An introduction to Apache Spark

Strumenti di programmazione per sistemi paralleli e distribuiti
2014/2015
Prof. Massimo Coppola

Speaker: Alessandro Lulli - lulli@di.unipi.it

Summary

- introduction to distributed graph processing
- RDDs: Resilient Distributed Datasets
- Apache Spark
- GraphX
- my experience: a case study with Connected Components

“text on cloud” generated from Zaharia, Matei, et al. "Spark: cluster computing with working sets”

A (big) data problem

- exponential growth of structured and unstructured data
- significant part of the data produced can be modelled as a graph
- data and graphs are produced continuously by internet users
- some examples:
  - US road network 10^11 nodes and edges
  - Web Page Graph 3.5 x 10^9 nodes and 128 x 10^9 edges
  - Twitter 40 x 10^6 nodes and 10^9 edges

http://wwwranking.webdatacommons.org/more.html
http://an.kaist.ac.kr/traces/WWW2010.html

Why data analytics?

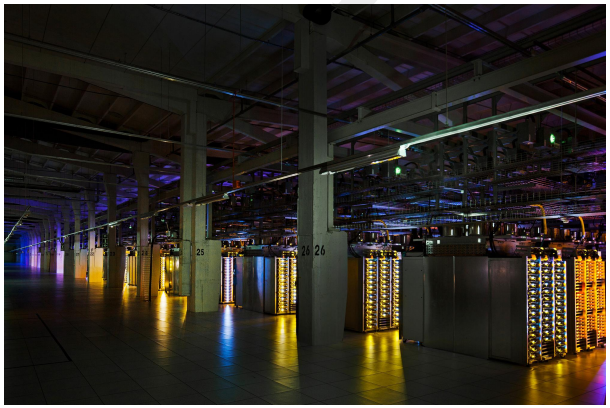
- to provide new products and services tailored for users: ad-hoc advertisements and search results
- improve decision making: ranking methods
- minimise risks: search for vulnerability in road networks
- unearth valuable insights that would otherwise remain hidden: study human brain connectome

Facts

- we have a lot of data
- it is useful to elaborate data
- processing capacity is beyond the capacity of a single machine

Our target

- efficient graph processing (GP) on distributed commodity hardware

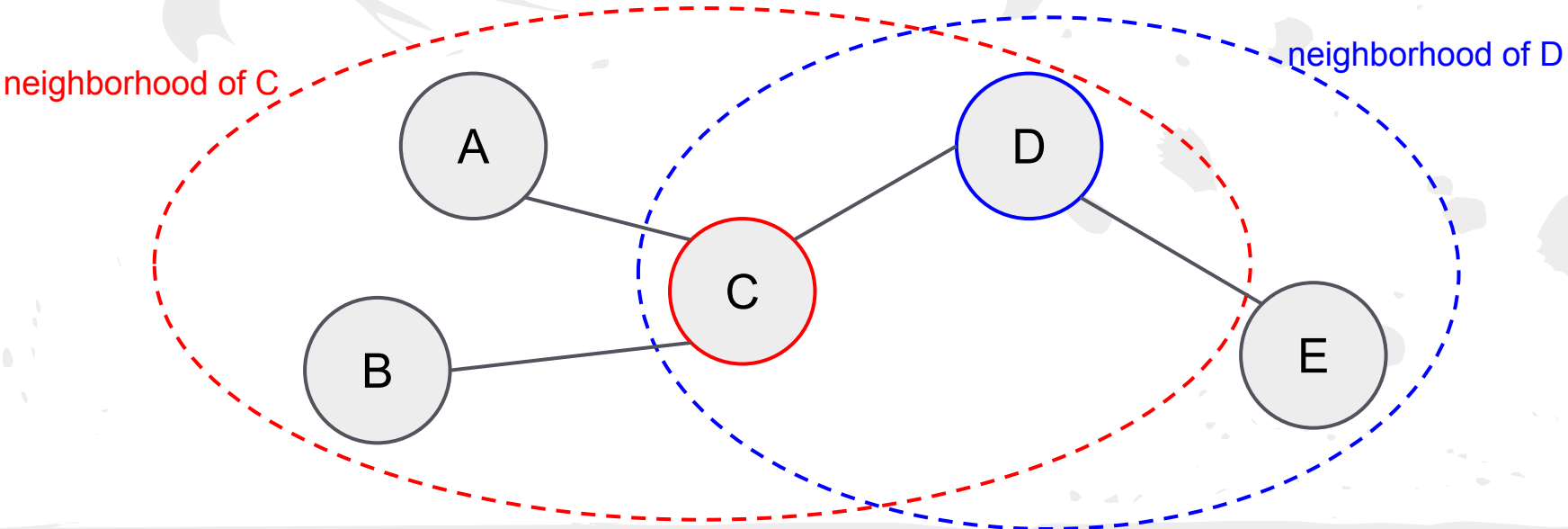


Distributed frameworks for GP

- aim at easing the development task by:
  - letting programmers focus only on problem solving
  - automating data distribution
  - managing failures
- the vertex-centric computing model has momentum
  - computation follows the structure of the graph
- two main approaches:
  - data parallel, mainly based on Google's MapReduce:
    - Hadoop
    - Spark
  - BSP-Like:
    - Pregel
    - GraphX, built on top of Spark (2013)

