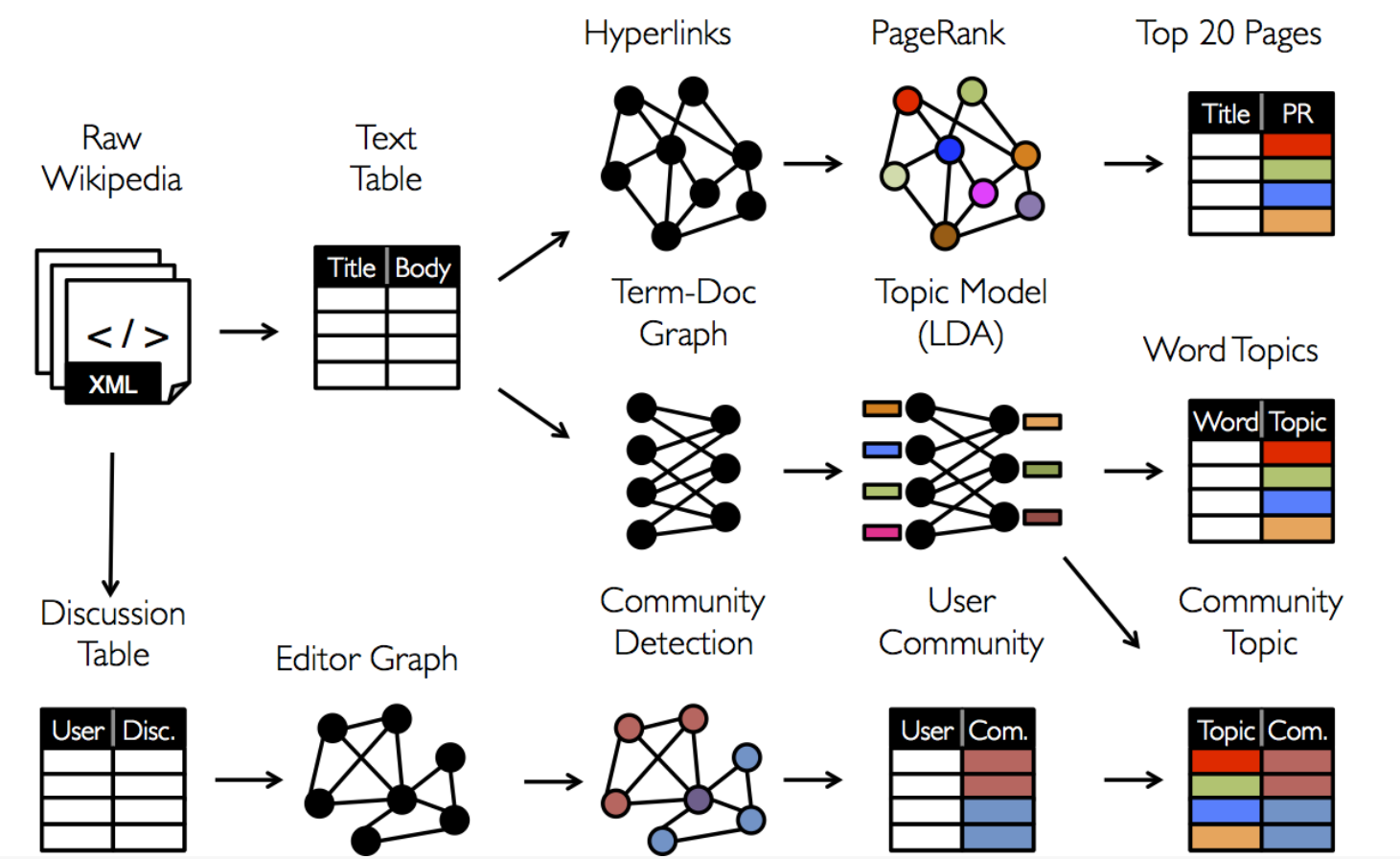
Graph-Parallel Computation

- The growing scale and importance of graph data has driven the development of numerous new graph-parallel systems (e.g., Giraph and GraphLab)
- These systems can efficiently execute sophisticated graph algorithms orders of magnitude faster than more general *data-parallel* systems.
  - Expose specialized APIs to simplify graph programming



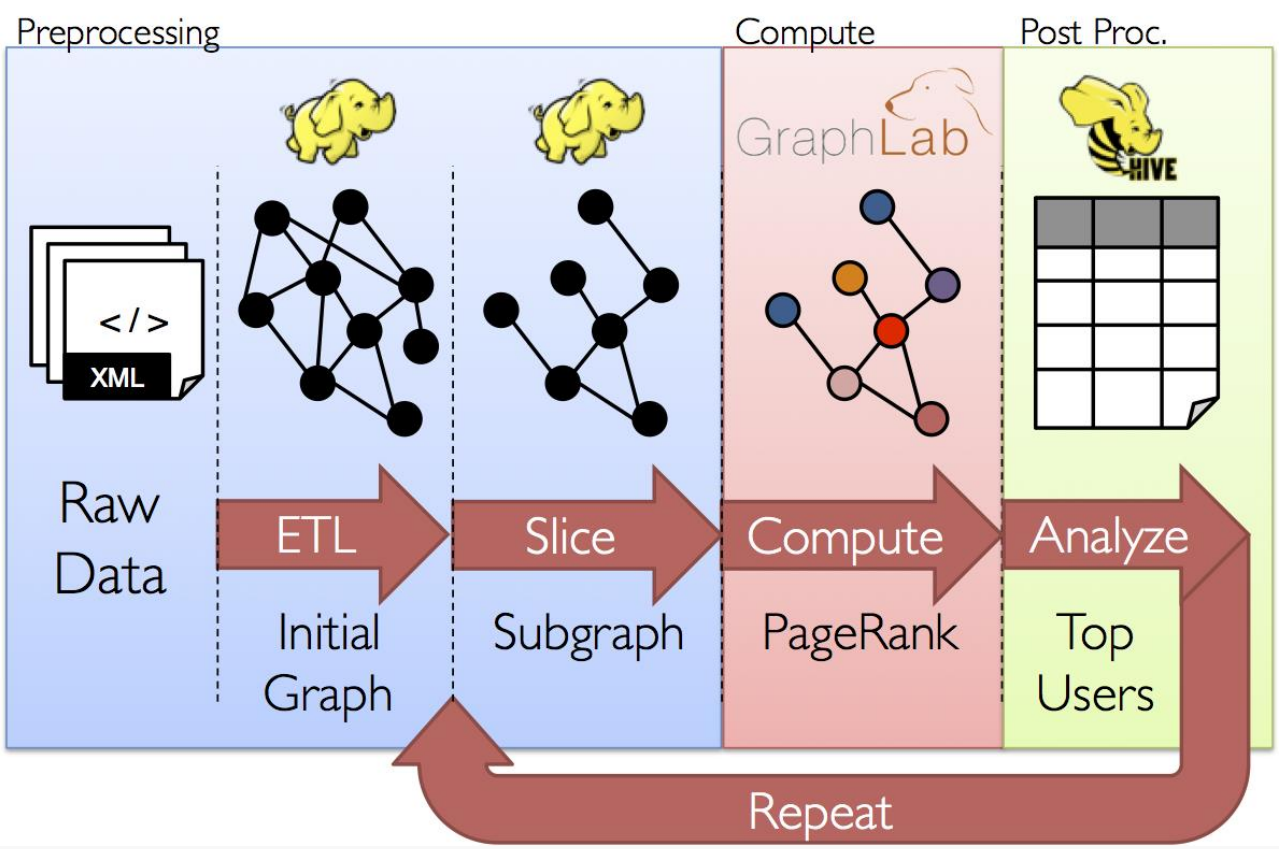
Specialized Systems May Miss the Bigger Picture

- It is often desirable to be able to move between table and graph views of the same physical data and to leverage the properties of each view to easily and efficiently express computation



