BIG DATA

Scala Programming Language

Prof. J.Ruby Dinakar
Department of Computer Science and Engineering

Acknowledgements:
Significant information in the slide deck presented through the Unit 3 of the course have been created by **Dr. K V Subramaniam's** and would like to acknowledge and thank him for the same. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for classroom presentation only.

References

Programming in Scala

Martin Odersky, Lex Spoon, Bill Venners

BIG DATA

Issues with Hadoop

Issue with Small Files

- Hadoop does not suit for small data files due to its high capacity design
- A small file is significantly smaller than the HDFS block size (default 128MB)
- Too many small files cause problem for managing the Namenode.

Slow Processing Speed

- Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

Support for Batch Processing only and No Real-time Data Processing

- Hadoop supports batch processing only, it does not process streamed data, and hence overall performance is slower

No Delta Iteration

- Hadoop is not so efficient for iterative processing, as Hadoop does not support cyclic data flow(i.e. a chain of stages in which each output of the previous stage is the input to the next stage).

BIG DATA

Apache Spark vs. Apache Hadoop

- Apache Hadoop and Apache Spark are both open-source frameworks for big data processing with some key differences.
- Hadoop uses the MapReduce to process data, while Spark uses resilient distributed datasets (RDDs).
- Hadoop has a distributed file system (HDFS), meaning that data files can be stored across multiple machines.
- The file system is scalable, since servers and machines can be added to accommodate increasing volumes of data. Spark does not provide a distributed file storage system, so it is mainly used for computation, on top of Hadoop.
- Spark does not need Hadoop to run, but can be used with Hadoop since it can create distributed datasets from files stored in the HDFS.

BIG DATA

Apache Spark vs. Scala

- Spark is an open-source distributed general-purpose cluster-computing framework.
- Scala is a general-purpose programming language providing support for functional programming and a strong static type system.
- Spark is used to increase the Hadoop computational process.
- Scala can be used for web applications, streaming data, distributed applications and parallel processing.

BIG DATA

What is Scala?

- SCAlable Language developed by Marvin Odersky at EPFL (Switzerland)
- Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way.
- It seamlessly integrates features of object-oriented and functional languages.

Scala is object-oriented

- every value is an object. Types and behaviors of objects are described by classes and traits.
- Classes can be extended by subclassing, and by using a flexible mixin-based composition mechanism as a clean replacement for multiple inheritance.

Scala is functional

- every function is a value. Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and it supports currying(transforms functions with multiple arguments into a single function arguments).
- Scala's case classes and its built-in support for pattern matching provide the functionality of algebraic types, which are used in many functional languages.
- Singleton objects provide a convenient way to group functions that aren't members of a class.
- ideal for developing applications like web services, data intensive applications.

BIG DATA

What is Scala?

Scala is statically typed

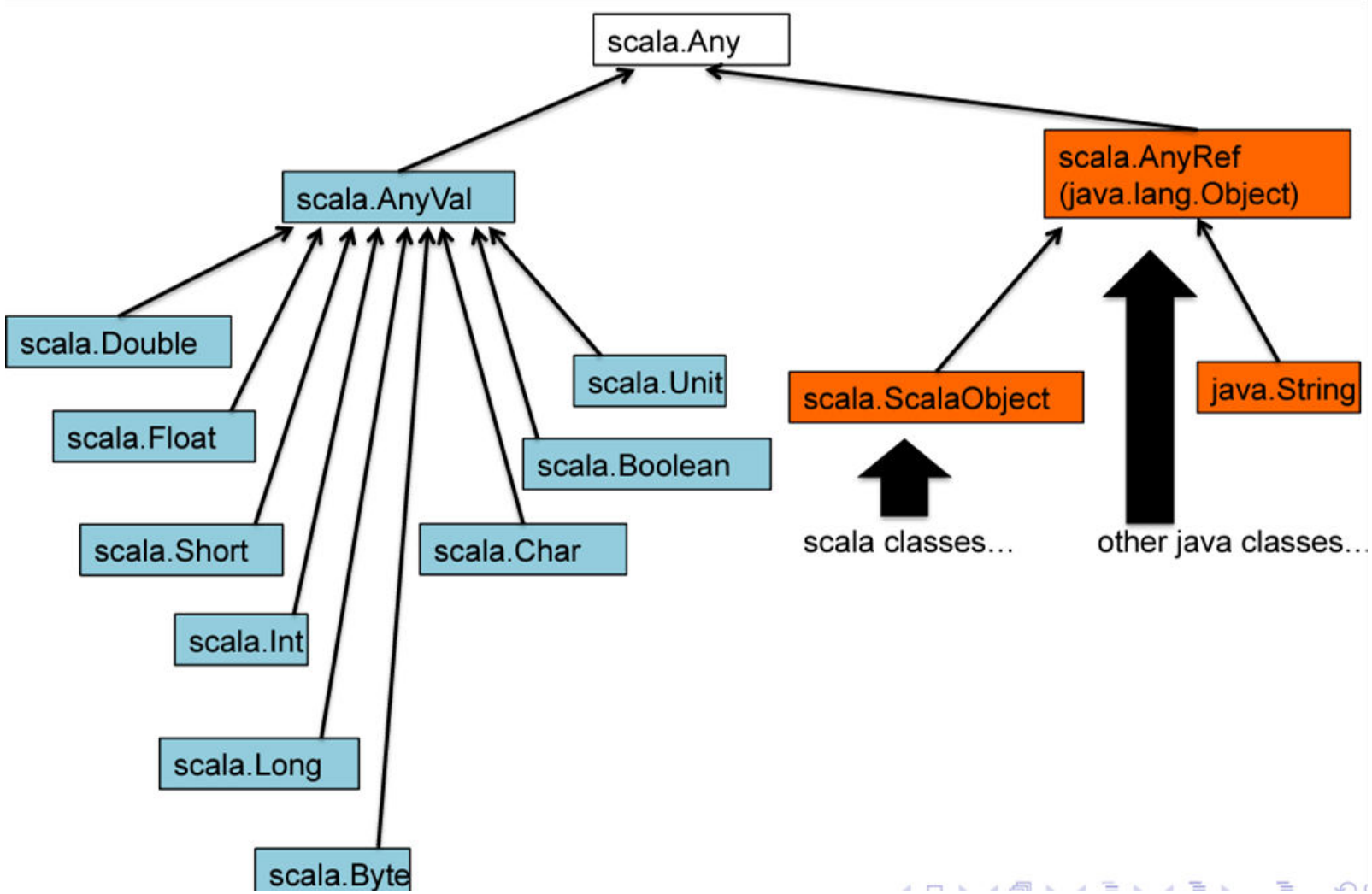
It supports:

- Generic classes
- Variance annotations
- Upper and lower type bounds
- Inner classes and abstract type members as object members
- Compound types
- Explicitly typed self references
- Implicit parameters and conversions
- Polymorphic methods

Type inference means the user is not required to annotate code with redundant type information.

The Scala compiler can often infer the type of an expression so you don't have to declare it explicitly.

Scala class hierarchy



BIG DATA

Java Vs Scala

- Scala is very similar to Java Language. It compiled to byte codes and use Java Virtual Machine(JVM) to execute code.
- Scala reduces the number of lines from a Java application by making clever use of type inference, treating everything as an object, function passing, and several other features.
- Scala has built-in lazy evaluation, which allows deferring time-consuming computation until absolutely needed and we can do this by using a keyword called “lazy”.
- Scala supports Operator overloading but Java does not support Operator overloading .
- Functions are treated as objects in Java but Scala treats any method or function as variables.
- Syntax in Scala is more complicated than Java.
- Java is backward compatible but Scala provides limited backward compatibility

BIG DATA

Scala is interoperable

Scala programs interoperate seamlessly with Java class libraries:

- Method calls
- Field accesses
- Class inheritance
- Interface implementation

all work as in Java.

Scala programs compile to JVM bytecodes.

Scala's syntax resembles [illegible] but there [illegible] differenc[illegible]

```
object Example1 {
  def main(args: Array[String]) {
    val b = new StringBuilder()
    for (i ← 0 until args.length) {
      if (i > 0) b.append(" ")
      b.append(args(i).toUpperCase)
    }
    Console.println(b.toString)
  }
}
```

Array[String] instead of String[]

Scala's version of the extended **for** loop

Arrays are indexed args(i) instead of args[i]

Program to change the command line argument input to uppercase

BIG DATA

Java and Scala: Spot the differences

Declaring variables:

```
var x: Int = 7
var x = 7 // type inferred
val y = "hi" // read-only
```

Java equivalent:

```
int x = 7;

final String y = "hi";
```

Functions:

```
def square(x: Int): Int = x*x
def square(x: Int): Int = {
  x*x
}
def announce(text: String) =
{
  println(text)
}
```

Java equivalent:

```
int square(int x) {
  return x*x;
}

void announce(String text) {
  System.out.println(text);
}
```

BIG DATA

Java and Scala: Class

... in Java:

```
public class Person {
  public final String name;
  public final int age;
  Person(String name, int age) {
      this.name = name;
      this.age = age;
  }
}
```

... in Scala:

```
class Person(val name: String,
             val age: Int) {}
```

BIG DATA

Java and Scala: Usage of Class

... in Java:

```
import java.util.ArrayList;
...
Person[] people;
Person[] minors;
Person[] adults;
{  ArrayList<Person> minorsList = new ArrayList<Person>();
   ArrayList<Person> adultsList = new ArrayList<Person>();
   for (int i = 0; i < people.length; i++)
       (people[i].age < 18 ? minorsList : adultsList)
                     .add(people[i]);
   minors = minorsList.toArray(people);
   adults = adultsList.toArray(people);
}
```

This program partition the data based on age. <18 is stored in minors list and >18 is stored in adults list.

... in Scala:

```
val people: Array[Person]
val (minors, adults) = people partition (_.age < 18)
```

A function value

An infix method call

A simple pattern match

Java and Scala: Definitions

Scala method definitions:

```
def fun(x: Int): Int = {
   result
}

def fun = result
```

Scala variable definitions:

```
var x: int = expression
val x: String = expression
```

Java method definition:

```
int fun(int x) {
   return result
}
```

(no parameterless methods)

Java variable definitions:

```
int x = expression
final String x = expression
```

BIG DATA

Java and Scala: Expressions

Scala method calls:

obj.meth(arg)

or: obj meth arg

Scala choice expressions:

if (cond) expr1 else expr2

expr match {
  case pat_1 => $expr_1$
  ....
  case pat_n => $expr_n$
}

Java method call:

obj.meth(arg)

(no operator overloading)

Java choice expressions, stats:

cond ? expr1 : expr2 // expression

if (cond) return expr1; // statement
else return expr2;

switch (expr) {
  case pat_1 : return $expr_1$;
  ...
  case pat_n : return $expr_n$;
} // statement only