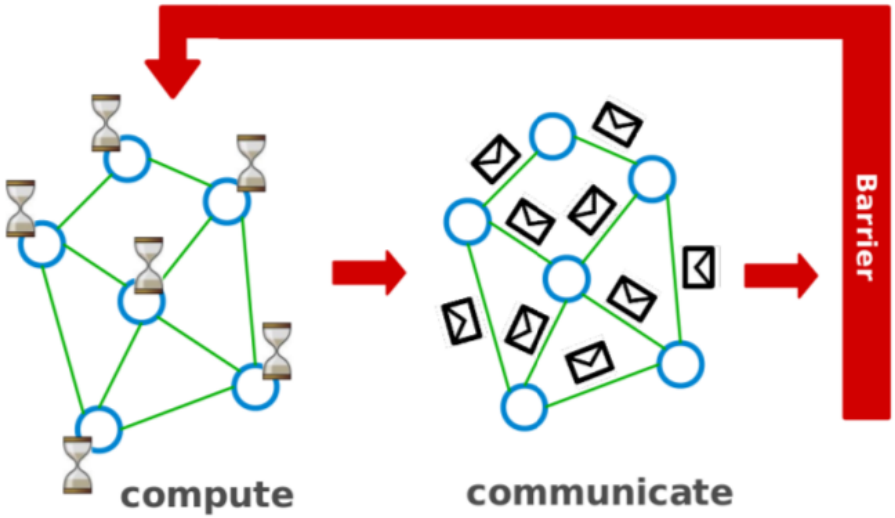
Vertex centric computing model

- each vertex
  - is a computational unit
  - local-knowledge computation: the computation for a node can rely only on its own data and the neighborhood data
  - there is no global knowledge
- each edge
  - is a communication channel
  - vertices interact with other vertices using messages



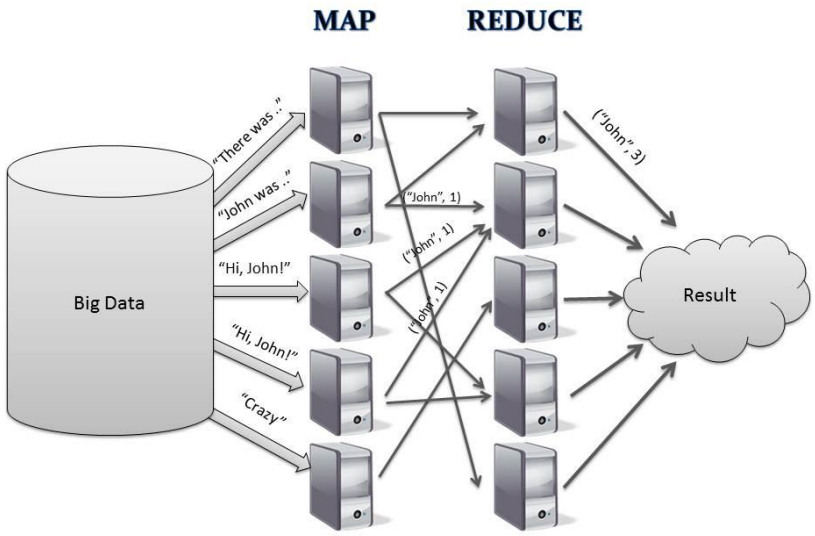
Bulk Synchronous Parallel

- define a computation as a sequence of iterations called *superstep*
- synchronization barrier at the end of each superstep
- all communications are from superstep S to superstep S+1
- re-introduced by Google with Pregel and Giraph based on Hadoop MapReduce



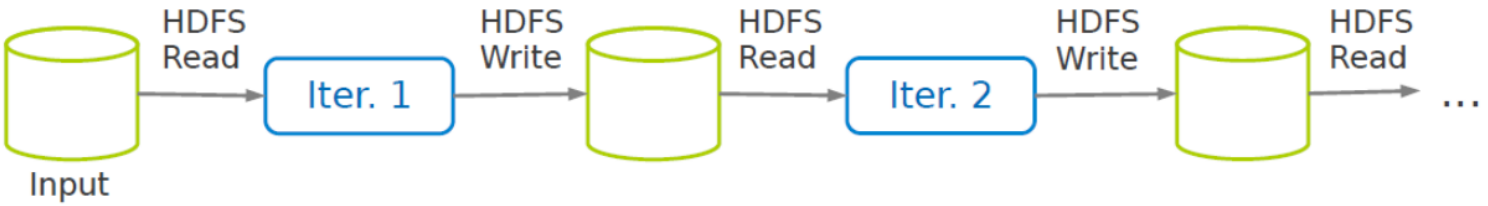
MapReduce

- three step model:
  - *Map*: each sub-problem takes an input key/value pair and produces a set of intermediate key/value pairs
  - *Shuffle*: key value pairs having the same key are assigned to the same worker
  - *Reduce*: all the results from the Map tasks are gathered from the worker nodes and then merged to form the output solution
- a graph is usually represented by assigning each vertex to a key



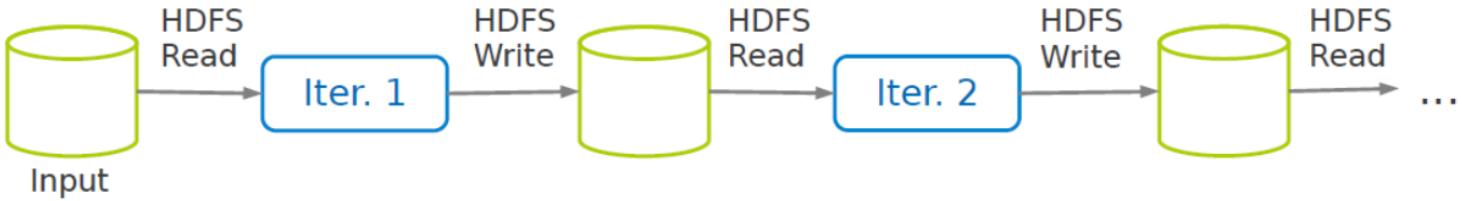
Motivation to move forward

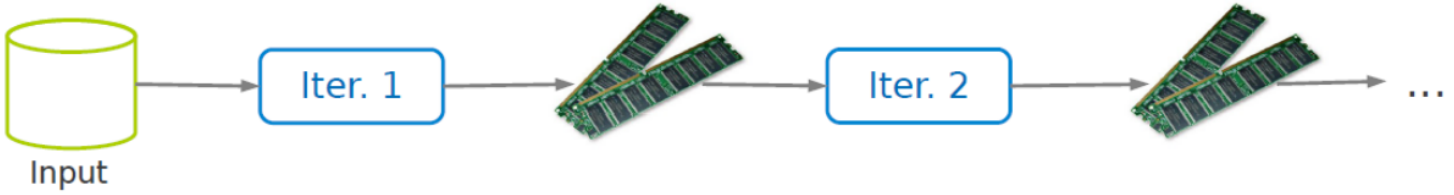
- MapReduce greatly simplified big data analysis on large, unreliable clusters
- it provides fault-tolerance, but also has drawbacks
  - iterative computation: hard to reuse intermediate results across multiple computations
  - efficiency: the only way to share data across jobs is stable storage, which is slow



Proposed solution: Spark

- challenge: how to design a distributed memory abstraction that is both fault tolerant and efficient?
- idea: in-memory data processing and sharing using Resilient Distributed Datasets (RDDs)





Resilient Distributed Datasets (RDDs)

- a distributed memory abstraction
- restricted form of distributed shared memory
  - read-only, partitioned collection of records
  - immutable collections of objects spread across a cluster
- can only be built through coarse-grained deterministic transformations from:
  - data in stable storage
  - transformations from other RDDs
- express computation by defining RDDs

What is an RDD?

