COMP9313: Big Data Management

Lecturer: Xin Cao

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

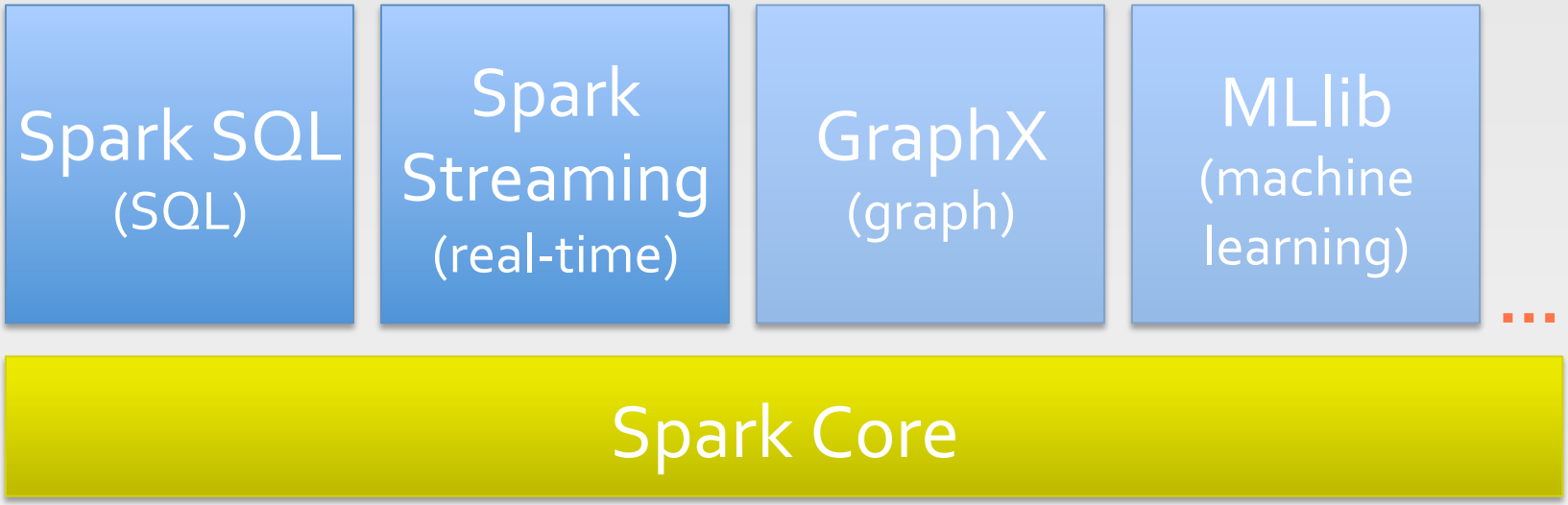
Chapter 7: Spark II

Review of Chapter 6

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

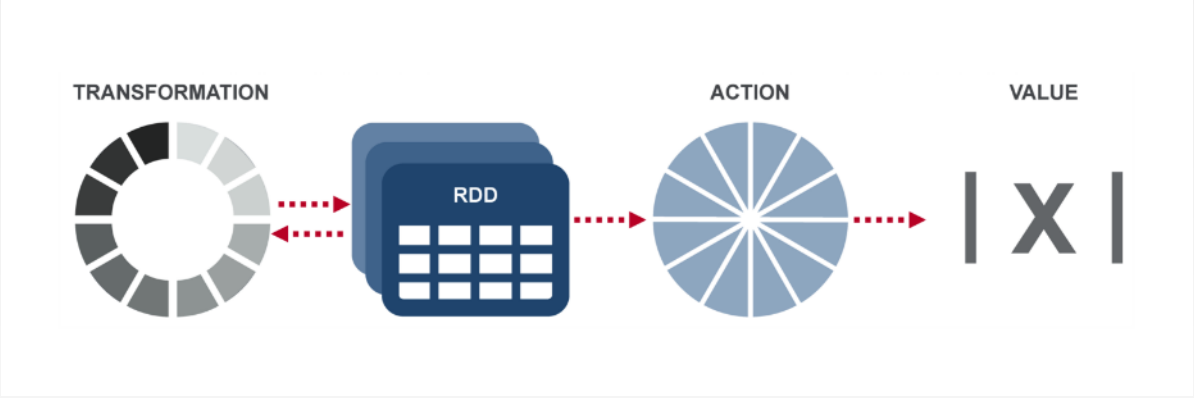
Spark Components



- 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



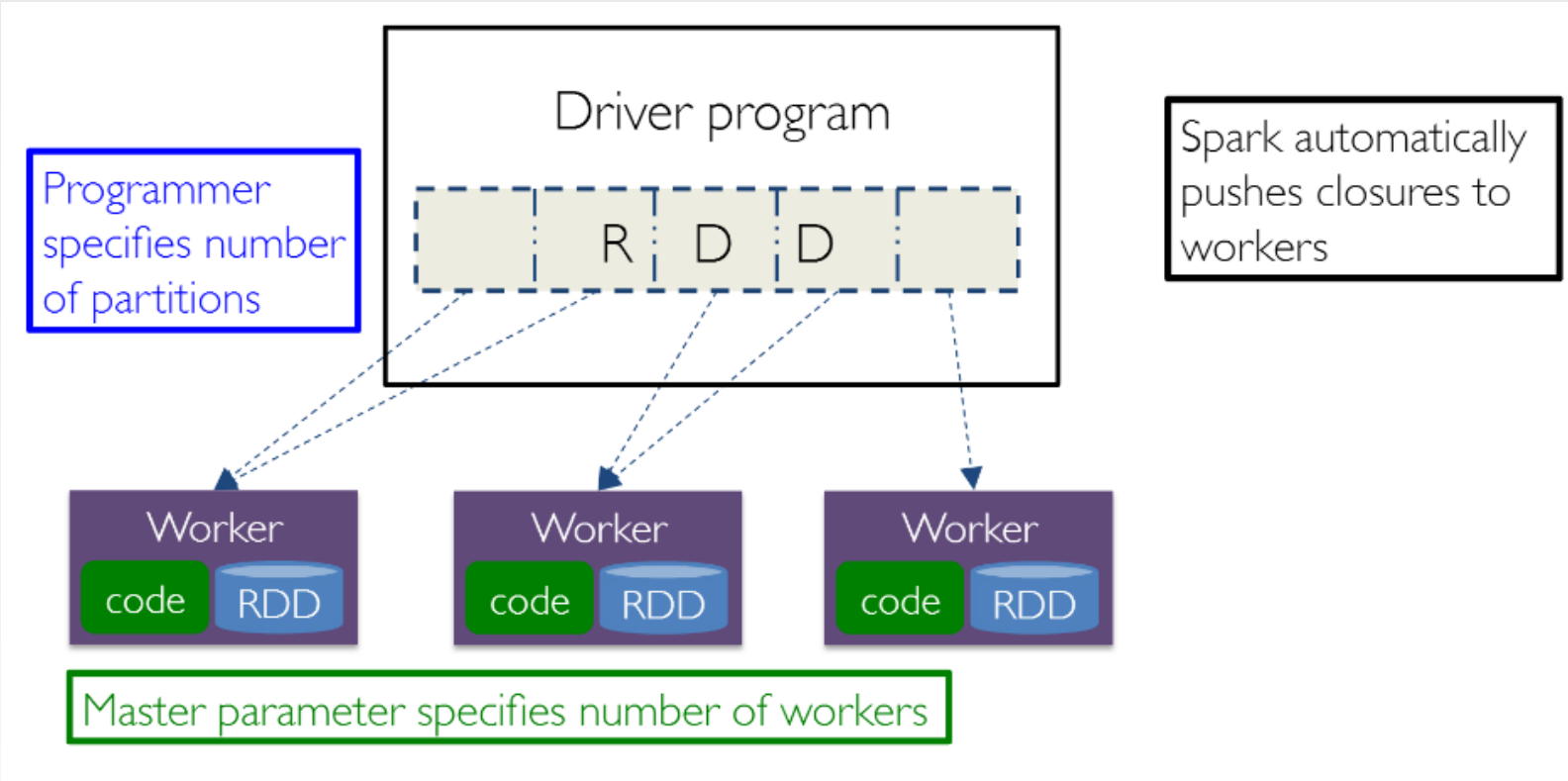
Pair RDD

- RDDs of key/value pairs
- Spark provides special operations on pair RDDs
- Creating pair RDD
  - val pairs = lines.map(x => (x.split(" ")(0), x))
- Transformations on pair RDDs (example: {(1, 2), (3, 4), (3, 6)})
  - reduceByKey(_+_), {(1, 2), (3, 10)}
  - groupByKey, {(1, [2]), (3, [4, 6])}
  - mapValues(x=>x+1), {(1, 3), (3, 5), (3, 7)}
  - flatMapValues(x=>(x to 5)), {(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}
- Actions on pair RDDs (example: {(1, 2), (3, 4), (3, 6)})
  - countByKey(), {(1, 1), (3, 2)}
  - collectAsMap(), Map{(1, 2), (3, 4), (3, 6)}
  - lookup(3), [4, 6]

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

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
```

- Creating a Spark Context Object
  - 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

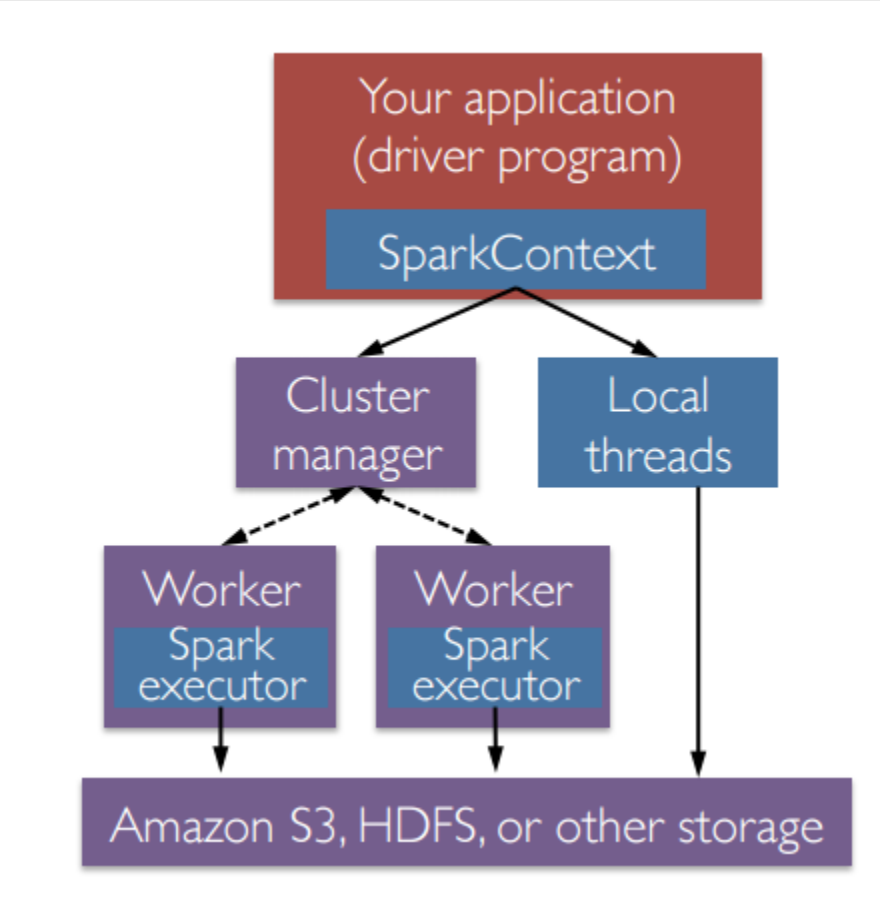
```
val input =  sc.textFile(inputFile)
```

- 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)
```