BIG DATA

Java and Scala: Expressions

Scala method calls:

obj.meth(arg)

or: obj meth arg

Scala choice expressions:

if (cond) expr1 else expr2

expr match {
  case pat_1 => $expr_1$
  ....
  case pat_n => $expr_n$
}

Java method call:

obj.meth(arg)

(no operator overloading)

Java choice expressions, stats:

cond ? expr1 : expr2 // expression

if (cond) return expr1; // statement
else return expr2;

switch (expr) {
  case pat_1 : return $expr_1$;
  ...
  case pat_n : return $expr_n$;
} // statement only

BIG DATA

Java and Scala: Traits

Scala Trait

```
trait T {
   def abstractMeth(x: String): String

   def concreteMeth(x: String) =
      x+field

   var field = "!"
}
```

Scala mixin composition:

```
class C extends Super with T
```

Java Interface

```
interface T {
   String abstractMeth(String x)

   (no concrete methods)


   (no fields)
}
```

Java extension + implementation:

```
class C extends Super implements T
```

Scala:

- Minimal Verbosity
- Referential Transparency – type inferencing
- Concurrency
- Functional Programming

BIG DATA

Minimal Verbosity

- Java:
  - class Person {
    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName, int age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    }

    public void setFirstName(String firstName) { this.firstName = firstName; }
    public void String getFirstName() { return this.firstName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
    public void String getLastName() { return this.lastName; }
    public void setAge(int age) { this.age = age; }
    public void int getAge() { return this.age; }
    }
- Scala:
  - class Person(var firstName: String, var lastName: String, var age: Int)
- Source: http://blog.objectmentor.com/articles/2008/08/03/the-seductions-of-scala-part-i

BIG DATA

Type Inferencing

- Java is statically typed--a variable has a type, and can hold only values of that type
  - You must specify the type of every variable
  - Type errors are caught by the compiler, not at runtime--this is a big win
  - However, it leads to a lot of typing (pun intended)
- Languages like Ruby and Python don’t make you declare types
  - Easier (and more fun) to write programs
  - Less fun to debug, especially if you have even slightly complicated types

BIG DATA

Type Inferencing

- Scala is also statically typed, but it uses type inferencing--that is, it figures out the types, so you don't have to
  - **The good news**: Less typing, more fun, type errors caught by the compiler
  - **The bad news**: More kinds of error messages to get familiar with

- Example
  - `val collegeName = "PES University"`
  - `def squareOf(x: Int) : x * x`

Source: https://docs.scala-lang.org/tour/type-inference.html

Consistency

- In Java, every value is an object--unless it's a primitive
  - Numbers and booleans are primitives for reasons of efficiency, so we have to treat them differently (you can't "talk" to a primitive)
  - In Scala, all values are objects.
  - The compiler turns them into primitives, so no efficiency is lost (behind the scenes, there are objects like RichInt)

- Java has operators (+, <, ...) and methods, with different syntax
  - In Scala, operators are just methods, and in many cases you can use either syntax

BIG DATA

Concurrency

- Broadly speaking, concurrency can be either:
  - Fine-grained: Frequent interactions between threads working closely together (extremely challenging to get right)
  - Coarse-grained: Infrequent interactions between largely independent sequential processes (much easier to get right)
- Java 5 and 6 provide reasonable support for traditional fine-grained concurrency
  - Threads
- Scala has total access to the Java API
  - Hence, it can do anything Java can do
  - And it can do much more (see next slide)
- Scala also has Actors for coarse-grained concurrency
  - Sending messages (use the send ! Abstraction)

Concurrency

- The best-known functional languages are ML, OCaml, and Haskell
- Functional languages are regarded as:
  - "Ivory tower languages," used only by academics (mostly but not entirely true)
  - Difficult to learn (mostly true)
  - The solution to all concurrent programming problems everywhere (exaggerated, but not entirely wrong)
- Scala is an "impure" functional language--you can program functionally, but it isn't forced upon you

BIG DATA

Functional Programming

- The big problem with concurrency is dealing with shared state--multiple threads all trying to read and maybe change the value of same variables
- If all data were immutable, then any thread could read it any time, no synchronization, no locks, no problem
- But if your program couldn’t ever change anything, then it couldn’t ever do anything, right?
- Wrong!
- There is an entire class of programming languages that use only immutable data, but work just fine: the functional languages

BIG DATA

Functional Programming

- Immutable – functional operations create new structures
  - Don’t modify existing structures
- Program implicitly captures data flow
- Order of operations not significant
- Functions
  - Are objects
  - can be passed as arguments to functions
  - can return functions
  - Can operate on collections

Functional Programming Example

BIG DATA

Functional Programming – differences with java

Quicksort program in Scala – java style

Observe

Focus on HOW?

Explicitly iterate

Determine when and how to swap()

```
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort1(l: Int, r: Int) {
    val pivot = xs((l + r) / 2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) i += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1
        j -= 1
      }
    }
    if (l < j) sort1(l, j)
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
}
```

Functional Programming – differences with java

Focus on solving the problem

Observe

Focus on WHAT, not HOW

```
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
           xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
}
```

Pick a pivot

Sort values smaller than the pivot

Sort values larger than the pivot

Concatenate the result

BIG DATA

Self Learning Exercise

- Does this sort array in ascending order or descending order?
- Consider the program with array
  - xs=3,1,2,0,7,6,4,5
- Write a program to sort in the reverse order (if ascending, sort descending)
- How can we parallelize this?

```
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
           xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
}
```

Functional Programming and Big Data

- Big data architectures leverage

  Parallel disk, memory and CPU in clusters

- Operations consist of independently parallel operations

  Similar to *map()* operator in a functional language

- Parallel operations have to be consolidated

  Similar to *aggregation()* operators in functional languages

- In order to achieve the desired efficiency for big data applications, Concurrency, Parallelism and Referential Transparency are key, and in a functional environment, all of them are naturally present.