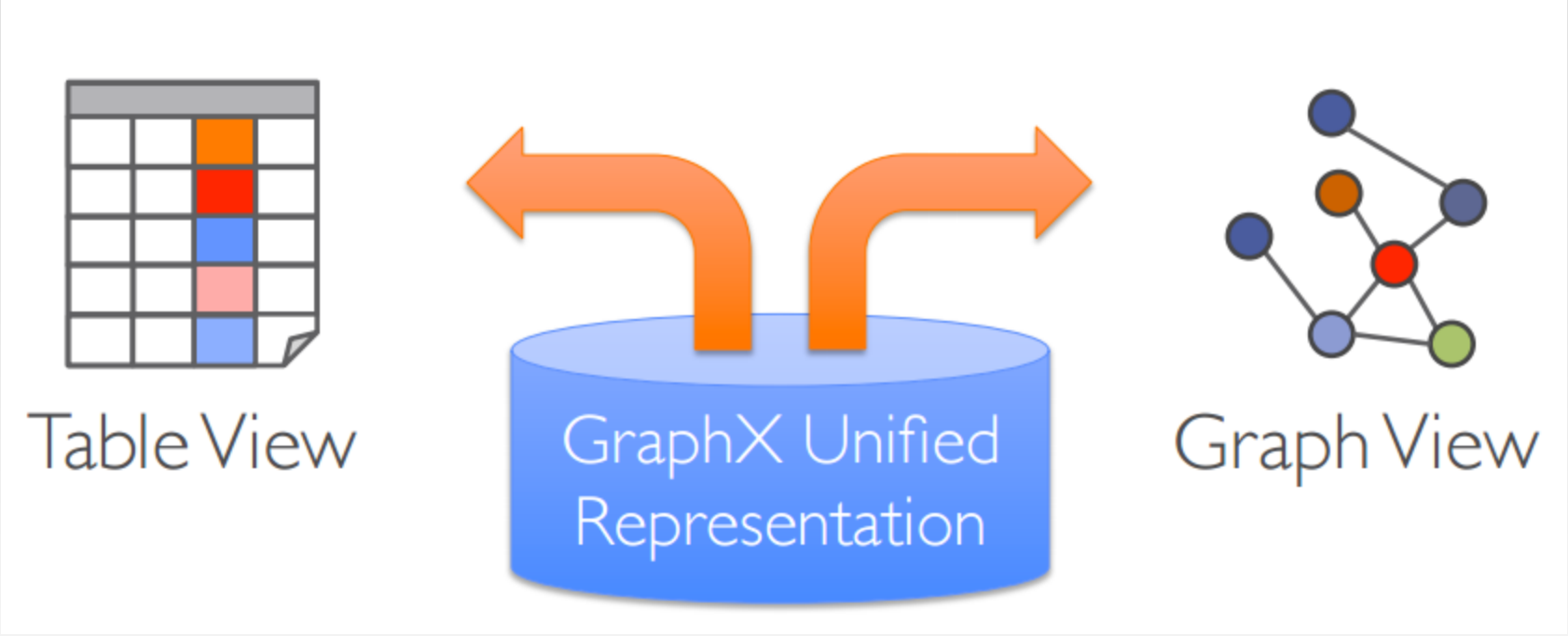
GraphX Motivation

- The goal of the GraphX project is to unify graph-parallel and data-parallel computation in one system with a single composable API.
- The GraphX API enables users to view data both as graphs and as collections (i.e., RDDs) without data movement or duplication.



GraphX Motivation

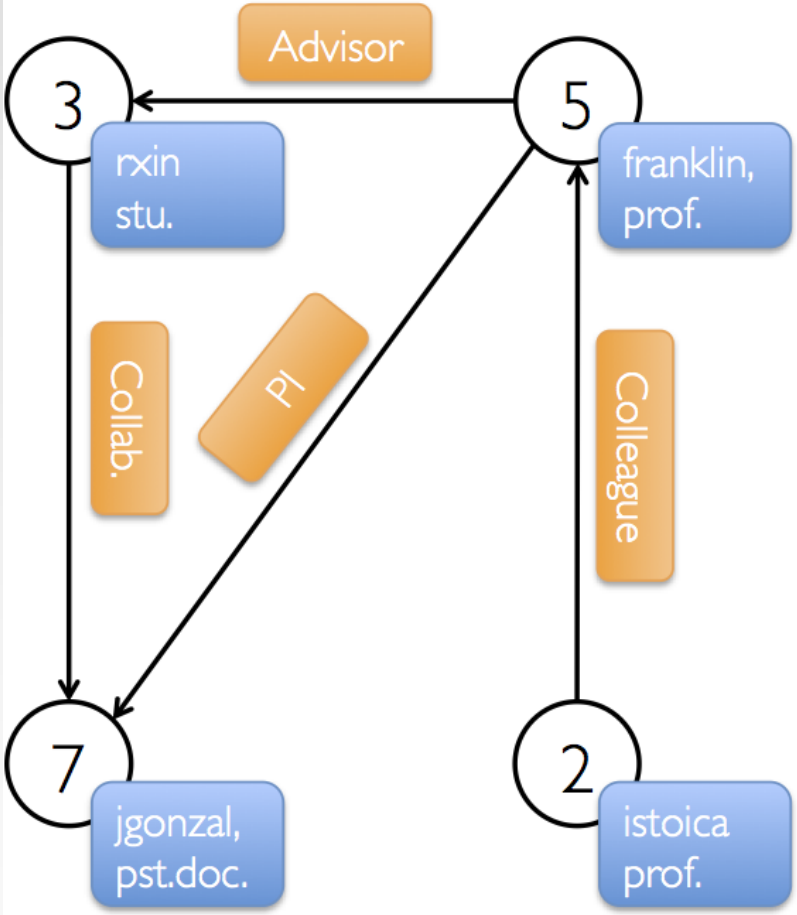
- Tables and Graphs are composable views of the same physical data



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

View a Graph as a Table

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

Table Operators

- Table (RDD) operators are inherited from Spark:

map	reduce	sample
filter	count	take
groupBy	fold	first
sort	reduceByKey	partitionBy
union	groupByKey	mapWith
join	cogroup	pipe
leftOuterJoin	cross	save
rightOuterJoin	zip	...

Graph Operators

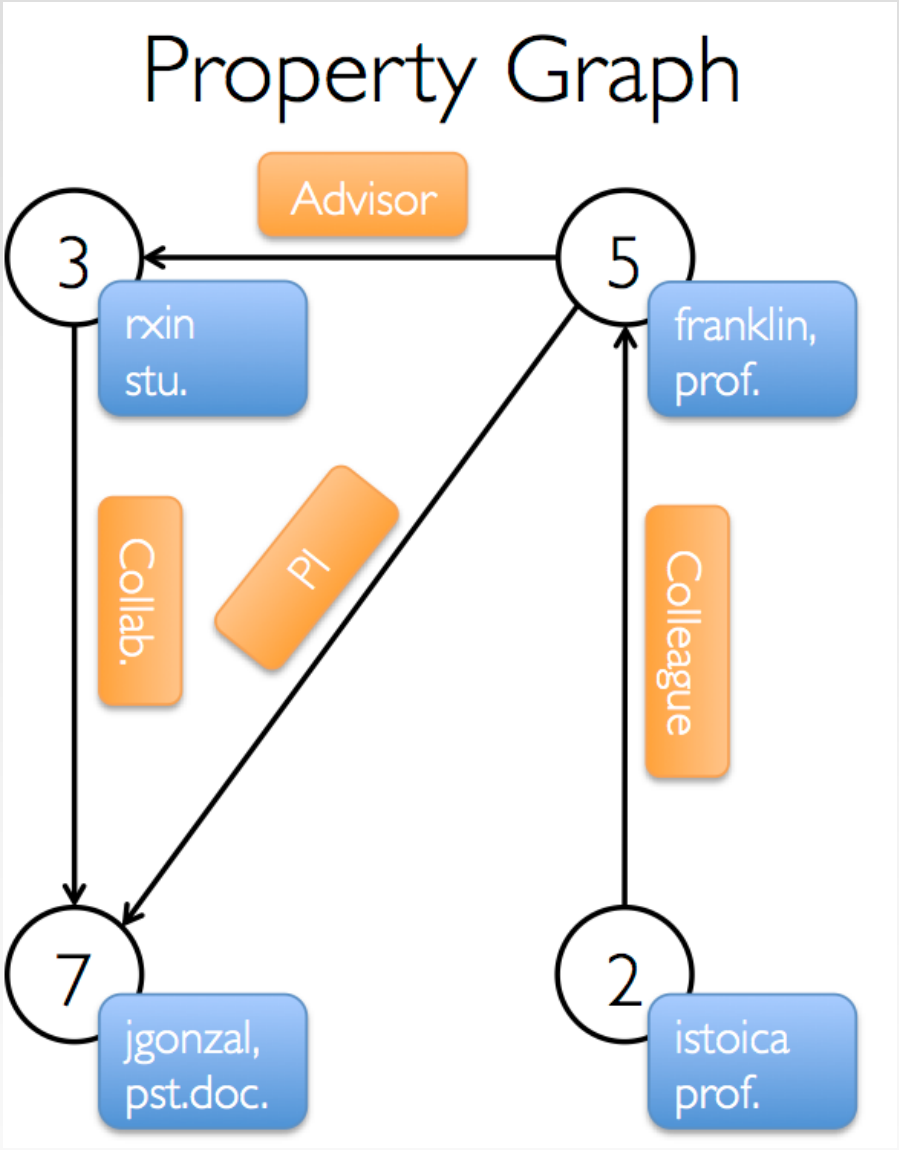
```
class Graph[VD, ED] {
        // Information about the Graph
        val numEdges: Long
        val numVertices: Long
        val inDegrees: VertexRDD[Int]
        val outDegrees: VertexRDD[Int]
        val degrees: VertexRDD[Int]
        // Views of the graph as collections
        val vertices: VertexRDD[VD]
        val edges: EdgeRDD[ED]
        val triplets: RDD[EdgeTriplet[VD, ED]]
        // Transform vertex and edge attributes
        def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
        def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
        def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]):
Graph[VD, ED2]
        def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
        // Modify the graph structure
        def reverse: Graph[VD, ED]
        def subgraph(
                epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
                vpred: (VertexID, VD) => Boolean = ((v, d) => true))
                : Graph[VD, ED]
          def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
        // …other operators…
}
```

The Property Graph

- The property graph is a directed multigraph with user defined objects attached to each vertex and edge.
- A directed multigraph is a directed graph with potentially multiple parallel edges sharing the same source and destination vertex
- The property graph is parameterized over the vertex (VD) and edge (ED) types. These are the types of the objects associated with each vertex and edge respectively.
- Each vertex is keyed by a *unique* 64-bit long identifier (VertexID). Similarly, edges have corresponding source and destination vertex identifiers.
- Logically the property graph corresponds to a pair of typed collections (RDDs) encoding the properties for each vertex and edge.