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" % "2.3.0"
```

  - 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

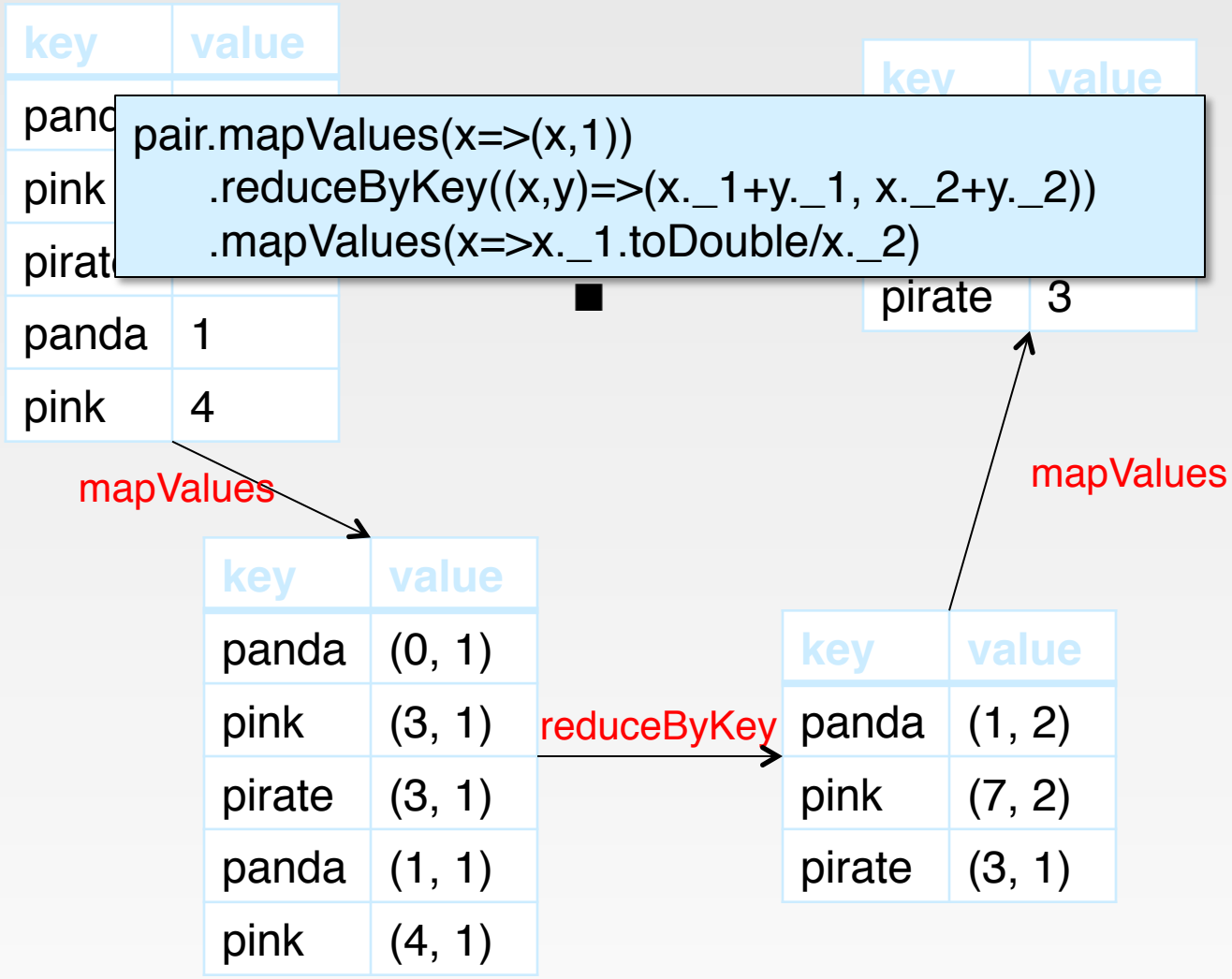
- spark-submit

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"

Practice

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



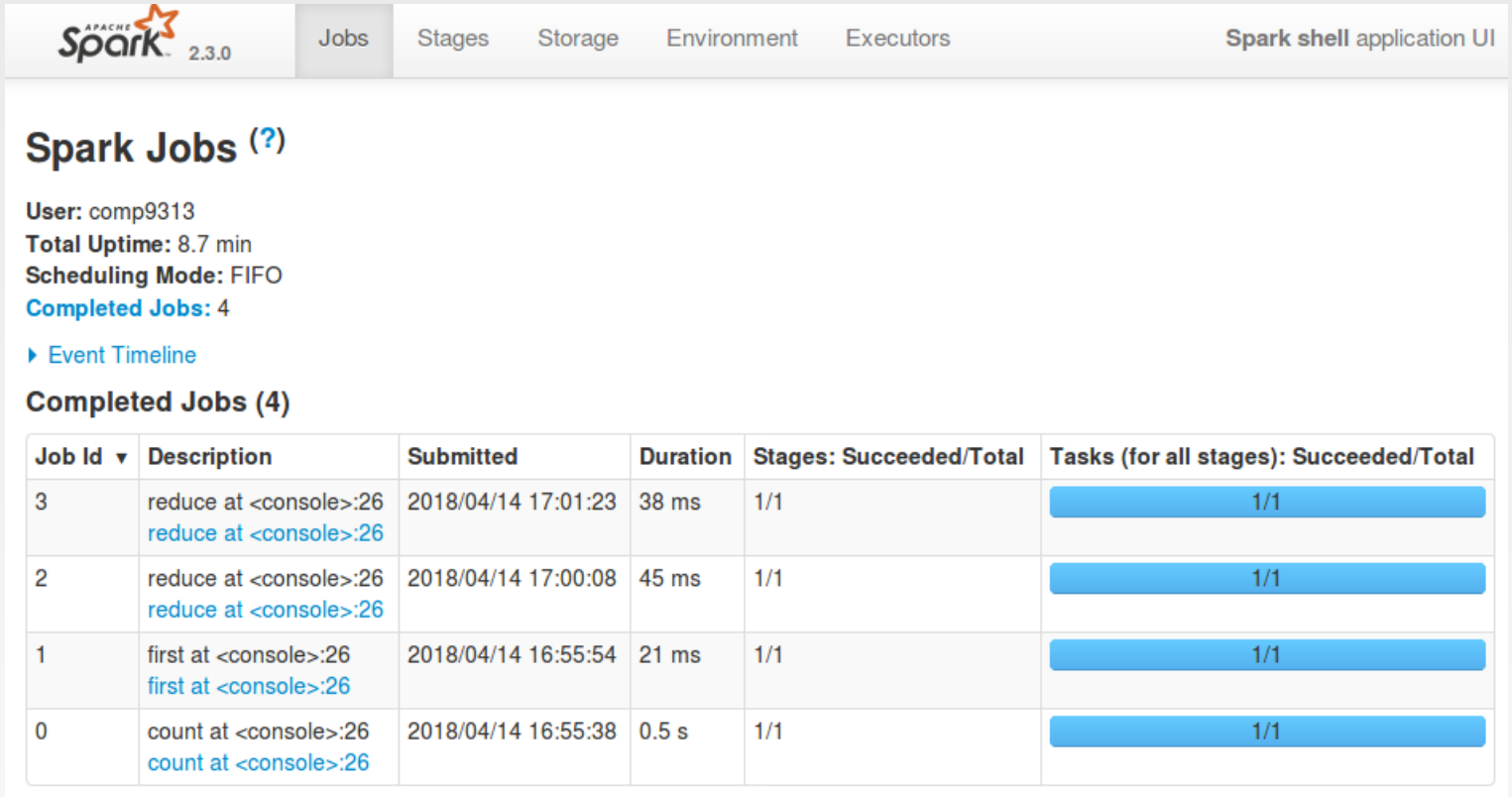
Practice

- Problem 2: 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))
```

Spark Web Console

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



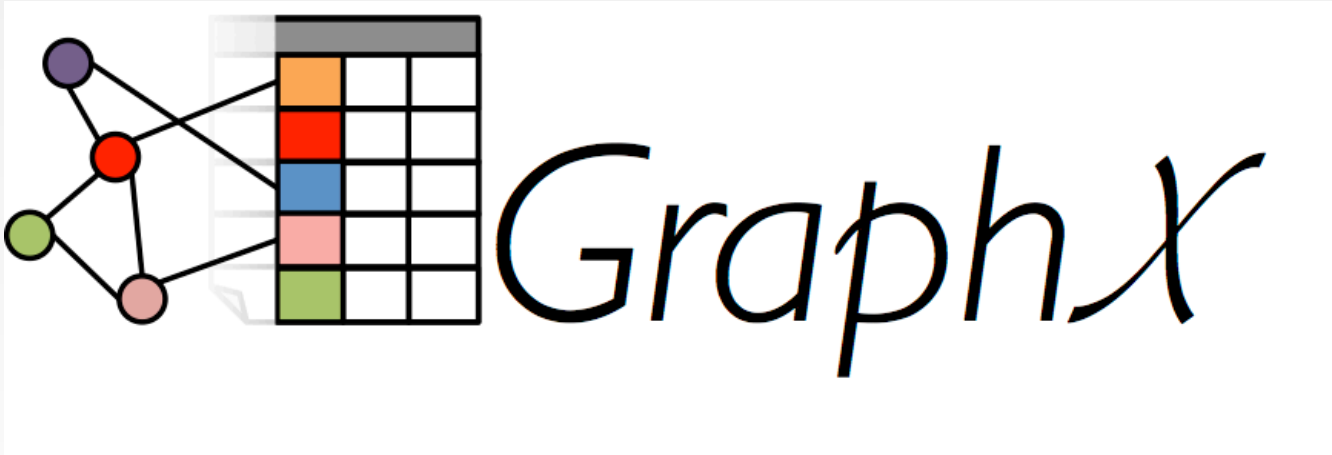
In-Memory Can Make a Big Difference

- Two iterative Machine Learning algorithms:

