Data Intensive Computing Platforms

Amir H. Payberah
amir@sics.se

SICS/KTH

Big Data

Big Data

... everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it.

- Dan Ariely

Big Data

> Big data is data that exceeds the processing capacity of conventional database systems. The data is too big, moves too fast, or doesn't fit the strictures of your database architectures. To gain value from this data, you must choose an alternative way to process it.
>
> - O'Reilly

O'REILLY®

Big Data

Big data is data that exceeds the processing capacity of conventional database systems. The data is too big, moves too fast, or doesn't fit the strictures of your database architectures. To gain value from this data, you must choose an alternative way to process it.

- O'Reilly

Too Many Words

O'REILLY®

Big Data

> Big data is the data characterized by 3 attributes: volume, variety and velocity.
>
> - IBM

Big Data

> Big data is the data characterized by 3 attributes: volume, variety and velocity.
>
> - IBM

Big Data

> Big data is the data characterized by 4 key attributes: volume, variety, velocity and value.
>
> - Oracle

Big Data

> Big data is the data characterized by 4 key attributes: volume, variety, velocity and value.
>
> - Oracle

Buzzwords

Let's Define Big Data In Simple Words

Big Data

DevOps Borat

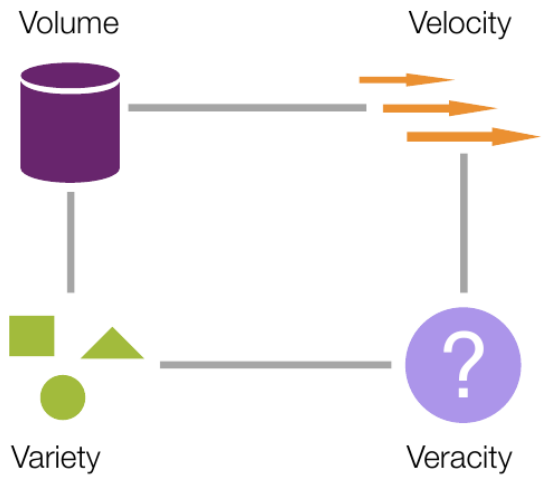
@DEVOPS_BORAT

Small Data is when is fit in RAM. Big Data is when is crash because is not fit in RAM.

2/6/13, 8:22 AM

The Four Dimensions of Big Data

- Volume: data size
- Velocity: data generation rate
- Variety: data heterogeneity
- This 4th V is for Vacillation: Veracity/Variability/Value



Big Data Sources

How Much Data?

- Google: 100 PB/day process, 15000 PB storage
- EBay: 100 PB/day, 90 PB storage
- Baidu: 10-100 PB/day, 2000 PB storage
- Facebook: 600 TB/day, 300 PB storage
- Spotify: 2.2 TB/day, 100 PB storage

[https://followthedata.wordpress.com/2014/06/24/data-size-estimates]

Two Driving Factors

- Cloud computing
- Open source communities

Who Uses Big Data?

- Banking
- Government
- Manufacuring
- Education
- Health care
- ...

How To Store and Process Big Data?

Scale Up vs. Scale Out (1/2)

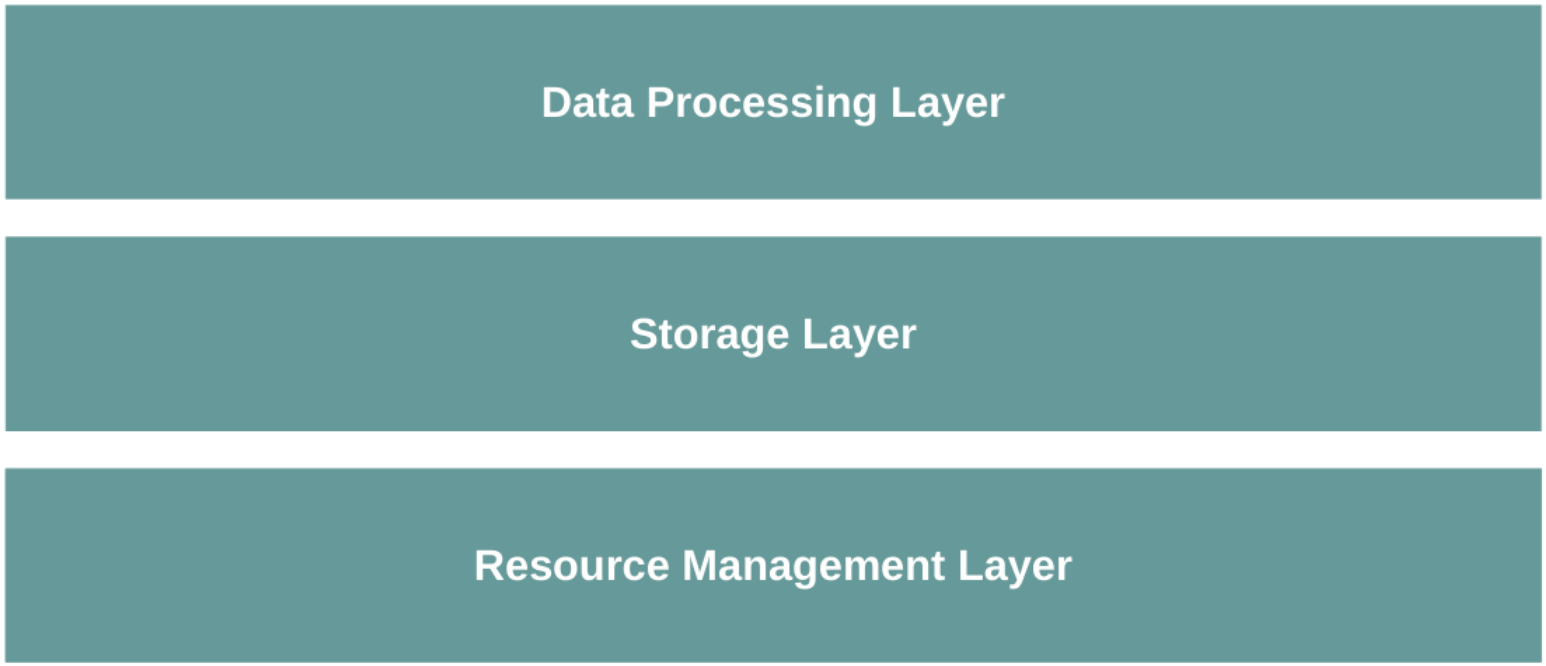
- Scale up or scale vertically: adding resources to a single node in a system.
- Scale out or scale horizontally: adding more nodes to a system.

Scale Up vs. Scale Out (2/2)

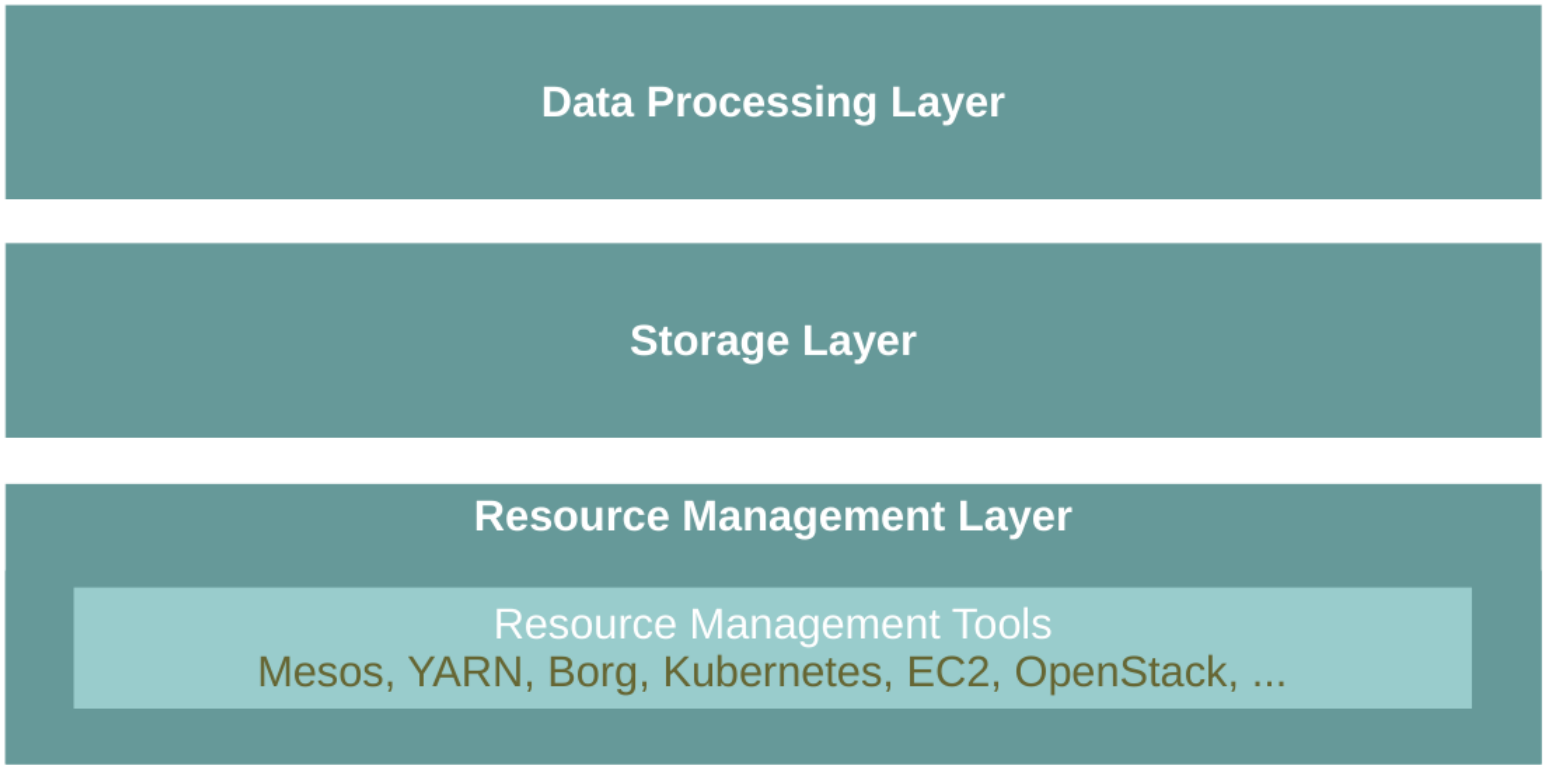
- Scale up: more expensive than scaling out.
- Scale out: more challenging for fault tolerance and software development.


APACHE
HBASE
mahout
GraphX
hadoop
kafka
Storm
Dato
Spark
HIVE
S4 distributed stream computing platform
cassandra
Google Cloud Platform