Functional programming and collections

Original List is left unchanged

```
val list = List(1, 2, 3)
list.foreach(x => println(x)) // prints 1, 2, 3
list.foreach(println)         // same

list.map(x => x + 2)    // returns a new List(3, 4, 5)
list.map(_ + 2)         // same

list.filter(x => x % 2 == 1)// returns a new List(1, 3)
list.filter(_ % 2 == 1)     // same

list.reduce((x, y) => x + y) // => 6
list.reduce(_ + _)           // same
```

THANK YOU

Prof. J.Ruby Dinakar
Dept. of Computer Science and Engineering
rubydinakar@pes.edu

BIG DATA

In memory analytics with Spark : Introduction

Prof. J.Ruby Dinakar
Department of Computer Science and Engineering

Acknowledgements:
Significant information in the slide deck presented through the Unit 3 of the course have been created by **Dr. K V Subramaniam's** and would like to acknowledge and thank him for the same. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for classroom presentation only.

BIG DATA

Overview of lecture – Spark Introduction

- Apache Spark
- Why Spark – the motivation?
- Moving to in-memory compute
- Distribute data in-memory
- Putting it all together : Word count in Spark
- Spark Use cases

Apache Spark

BIG DATA

Apache Spark

Apache® Spark™ is a powerful open source unified data processing engine built around speed, ease of use, and sophisticated analytics. It was originally developed at UC Berkeley in 2009.

It can be used instead of map reduce.

Internet powerhouses such as Netflix, Yahoo, and eBay have deployed Spark at massive scale, collectively processing multiple petabytes of data on clusters of over 8,000 nodes.

It has quickly become the largest open source community in big data

With over 1000 contributors from 250+ organizations.

Motivation for Spark

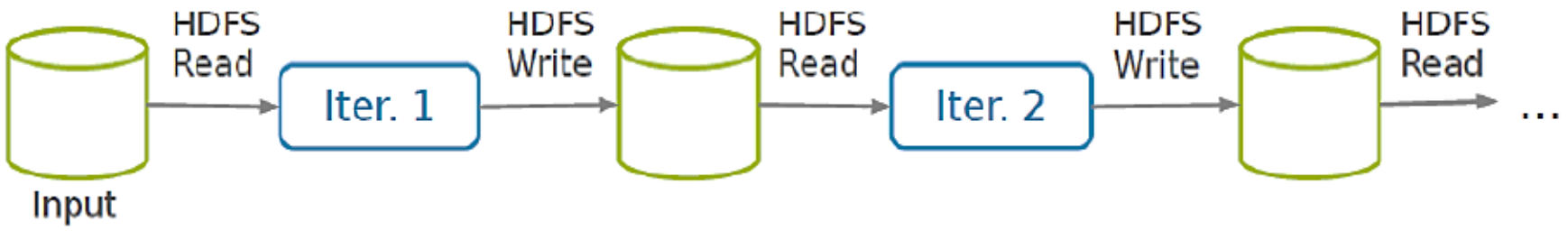
MapReduce : Weakness and Limitations

Programming Model

- Hard to implement everything as a mapreduce program.
- Multiple MapReduce steps are required for even simple operations.
- E.g. Word count that sorts by their frequency.
- Lack of control, structure and data types.

No native support for iteration

- Each iteration reads/writes data from/to disk : overhead
- Need to design algorithms to reduce the number of iterations.



Mapreduce Limitations

Efficiency

Higher communication cost due to map, shuffle, combine and reduce tasks

Frequent writing of output to disk

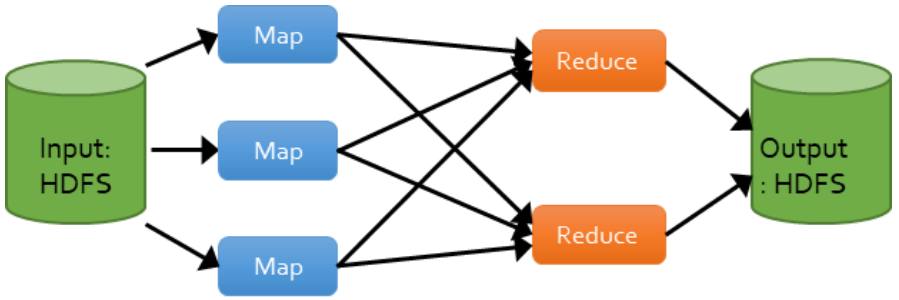
Limited exploitation of Main memory

Not feasible for Real time stream processing

MapReduce job needs to scan the entire input.