Part 3: Spark GraphX

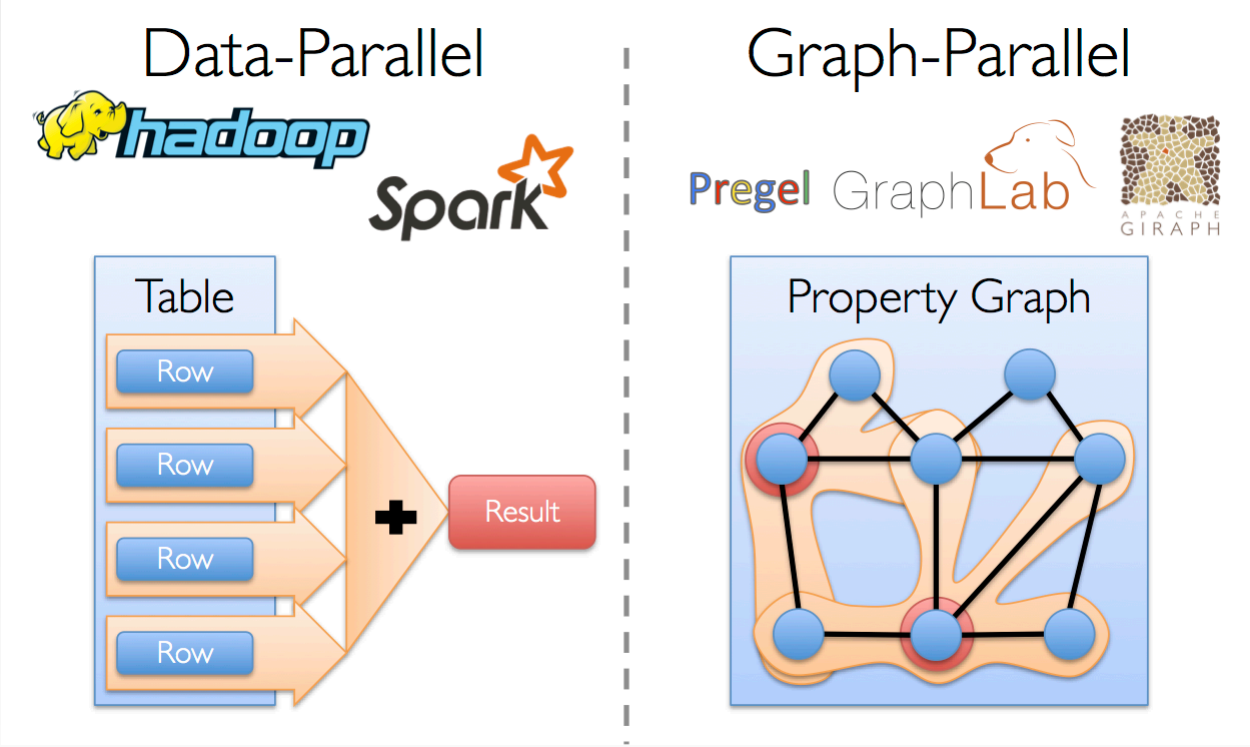
Spark GraphX

- **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.



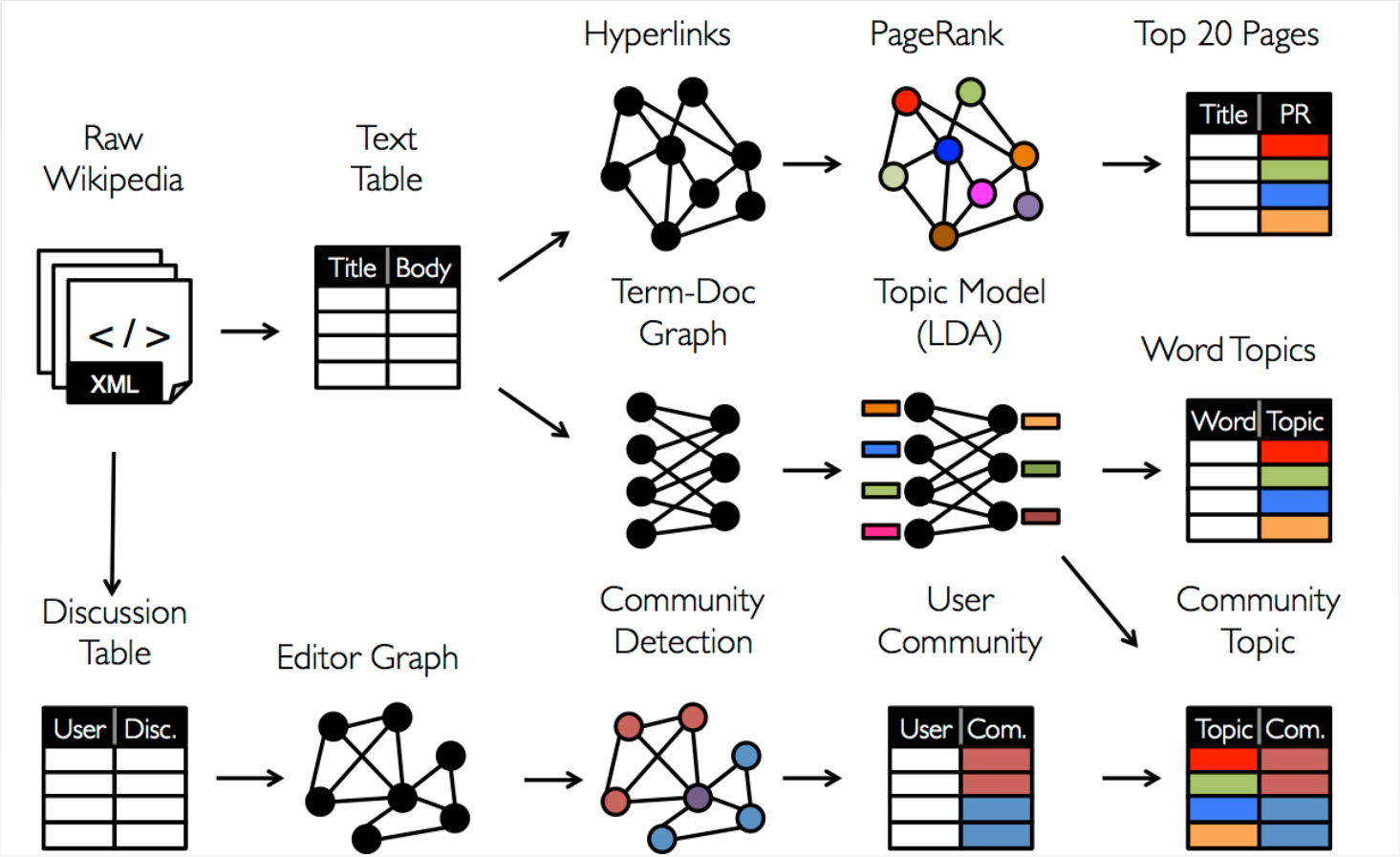
Graph-Parallel Computation

- 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



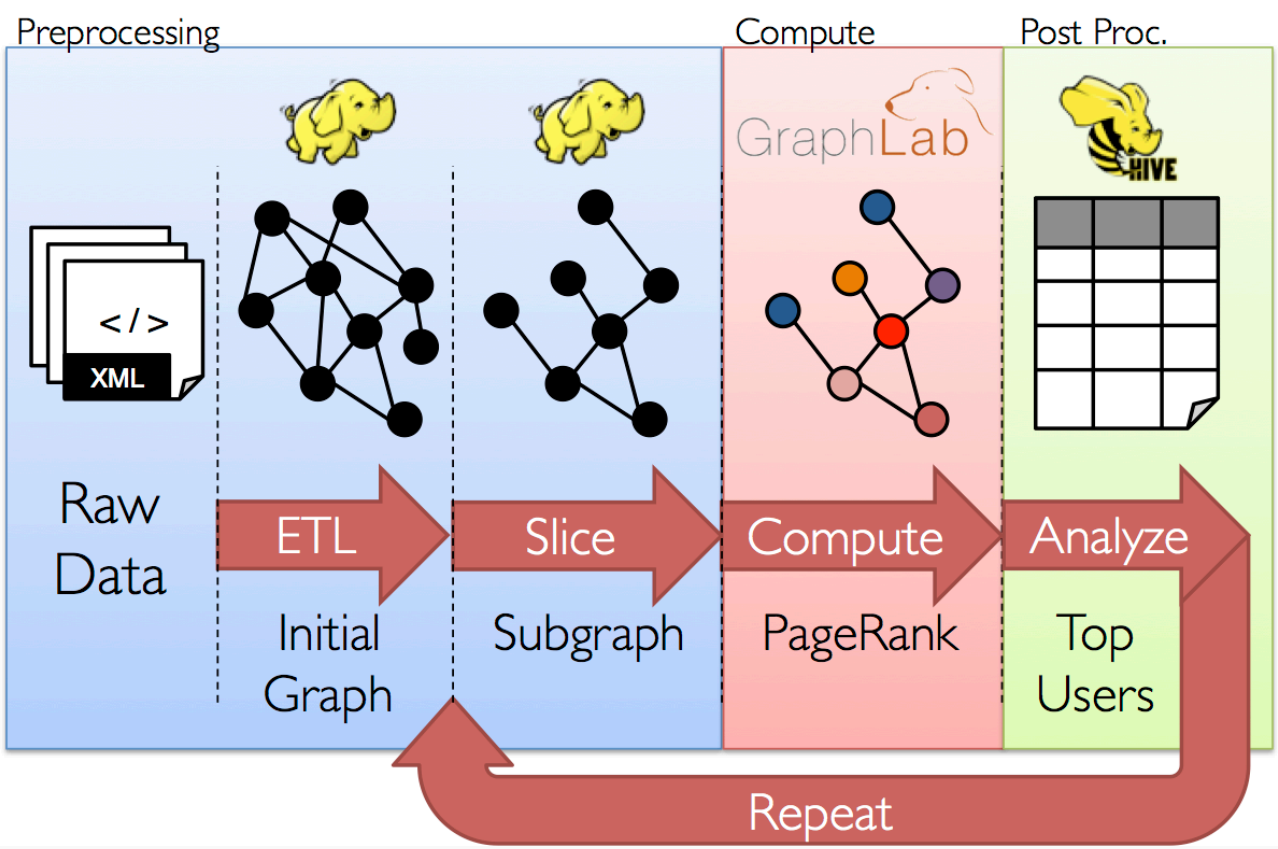
Specialized Systems May Miss the Bigger Picture