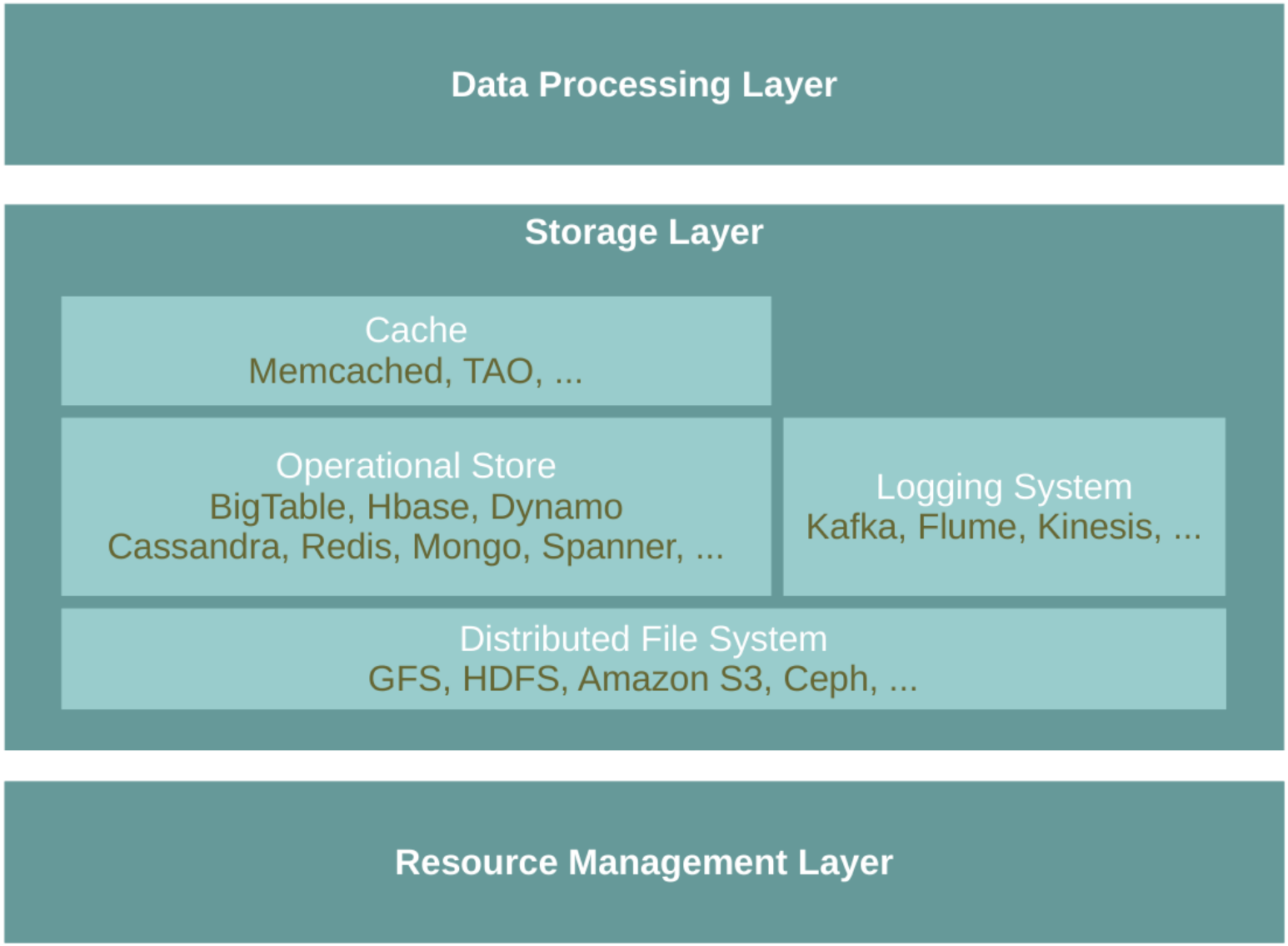
Three Main Layers: Big Data Stack



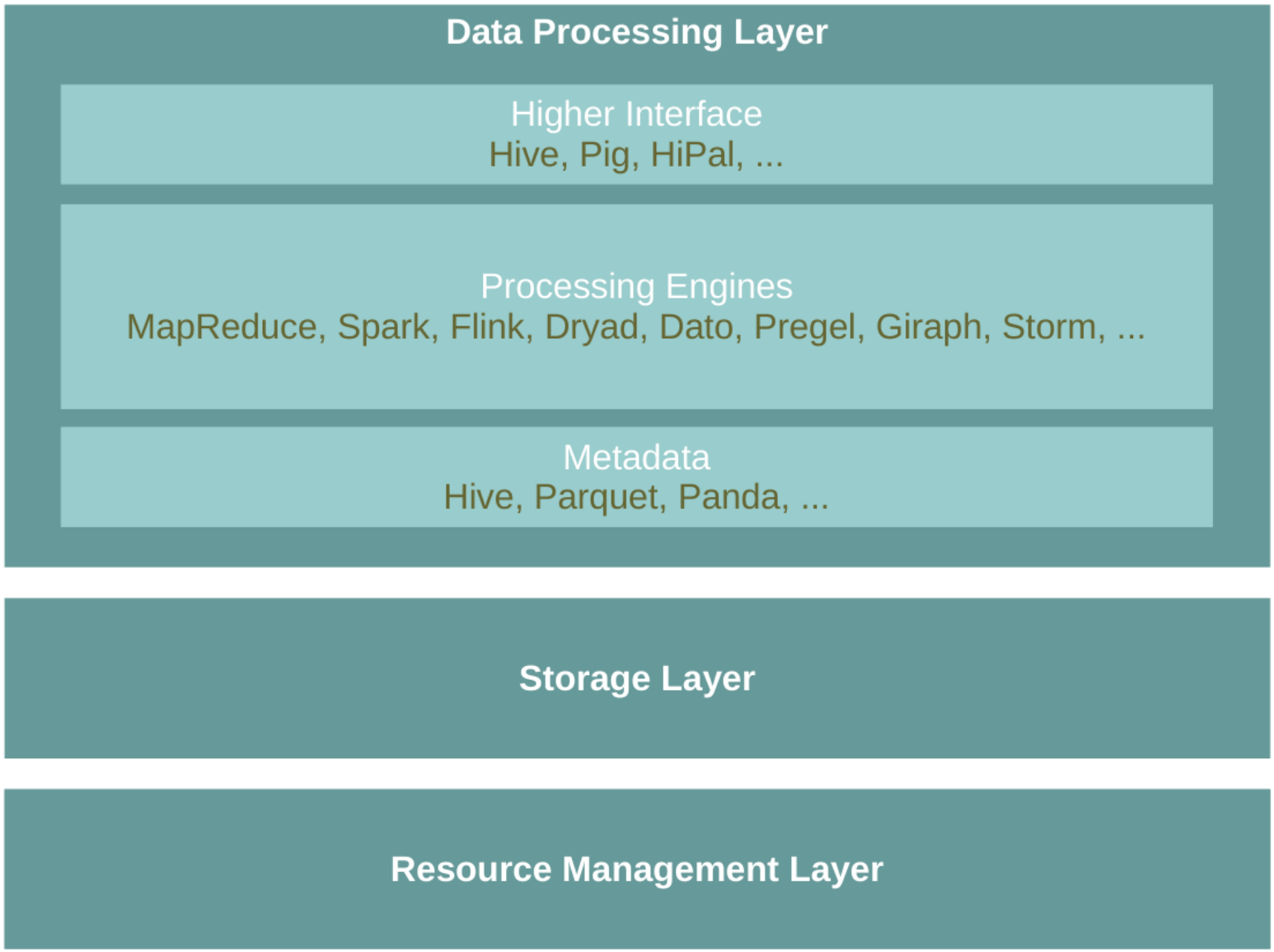
Resource Management Layer



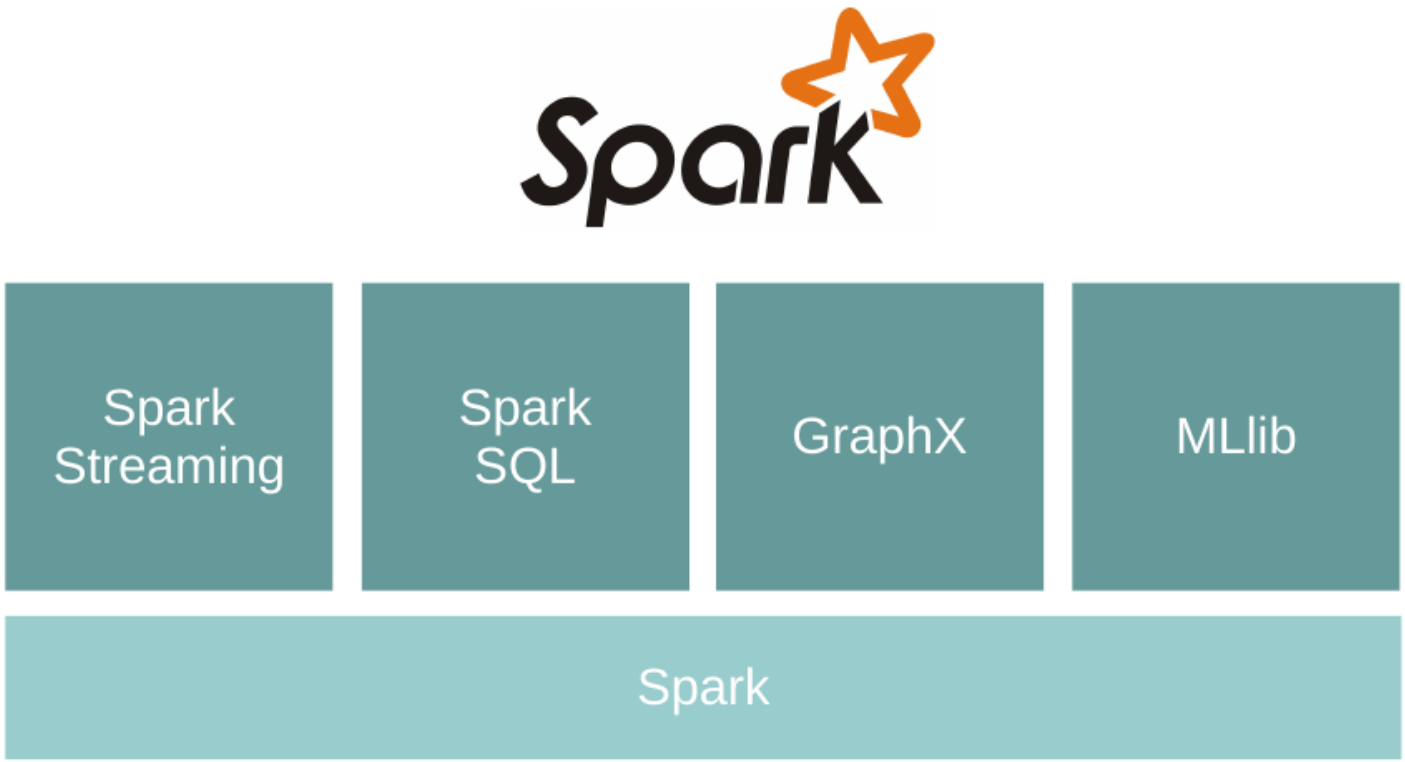
Storage Layer



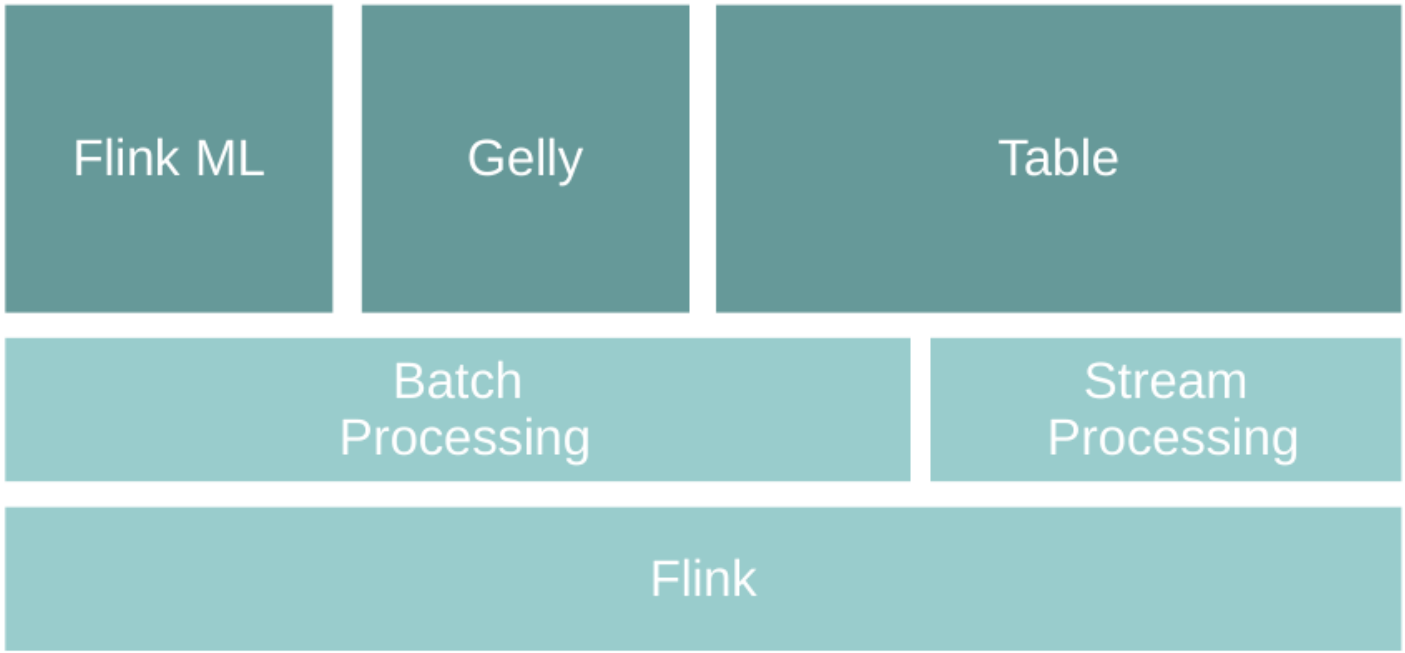
Processing Layer



Spark Processing Engine



Flink Processing Engine



Scala

Scala

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

Functional Programming Languages

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

Scala Variables

- Values: immutable
- Variables: mutable

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

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

If ... Else

```
var x = 30;

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

Loop

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

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

Functions

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

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

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

Anonymous Functions

- Lightweight syntax for defining functions.

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

Higher-Order Functions


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

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

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

Collections (1/2)

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

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

Collections (1/2)

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

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

- List: sequential collection of elements of the same type

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

Collections (1/2)

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

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

- List: sequential collection of elements of the same type

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

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

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

Collections (2/2)

- Map: collection of key/value pairs

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

Collections (2/2)

- Map: collection of key/value pairs

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

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

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

Functional Combinators

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

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

- flatten: it collapses one level of nested structure

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

- flatMap: map + flatten

- foreach: it is like map but returns nothing

Classes and Objects

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

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

Classes and Objects

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

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

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

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

Test.main(null)
```

Case Classes and Pattern Matching

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

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

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

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

Data Intensive Computing

Motivation

- We have a huge text document.
- Count the number of times each distinct word appears in the file

Motivation

- We have a huge text document.
- Count the number of times each distinct word appears in the file
- If the file fits in memory: `words(doc.txt) | sort | uniq -c`

Motivation

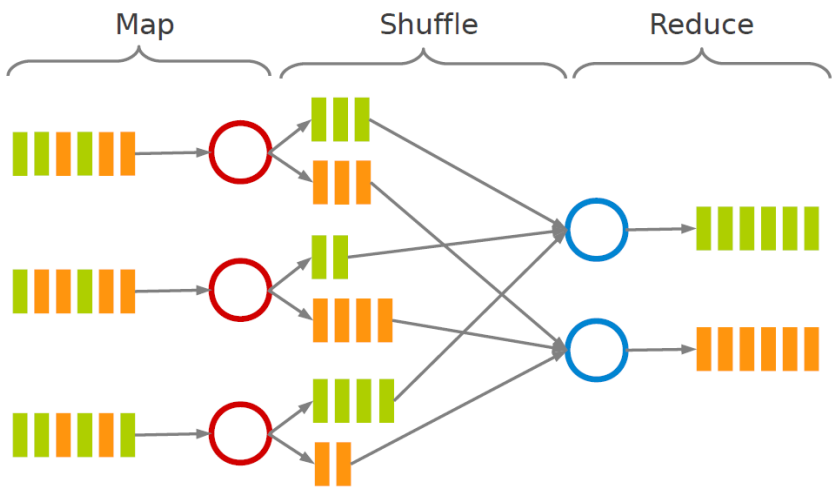
- We have a huge text document.
- Count the number of times each distinct word appears in the file
- If the file fits in memory: `words(doc.txt) | sort | uniq -c`
- If not?

MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`

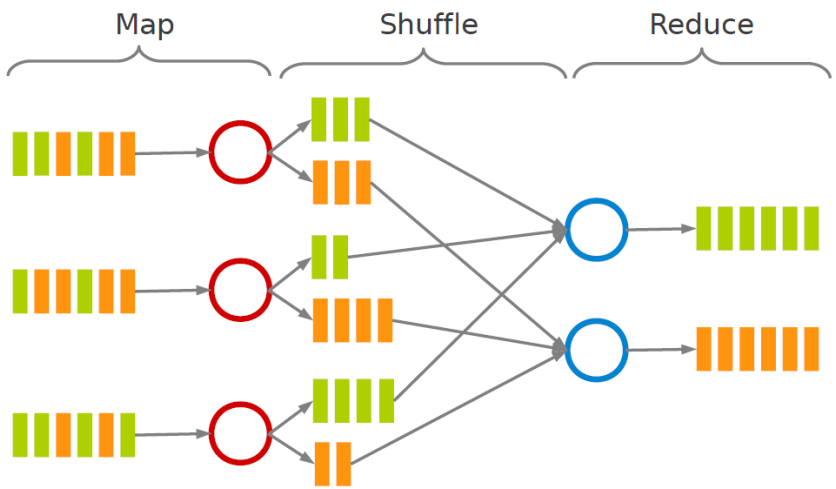
MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.