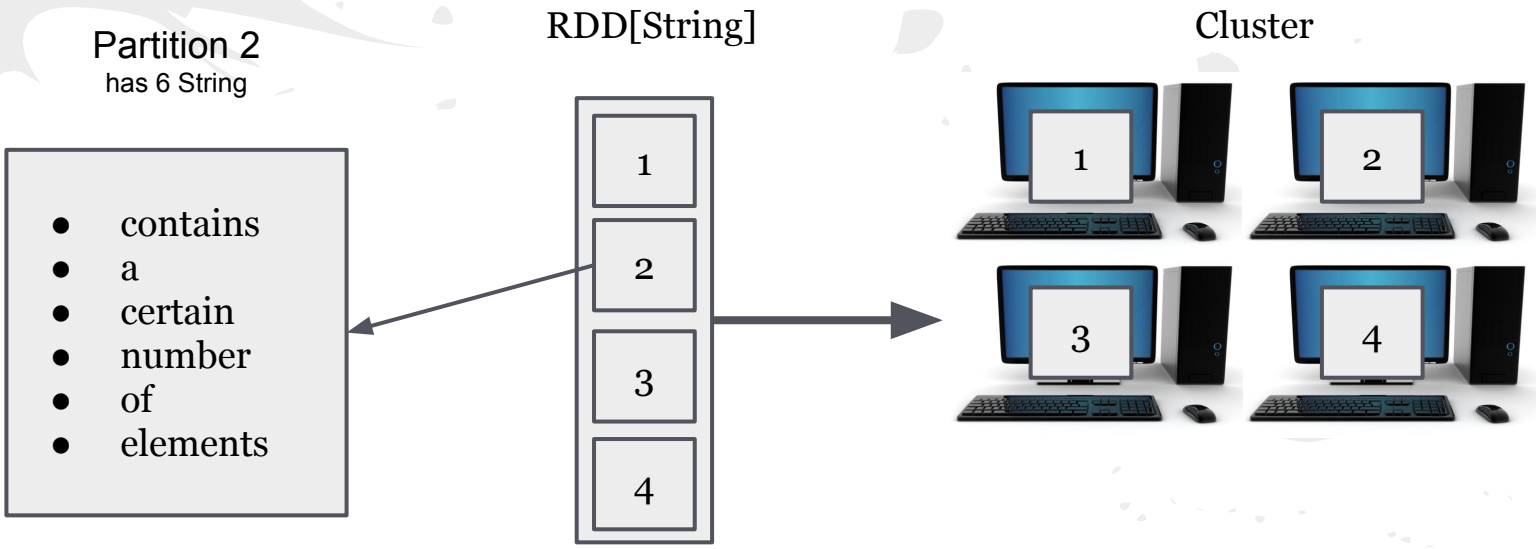
- imagine to have a list distributed on different computer

a “normal” list

RDD

RDD Partitions

- an RDD is divided into a number of partitions, which are atomic pieces of information
- each partition:
  - contains a certain number of elements of type T
  - can be stored on different nodes of a cluster

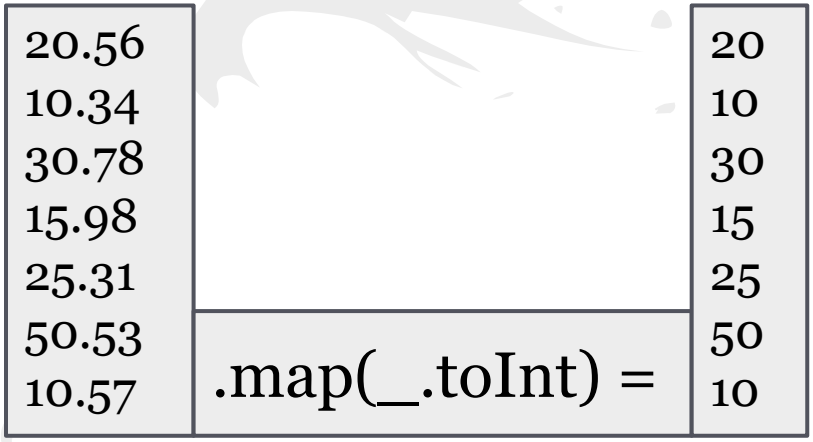


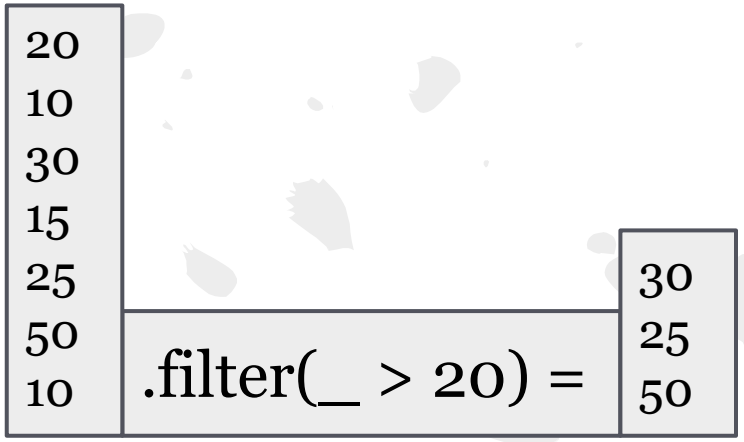
RDD Generic

- an RDD is generic (the same of a Java List) -> RDD[T]
- it is possible to make an RDD[String], RDD[Int], RDD [List<String>]...
- it is enough to have type T Serializable