- 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



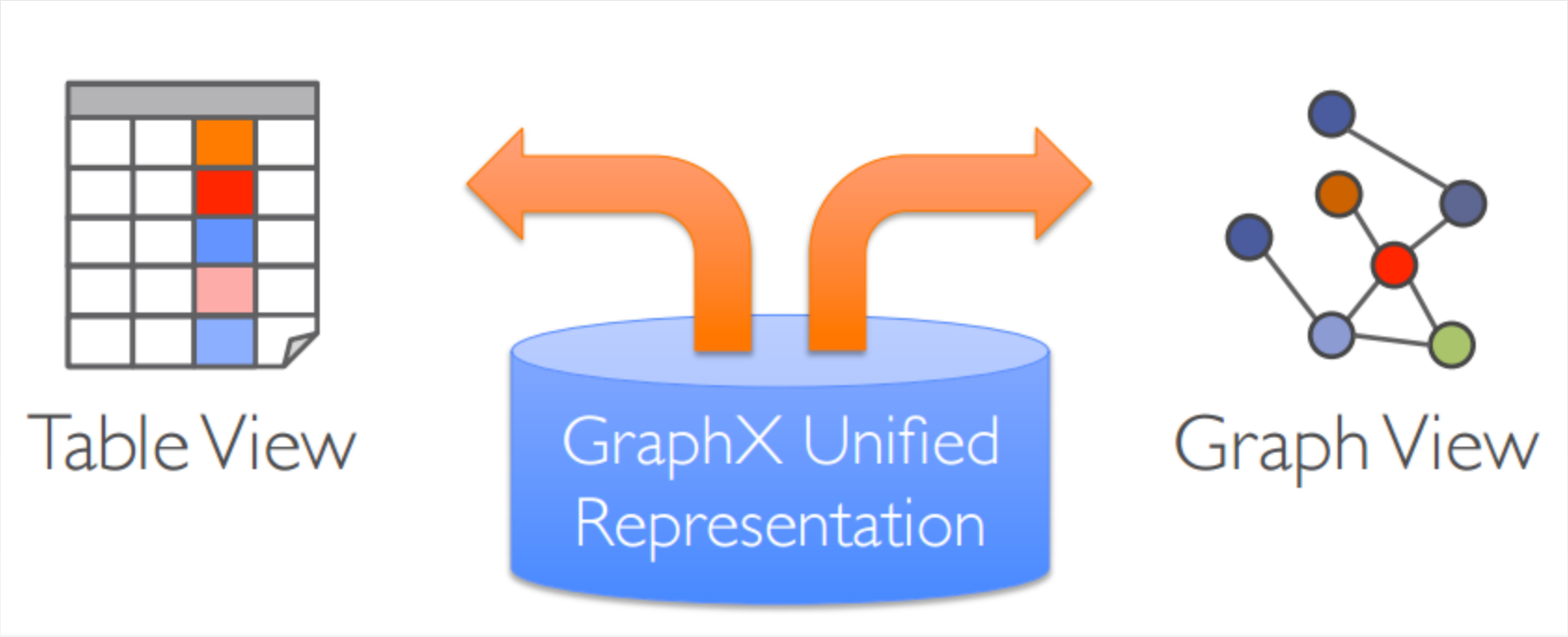
GraphX Motivation

- 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

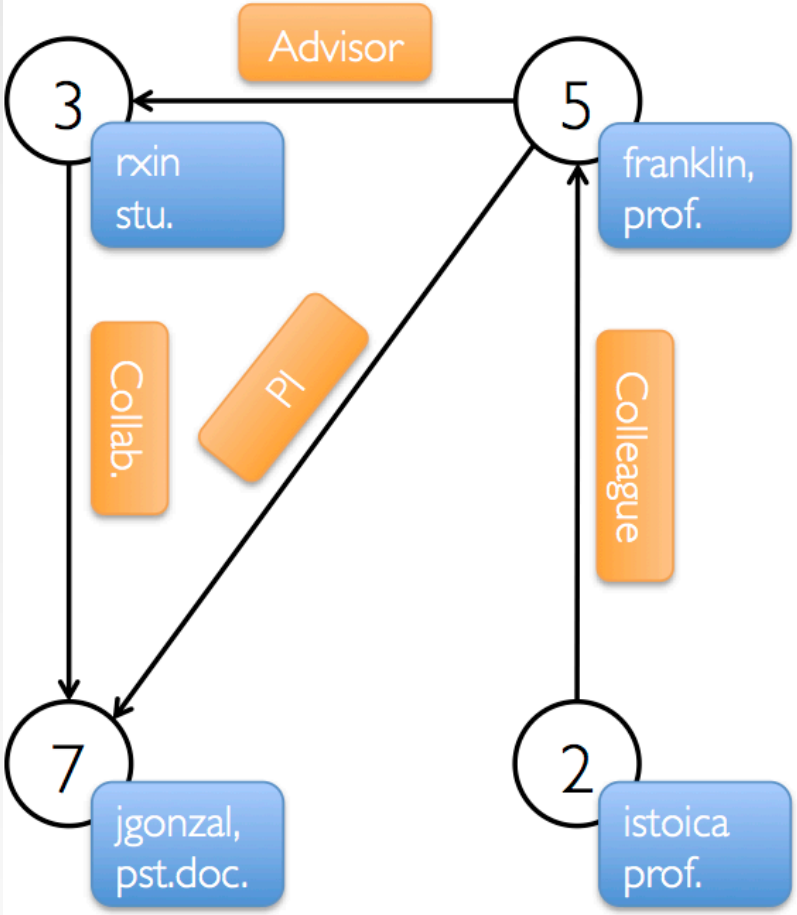
- Tables and Graphs are composable views of the same physical data



- 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

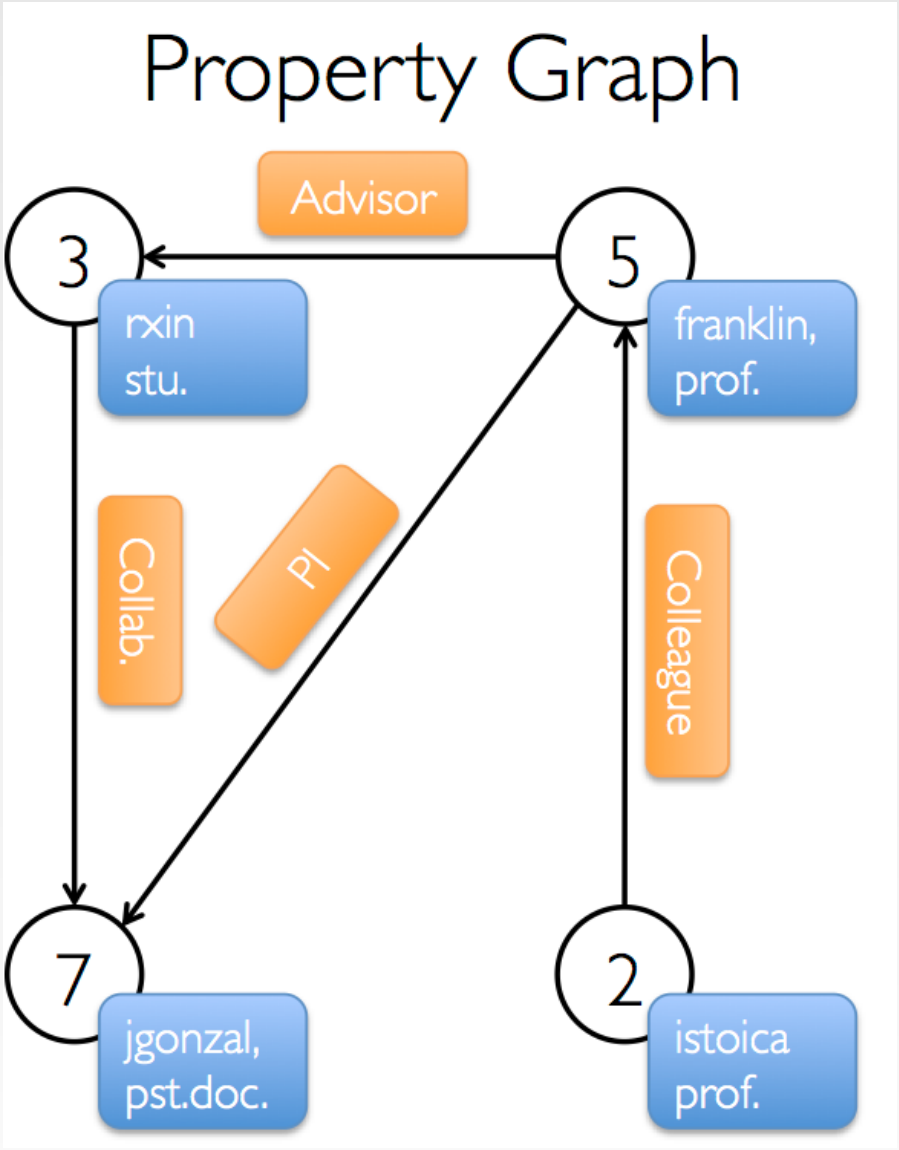
```
class Graph[VD, ED] {
        // Information about the Graph
        val numEdges: Long
        val numVertices: Long
        val inDegrees: VertexRDD[Int]
        val outDegrees: VertexRDD[Int]
        val degrees: VertexRDD[Int]
        // Views of the graph as collections
        val vertices: 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



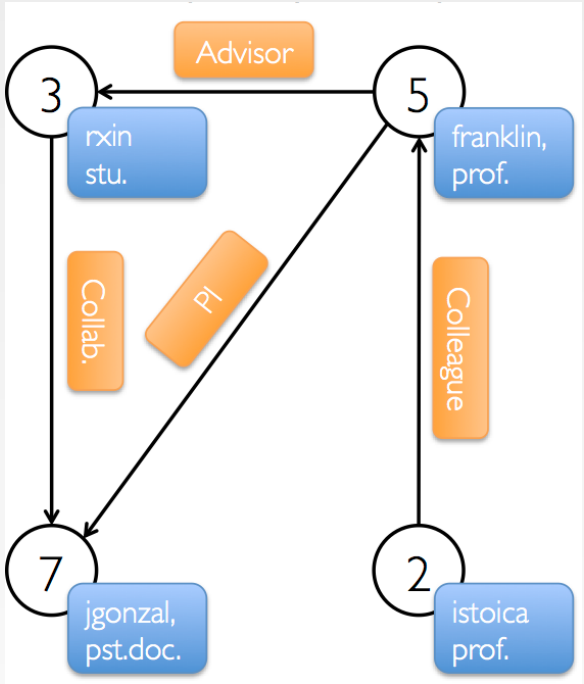
GraphX Example

- 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

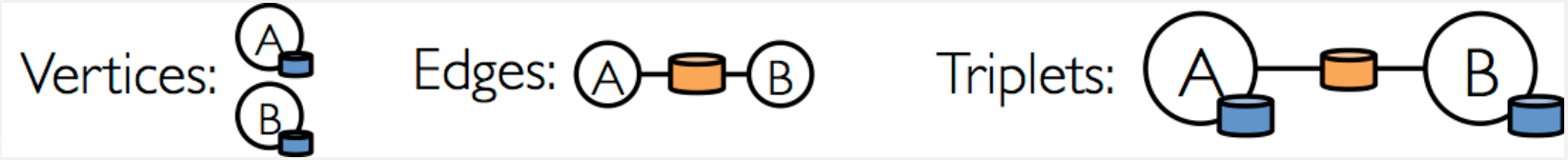
```
graph.vertices.filter {
  case (id, (name, pos)) => pos == “prof”
}.collect.foreach {
  case (id, (name, age)) => println(s"$name is Professor")
}
```

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

- Bulk Synchronous Parallel Model (BSP)
  - 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)

Vertex-centric, message passing

- The Pregel operator in GraphX is a BSP messaging abstraction *constrained to the topology of the graph.*

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)

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
)
```