Alternate Model

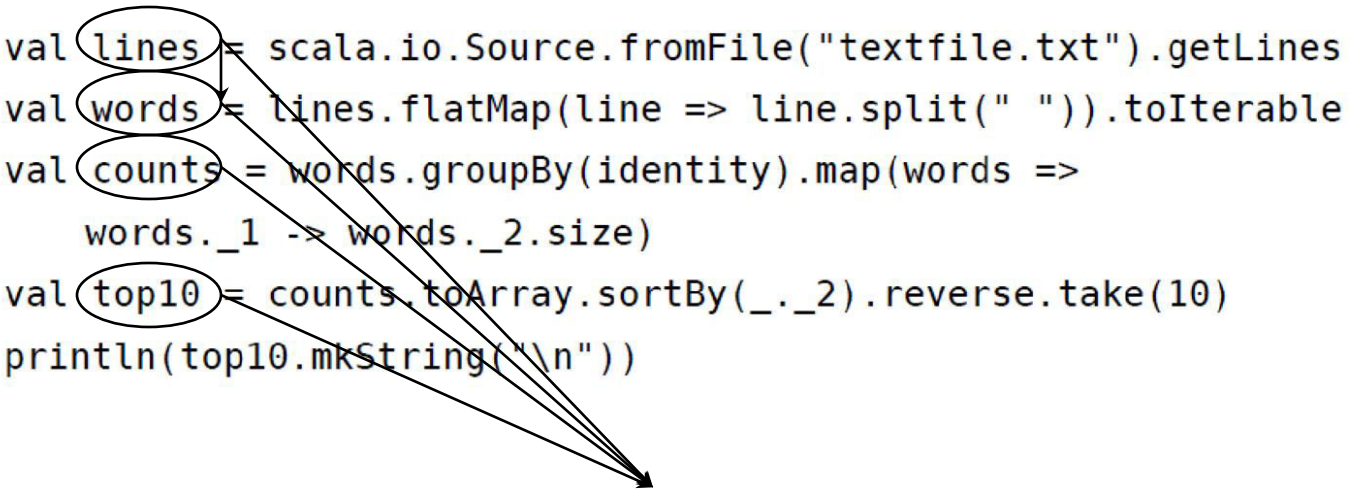
- Acyclic data flow
  - consider operating on a *data working set*
  - **Working set:** same set of data reused
  - in page rank, we keep computing importance vector and reusing in next iteration.
  - Hadoop – inefficient in such cases.
- Example of use
  - Where we need to *iterate*
    - Graph processing
    - Machine Learning
  - Where we need to do *interactive analysis*
    - Python, R
- On every iteration, storing/reloading of data from persistent storage is time consuming.