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

Example Property Graph



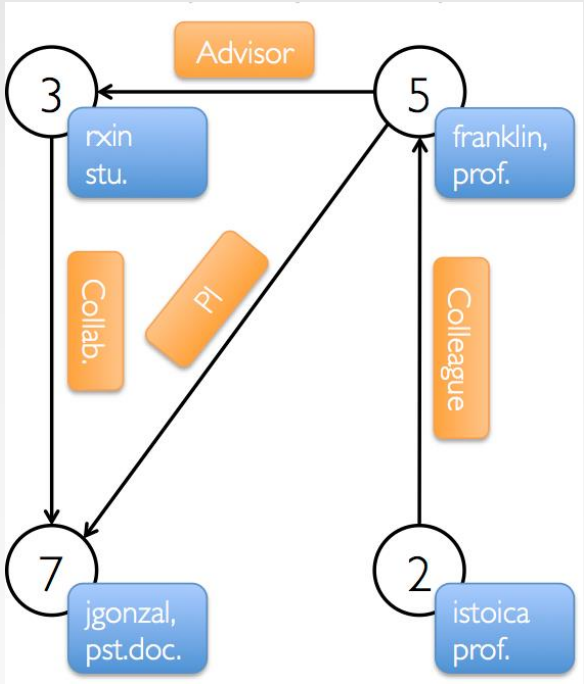
GraphX Example

- Import Spark and GraphX into your project

```
import org.apache.spark._
import org.apache.spark.graphx._
// To make some of the examples work we will also need RDD
import org.apache.spark.rdd.RDD
```

- We begin by creating the property graph from arrays of vertices and edges

```
val vertexArray = Array(
  (3L, ("rxin", "student")),
  (7L, ("jgonzal", "postdoc")),
  (5L, ("franklin", "prof")),
  (2L, ("istoica", "prof"))
)
val edgeArray = Array(
  Edge(3L, 7L, "collab"),
  Edge(5L, 3L, "advisor"),
  Edge(2L, 5L, "colleague"),
  EdgeEdge(5L, 7L, "pi"),
)
```



Construct a Property Graph

- The most general method of constructing a property graph is to use the Graph object

```
// Assume the SparkContext has already been constructed
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(vertexArray)
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(edgeArray)
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

  - Edges have a srcId and a dstId corresponding to the source and destination vertex identifiers.
  - In addition, the Edge class has an attr member which stores the edge property

Deconstruct a Property Graph

- We can deconstruct a graph into the respective vertex and edge views by using the graph.vertices and graph.edges members respectively

```
// Constructed from above
val graph: Graph[(String, String), String]
// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count
// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

  - Note that graph.vertices returns an VertexRDD[(String, String)] which extends RDD[(VertexId, (String, String))] and so we use the scala case expression to deconstruct the tuple.
  - graph.edges returns an EdgeRDD containing Edge[String]objects. We could have also used the case class type constructor as in the following:

    graph.edges.filter { **case Edge**(src, dst, prop) **=>** src > dst }.count

Graph Views

- In many cases we will want to extract the vertex and edge RDD views of a graph
- The graph class contains members (graph.vertices and graph.edges) to access the vertices and edges of the graph

- Example: use graph.vertices to display the names of the users who are professors

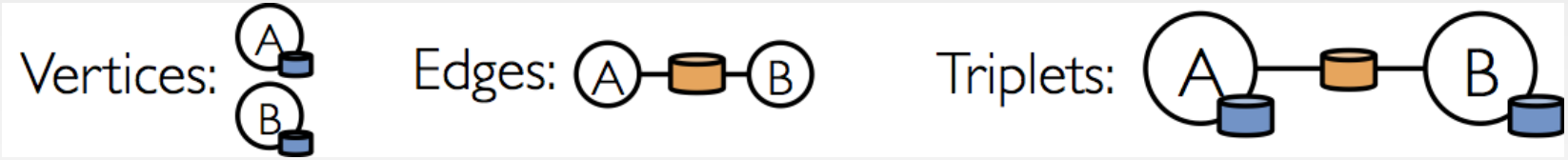
```
graph.vertices.filter {
  case (id, (name, pos)) => pos == “prof”
}.collect.foreach {
  case (id, (name, age)) => println(s"$name is Professor")
}
```

Triplet View

- The triplet view logically joins the vertex and edge properties yielding an RDD[EdgeTriplet[VD, ED]] containing instances of the EdgeTriplet class
- This *join* can be expressed in the following SQL expression:

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

or graphically as:



EdgeTriplet class

- The EdgeTriplet class extends the Edge class by adding the srcAttr and dstAttr members which contain the source and destination properties respectively.
- We can use the triplet view of a graph to render a collection of strings describing relationships between users.

```
// Constructed from above
val graph: Graph[(String, String), String]
// 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)
facts.collect.foreach(println(_))
```

Pregel Operators

```
def pregel[A]
      (initialMsg: A,
       maxIter: Int = Int.MaxValue,
       activeDir: EdgeDirection = EdgeDirection.Out)
      (vprog: (VertexId, VD, A) => VD,
       sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
       mergeMsg: (A, A) => A)
    : Graph[VD, ED] = {
              … …
     }
```

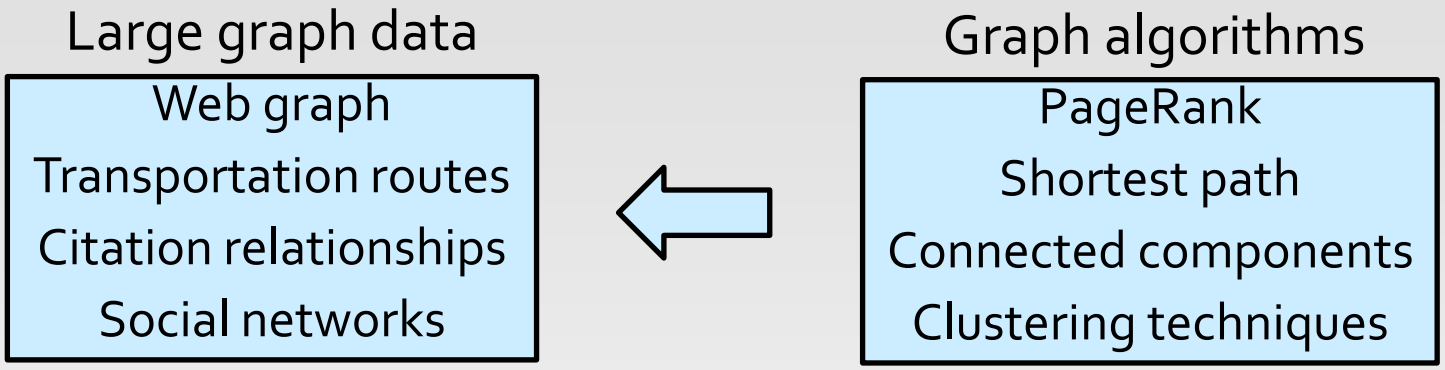
- The first argument list contains configuration parameters including the initial message, the maximum number of iterations, and the edge direction in which to send messages (by default along out edges).

- The second argument list contains the user defined functions for receiving messages (the vertex program vprog), computing messages (sendMsg), and combining messages mergeMsg.