Part 4: Spark SQL and DataFrames

Spark SQL Overview

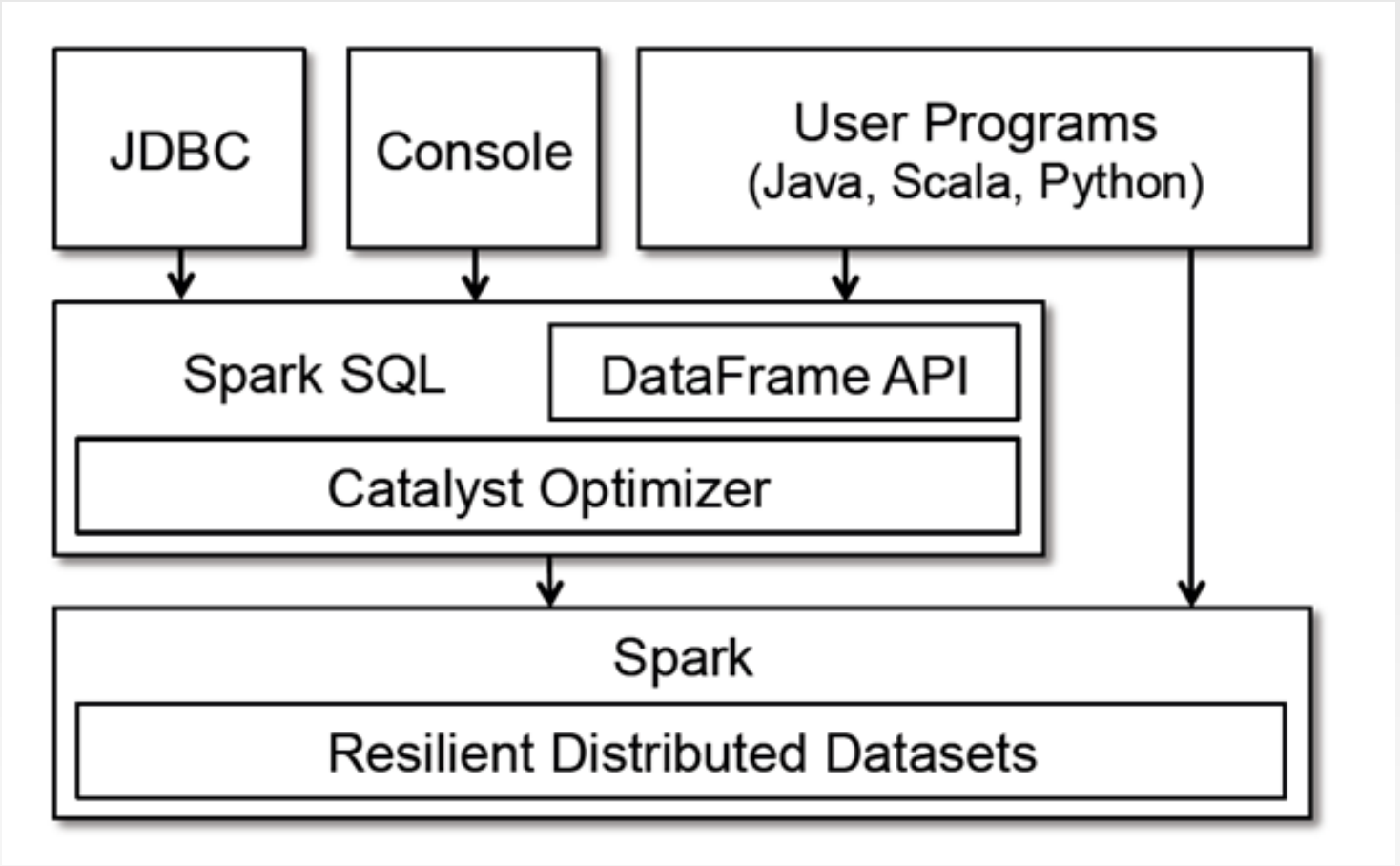
- Part of the core distribution since Spark 1.0 (April 2014)
- Tightly integrated way to work with structured data (tables with rows /columns)
- Transform RDDs using SQL
- Data source integration: Hive, Parquet, JSON, and more

- Spark SQL is **not** about SQL.
  - Aims to Create and Run Spark Programs Faster:

Relationship to Shark

- Shark has been subsumed by Spark SQL
- Shark modified the Hive backend to run over Spark, but had two challenges:
  - Limited integration with Spark programs
  - Hive optimizer not designed for Spark
- Spark SQL reuses the best parts of Shark:
  - Borrows
    - Hive data loading
    - In-memory column store
  - Adds
    - RDD-aware optimizer
    - Rich language interfaces

Spark Programming Interface



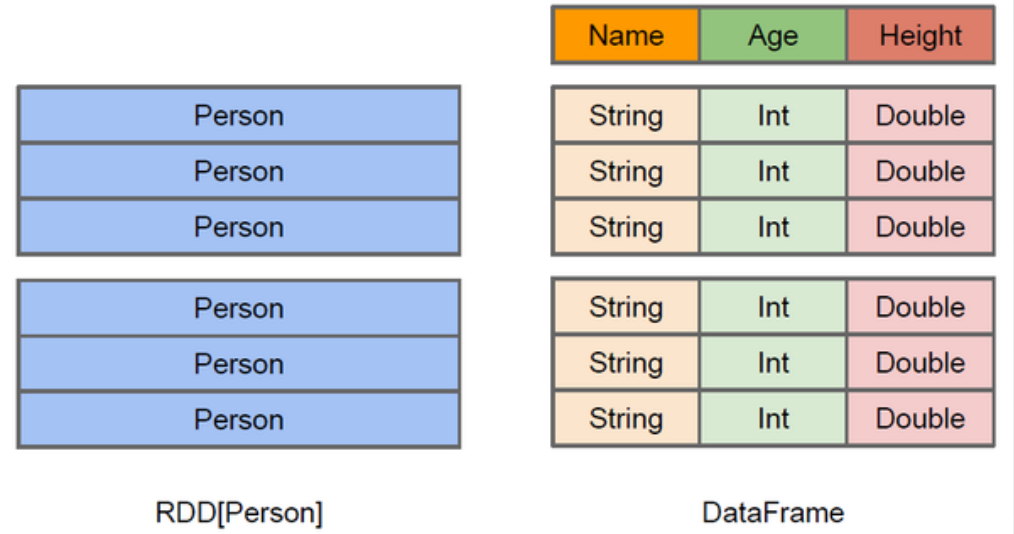
Datasets and DataFrames

- A *Dataset* is a distributed collection of data
  - provides the benefits of RDDs (e.g., strong typing) with the benefits of Spark SQL's optimized execution engine
  - A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, etc.)
  - The Dataset API is available in Scala and Java

- A *DataFrame* is a *Dataset* organized into named columns
  - It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations
  - An abstraction for selecting, filtering, aggregating and plotting structured data
  - In Scala and Java, a DataFrame is represented by a Dataset of Rows
    - Scala: DataFrame is simply a type alias of Dataset[Row]
    - Java: use Dataset<Row> to represent a DataFrame

Difference between DataFrame and RDD

- DataFrame more like a traditional database of two-dimensional form, in addition to data, but also to grasp the structural information of the data, that is, schema



  - RDD[Person] although with Person for type parameters, but the Spark framework itself does not understand internal structure of Person class
  - DataFrame has provided a detailed structural information, making Spark SQL can clearly know what columns are included in the dataset, and what is the name and type of each column. Thus Spark SQL query optimizer can target optimization

DataFrame Data Sources

- Spark SQL's Data Source API can read and write DataFrames using a variety of formats.
  - E.g., structured data files, tables in Hive, external databases, or existing RDDs
  - In the Scala API, DataFrame is simply a type alias of Dataset[Row]

Starting Point: SparkSession

- The entry point into all functionality in Spark is the SparkSession class.

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

  - SparkSession in Spark 2.0 provides built-in support for Hive features including the ability to write queries using HiveQL, access to Hive UDFs, and the ability to read data from Hive tables

Creating DataFrames

- With a SparkSession, applications can create DataFrames from *an existing RDD*, from *a Hive table*, or from *Spark data sources*.
  - creates a DataFrame based on the content of a JSON file:

```
val df = spark.read.json("examples/src/main/resources
/people.json")

// Displays the content of the DataFrame to stdout

df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

DataFrame Operations

- DataFrames are just Dataset of Rows in Scala and Java API.
  - These operations are also referred as “untyped transformations” in contrast to “typed transformations” come with strongly typed Scala/Java Datasets

```
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+
```

DataFrame Operations

```
// Select everybody, but increment the age by 1
df.select($"name", $"age" + 1).show()
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter($"age" > 21).show()
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+

// Count people by age
df.groupBy("age").count().show()
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

Running SQL Queries Programmatically

- The sql function on a SparkSession enables applications to run SQL queries programmatically and returns the result as a DataFrame.