RDD Operations: Transformations

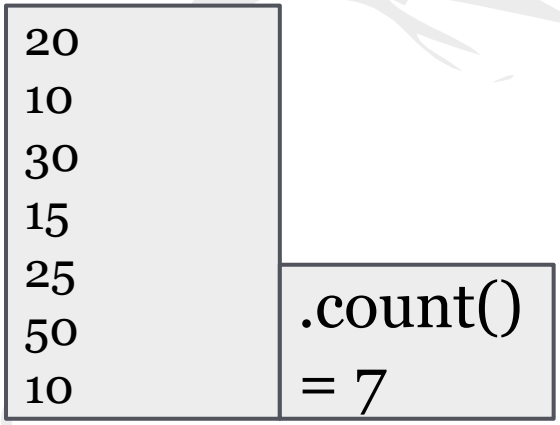
- lazy operators that create new RDDs from existing ones
  - map (f: T->U)
  - filter (f: T->Bool)
  - reduceByKey()
  - join()
  - union()
  - … (and lots of others)

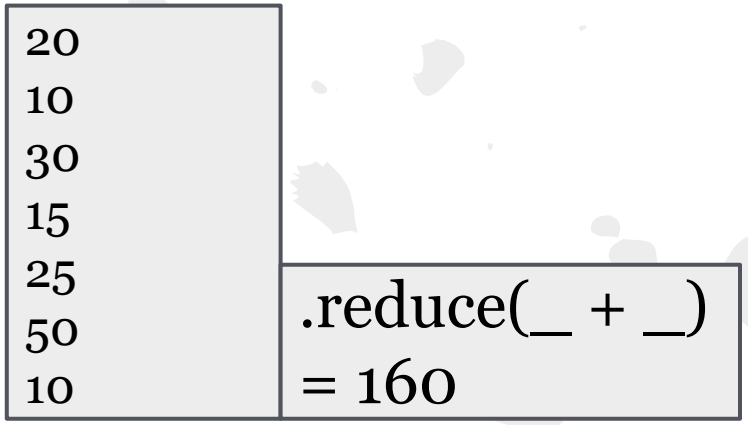




RDD Operations: Actions

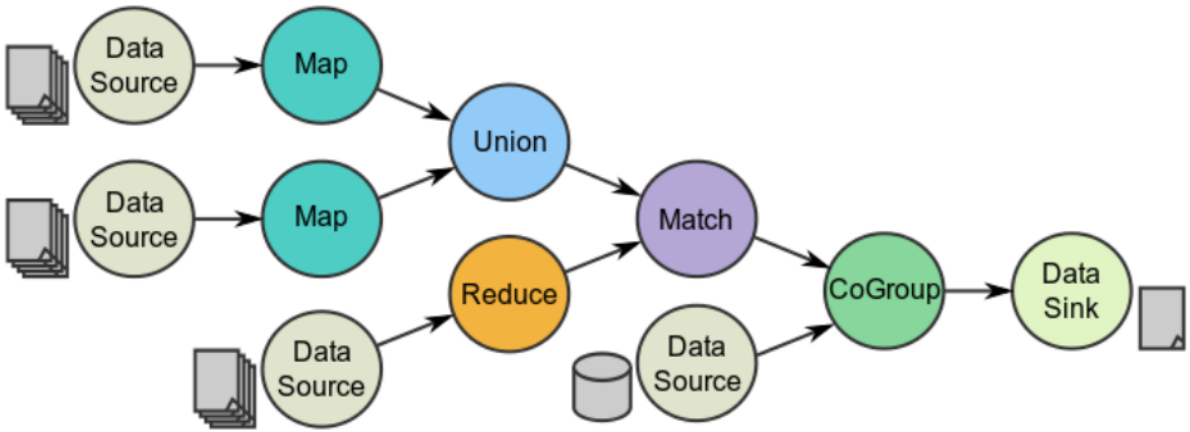
- launch a computation on a RDD and return a value to the program, or write data to the external storage
  - count()
  - reduce()
  - save(): actually saves an RDD to permanent storage
  - … (and lots of others)