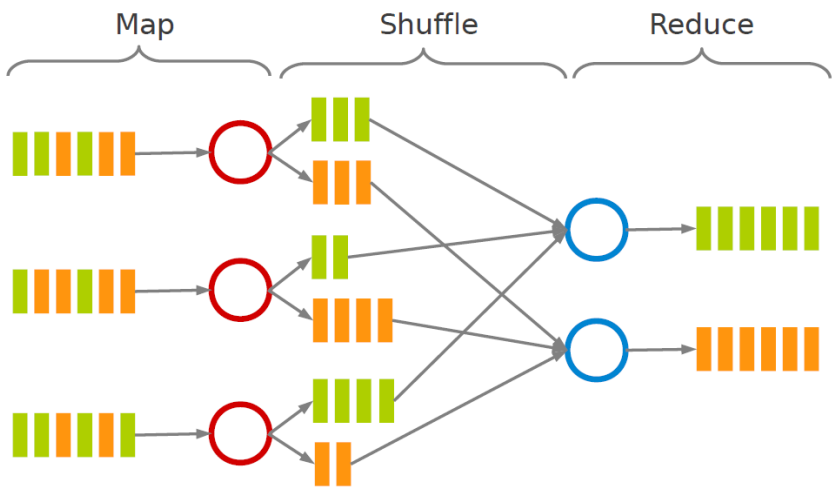
MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.



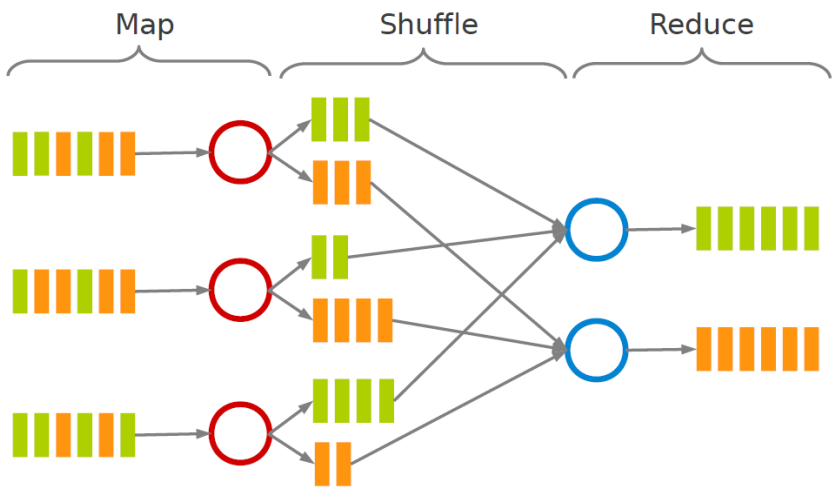
MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.



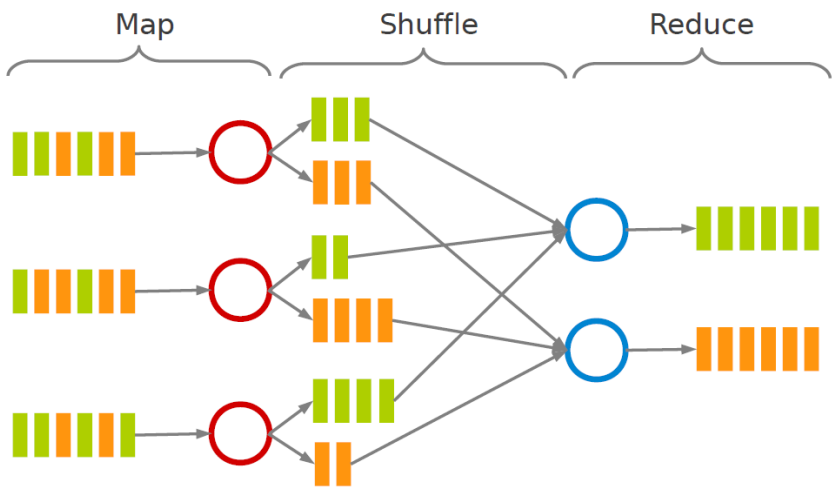
MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.
- Reduce: aggregate, summarize, filter or transform.



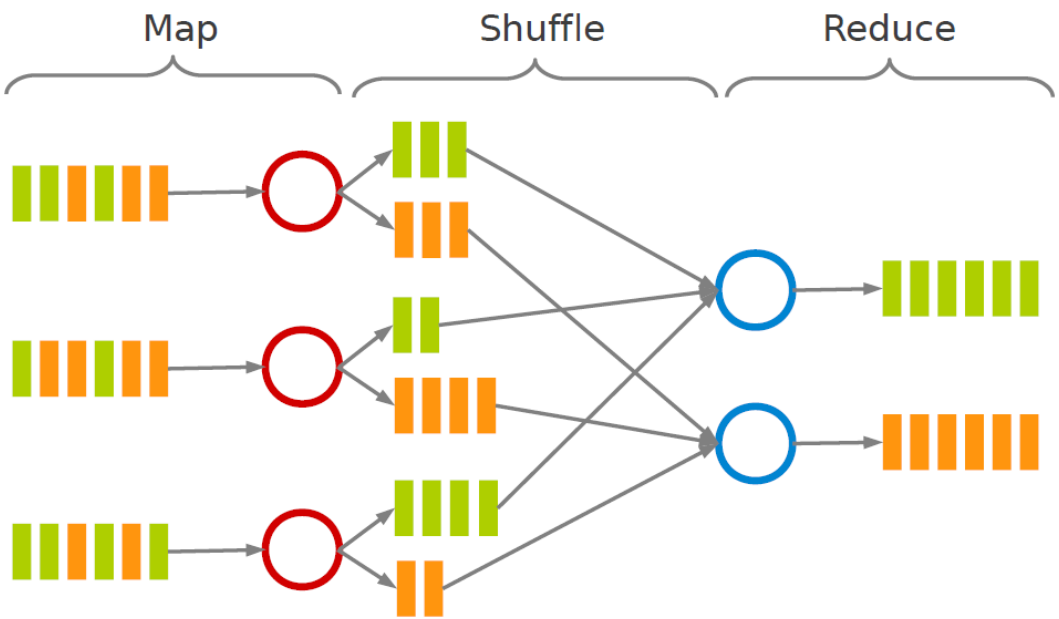
MapReduce Programming Model

- `words(doc.txt) | sort | uniq -c`
- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.
- Reduce: aggregate, summarize, filter or transform.
- Write the result.



MapReduce Dataflow

- map function: processes data and generates a set of intermediate key/value pairs.
- reduce function: merges all intermediate values associated with the same intermediate key.



Word Count in MapReduce

- Consider doing a word count of the following file using MapReduce:

```
Hello World Bye World
Hello Hadoop Goodbye Hadoop
```

Word Count in MapReduce - map

- The map function reads in words one a time and outputs (word, 1) for each parsed input word.
- The map function output is:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

Word Count in MapReduce - shuffle

- The shuffle phase between map and reduce phase creates a list of values associated with each key.
- The reduce function input is:

```
(Bye, (1))
(Goodbye, (1))
(Hadoop, (1, 1))
(Hello, (1, 1))
(World, (1, 1))
```

Word Count in MapReduce - reduce

- The reduce function sums the numbers in the list for each key and outputs (word, count) pairs.
- The output of the reduce function is the output of the MapReduce job:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

The Word Count Mapper

```
public static class MyMap extends Mapper<...> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);

    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(word, one);
    }
  }
}
```

The Word Count Reducer

```
public static class MyReduce extends Reducer<...> {
  public void reduce(Text key, Iterator<...> values, Context context)
    throws IOException, InterruptedException {
    int sum = 0;

    while (values.hasNext())
      sum += values.next().get();

    context.write(key, new IntWritable(sum));
  }
}
```

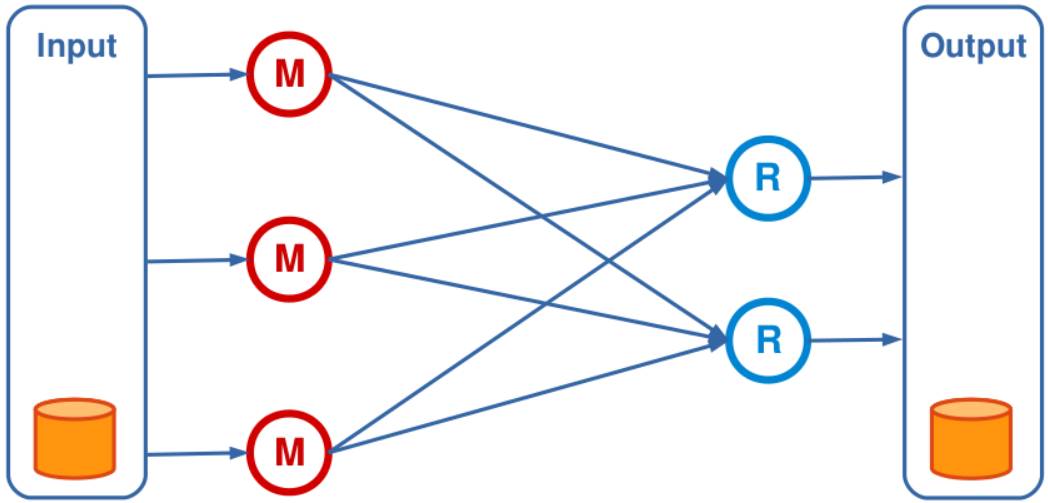
The Word Count Driver

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = new Job(conf, "wordcount");

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  job.setMapperClass(MyMap.class);
  job.setCombinerClass(MyReduce.class);
  job.setReducerClass(MyReduce.class);

  job.setInputFormatClass(TextInputFormat.class);
  job.setOutputFormatClass(TextOutputFormat.class);