In Memory Computation

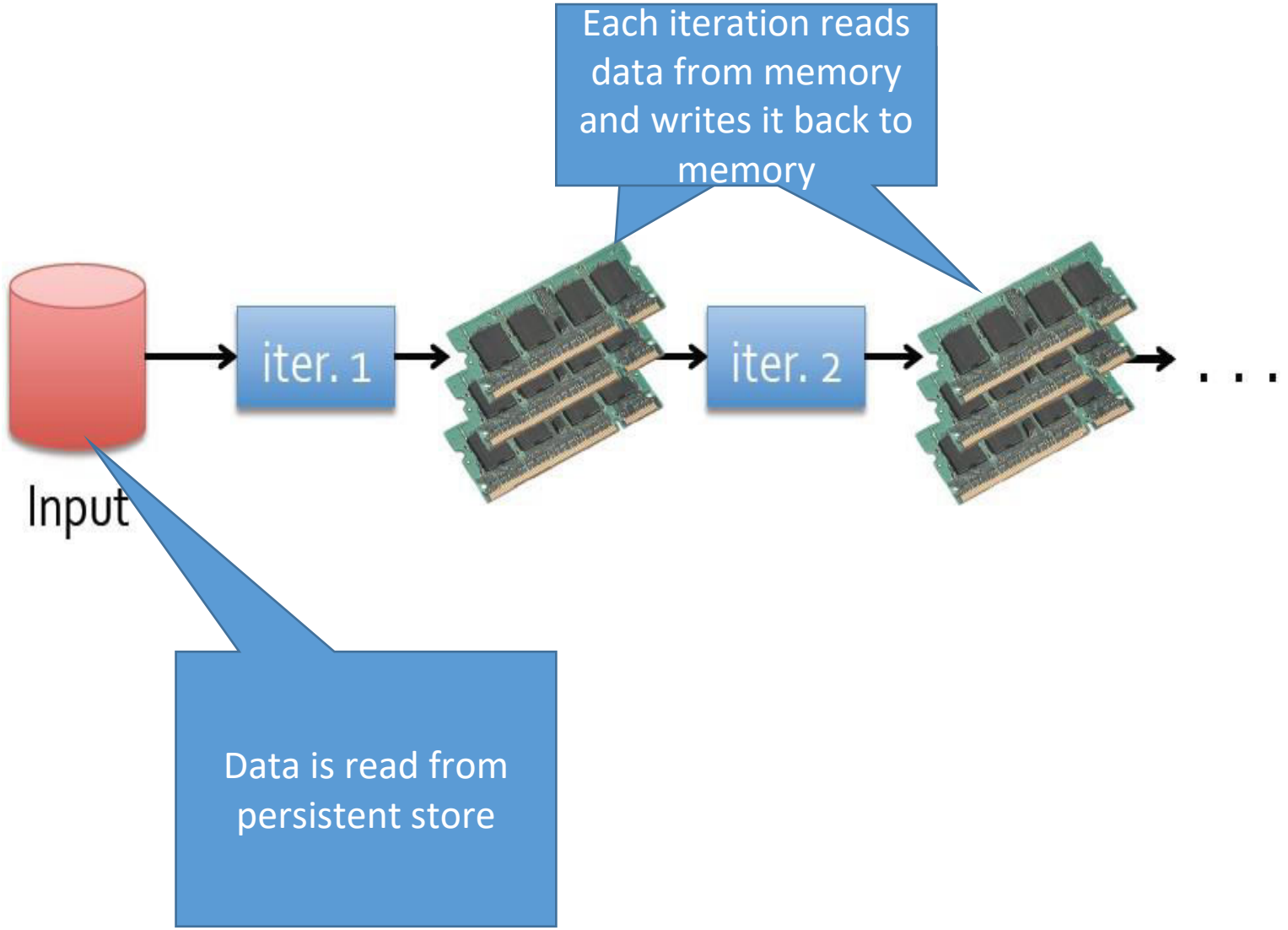
BIG DATA

Word count in scala



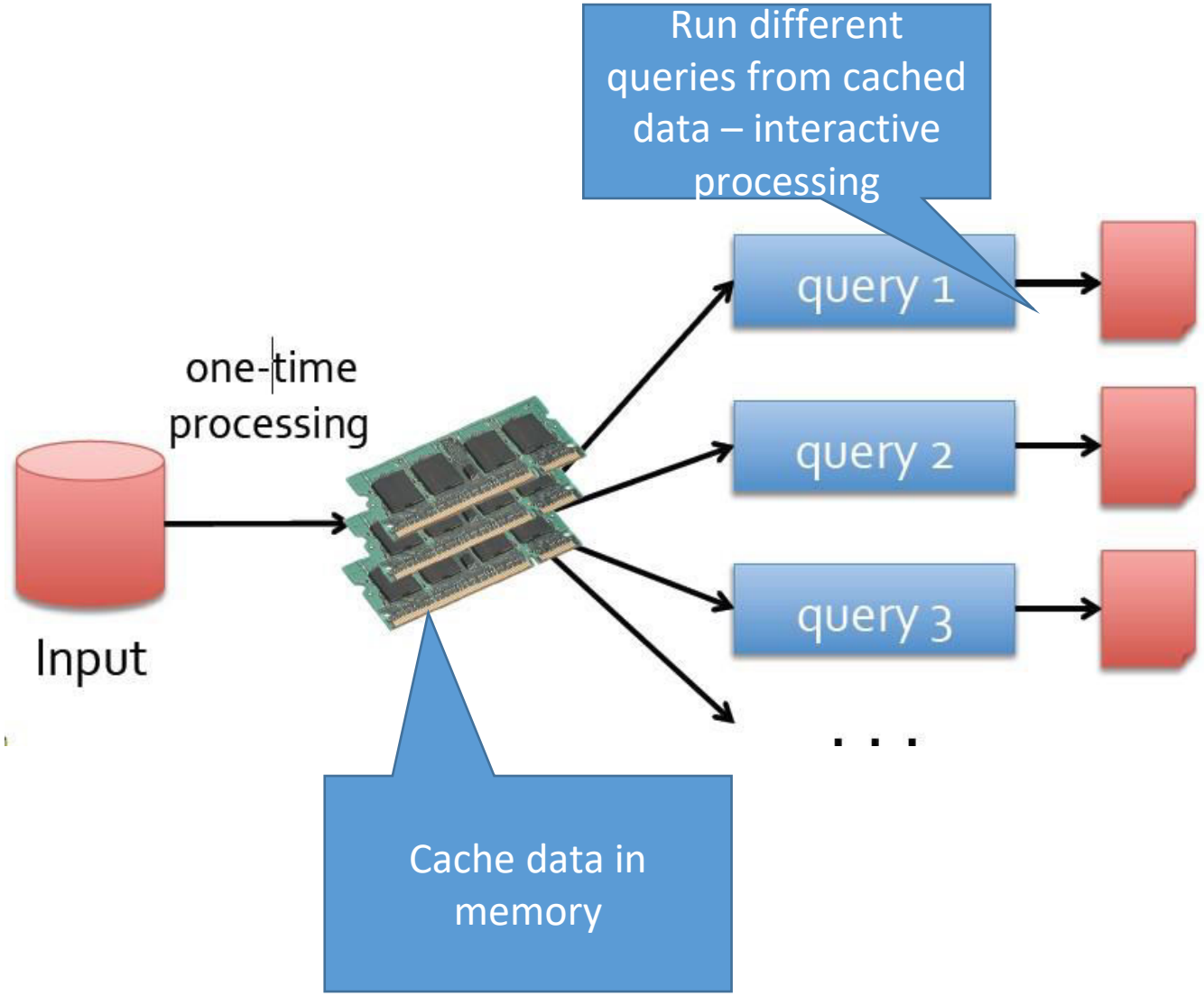
Each operation creates
A data value that can be kept in
Memory and reused.

Doing in memory processing – Iterative processing



BIG DATA

Doing in memory processing – interactive processing



Challenges of in memory processing

- How do we distribute the data among the DRAM of the cluster?
- What happens if this memory is not sufficient?
- How do we handle failures because memory is volatile?

Distributed Dataset

What is an RDD?

- When we add lineage information to the concept of a Distributed Dataset
  - We add ability to recreate it in case of failure
  - So, this data is now *resilient* to failures.
  - Hence called an ***RDD:*** *Resilient Distributed Dataset*

How is lineage information stored

- Lineage information is stored by keeping track of
  - Operations that are performed on
    - An RDD
    - That results in another RDD
  - What types of operations are supported?