Pregel Introduction

Motivation of Pregel

- Many practical computing problems concern large graphs



- Single computer graph library does not scale
- MapReduce is ill-suited for graph processing
  - Many iterations are needed for parallel graph processing
  - Materializations of intermediate results at every MapReduce iteration harm performance

Pregel

- **Pregel**: A System for Large-Scale **Graph** Processing (Google) - Malewicz et al. SIGMOD 2010.
- Scalable and Fault-tolerant platform
- API with flexibility to express arbitrary algorithm
- Inspired by Valiant's Bulk Synchronous Parallel model
  - Leslie G. Valiant: A Bridging Model for Parallel Computation. Commun. ACM 33 (8): 103-111 (1990)
- Vertex centric computation (Think like a vertex)

Bulk Synchronous Parallel Model (BSP)

analogous to MapReduce rounds

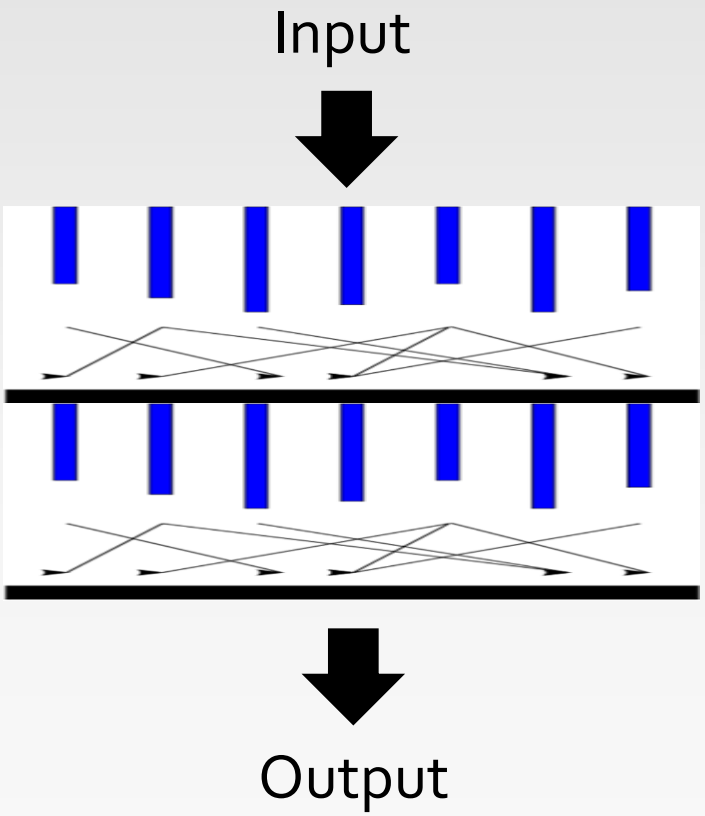
- Processing: a series of supersteps
- Vertex: computation is defined to run on each vertex
- Superstep S: *all vertices compute in parallel; each vertex v may*
  - receive messages sent to v from superstep S – 1;
  - perform some computation: modify its states and the states of its outgoing edges
  - Send messages to other vertices (to be received in the next superstep)

Message passing

Vertex-centric, message passing

Pregel Computation Model

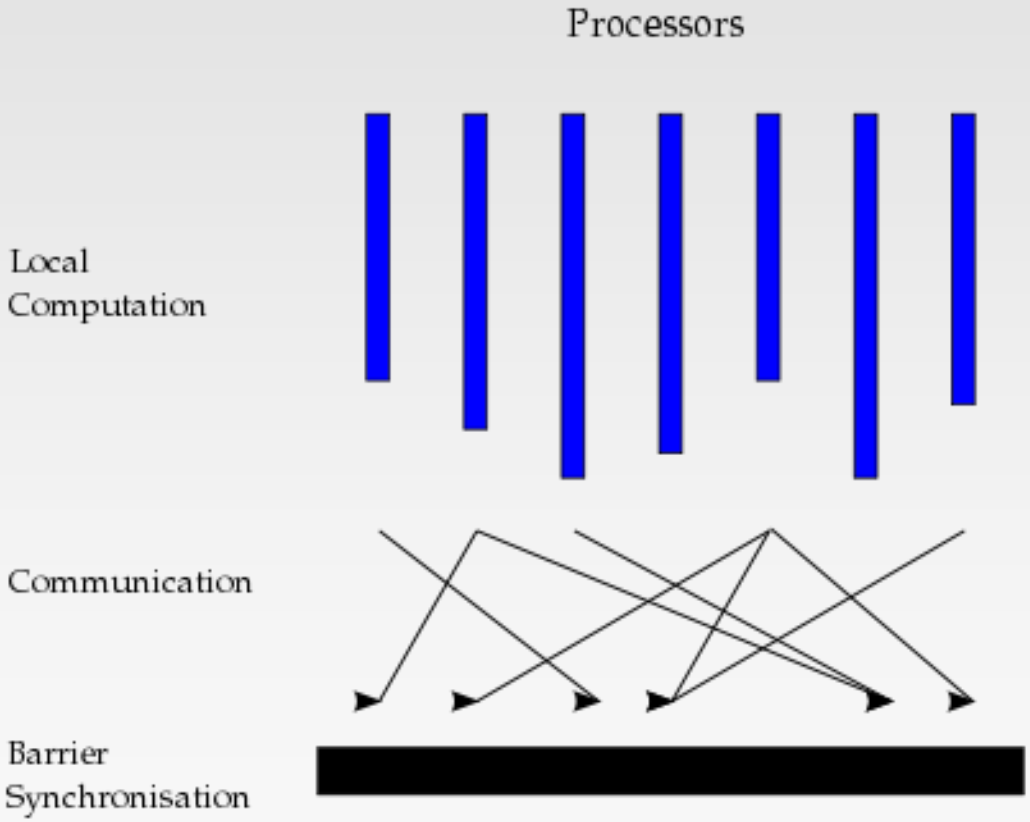
- Based on Bulk Synchronous Parallel (BSP)
  - Computational units encoded in a directed graph
  - Computation proceeds in a series of supersteps
  - Message passing architecture



Supersteps
(a sequence of iterations)

Pregel Computation Model (Cont')

- Concurrent computation and Communication need not be ordered in time
- Communication through message passing

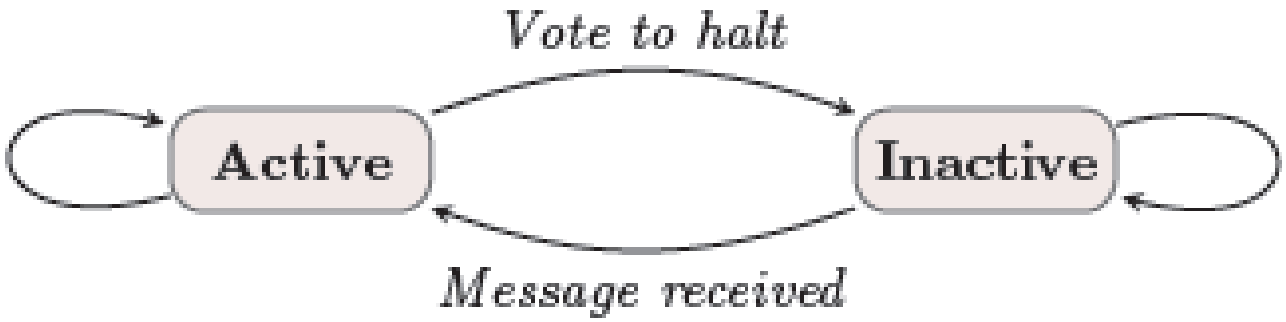


Source: http://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Pregel Computation Model (Cont')

- Superstep: the vertices compute in parallel
  - Each vertex

State machine for a vertex

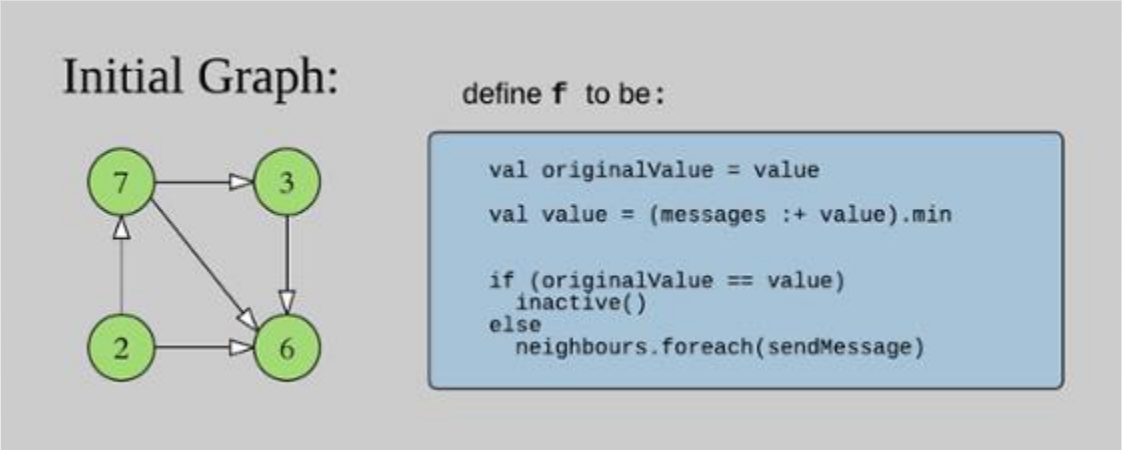


  - Termination condition
    - All vertices are simultaneously inactive
    - A vertex can choose to deactivate itself
    - Is "woken up" if new messages received

Superstep

- During a superstep, the following can happen in the framework:
  - It receives and reads messages that are sent to v from the previous superstep s-1.
  - It applies a user-defined function f to each vertices in parallel, so f essentially specifies the behaviour of a single vertex v at a single superstep s.
  - It can mutate the state of v.
  - It can send messages to other vertices (typically along outgoing edges) that the vertices will receive in the next superstep s+1.
- All communations are between supersteps s and s+1

Example: Find the minimum value in a graph

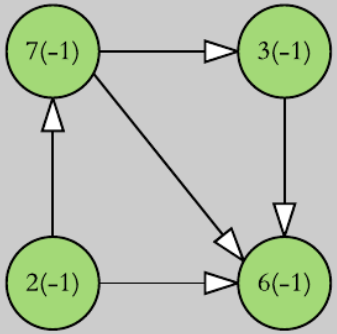


- The pseudo-code definition of f is also given above, it will:
  - Set originalValue to the current value of the vertex.
  - Mutate the value of the vertex to the minimum of all the incoming messages and originalValue.
  - If originalValue and value are the same, then we will render the vertex inactive. Otherwise, send message out to all its outgoing neighbour.