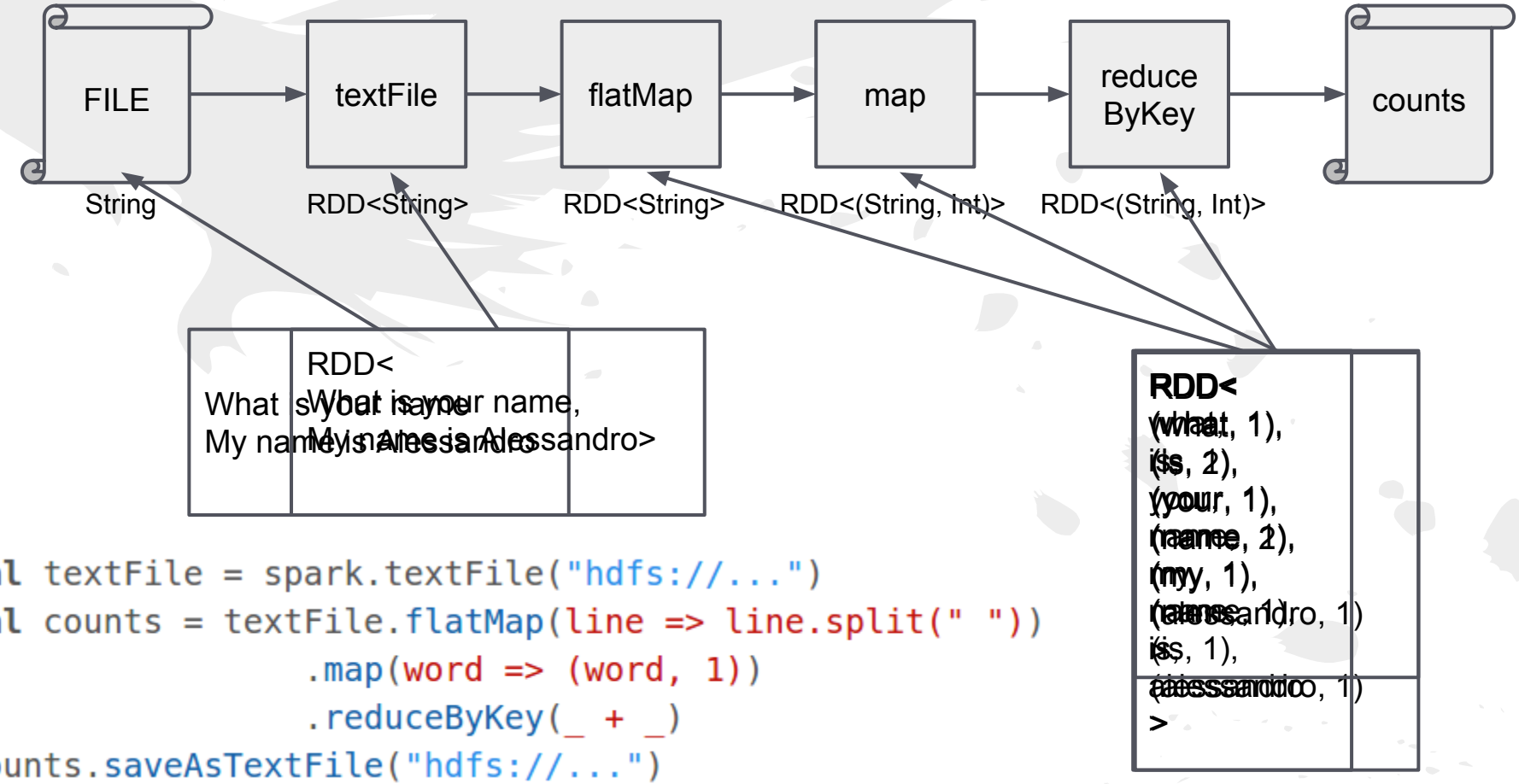
Programming Model

- based on parallelizable operators:
  - higher-order functions that execute user defined functions in parallel
- a data flow is composed of any number of data sources and operators
  - data sinks by connecting their inputs and outputs
  - job description based on directed acyclic graphs (DAG)



RDD Example: Word Count

- build a dataset of (String, Int) pairs called counts and then save it to a file

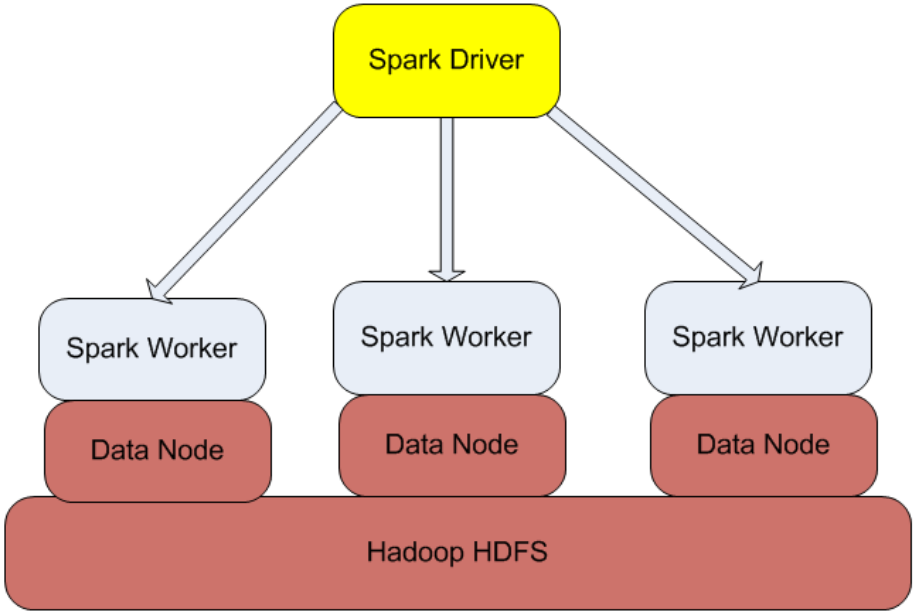


Spark

- implementation of the RDD abstraction
  - Scala interface
  - each RDD is represented as an object in Spark
- two components
  - Driver
  - Workers
- API available in Scala, Java and Python

Spark Runtime

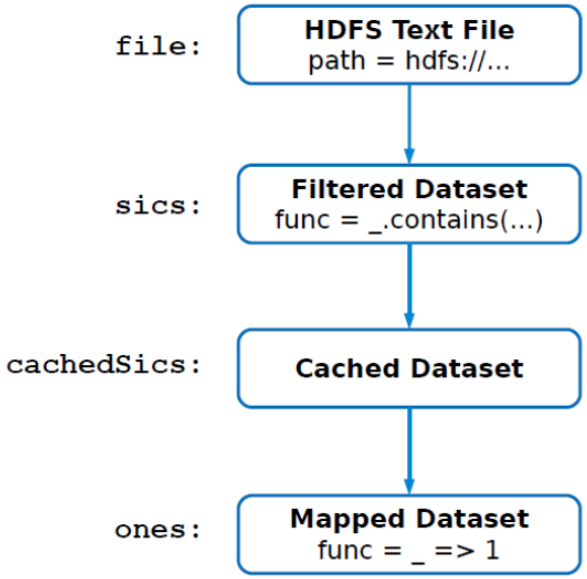
- Driver
  - defines and invokes actions on RDDs
  - tracks the RDDs' lineage
- Workers
  - store RDD partitions
  - perform RDD transformations