  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  job.waitForCompletion(true);
}
```

Spark

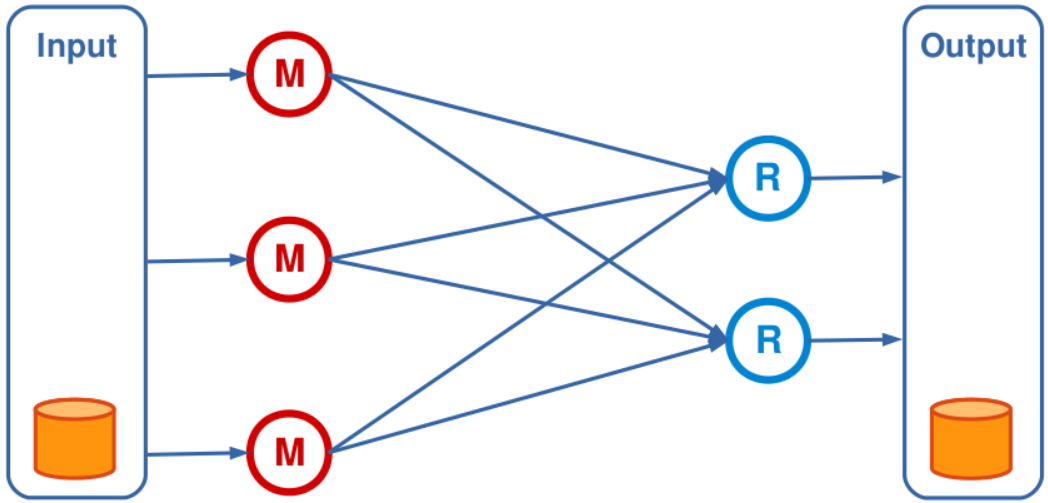
Motivation (1/4)

- Most current cluster programming models are based on acyclic data flow from stable storage to stable storage.



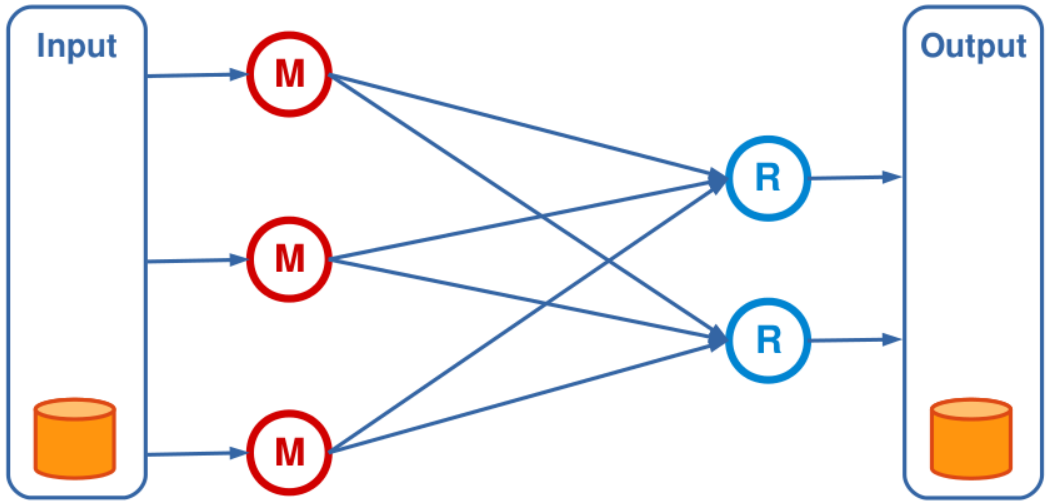
Motivation (1/4)

- Most current cluster programming models are based on acyclic data flow from stable storage to stable storage.
- Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures.



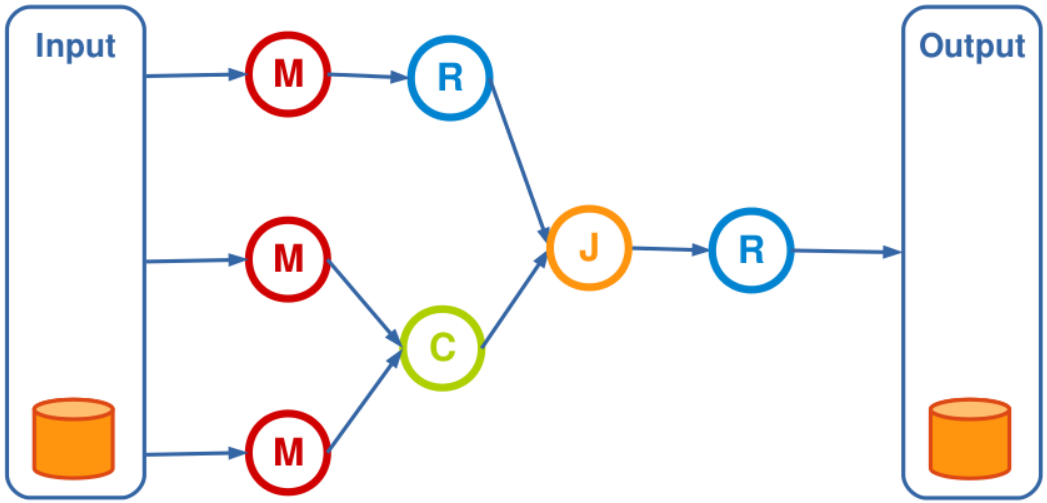
Motivation (1/4)

- Most current cluster programming models are based on acyclic data flow from stable storage to stable storage.
- Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures.
- E.g., MapReduce



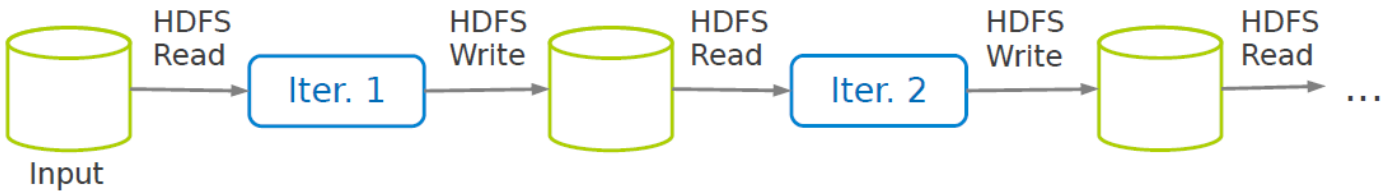
Motivation (2/4)

- MapReduce programming model has not been designed for complex operations, e.g., data mining.



Motivation (3/4)

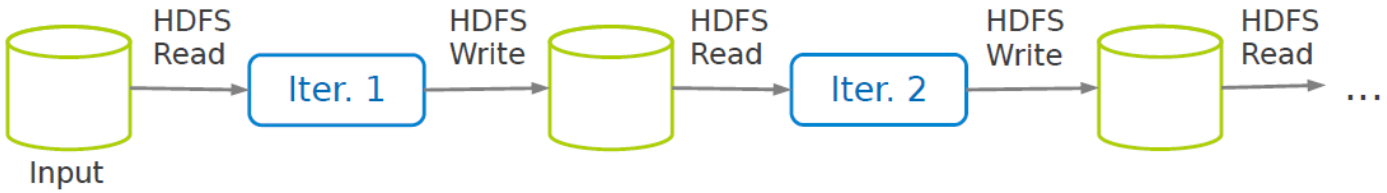
- Very expensive (slow), i.e., always goes to disk and HDFS.



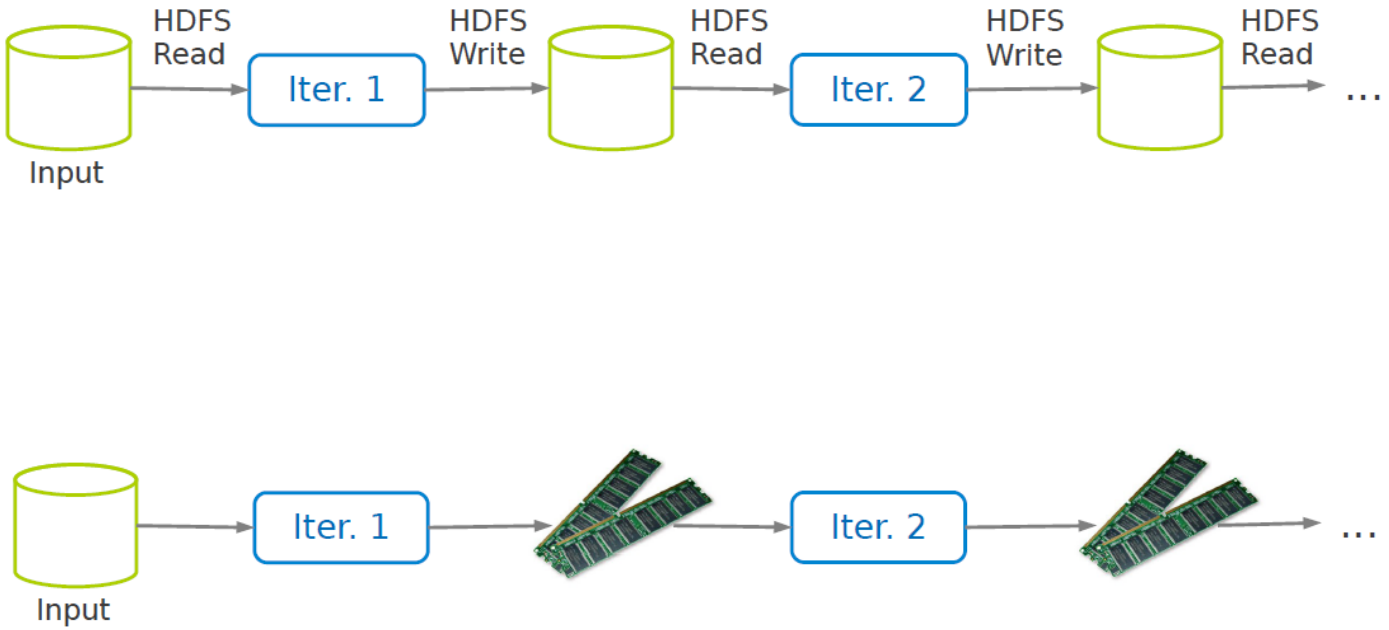
Motivation (4/4)

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

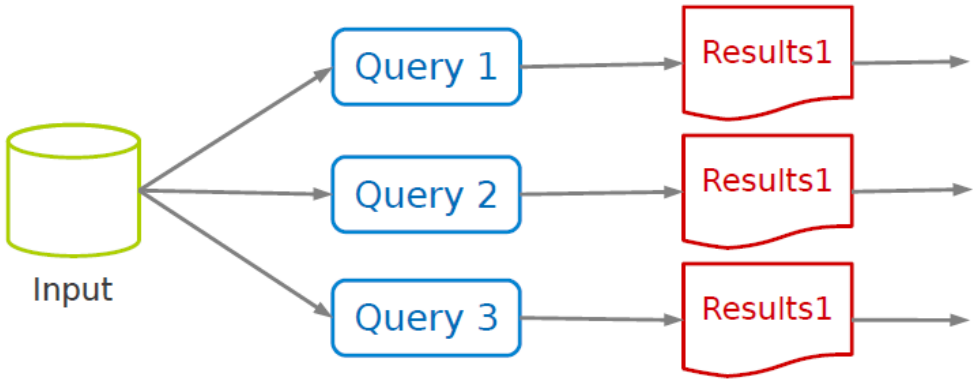
Spark vs. MapReduce (1/2)



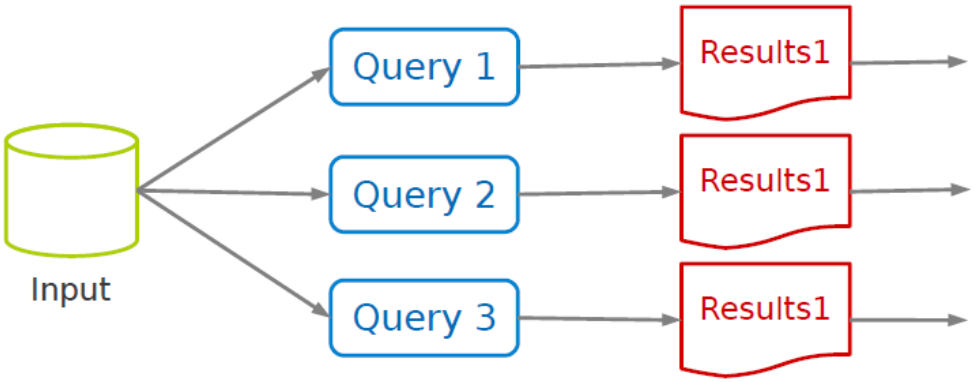
Spark vs. MapReduce (1/2)

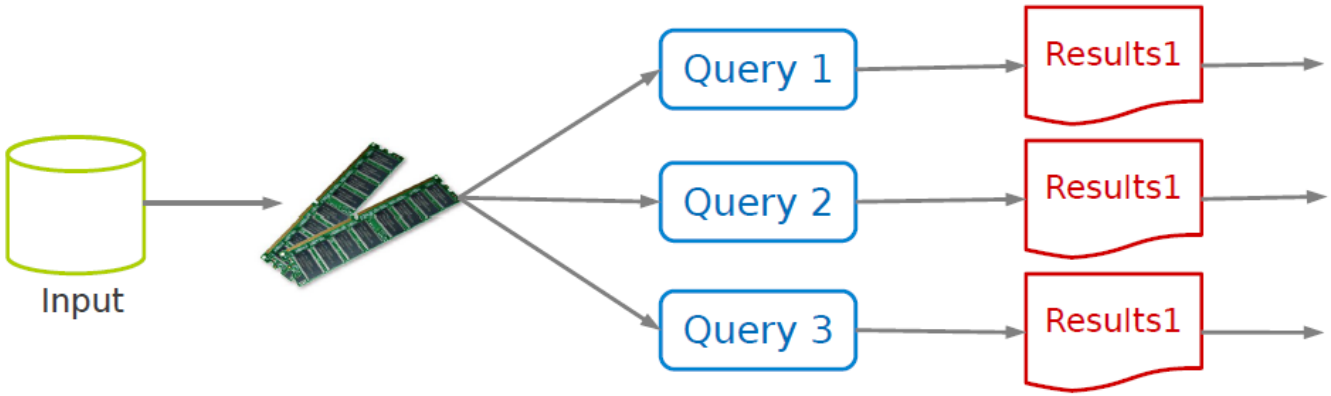


Spark vs. MapReduce (2/2)



Spark vs. MapReduce (2/2)





Challenge

How to design a distributed memory abstraction that is both fault tolerant and efficient?

Challenge

How to design a distributed memory abstraction that is both fault tolerant and efficient?

Solution

Resilient Distributed Datasets (RDD)

Resilient Distributed Datasets (RDD) (1/2)

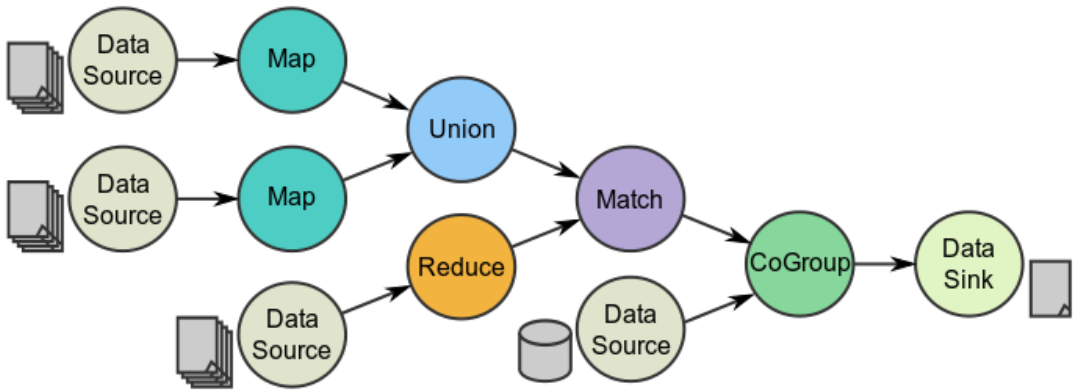
- A distributed memory abstraction.
- Immutable collections of objects spread across a cluster.
  - Like a `LinkedList <MyObjects>`

Resilient Distributed Datasets (RDD) (2/2)

- An RDD is divided into a number of partitions, which are atomic pieces of information.
- Partitions of an RDD can be stored on different nodes of a cluster.

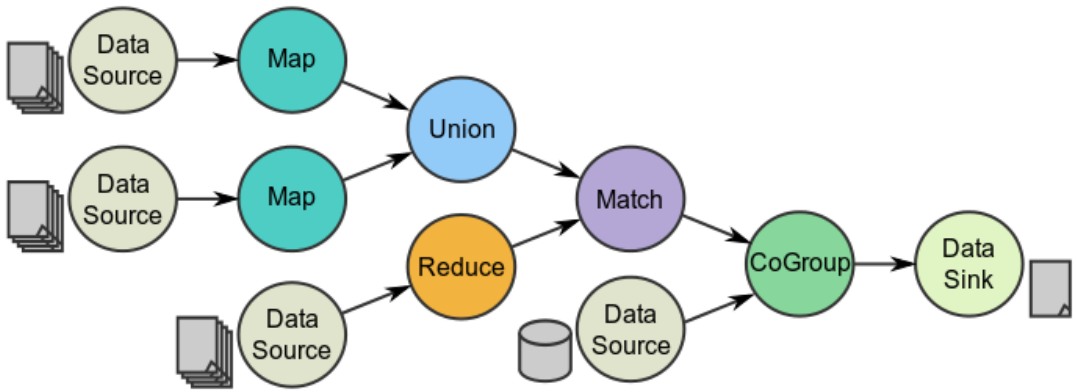
Spark Programming Model

- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.



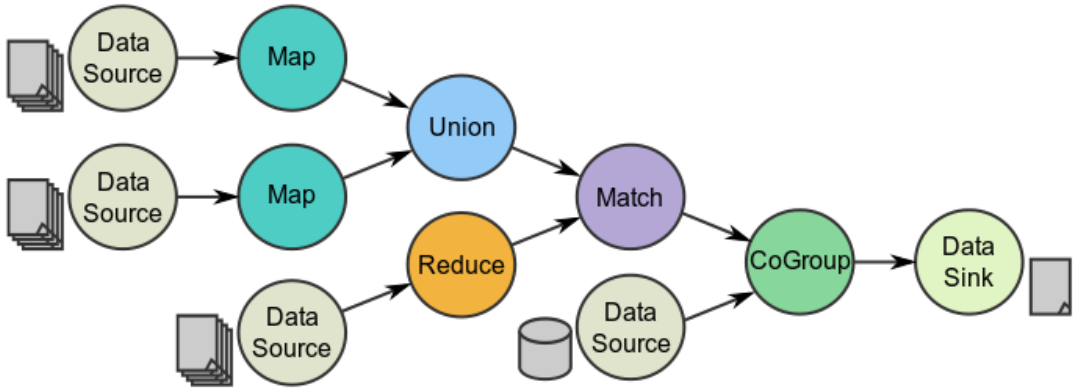
Spark Programming Model

- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.
- Operators are higher-order functions that execute user-defined functions in parallel.



Spark Programming Model

- A data flow is composed of any number of data sources, operators, and data sinks by connecting their inputs and outputs.
- Operators are higher-order functions that execute user-defined functions in parallel.
- Two types of RDD operators: transformations and actions.



RDD Operators (1/2)

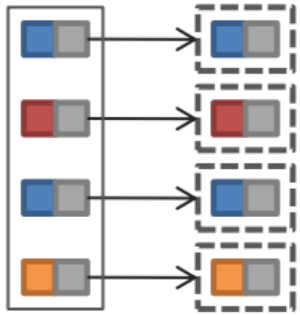
- Transformations: lazy operators that create new RDDs.
- Actions: lunch a computation and return a value to the program or write data to the external storage.

RDD Operators (2/2)

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

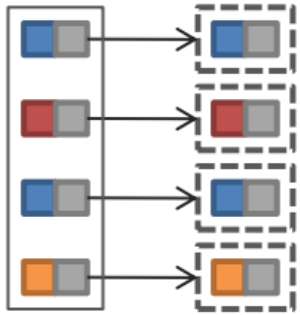
RDD Transformations - Map

- All pairs are independently processed.



RDD Transformations - Map

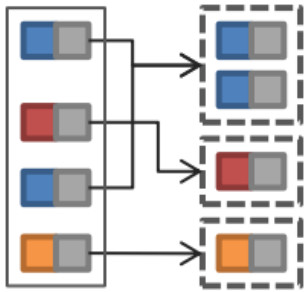
- All pairs are independently processed.



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

// selecting those elements that func returns true.
val even = squares.filter(_ % 2 == 0) // {4}
```