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

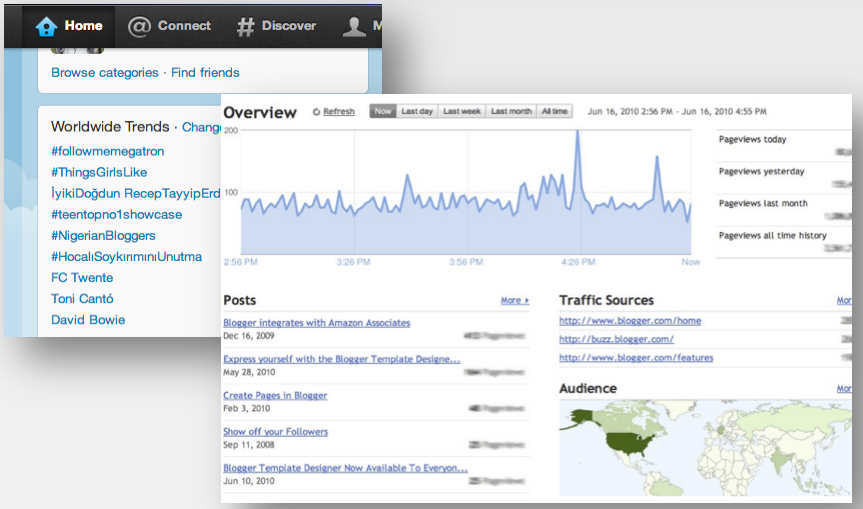
- Find full example code at

  https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala

Part 5: Spark Streaming

Motivation

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Ad impressions
  - … …



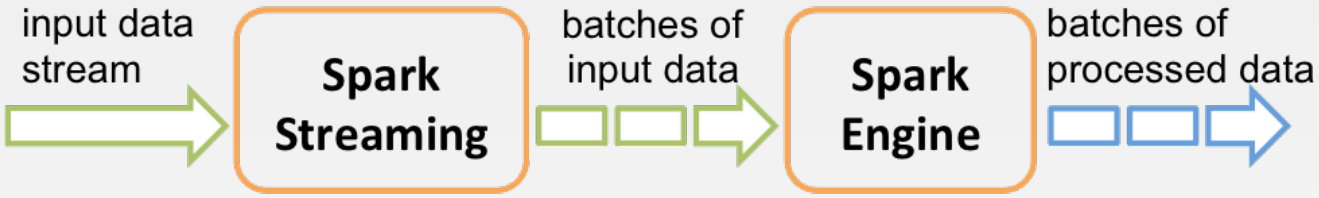
- Distributed stream processing framework is required to
  - Scale to large clusters (100s of machines)
  - Achieve low latency (few seconds)

What is Spark Streaming

- Spark Streaming is an extension of the core Spark API that enables **scalable**, **high-throughput**, **fault-tolerant** stream processing of live data streams
- Receive data streams from input sources, process them in a cluster, push out to filesystems, databases, and live dashboards
  - Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets
  - Data can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window

How does Spark Streaming Work

- 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



Spark Streaming Programming Model

- Spark Streaming provides a high-level abstraction called *discretized stream* (*Dstream*)
  - Represents a continuous stream of data.
  - DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams.
  - Internally, a DStream is represented as a sequence of RDDs.

- DStreams API very similar to RDD API
  - Functional APIs in Scala, Java
  - Create input DStreams from different sources
  - Apply parallel operations

An Example: Streaming WordCount

- Use StreamingContext, rather then SparkContext

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

object NetworkWordCount {
         val conf = new
         SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
         val ssc = new StreamingContext(conf, Seconds(10))

         val lines = ssc.socketTextStream("localhost", 9999)
         val words = lines.flatMap(_.split(" "))
         val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
         wordCounts.print()
         ssc.start()
         ssc.awaitTermination()
}
```

Streaming WordCount

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

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

- Create a local StreamingContext with two working thread and batch interval of 10 second.

```
val conf = new

SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(10))
```

  - A **StreamingContext** object has to be created which is the main entry point of all Spark Streaming functionality.
  - At least two local threads must be used (two cores)
  - Do the count for each 10 seconds
    - The batch interval must be set based on the latency requirements of your application and available cluster resources

Streaming WordCount

- After a streaming context is defined, you have to do the following:
  - Define the input sources by creating input DStreams.
  - Define the streaming computations by applying transformation and output operations to DStreams.
  - Start receiving data and processing it using streamingContext.start().
  - Wait for the processing to be stopped (manually or due to any error) using streamingContext.awaitTermination().
  - The processing can be manually stopped using streamingContext.stop().

Streaming WordCount

- Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. localhost) and port (e.g. 9999).

```
val lines = ssc.socketTextStream("localhost", 9999)
```

  - This lines DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text.

- Split the lines by space characters into words and do the count

```
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
```

- No real processing has started yet now. To start the processing after all the transformations have been setup, we finally call

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

- The complete code can be found in the Spark Streaming example NetworkWordCount.

Linking the Application

- Add the following dependency to your SBT configuration:
  - libraryDependencies += "org.apache.spark" % "spark-streaming_2.11" % "2.3.0"

- For data sources like Kafka, Flume, and Kinesis that are not present in the Spark Streaming core API, you will have to add the corresponding artifact spark-streaming-xyz_2.11 to the dependencies

Kafka	spark-streaming-kafka-0-8_2.11
Flume	spark-streaming-flume_2.11
Kinesis	spark-streaming-kinesis-asl_2.11 [Amazon Software License]

Run Streaming WordCount

- First need to run Netcat (a small utility found in most Unix-like systems) as a data server by using:

  $ nc -lk 9999

- sbt configuration file:

```
name := “Network WordCount"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark
-streaming" % "2.3.0"
```

- Then, in a different terminal, you can start the example by using

  $ spark-submit --class NetworkWordCount ~/sparkapp/target /scala-2.11/network-wordcount_2.11-1.0.jar

Results of Streaming WordCount

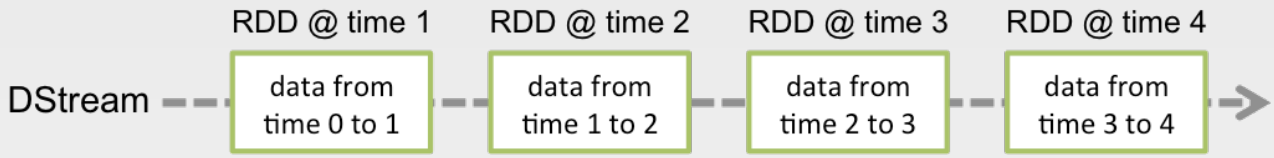
```
comp9313@comp9313-VirtualBox:~$ nc -lk 9999
hello world hello
world world
hello hello
```

```
-------------------------------------------
Time: 1493001960000 ms
-------------------------------------------


-------------------------------------------
Time: 1493001970000 ms
-------------------------------------------
(hello,2)
(world,1)

-------------------------------------------
Time: 1493001980000 ms
-------------------------------------------
(hello,2)
(world,2)
```

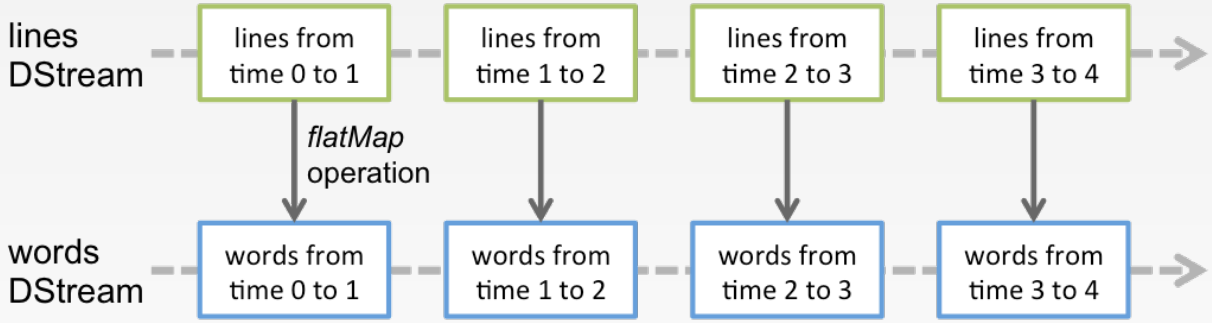
- The first 10 seconds, receives no data
- The next 10 seconds, receives one line
- The last 10 seconds, receives two lines

Discretized Streams (DStreams)

- A DStream is represented by a continuous series of RDDs
  - Each RDD in a DStream contains data from a certain interval, as shown in the following figure.



- Any operation applied on a DStream translates to operations on the underlying RDDs.
  - in the earlier example of converting a stream of **lines** to **words**, the flatMap operation is applied on each RDD in the **lines** DStream to generate the RDDs of the **words** DStream



Input DStreams and Receivers

- Input DStreams are DStreams representing the stream of input data received from streaming sources.
  - E.g., **lines** was an input DStream as it represented the stream of data received from the netcat server
- Every input DStream is associated with a **Receiver** object which receives the data from a source and stores it in Spark's memory for processing
- Spark Streaming provides two categories of built-in streaming sources.
  - *Basic sources*: Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
  - *Advanced sources*: Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes. These require linking against extra dependencies

Input DStreams and Receivers

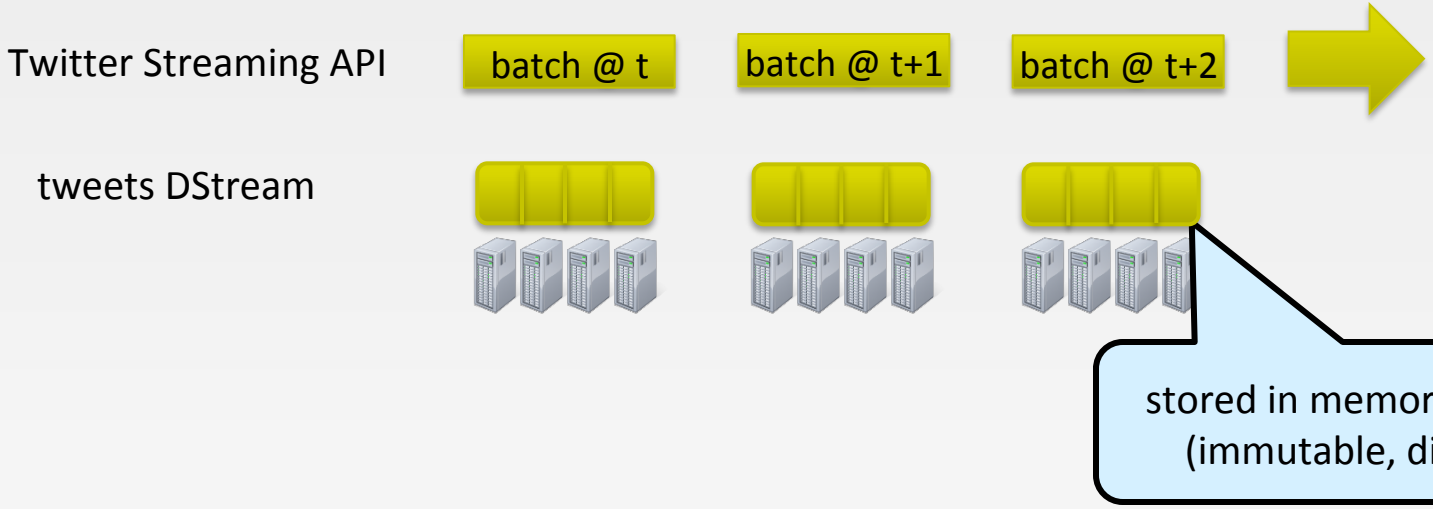
- When running a Spark Streaming program locally, do not use “local” or “local[1]” as the master URL
  - If you are using an input DStream based on a receiver (e.g., sockets), then the single thread will be used to run the receiver, leaving no thread for processing the received data
  - When running locally, always use “local[*n*]” as the master URL, where *n* > number of receivers to run

- The number of cores allocated to the Spark Streaming application must be more than the number of receivers. Otherwise the system will only receive data, but not be able to process it

Example – Get hashtags from Twitter