Lineage

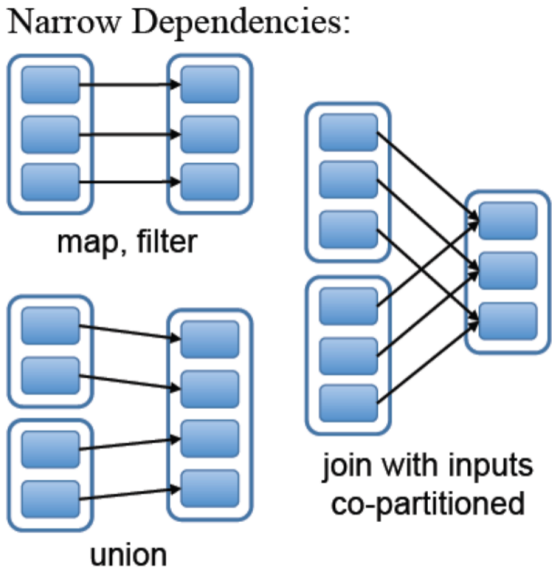
- it is the list of transformations (dependency) of a RDD
- how an RDD has been built from other RDDs or files
- transformations used to build an RDD
- RDDs are stored as a chain of objects capturing the lineage of each RDD
- important for job scheduling, fault tolerance and memory management

```
val file = sc.textFile("hdfs://...")
val sics = file.filter(_.contains("SICS"))
val cachedSics = sics.cache()
val ones = cachedSics.map(_ => 1)
val count = ones.reduce(_+_)
```

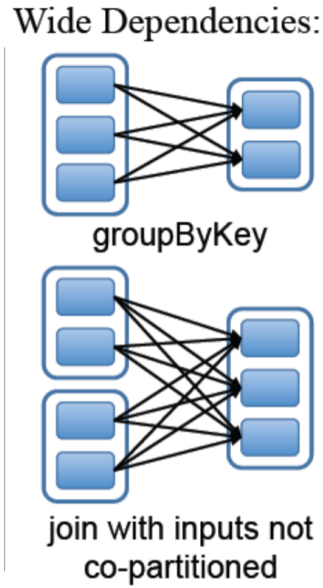


RDD Dependencies

- narrow dependencies
  - each partition of the parent RDD is used by at most one partition of the child RDD
  - allow for pipelined execution on one cluster node
  - easy fault recovery



- wide dependencies
  - multiple child partitions may depend on it
  - require data from all parent partitions to be available and to be shuffled across the nodes
  - a single failed node might cause a complete re-execution



Data Partitioning

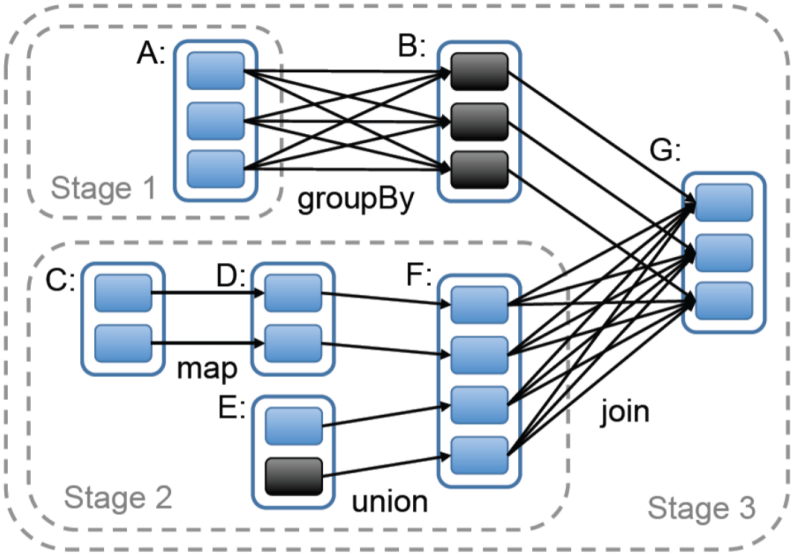
- users can decide how to assign data to partitions and the number of partitions
- but Spark decides how partitions are assigned to machines
  - no control where data are at runtime
  - spark adopts load balancing techniques

my what	name is
is your	ale name

Job Scheduling

- in order to execute an action on an RDD
  - scheduler decides the stages from the RDD's lineage graph
  - each stage contains as many pipelined transformations with narrow dependencies as possible

- tasks are assigned to machines based on data locality
- if a task needs a partition, which is available in the memory of a node, the task is sent to that node



RDD Fault Tolerance

- RDDs maintain lineage information that can be used to reconstruct lost partitions
- logging lineage rather than the actual data
  - no replication
- recompute only the lost partitions of an RDD
- recovery may be time-consuming for RDDs with long lineage chains and wide dependencies
- it can be helpful to checkpoint some RDDs to stable storage
- decision about which data to checkpoint is left to users

Checkpointing

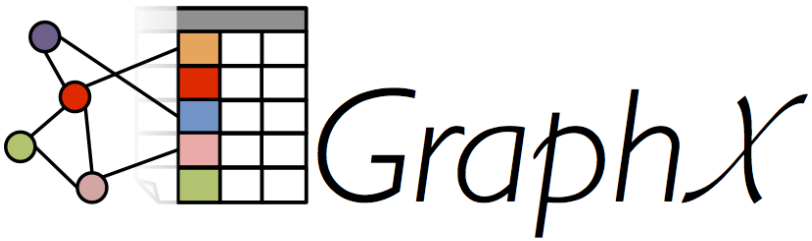
- BEWARE! checkpointing of RDDs is necessary to prevent long lineage chains during fault recovery
- low overhead and complexity than checkpointing of other shared memory structures
  - no need to synchronize
  - read-only, immutable nature of RDDs

Memory Management

- three options for persistent RDDs
  - in-memory storage as deserialized Java objects
  - in-memory storage as serialized data
  - on-disk storage
- more, bigger RDDs require memory management
- LRU eviction policy at the level of RDDs
  - when there is not enough memory, evict a partition from the least recently accessed RDD
  - possible thanks to the functional semantics and immutable nature of RDDs
  - exploit lineage to recompute discarded partitions which the program may need again

GraphX

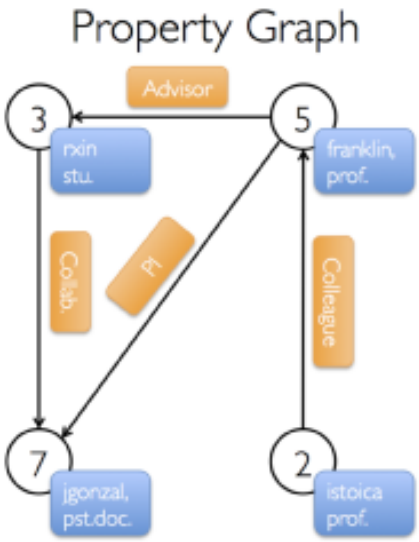
- it is implemented on top of Spark
- explore the design of graph processing systems on top of general-purpose distributed data-flow systems
- provides a new API collection targeting graph processing
  - enables composition of graphs with unstructured and tabular data
  - allows to view the same physical data both as a graph and as a collection without data movement or duplication