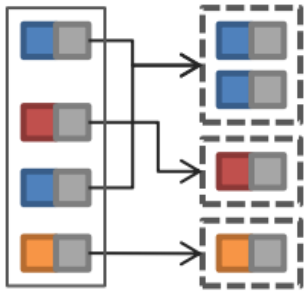
RDD Transformations - Reduce

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



RDD Transformations - Reduce

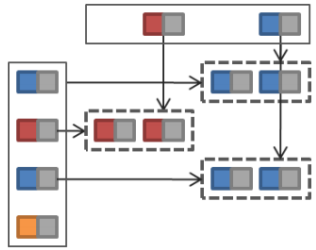
- Pairs with identical key are grouped.
- Groups are independently processed.



```
val pets = sc.parallelize(Seq(("cat", 1), ("dog", 1), ("cat", 2)))

pets.groupByKey()
// {(cat, (1, 2)), (dog, (1))}

pets.reduceByKey((x, y) => x + y)
or
pets.reduceByKey(_ + _)
// {(cat, 3), (dog, 1)}
```

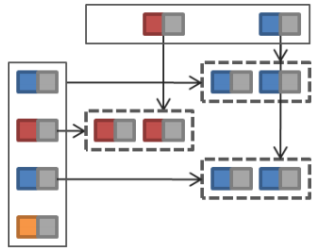
RDD Transformations - Join

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



RDD Transformations - Join

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



```
val visits = sc.parallelize(Seq(("h", "1.2.3.4"),
                                ("a", "3.4.5.6"),
                                ("h", "1.3.3.1")))

val pageNames = sc.parallelize(Seq(("h", "Home"),
                                   ("a", "About")))

visits.join(pageNames)
  // ("h", ("1.2.3.4", "Home"))
  // ("h", ("1.3.3.1", "Home"))
  // ("a", ("3.4.5.6", "About"))
```

Basic RDD Actions (1/2)

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

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

Basic RDD Actions (1/2)

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

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

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

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

Basic RDD Actions (1/2)

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

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

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

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

- Return the number of elements in the RDD.

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

Basic RDD Actions (2/2)

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

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

Basic RDD Actions (2/2)

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

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

- Write the elements of the RDD as a text file.

```
nums.saveAsTextFile("hdfs://file.txt")
```

SparkContext

- Main entry point to Spark functionality.
- Available in shell as variable sc.
- Only one SparkContext may be active per JVM.

```
// master: the master URL to connect to, e.g.,
// "local", "local[4]", "spark://master:7077"
val conf = new SparkConf().setAppName(appName).setMaster(master)

new SparkContext(conf)
```

Creating RDDs

- Turn a collection into an RDD.

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

Creating RDDs

- Turn a collection into an RDD.

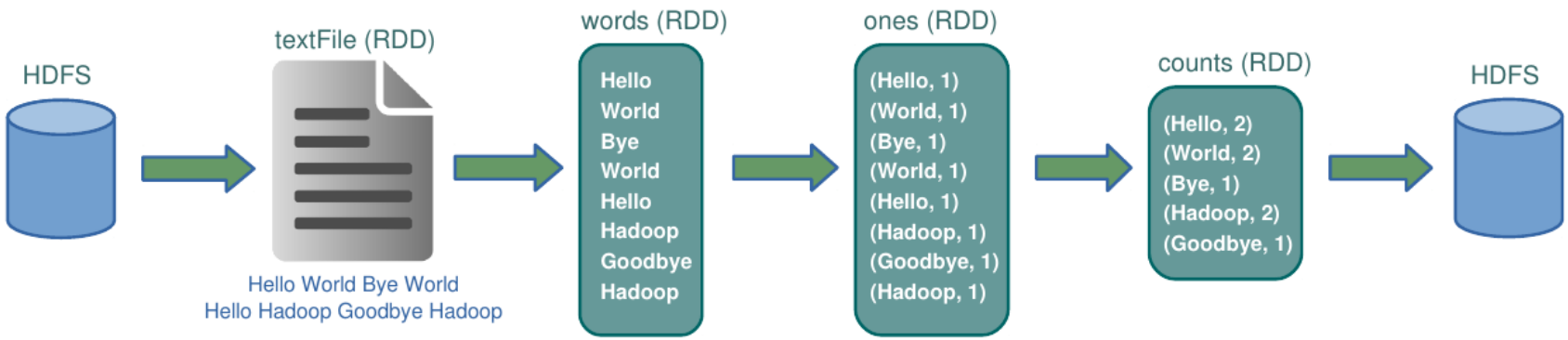
```
val a = sc.parallelize(Array(1, 2, 3))
```

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

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

Example 1

```
val textFile = sc.textFile("hdfs://...")

val words = textFile.flatMap(line => line.split(" "))
val ones = words.map(word => (word, 1))
val counts = ones.reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```



Example 2

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

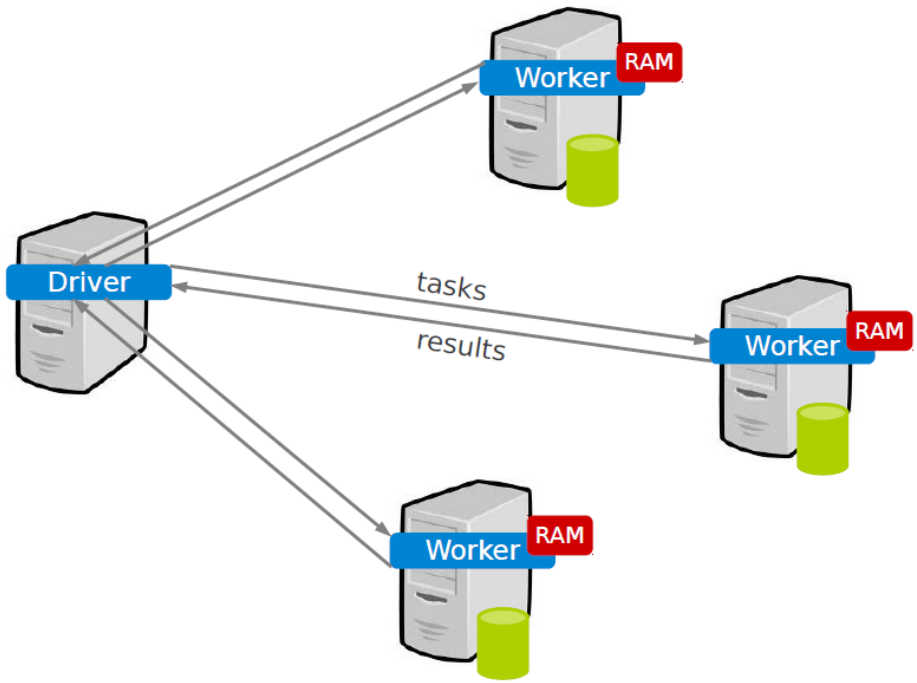
Example 2

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

```
val textFile = sc.textFile("hdfs://...")
val count = textFile.filter(_.contains("SICS")).count()
```

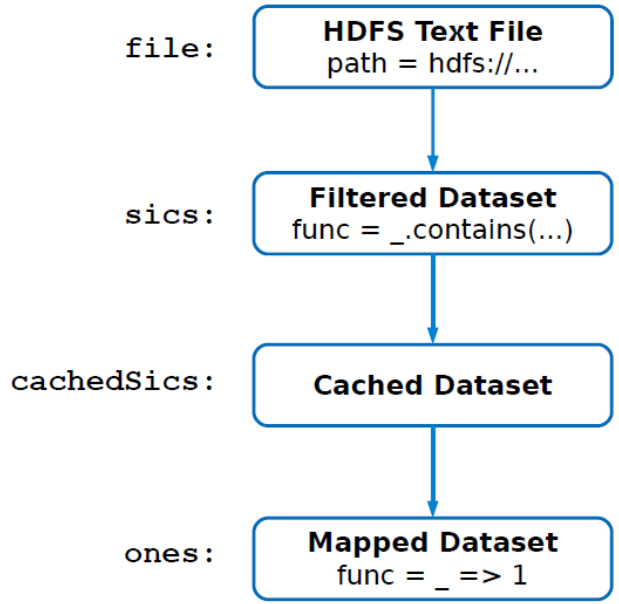
Spark Execution Model

- A Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster.



Lineage

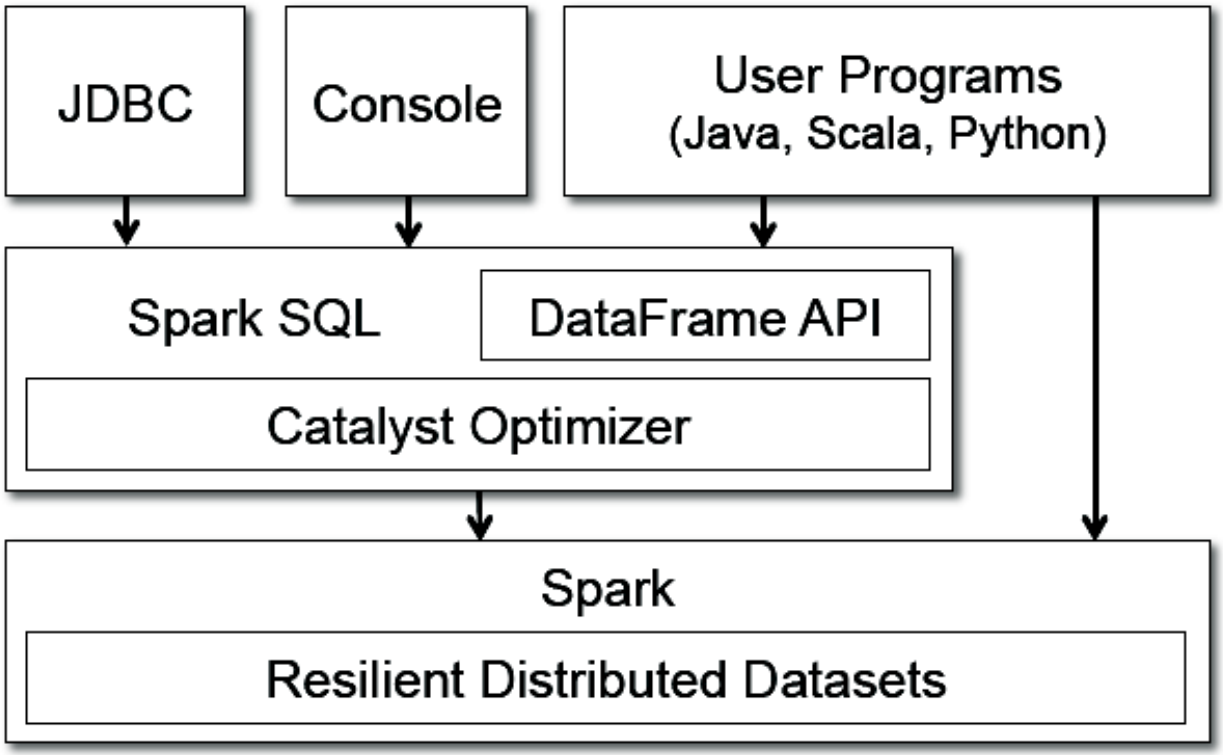
- Lineage: transformations used to build an RDD.
- RDDs are stored as a chain of objects capturing the lineage of each RDD.




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

Spark SQL

Spark and Spark SQL

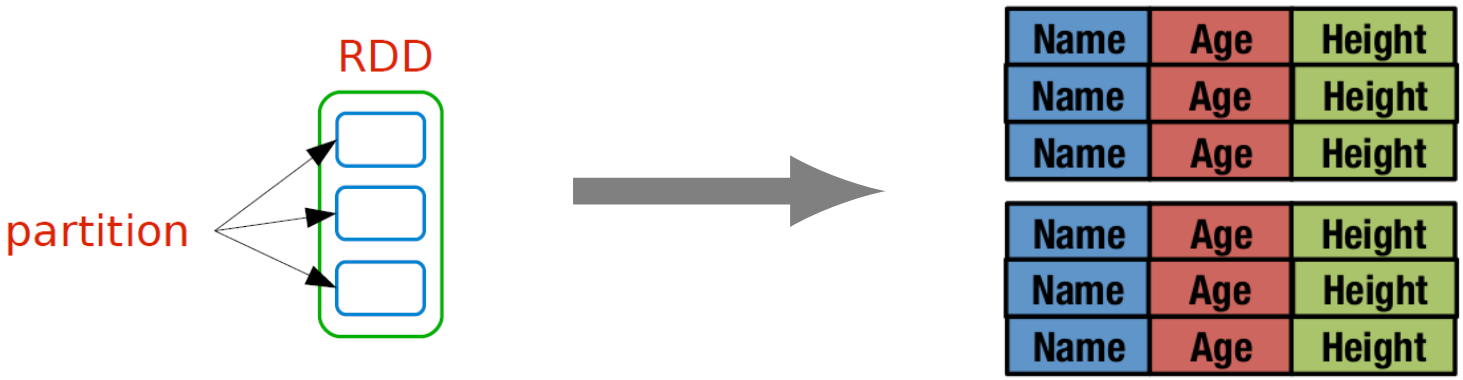


DataFrame

- A DataFrame is a distributed collection of rows
- Homogeneous schema.
- Equivalent to a table in a relational database.

Adding Schema to RDDs

- Spark + RDD: functional transformations on partitioned collections of opaque objects.
- SQL + DataFrame: declarative transformations on partitioned collections of tuples.



Creating DataFrames

- The entry point into all functionality in Spark SQL is the SQLContext.
- With a SQLContext, applications can create DataFrames from an existing RDD, from a Hive table, or from data sources.

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val df = sqlContext.read.json(...)
```