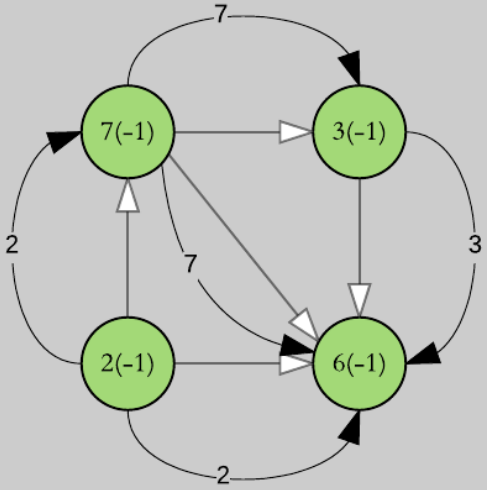
Superstep 0

Superstep 0:

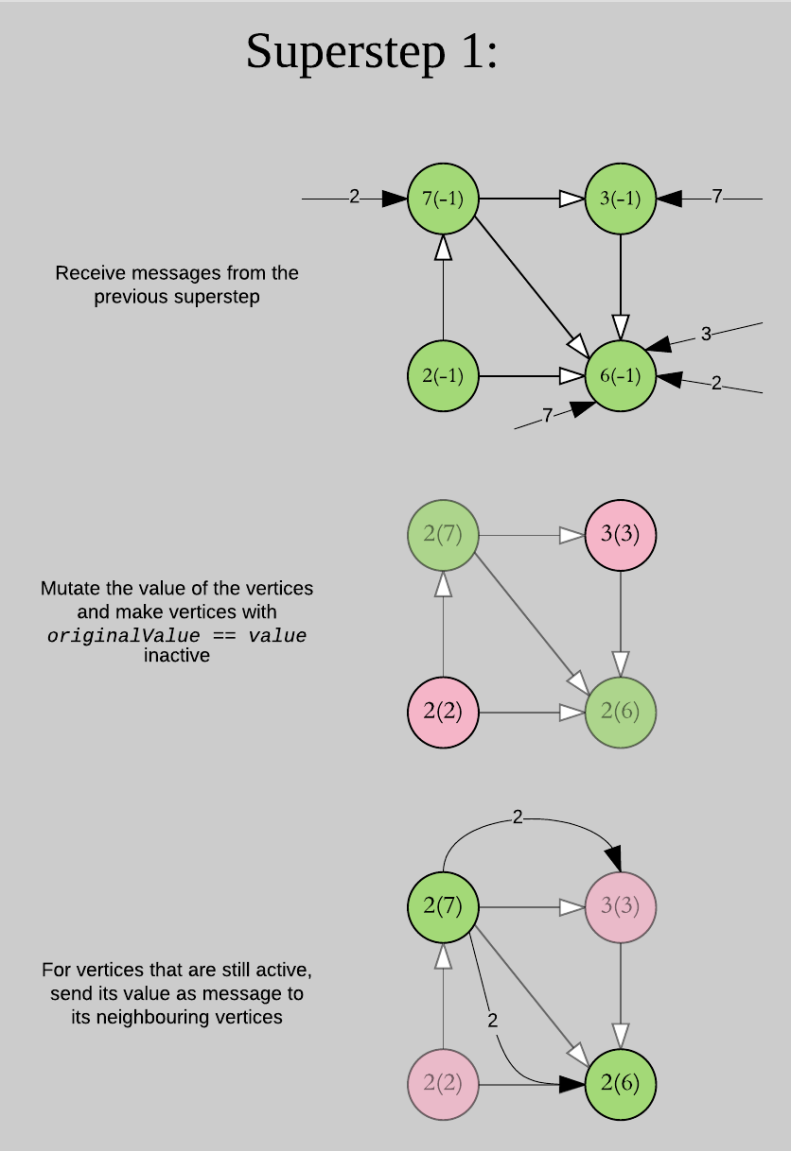
Initialise the graph
with `-1` picked as the
`originalValue`



All active vertices
send its value as message to
its neighbouring vertices



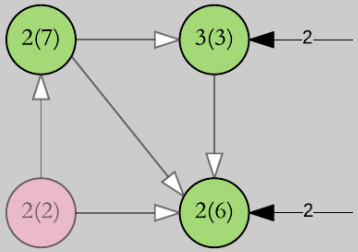
Superstep 1


Superstep 1:
Receive messages from the previous superstep
2
7(-1)
3(-1)
7
3
2(-1)
6(-1)
2
7
Mutate the value of the vertices and make vertices with originalValue == value inactive
2(7)
3(3)
2(2)
2(6)
For vertices that are still active, send its value as message to its neighbouring vertices
2
2(7)
3(3)
2
2(2)
2(6)

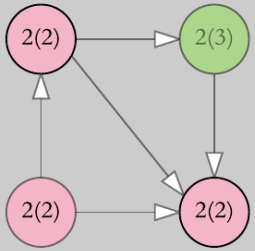
Superstep 2

Superstep 2:

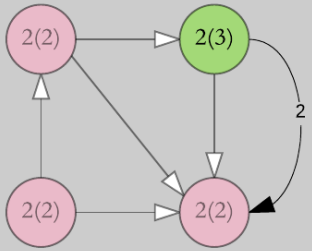
Receive message from the previous superstep



Mutate the value of the vertices and make vertices with *originalValue == value* inactive



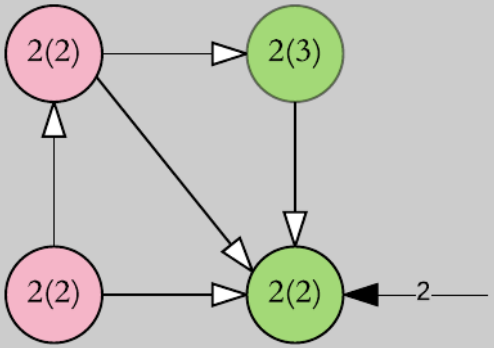
For vertices that are still active, send its value as message to its neighbouring vertices



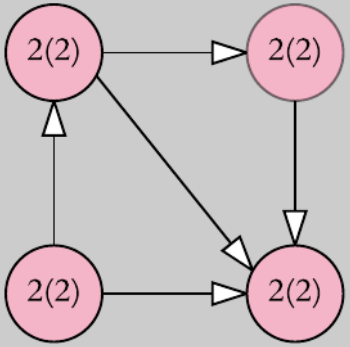
Superstep 3

Superstep 3:

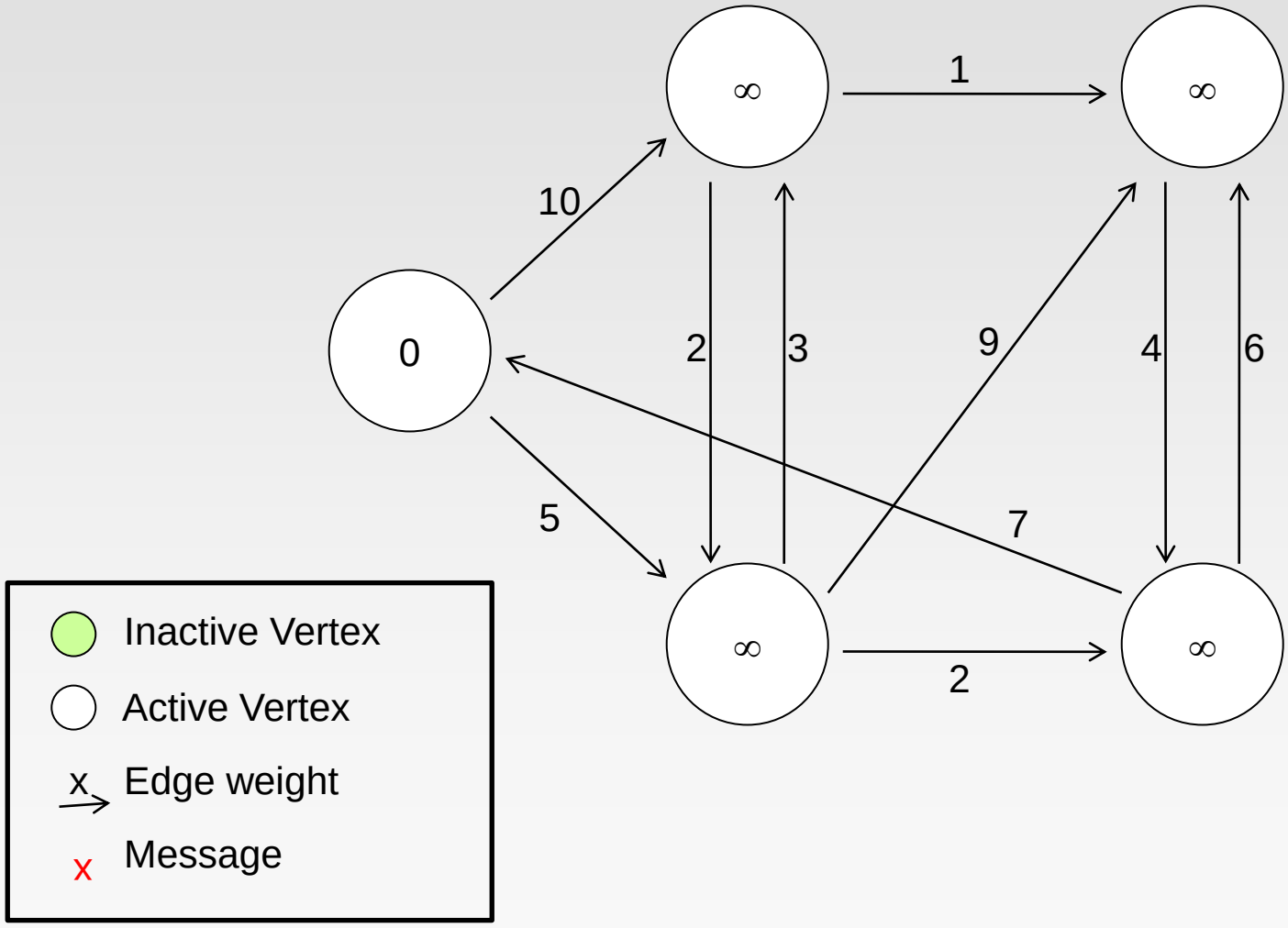
Receive message from the previous superstep