BIG DATA

Example: Log Processing

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(startsWithERROR())
messages = errors.map(split(“\t”),2)
cachedMsgs = messages.cache()

cachedMsgs.filter(containsfoo()).count()
cachedMsgs.filter(containsbar()).count()
. . .
```

Courtesy: Zaharia et al , “Spark: Fast, Interactive, Language-Integrated Cluster Computing”, www.spark-project.org

BIG DATA

Example: Log Processing

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(startsWithERROR())
messages = errors.map(split(“\t”),2)
cachedMsgs = messages.cache()

cachedMsgs.filter(containsfoo()).count()
cachedMsgs.filter(containsbar()).count()
...
```

Loads a file to an in memory struct called an RDD (think of it as a collection of strings)

Filter function to retain only those lines with an error. Creates another RDD

Applies a function to each element(string) in RDD and produces a new RDD

Keep it in memory as it will be reused

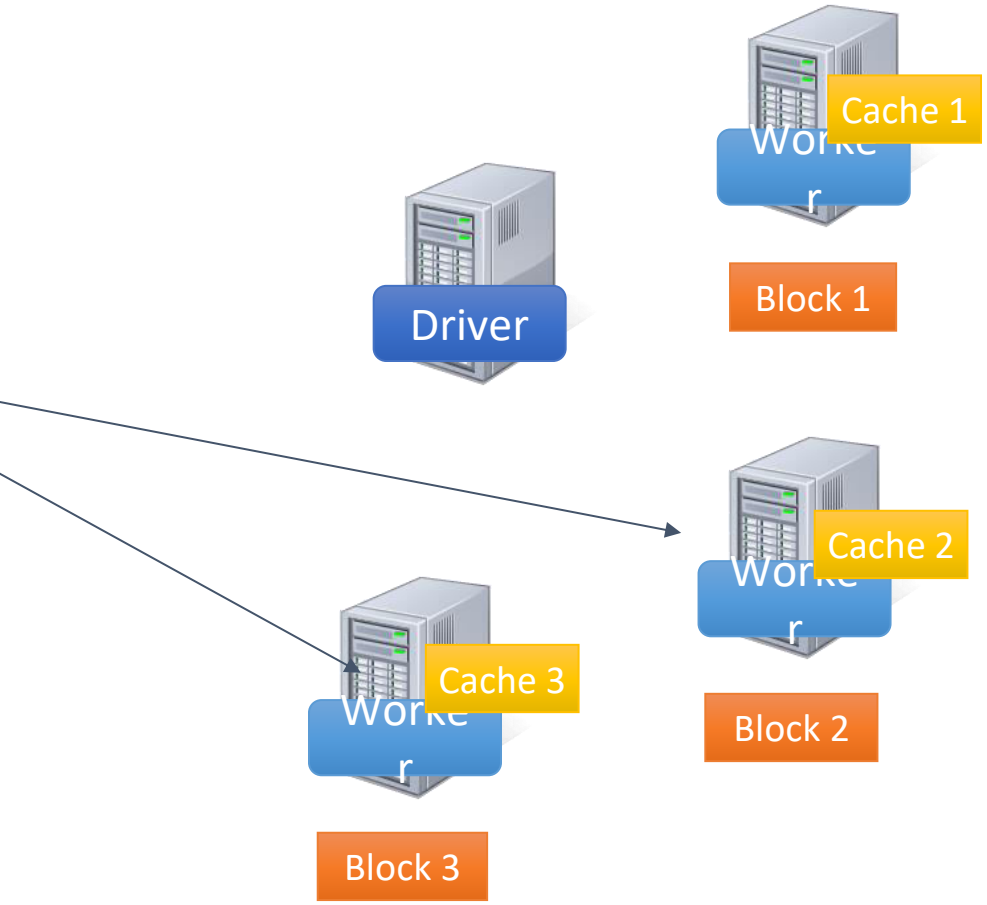
Counts #objects in the RDD

Courtesy: Zaharia et al , “Spark: Fast, Interactive, Language-Integrated Cluster Computing”, www.spark-project.org

BIG DATA

Log Processing: Distributing the computation

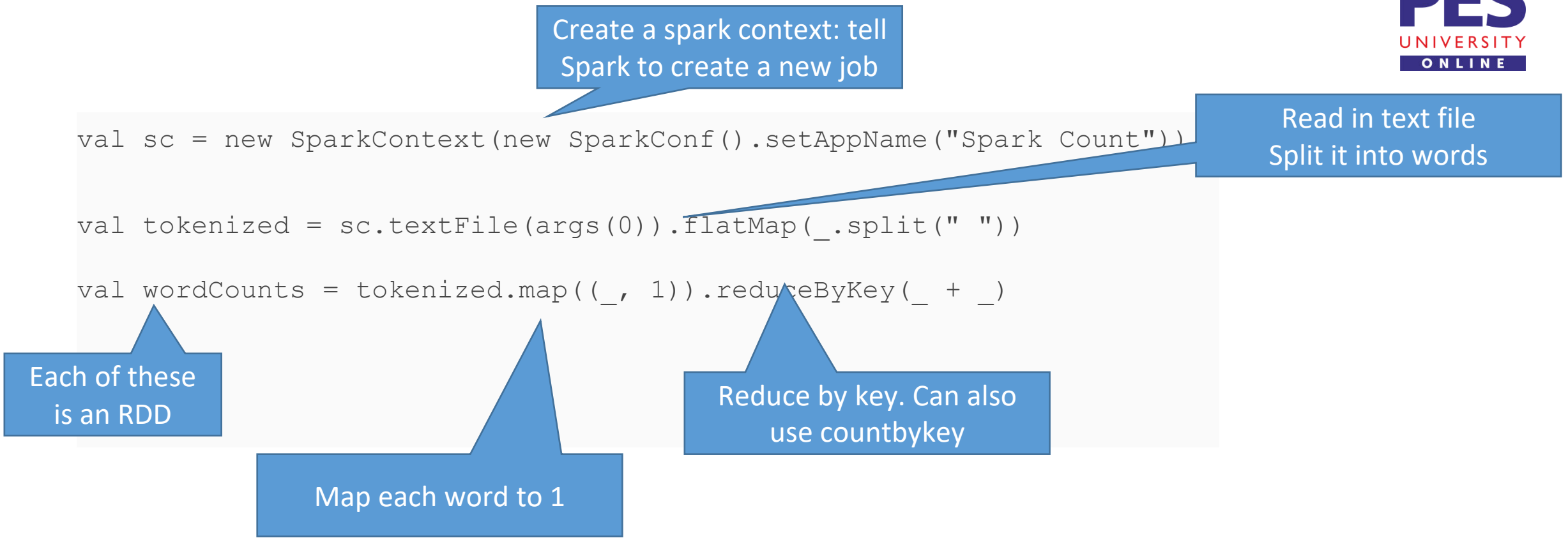
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(startsWithERROR())
messages = errors.map(split(“\t”),2)
cachedMsgs = messages.cache()