DataFrame Operations (1/2)

- Domain-specific language for structured data manipulation.

```
// Show the content of the DataFrame
df.show()
// age  name
// null Michael
// 30   Andy
// 19   Justin

// Print the schema in a tree format
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 (2/2)

- Domain-specific language for structured data manipulation.

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

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

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

Running SQL Queries Programmatically

- Running SQL queries programmatically and returns the result as a DataFrame.
- Using the `sql` function on a `SQLContext`.

```
val sqlContext = ...  // An existing SQLContext
val df = sqlContext.sql("SELECT * FROM table")
```

Converting RDDs into DataFrames

- Inferring the schema using reflection.

```
// Define the schema using a case class.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile(...).map(_.split(","))
               .map(p => Person(p(0), p(1).trim.toInt)).toDF()
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sqlContext
    .sql("SELECT name, age FROM people WHERE age >= 13 AND age <= 19")

// The results of SQL queries are DataFrames.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
teenagers.map(t => "Name: " + t.getAs[String]("name")).collect()
         .foreach(println)
```

Spark Streaming

Motivation

- Many applications must process large streams of live data and provide results in real-time.
  - Wireless sensor networks
  - Traffic management applications
  - Stock marketing
  - Environmental monitoring applications
  - Fraud detection tools
  - ...

Stream Processing Systems

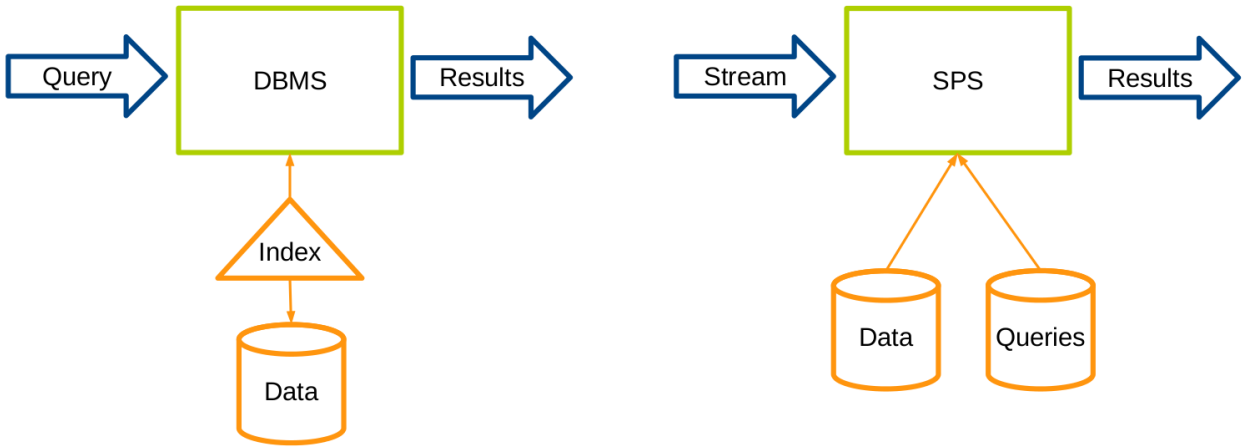
- Database Management Systems (DBMS): data-at-rest analytics
  - Store and index data before processing it.
  - Process data only when explicitly asked by the users.

Stream Processing Systems

- Database Management Systems (DBMS): data-at-rest analytics
  - Store and index data before processing it.
  - Process data only when explicitly asked by the users.
- Stream Processing Systems (SPS): data-in-motion analytics
  - Processing information as it flows, without storing them persistently.

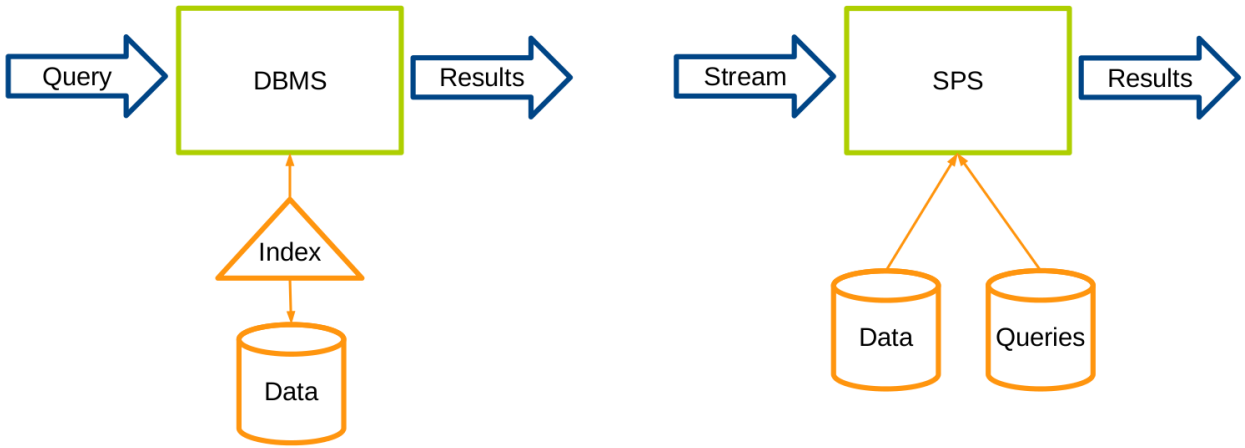
DBMS vs. SPS (1/2)

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



DBMS vs. SPS (2/2)

- DBMS: runs queries just once to return a complete answer.
- SPS: executes standing queries, which run continuously and provide updated answers as new data arrives.



Core Idea of Spark Streaming

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

Spark Streaming

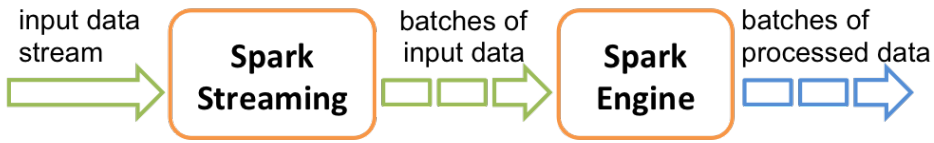
- Run a streaming computation as a series of very small, deterministic batch jobs.

Spark Streaming

- Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chop up the live stream into batches of X seconds.

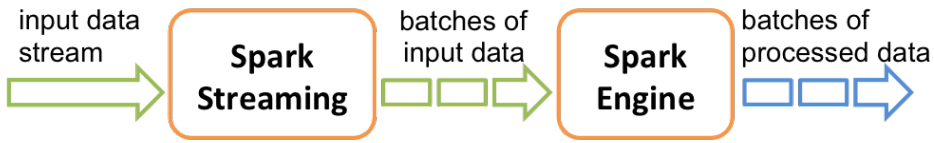
Spark Streaming

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



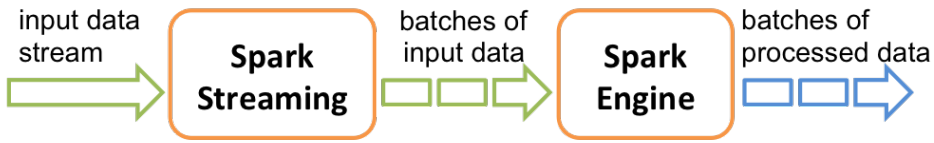
Spark Streaming

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



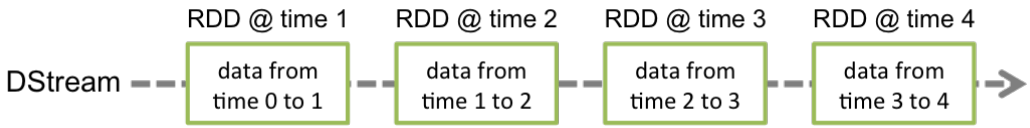
Spark Streaming

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



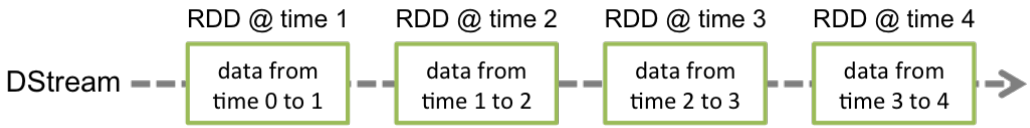
DStream

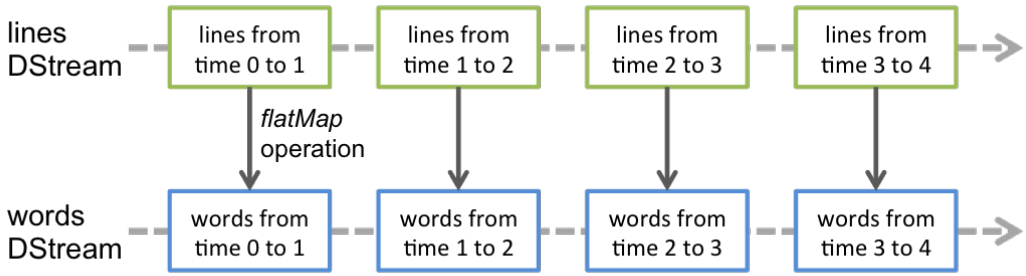
- DStream: sequence of RDDs representing a stream of data.
- Any operation applied on a DStream translates to operations on the underlying RDDs.



DStream

- DStream: sequence of RDDs representing a stream of data.
- Any operation applied on a DStream translates to operations on the underlying RDDs.





StreamingContext

- StreamingContext: the main entry point of all Spark Streaming functionality.
- To initialize a Spark Streaming program, a StreamingContext object has to be created.

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

Source of Streaming

- Two categories of streaming sources.
- Basic sources directly available in the StreamingContext API, e.g., file systems, socket connections, ...
- Advanced sources, e.g., Kafka, Flume, Kinesis, Twitter, ...

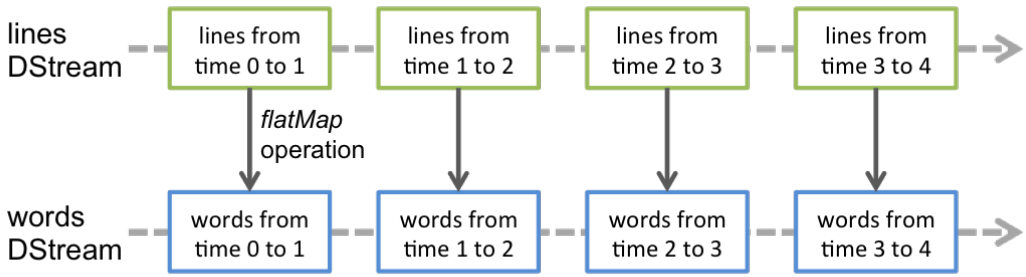
```
ssc.socketTextStream("localhost", 9999)

TwitterUtils.createStream(ssc, None)
```

DStream Transformations

- Transformations: modify data from on DStream to a new DStream.
- Standard RDD operations, e.g., map, join, ...
- DStream operations, e.g., window operations

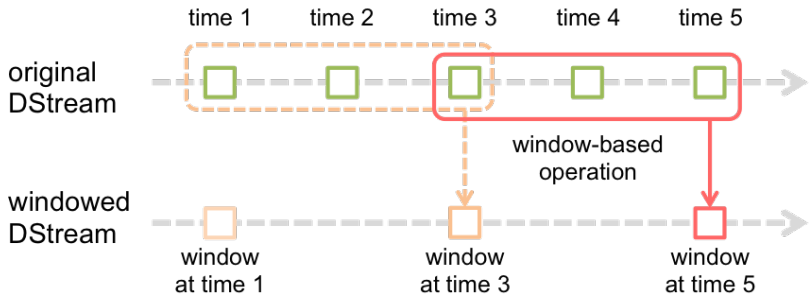
DStream Transformation Example

```
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))

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

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

wordCounts.print()
```



Window Operations

- Apply transformations over a sliding window of data: window length and slide interval.




```
val ssc = new StreamingContext(conf, Seconds(1))
val lines = ssc.socketTextStream(IP, Port)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))

val windowedWordCounts = pairs.reduceByKeyAndWindow(_ + _,
  Seconds(30), Seconds(10))
```

MapWithState Operation

- Maintains state while continuously updating it with new information.
- It requires the checkpoint directory.
- A new operation after updateStateByKey.