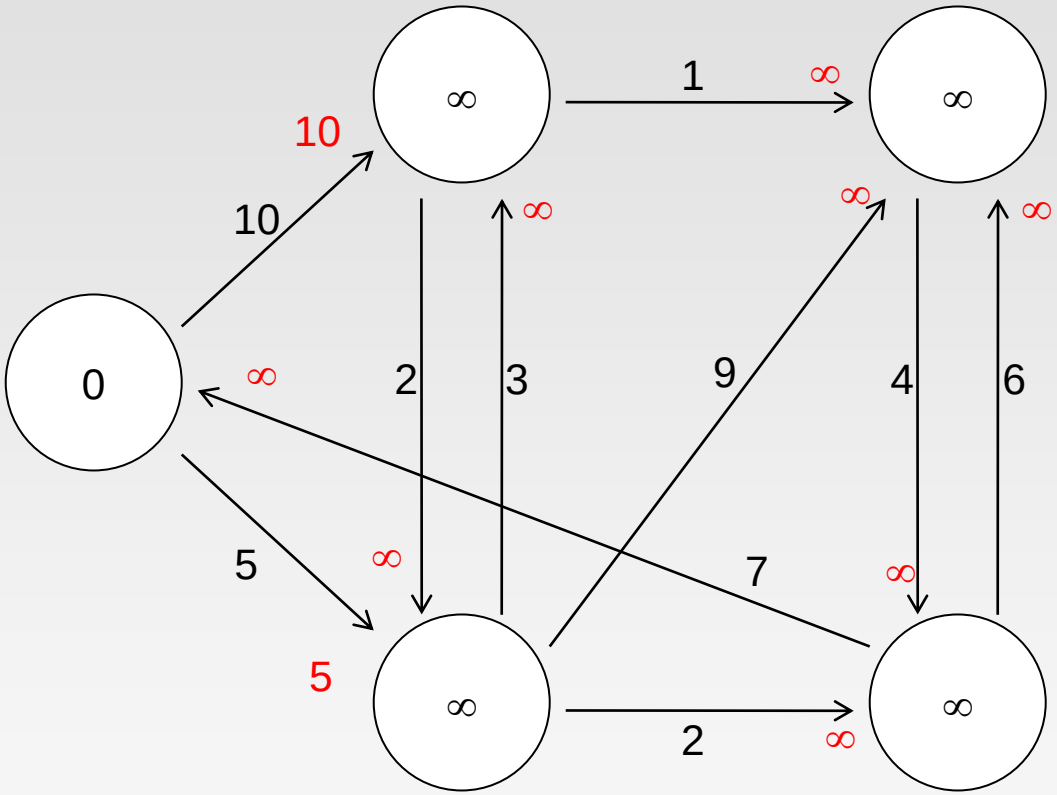
Mutate the value of the vertices and make vertices with `originalValue == value` inactive