Data Model

- property graph
- graphs represented using two Spark RDDs:
  - edge collection: VertexRDD
  - vertex collection: EdgeRDD

```
// 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]
}
```



Vertex Table

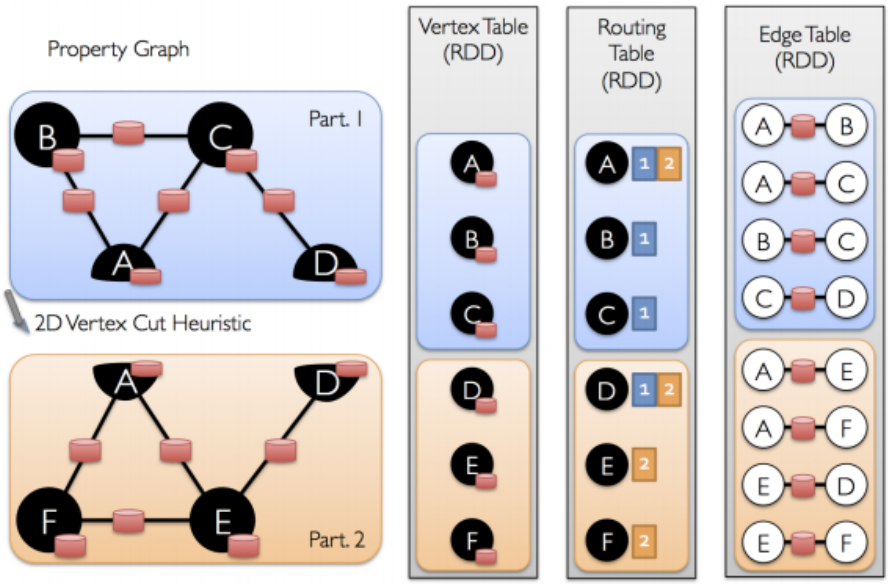
Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

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

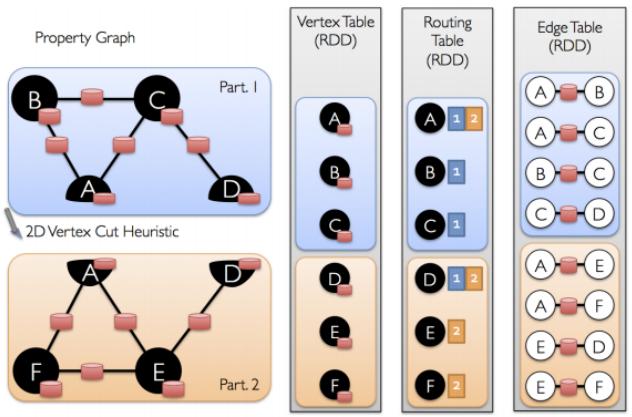
Edge Collection

- edges are split on different partitions
- different partitioning strategies can be applied
  - a few partitioning heuristics are predefined (4)
    - based on some hashing of the vertex identifier
  - more can be defined by the user
    - may need some vertex relabeling
    - getPartition(VertexId, VertexId, numParts): PartitionID



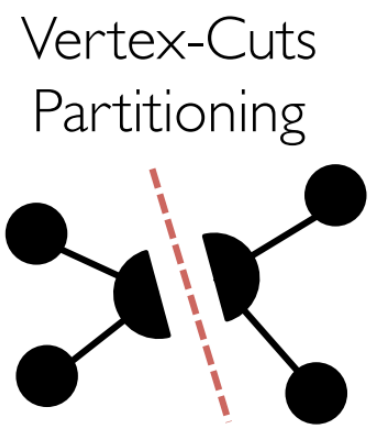
Vertex Collection

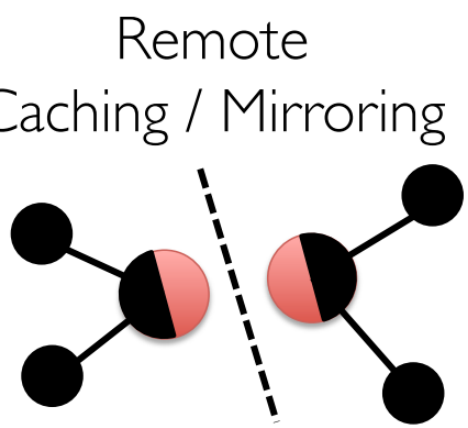
- *routing table*: a logical map from a vertex id to the set of edge partitions that contains adjacent edges
- *bitmask*: store the visibility of the vertices enabling soft deletion and index reuse
  - vertices bitmasks are updated after each operation (e.g. mapReduceTriplets).
  - vertices hidden by the bitmask do not participate in the graph operations (soft-deletion)
  - rebuild index structure is a costly operations, due to this RDD can reuse indices of previous RDD to rec
    and accelerate graph operations

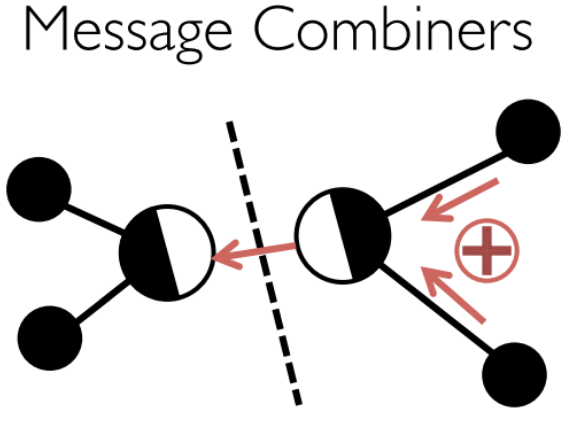


Vertex Cut

- vertex and edge collections partitioned independently
- edges belong to only one partition
- a vertex can have edges on different partitions, so
- vertex mirroring
  - vertex properties shipped across the network, to the edges
  - usually graphs have many more edges than vertices
    - cheaper than shipping edges
  - a vertex may have many edges in the same partition, enabling reuse of vertex property and message combiners