```
val ssc = new StreamingContext(conf, Seconds(1))
ssc.checkpoint(".")

val lines = ssc.socketTextStream(IP, Port)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))

val stateWordCount = pairs.mapWithState(
  StateSpec.function(mappingFunc))

val mappingFunc = (word: String, one: Option[Int], state: State[Int]) => {
  val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
  state.update(sum)
  (word, sum)
}
```

Spark Streaming and DataFrame

```
val words: DStream[String] = ...

words.foreachRDD { rdd =>
  // Get the singleton instance of SQLContext
  val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)
  import sqlContext.implicits._

  // Convert RDD[String] to DataFrame
  val wordsDataFrame = rdd.toDF("word")

  // Register as table
  wordsDataFrame.registerTempTable("words")

  // Do word count on DataFrame using SQL and print it
  val wordCountsDataFrame =
    sqlContext.sql("select word, count(*) as total from words group by word")
  wordCountsDataFrame.show()
}
```

GraphX

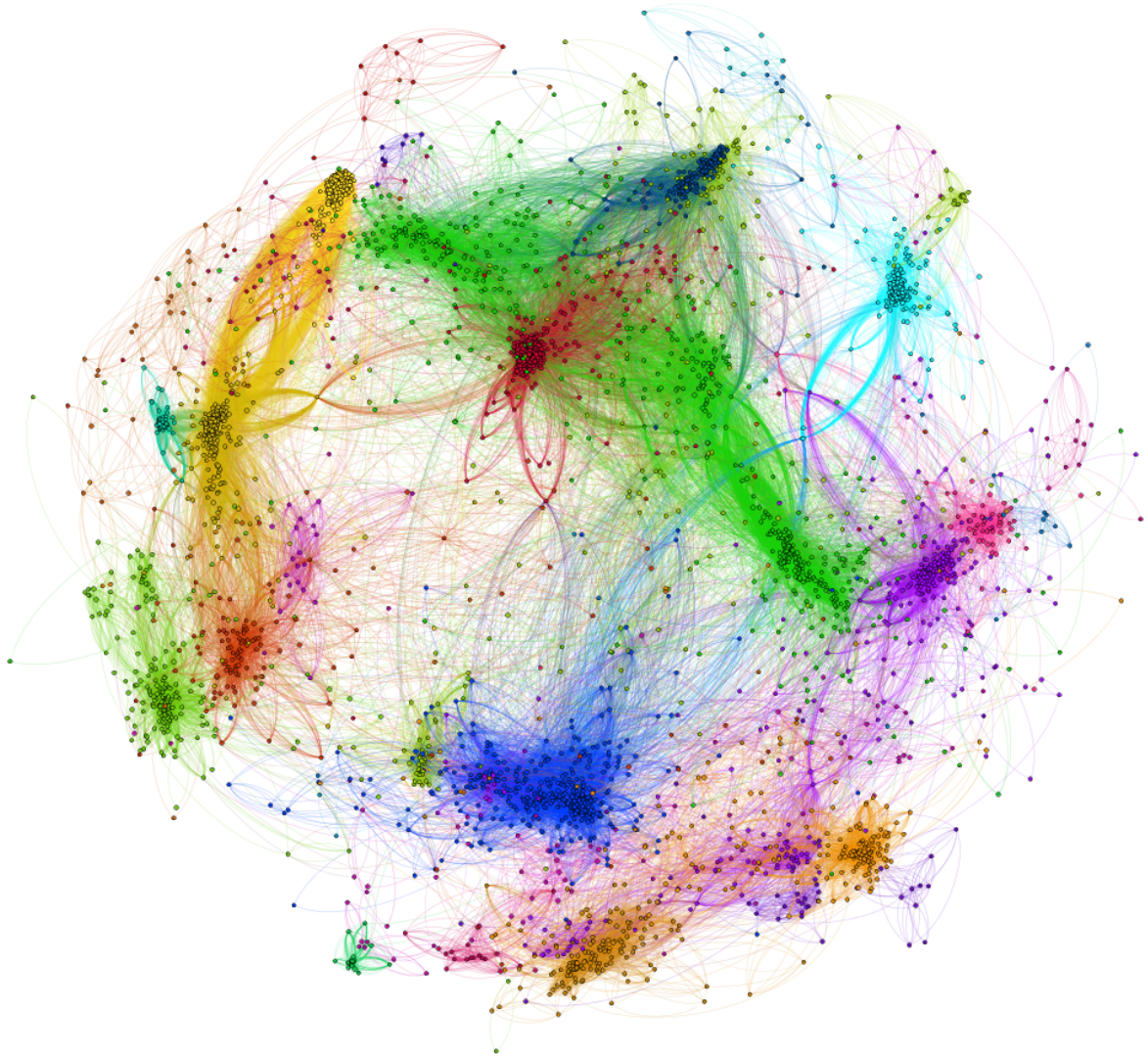

SOUNDCLOUD


You
Tube

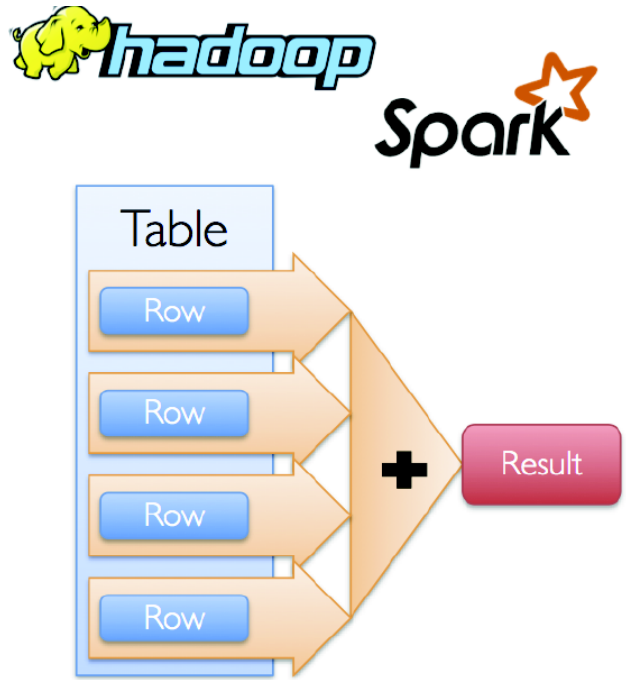
Introduction

- Graphs provide a flexible abstraction for describing relationships between discrete objects.
- Many problems can be modeled by graphs and solved with appropriate graph algorithms.

Large Graph



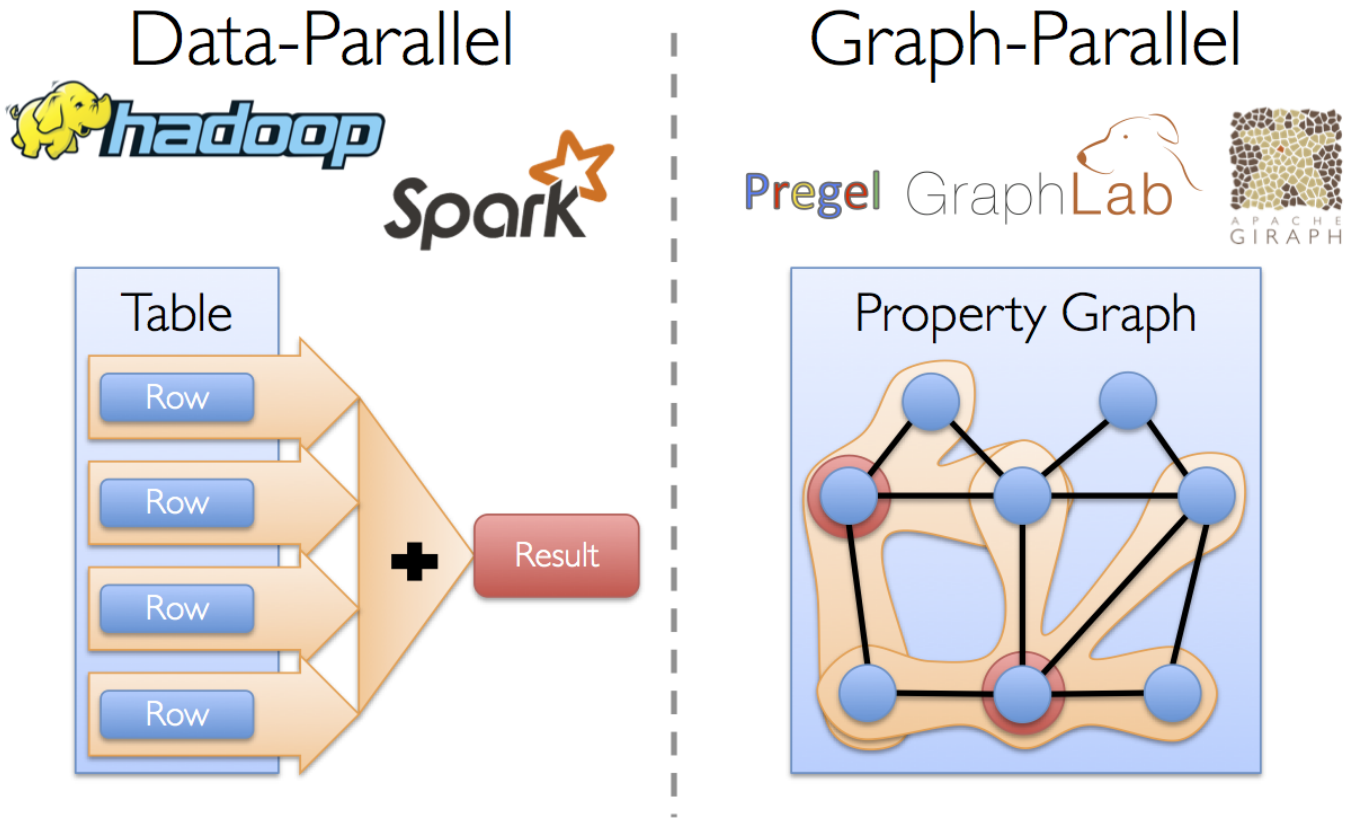
Can we use platforms like MapReduce or Spark, which are based on data-parallel model, for large-scale graph proceeding?



Graph-Parallel Processing

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

Data-Parallel vs. Graph-Parallel Computation (1/3)

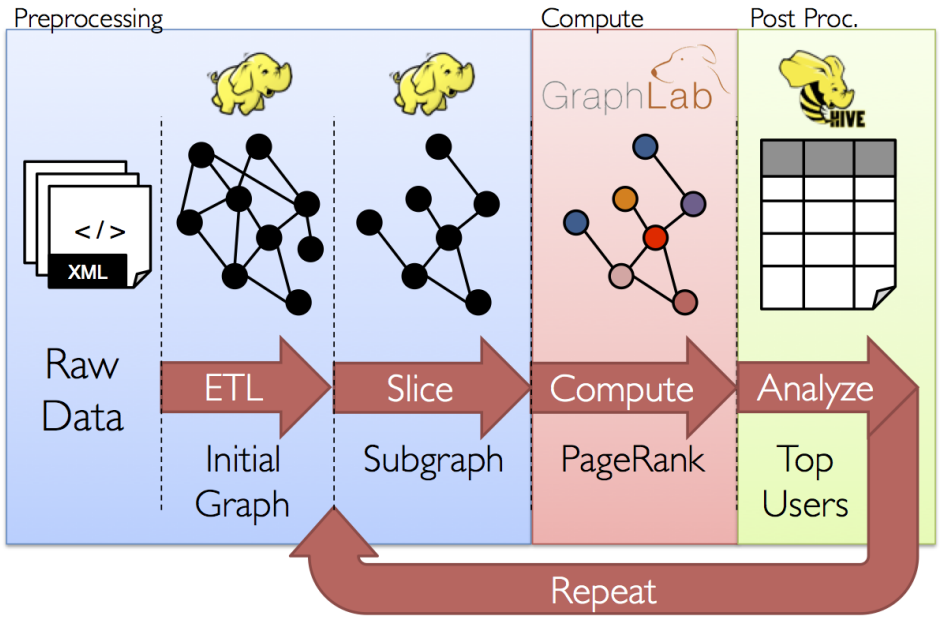


Data-Parallel vs. Graph-Parallel Computation (2/3)

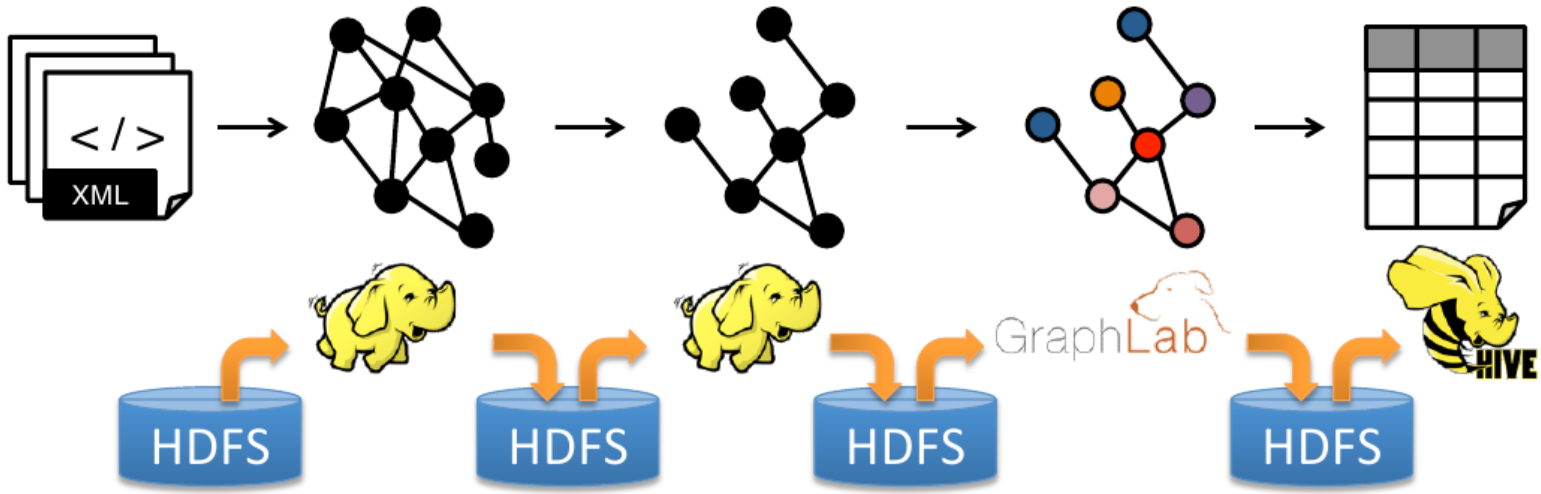
- Graph-parallel computation: restricting the types of computation to achieve performance.

Data-Parallel vs. Graph-Parallel Computation (2/3)

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



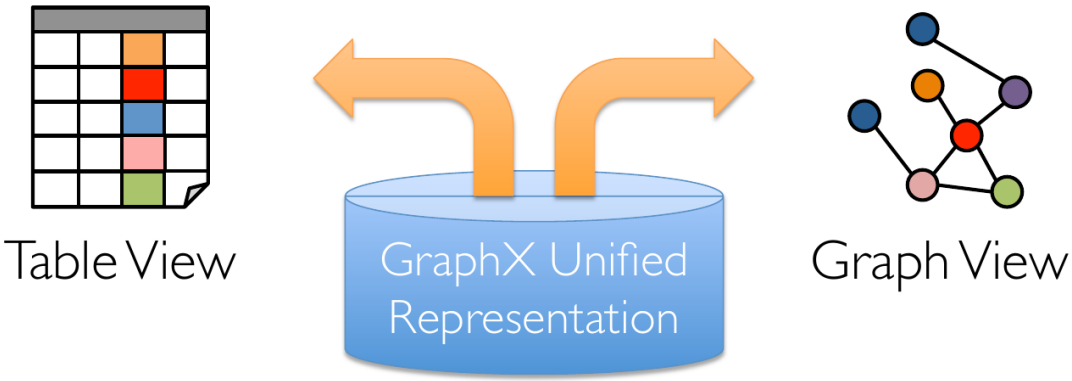
Data-Parallel vs. Graph-Parallel Computation (3/3)



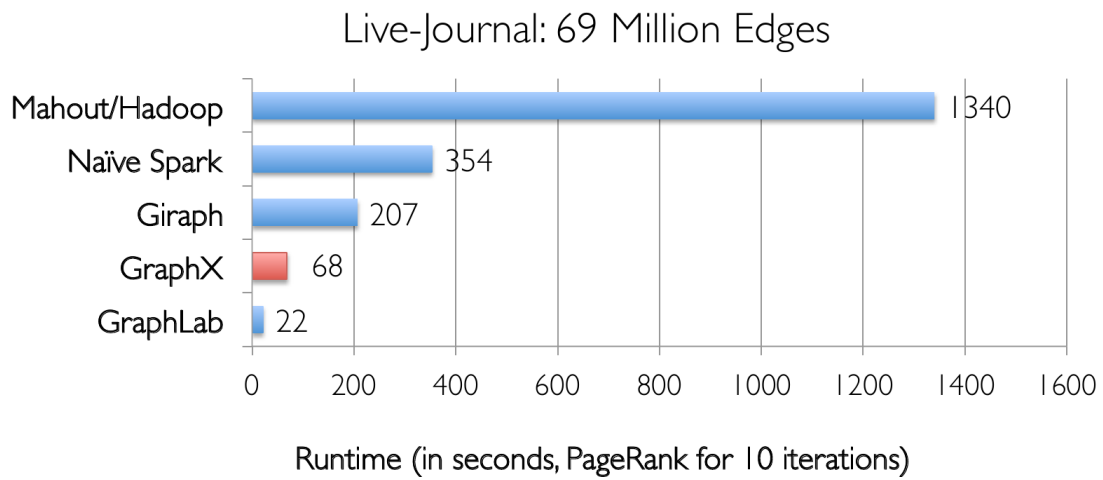
- Moving between table and graph views of the same physical data.
- Inefficient: extensive data movement and duplication across the network and file system.

GraphX

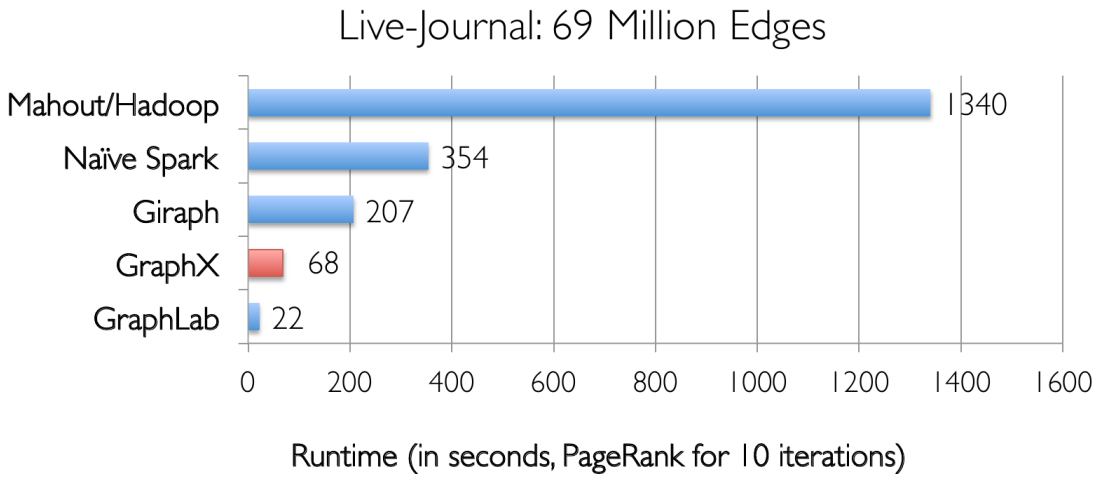
- Unifies data-parallel and graph-parallel systems.
- Tables and Graphs are composable views of the same physical data.
- Implemented on top of Spark.

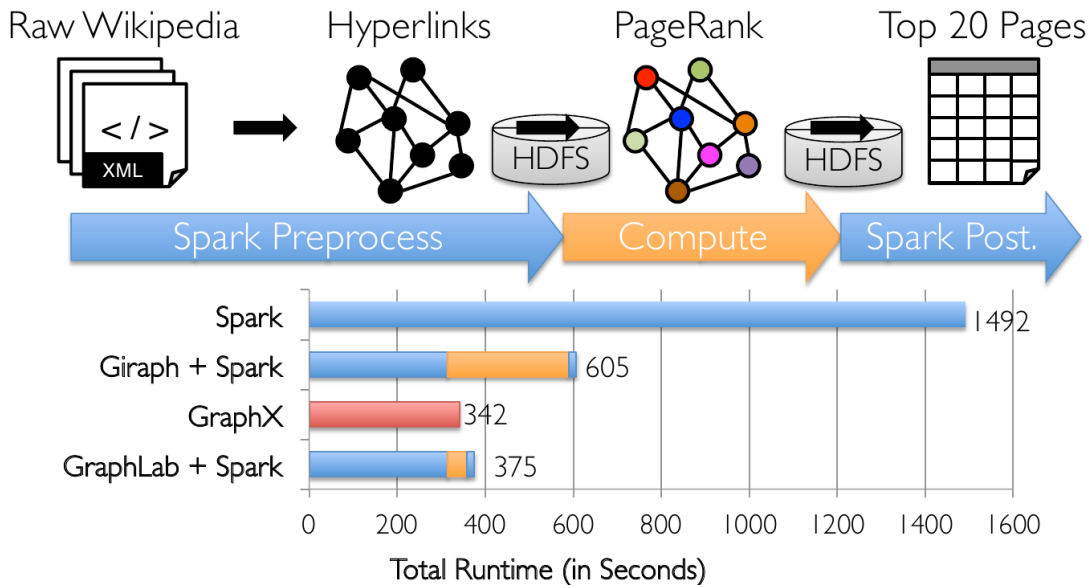


GraphX vs. Data-Parallel/Graph-Parallel Systems



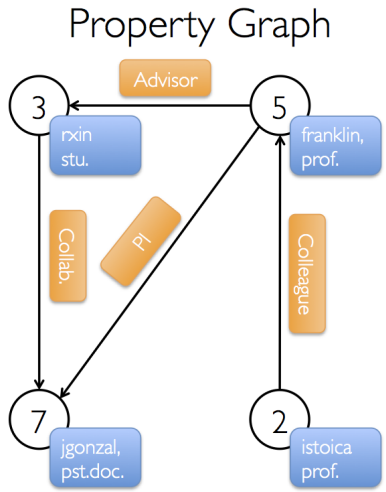
GraphX vs. Data-Parallel/Graph-Parallel Systems





Property Graph

- Represented using two Spark RDDs:
  - Edge collection: VertexRDD
  - Vertex collection: EdgeRDD



Vertex Table

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

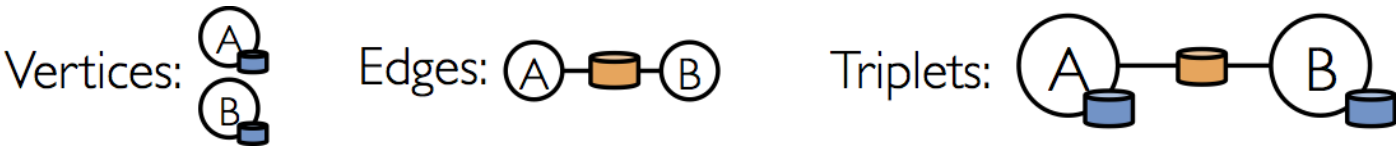
Edge Table

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