```
val ssc = new StreamingContext(conf, Seconds(10))
val tweets :DStream[Status] = TwitterUtils.createStream(ssc, None)
```

DStream: a sequence of RDDs representing a stream of data

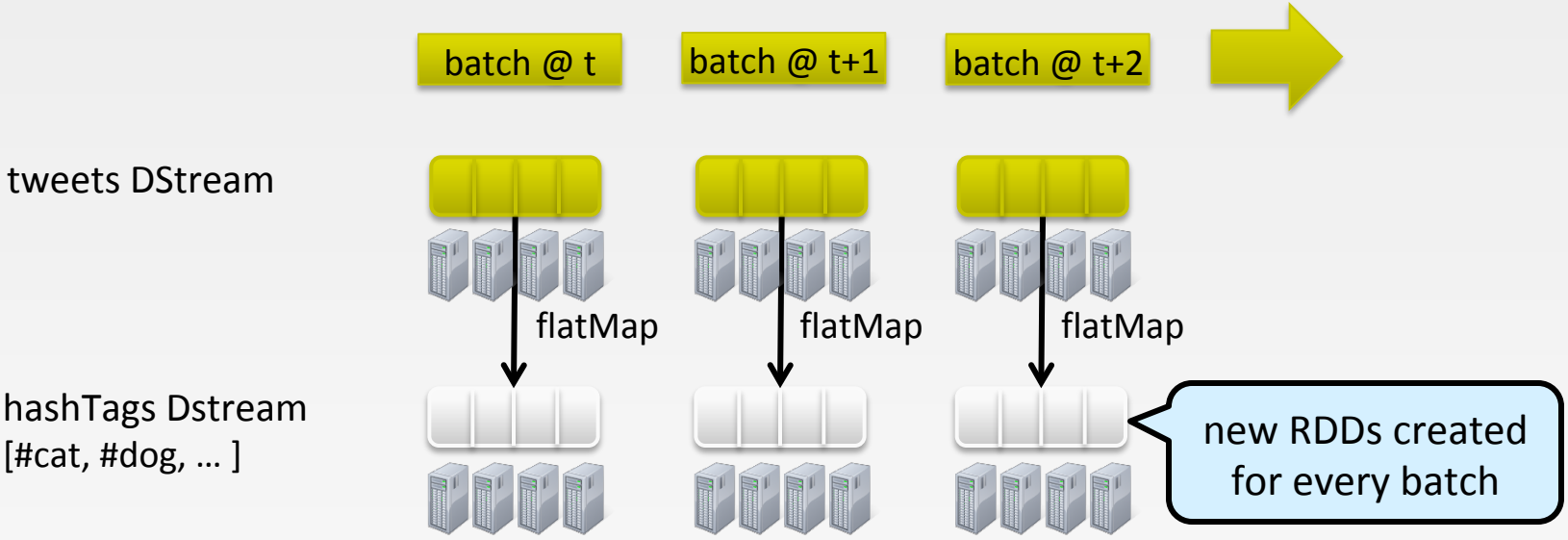


Example – Get hashtags from Twitter

```
val hashTags = tweets.flatMap (status => getTags(status))
```

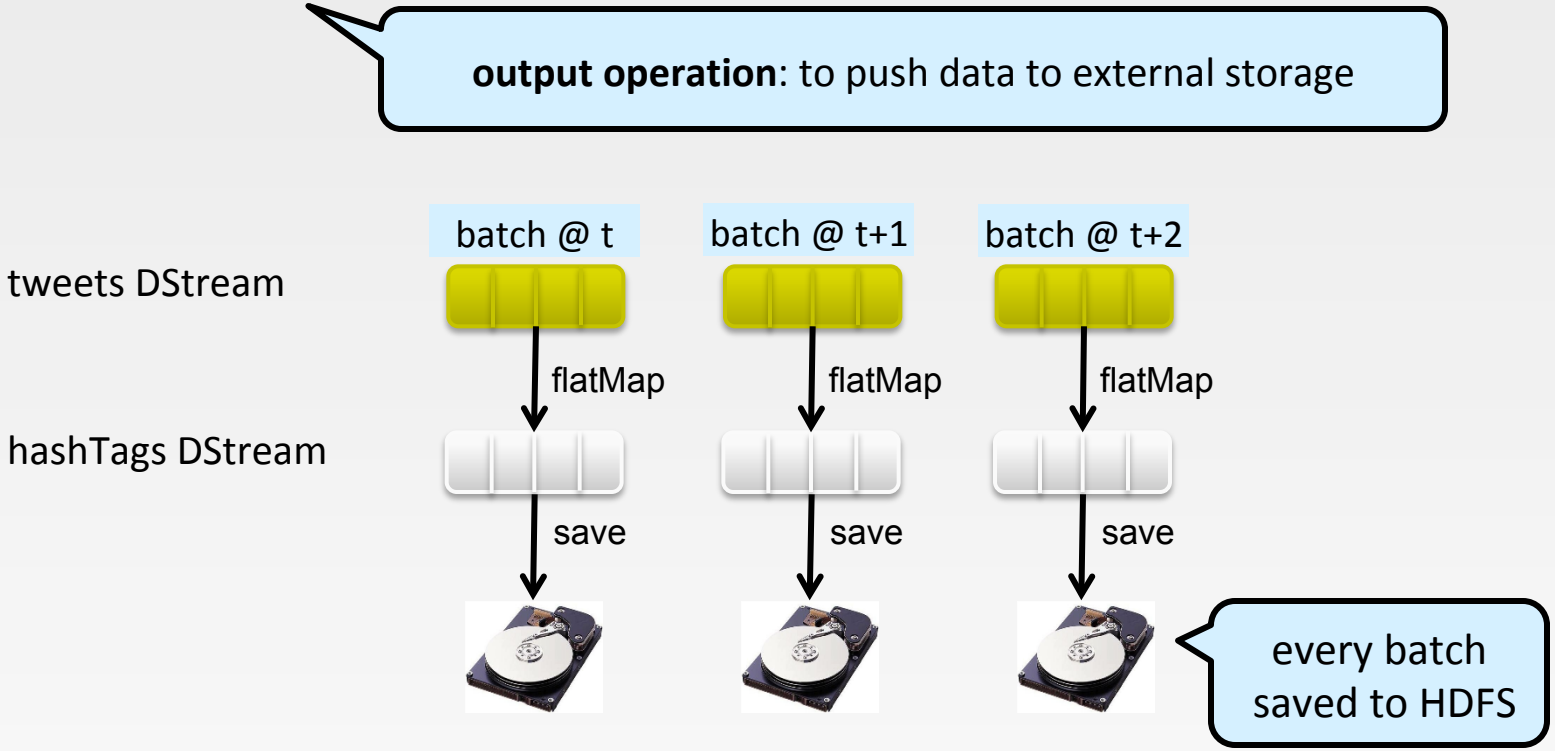
new DStream

transformation: modify data in one DStream to create another DStream



Example – Get hashtags from Twitter

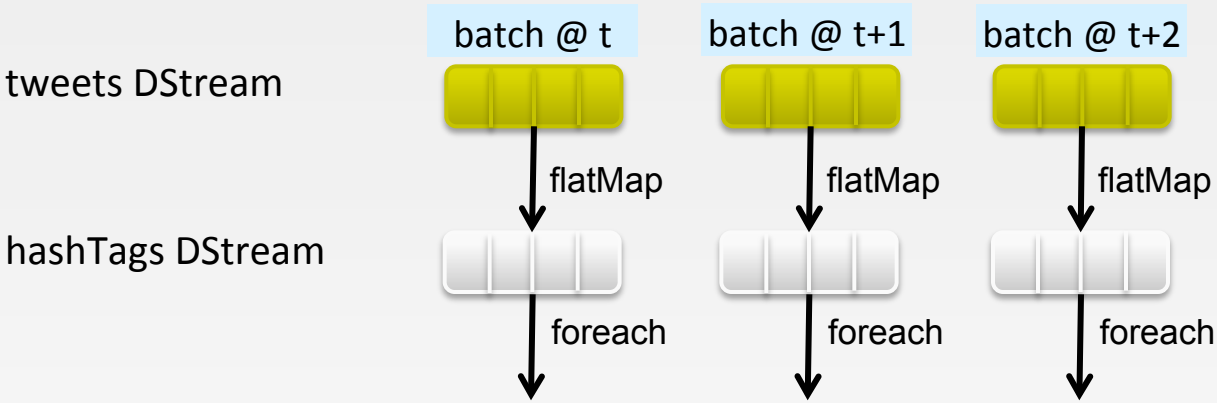
```
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```



Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

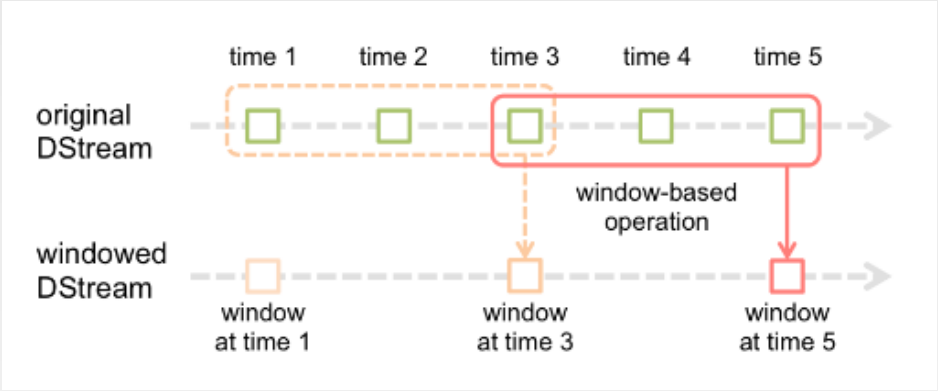
foreach: do whatever you want with the processed data



Write to database, update analytics UI, do whatever you want

Window Operations

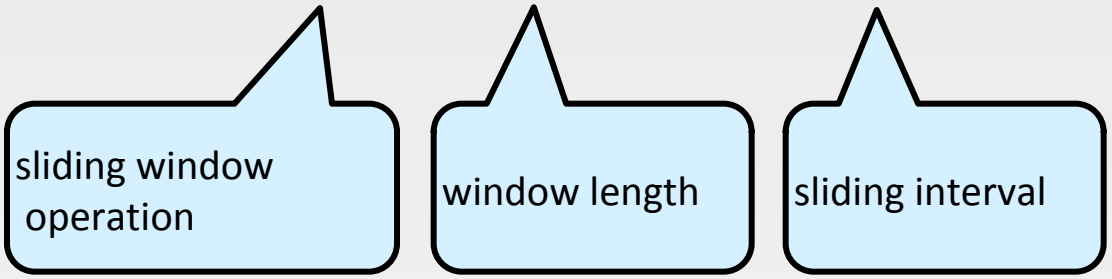
- Spark Streaming also provides *windowed computations*, which allow you to apply transformations over a sliding window of data

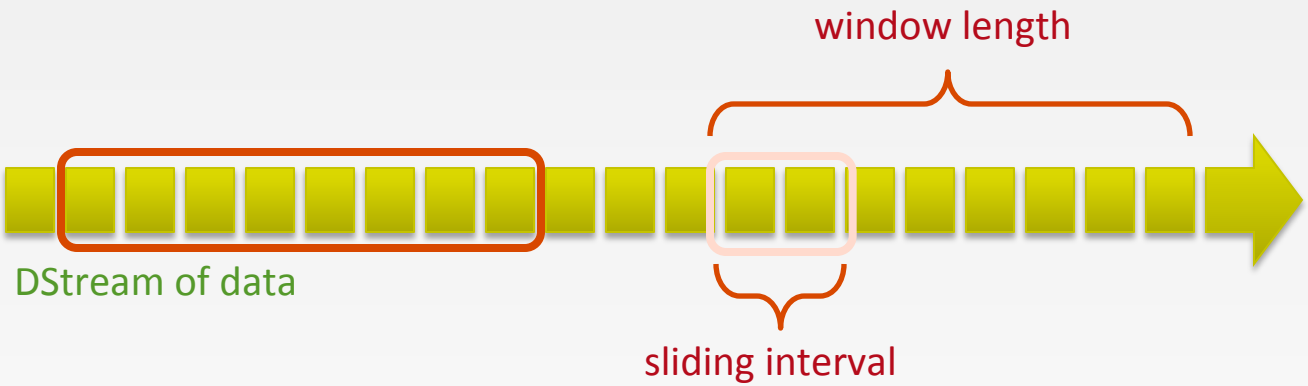


- Every time the window *slides* over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream.
  - E.g., the operation is applied over the last 3 time units of data, and slides by 2 time units
  - Any window operation needs to specify two parameters
    - *window length* - The duration of the window (3 in the figure).
    - *sliding interval* - The interval at which the window operation is performed (2 in the figure).

Window-based Transformations

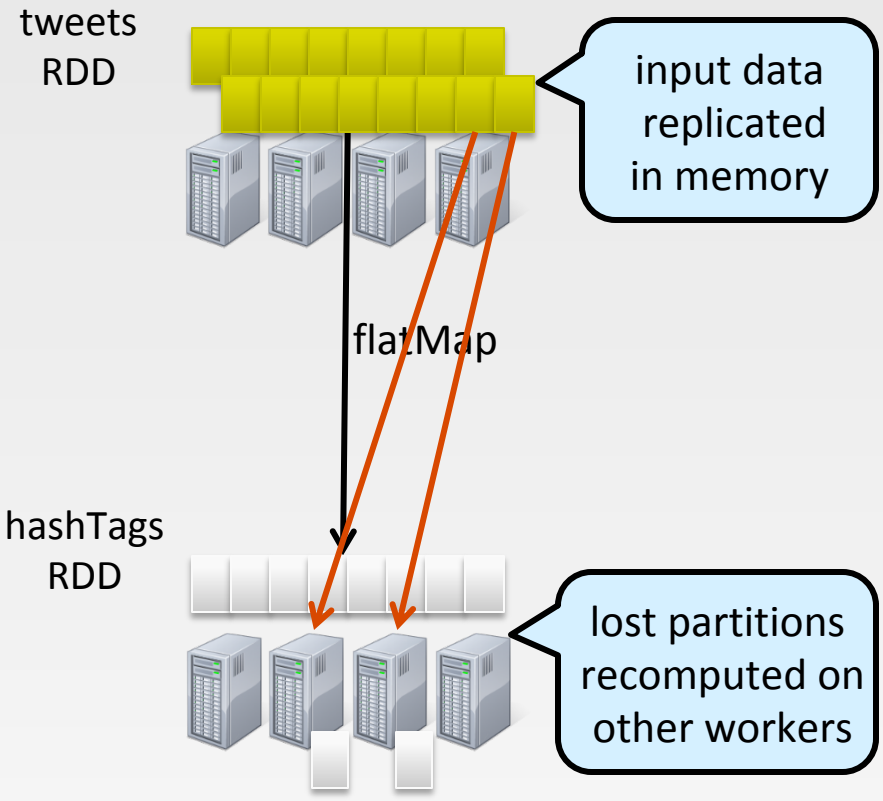
```
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(5)).countByValue()
```





Fault-tolerance: Worker

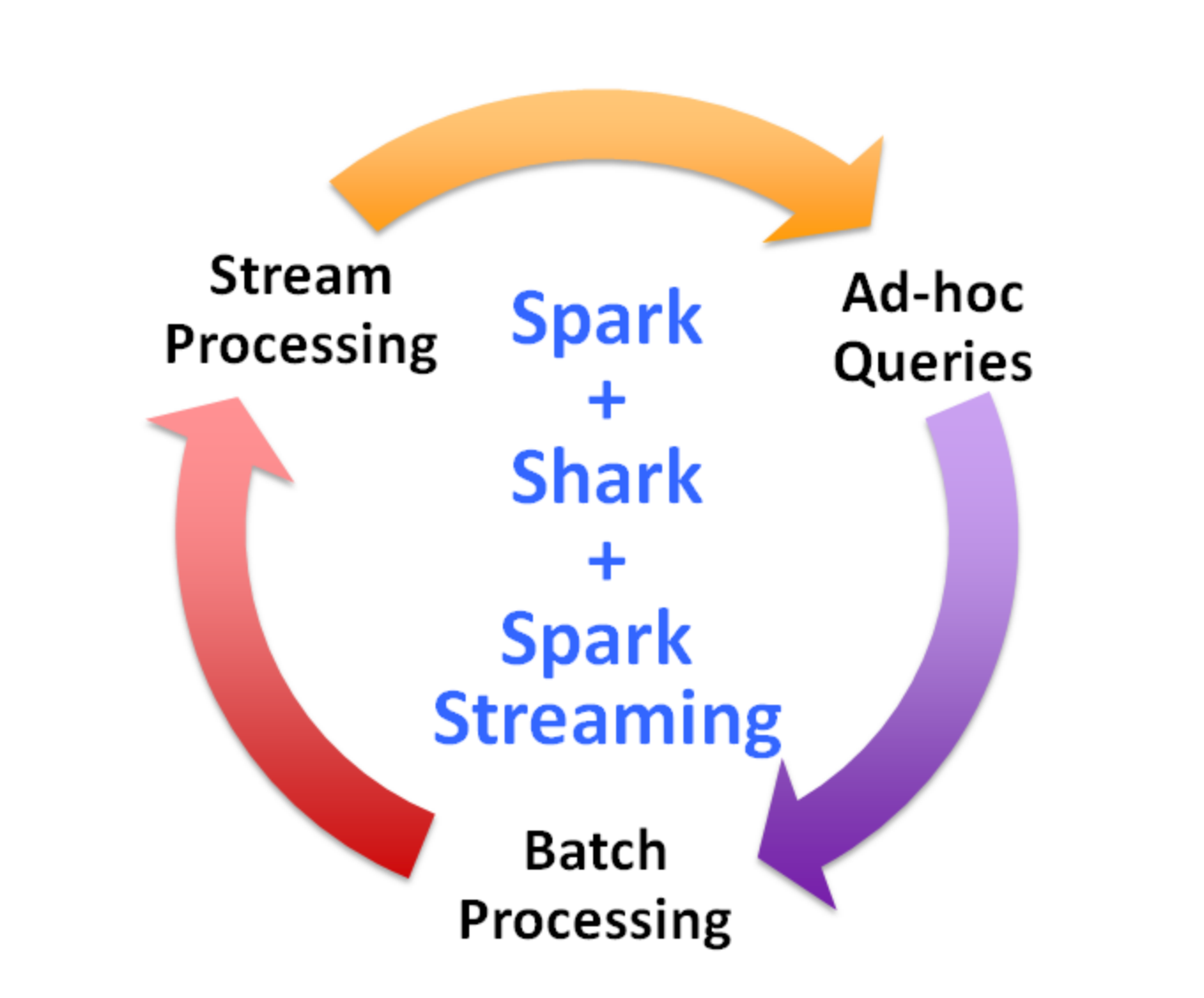
- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data
- All transformed data is fault-tolerant, and exactly-once transformations



Fault-tolerance: Master

- Master saves the state of the DStreams to a checkpoint file
  - Checkpoint file saved to HDFS periodically
- If master fails, it can be restarted using the checkpoint file
- More information in the Spark Streaming guide
- Automated master fault recovery coming soon

Vision - one stack to rule them all



References

- http://spark.apache.org/docs/latest/index.html
- Spark SQL guide:
  http://spark.apache.org/docs/latest/sql-programming-guide.html
- Spark Streaming guide:

  http://spark.apache.org/docs/latest/streaming-programming-guide.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
- Learning Spark. O'Reilly Media.

End of Chapter 7