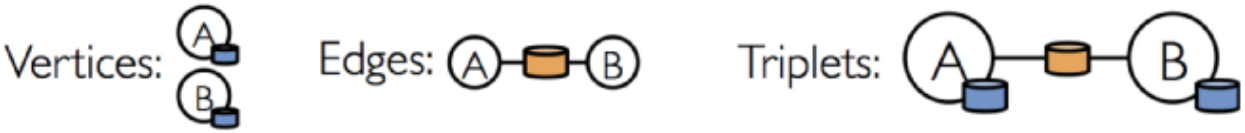


GAS Decomposition for the computation

- idea: most vertex programs interact with neighbouring vertices by collecting messages in the form of a generalized commutative associative sum
- three data parallel stages:
  - **Gather**: message to a vertex are merged using associative function i.e. (int a, int b) -> return a+b
  - **Apply**: from previous vertex state and the aggregated messages it is generated the new vertex state
  - **Scatter**: each edge produce message for the adjacent vertices based on vertices states
- prohibits direct communication between vertices that are not adjacent in the graph

Primitive Data Types: Triplet

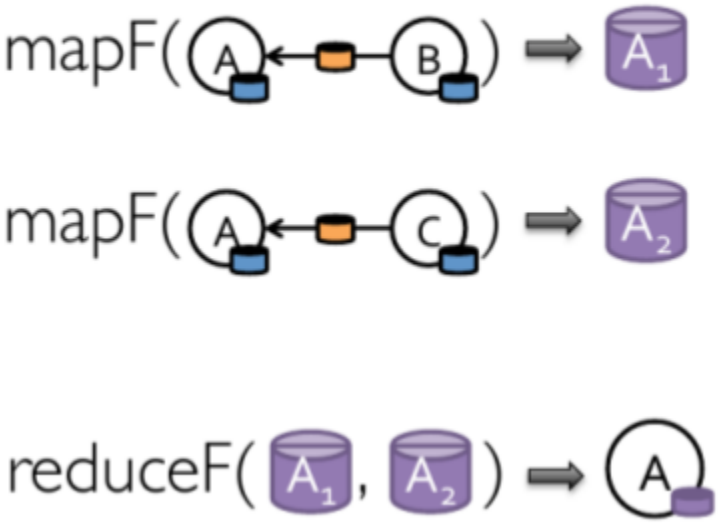
- EdgeTriplet represents an edge along with the vertex attributes of its neighboring vertices

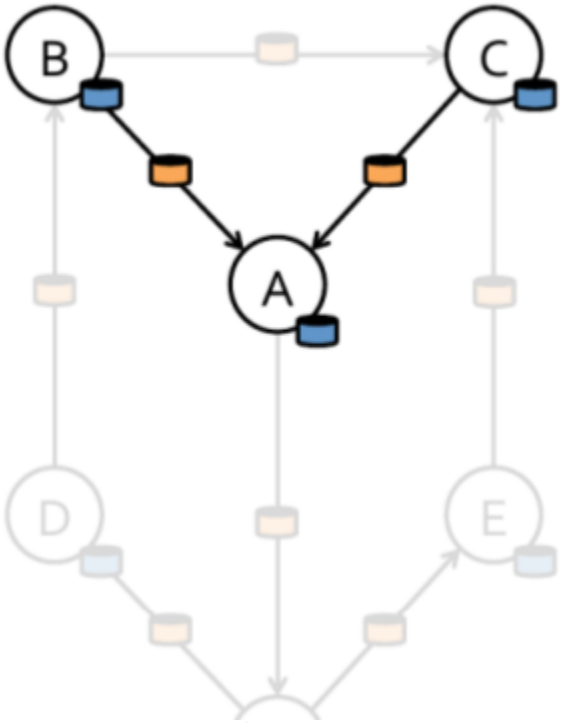



```
// Vertex collection
class VertexRDD[VD] extends RDD[(VertexId, VD)]

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

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

Map Reduce Triplets

- Map Reduce for each vertex
- Map is the action during the *Scatter* phase
- Reduce is the action during the *Gather* phase





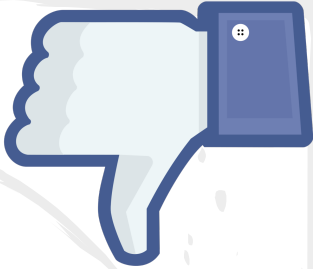
My Experience: Connected Components

- what is:
  - a CC is a subset of the vertices where there is a path between any pair of such vertices
- two algorithms
  - hashMin: each node propagates to neighbours the minimum known node identifier until no changes
  - cracker: the graph is simplified until only one vertex per CC remains (the seed), after the seed id is propagated to all the CC
- implementations:
  - Spark MapReduce
  - Spark GraphX

Spark MapReduce UP

- vertex-centric approach (both)
  - think from the point of view of the vertex
  - communicate with neighborhood
- neighborhood can easily change (cracker)
  - just have a Set of your neighbours identifier and modify the Set
  - also number of vertices can change
- explicit message passing seems to have control (both)

Spark MapReduce DOWN

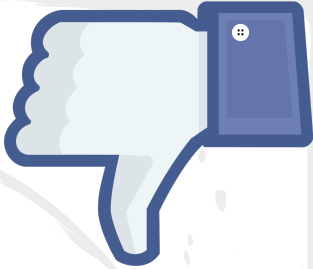


- the vertex state must be re-sent between iterations (both)
  - no automatic vertex state saving
- curse of the last reducer (both)
  - high degree vertices computation usually require a lot more time than low degree vertices

Spark GraphX UP

- vertex state is provided to you from the framework (both)
  - easy to change it
- in general GAS model (and vertex cut) improve performances on high degree vertices (both)

Spark GraphX DOWN



- it is not easy to think with the GAS model if you are used to vertex-centric (both)
- neighborhood can NOT change, although there exist tricky operations that require a full vertexRDD indices rebuilding (cracker)
  - the same apply to vertices inclusion / removal