```
// 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]
}
```

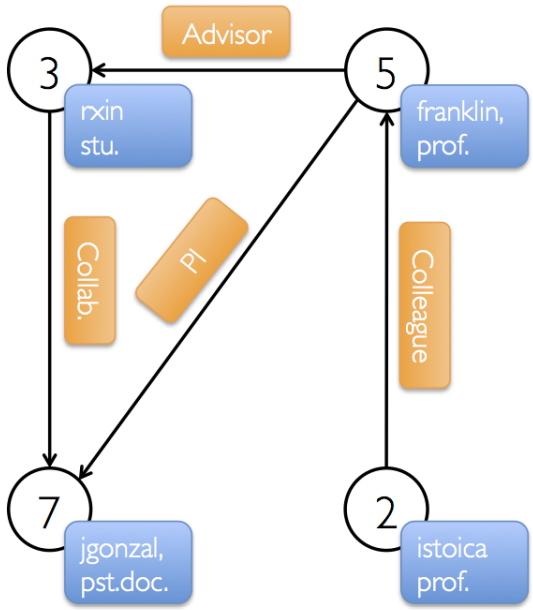
Triplets

- The triplet view logically joins the vertex and edge properties yielding an `RDD[EdgeTriplet[VD, ED]]`.



Example Property Graph (1/3)

Property Graph



Vertex Table

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

Edge Table

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

Example Property Graph (2/3)

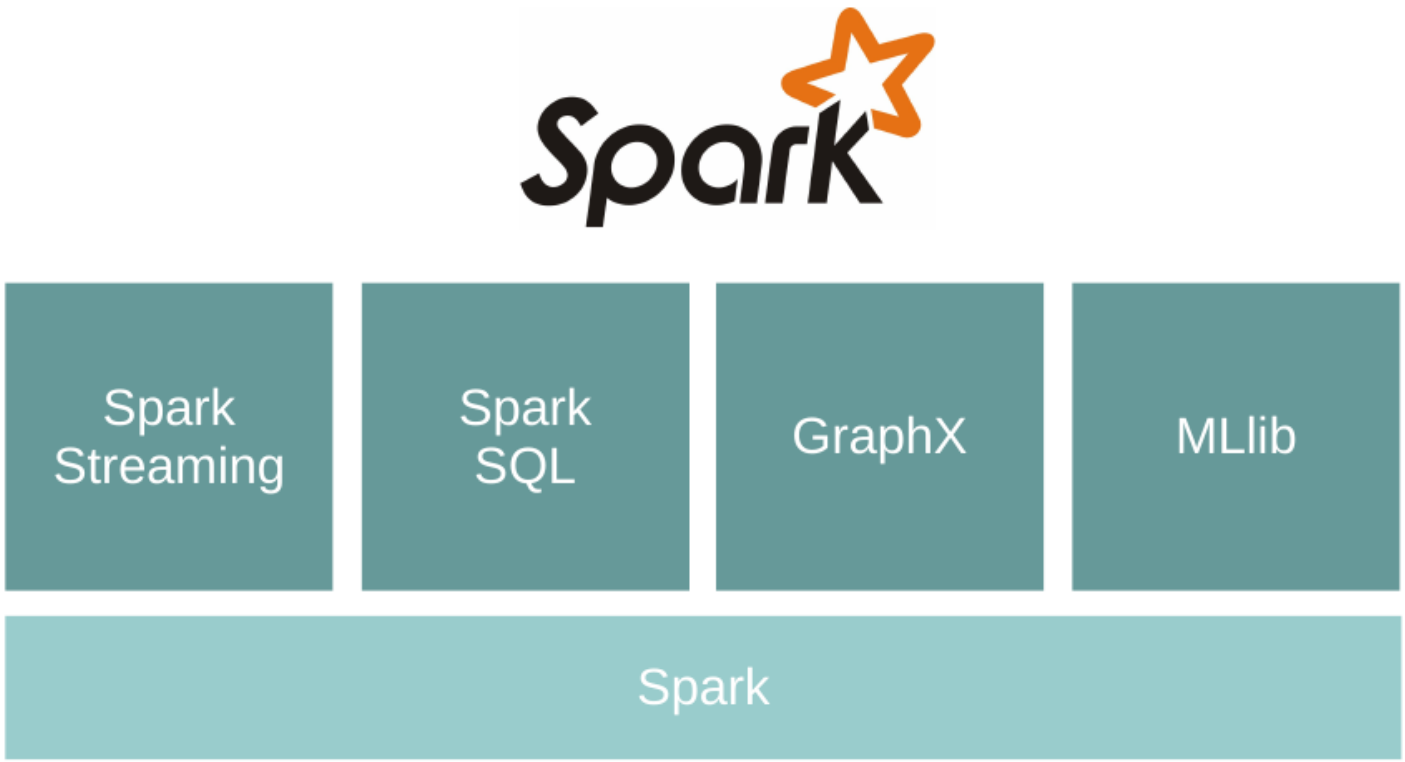
```
val sc: SparkContext

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

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

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

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

Example Property Graph (3/3)

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

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

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

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

Summary

Summary



Questions?