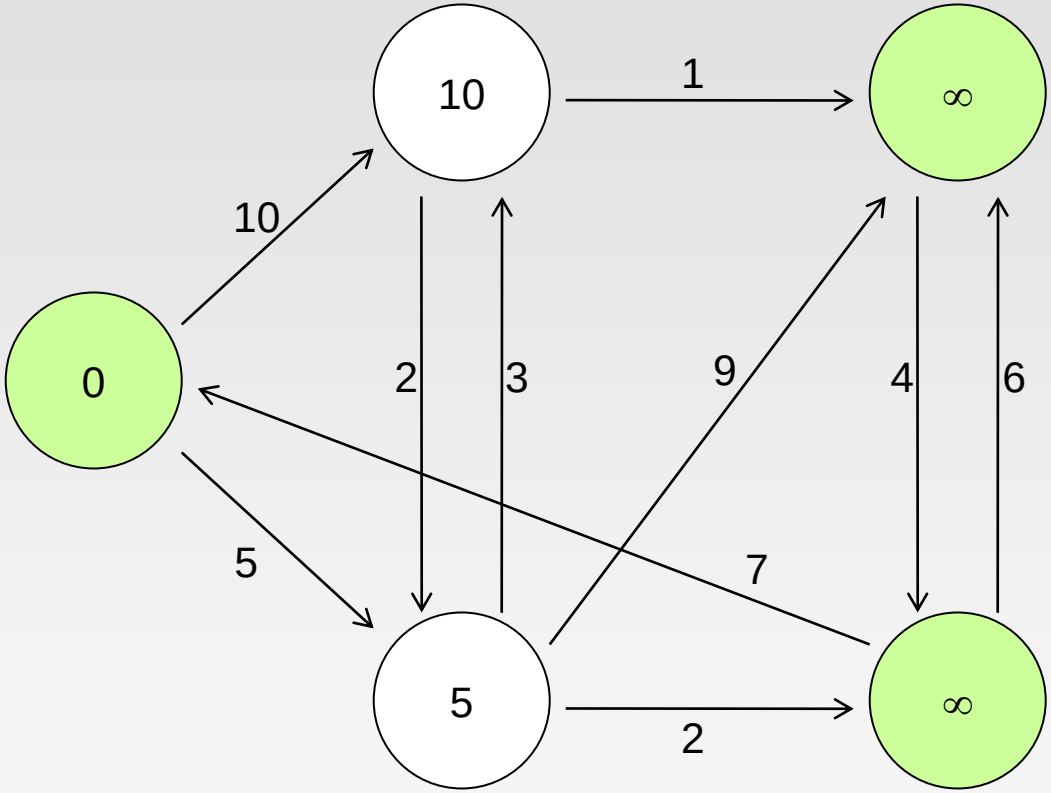
Example: SSSP – Parallel BFS in Pregel



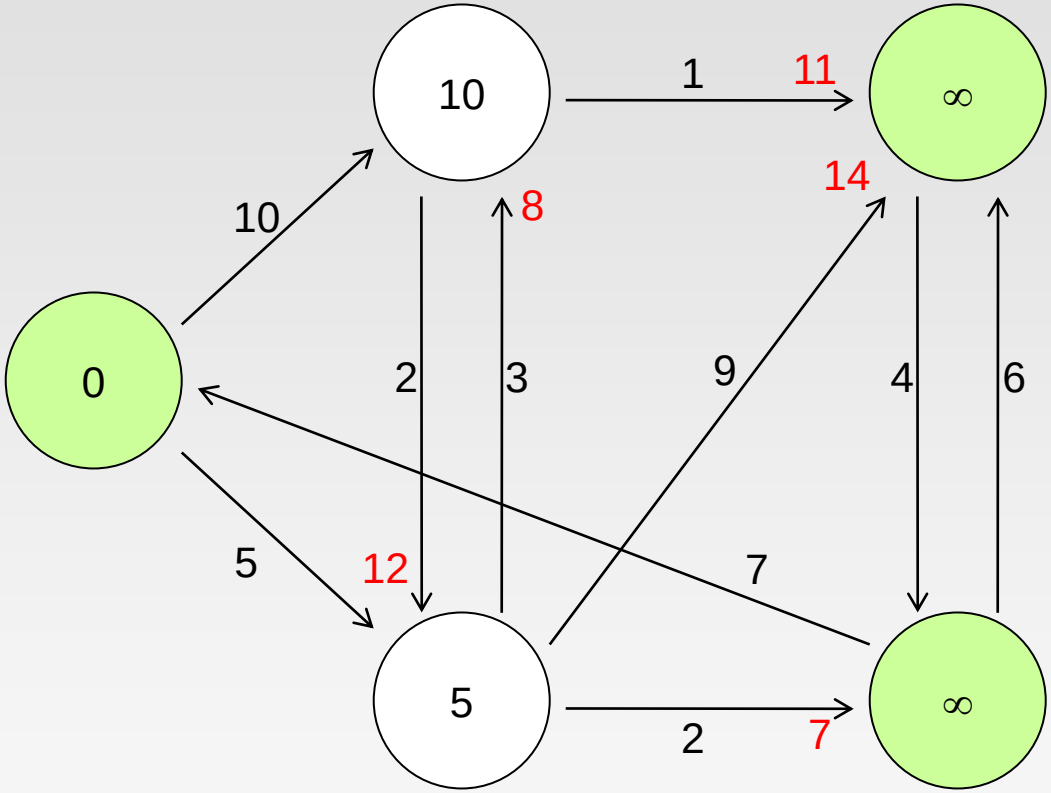
Example: SSSP – Parallel BFS in Pregel



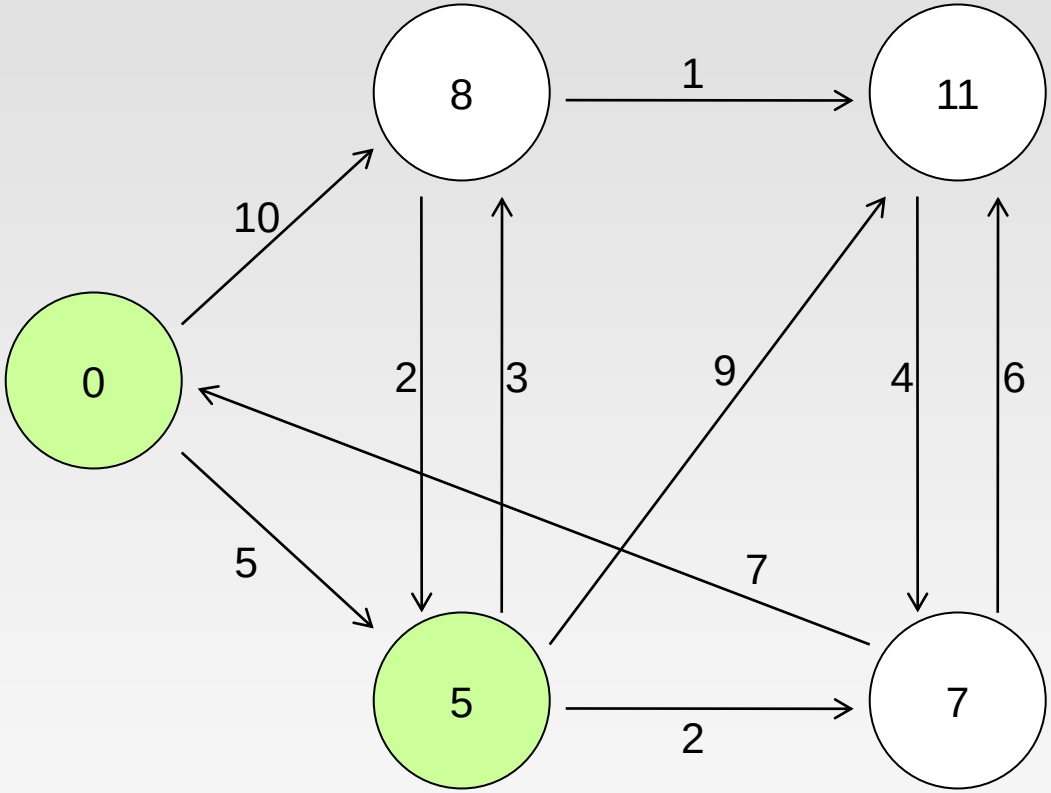
Example: SSSP – Parallel BFS in Pregel



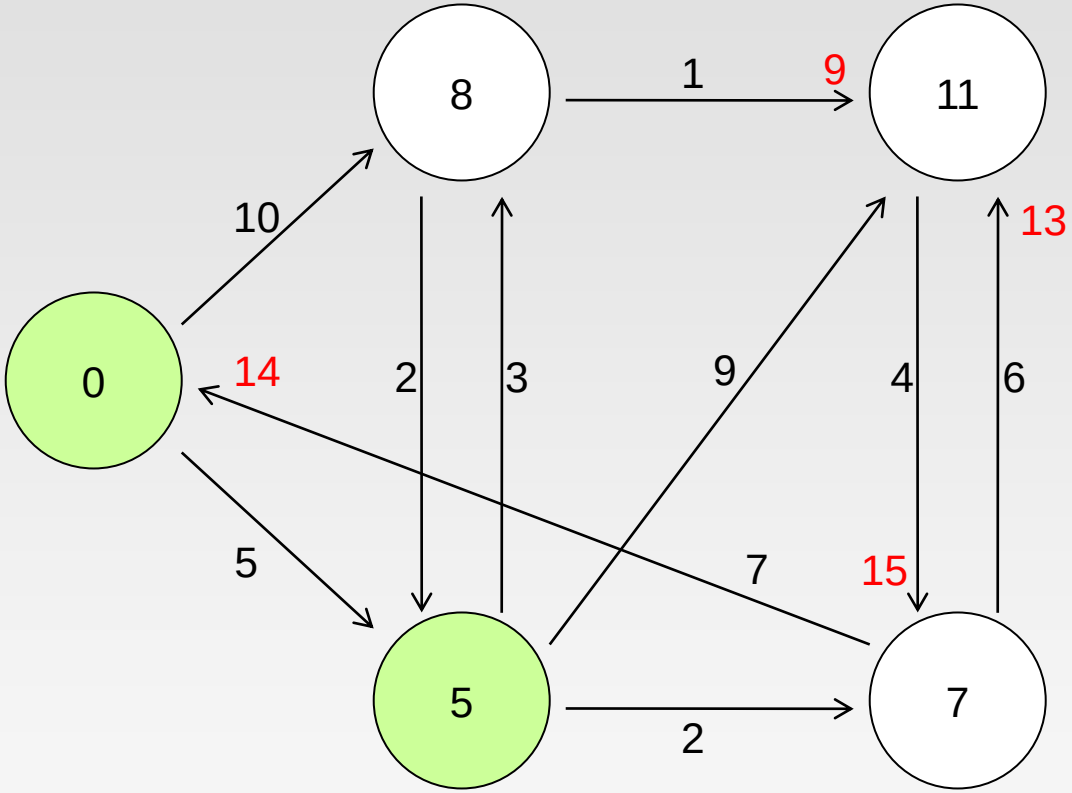
Example: SSSP – Parallel BFS in Pregel



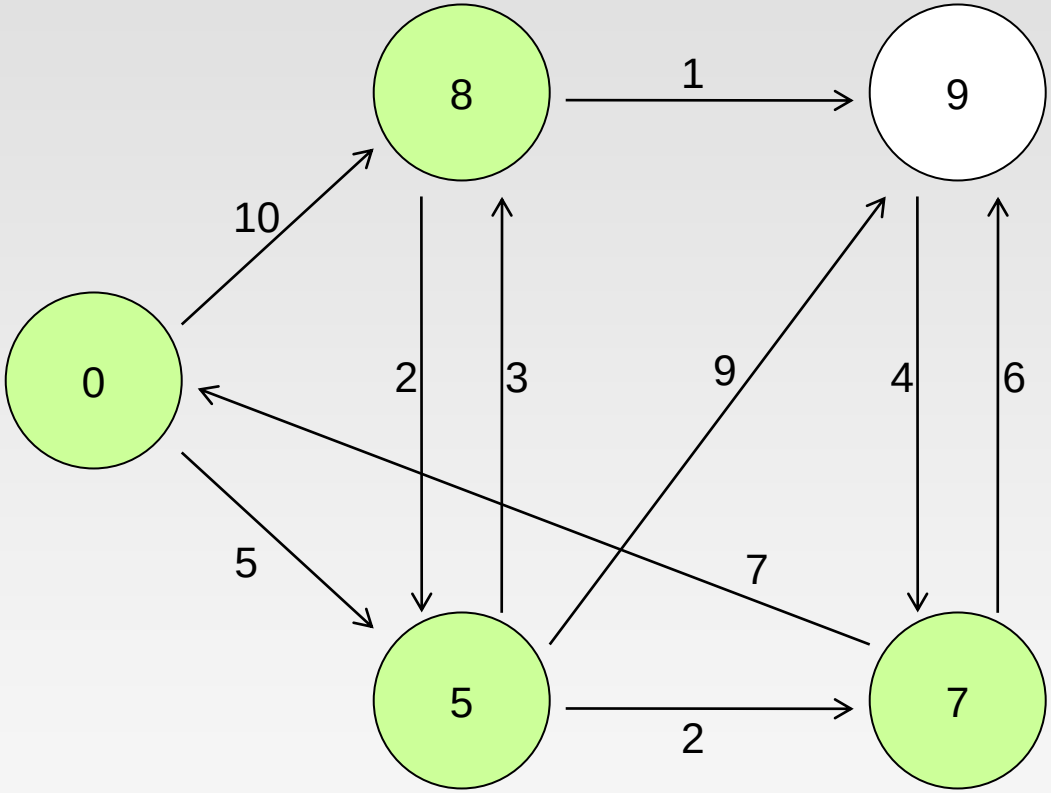
Example: SSSP – Parallel BFS in Pregel



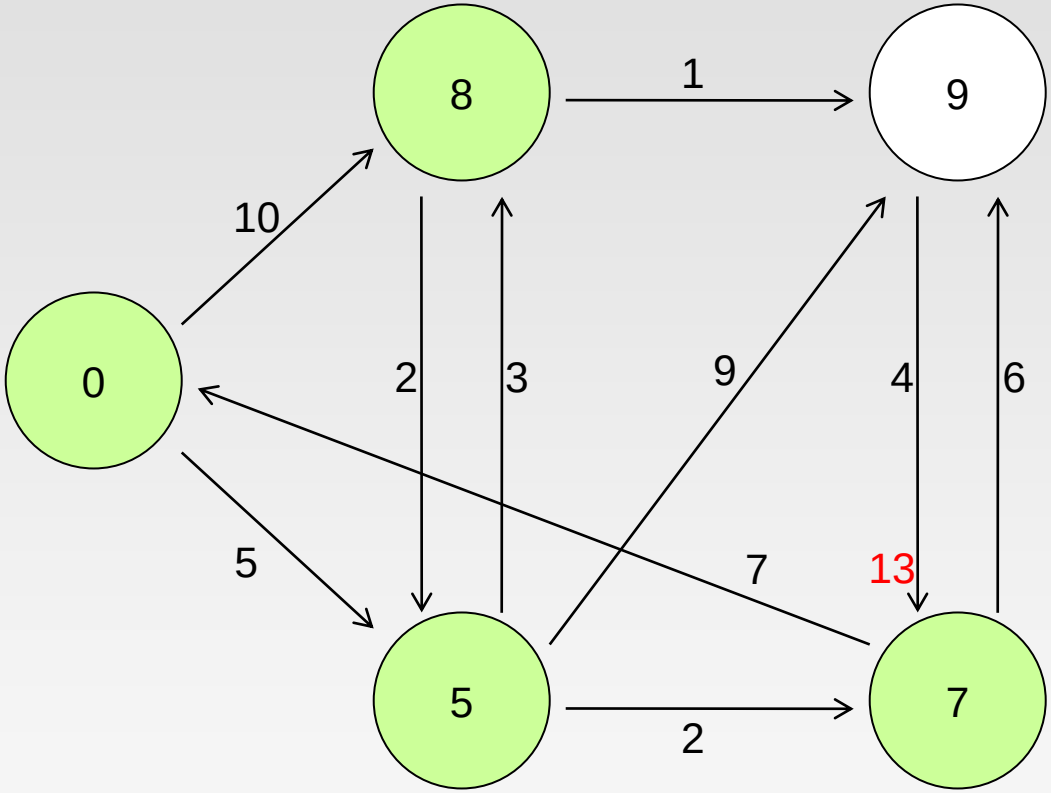
Example: SSSP – Parallel BFS in Pregel



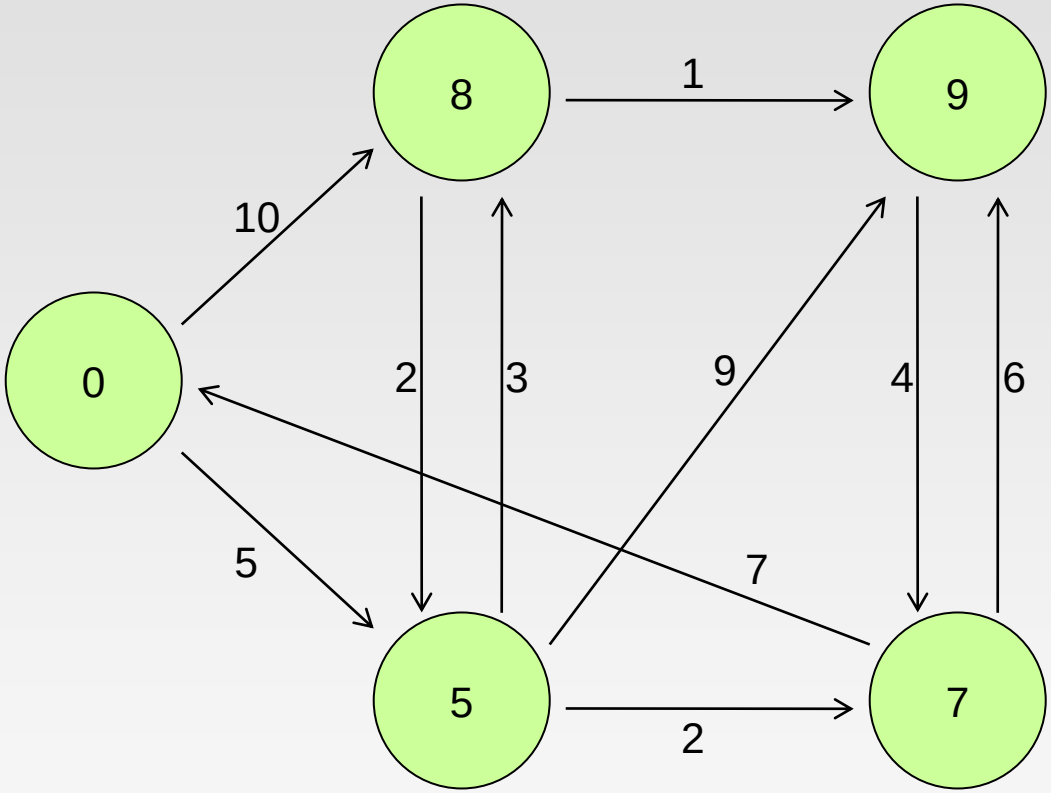
Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



Pregel Operators

```
def pregel[A]
      (initialMsg: A,
       maxIter: Int = Int.MaxValue,
       activeDir: EdgeDirection = EdgeDirection.Out)
      (vprog: (VertexId, VD, A) => VD,
       sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
       mergeMsg: (A, A) => A)
    : Graph[VD, ED] = {
              … …
     }
```

- The first argument list contains configuration parameters including the initial message, the maximum number of iterations, and the edge direction in which to send messages (by default along out edges).

- The second argument list contains the user defined functions for receiving messages (the vertex program vprog), computing messages (sendMsg), and combining messages mergeMsg.

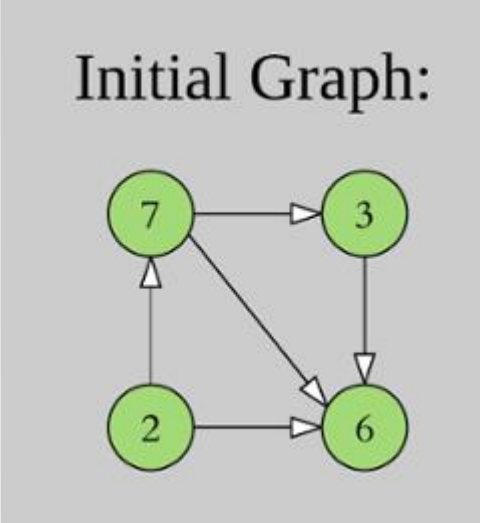
Scala Currying

- Methods may define multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments.

```
def modN(n: Int)(x: Int) = ((x % n) == 0)

val nums = List(1, 2, 3, 4, 5, 6, 7, 8)

nums.filter(modN(2))
```

- Results:
  - nums.filter(modN(2)) = nums.filter(x => modN(2)(x))
  - x is treated as the argument: List(2,4,6,8)

Find the minimum value in a graph




```
val initialMsg = 9999
def vprog(vertexId: VertexId, value: (Int, Int), message: Int): (Int, Int) =
{
        if (message == initialMsg)     value
        else     (message min value._1, value._1)
}

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))
}

def mergeMsg(msg1: Int, msg2: Int): Int = msg1 min msg2

val minGraph = graph.pregel(initialMsg)(vprog,  sendMsg,  mergeMsg)
```

Single Source Shortest Path

- vprog:
```
(id, dist, newDist) => math.min(dist, newDist)
```
- sendMsg:
```
triplet => {
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {  Iterator.empty  }
  }
```
- mergeMsg:
```
(a, b) => math.min(a, b)
```

https://github.com/apache/spark/blob/master/graphx/src/main/scala/org/apache/spark/graphx/lib/ShortestPaths.scala
or
https://spark.apache.org/docs/latest/graphx-programming-guide.html#pregel-api

- Full Pregel function call:

```
val initialGraph = graph.mapVertices((id, _) =>
    if (id == sourceId) 0.0 else Double.PositiveInfinity)
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {  Iterator.empty  }
  },
  (a, b) => math.min(a, b) // Merge Message
)
```

References

- http://spark.apache.org/docs/latest/index.html
- Spark GraphX guide: http://spark.apache.org/docs/latest/graphx-programming-guide.html
- Graph Analytics with Graphx. http://ampcamp.berkeley.edu/big-data-mini-course/graph-analytics-with-graphx.html
- https://www.cakesolutions.net/teamblogs/graphx-pregel-api-an-example
- Learning Spark. O'Reilly Media.

End of Chapter 6