cachedMsgs.filter(containsfoo()).count()
cachedMsgs.filter(containsbar()).count()
```



Word Count in Spark

BIG DATA

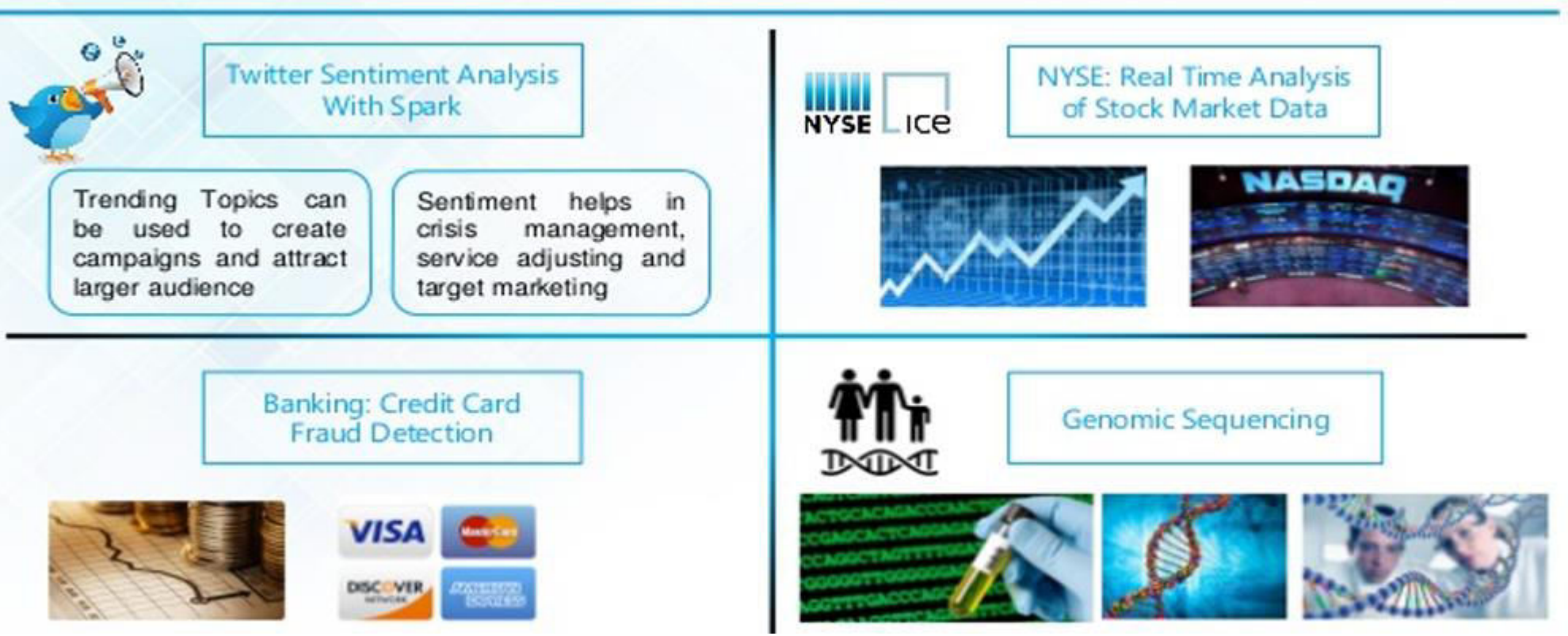
Word Count in Spark



https://docs.cloudera.com/documentation/enterprise/5-13-x/topics/spark_develop_run.html

BIG DATA

Spark Use cases



Additional References

- What is Apache Spark? Matei Zaharia
  - https://www.youtube.com/watch?v=p8FGC49N-zM

THANK YOU

Prof. J.Ruby Dinakar
Dept. of Computer Science and Engineering
rubydinakar@pes.edu

BIG DATA

Spark : Architecture

Prof. J.Ruby Dinakar

Department of Computer Science and Engineering

Acknowledgements:

Significant information in the slide deck presented through the Unit 3 of the course have been created by **Dr. K V Subramaniam's** and would like to acknowledge and thank him for the same. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for classroom presentation only.

BIG DATA

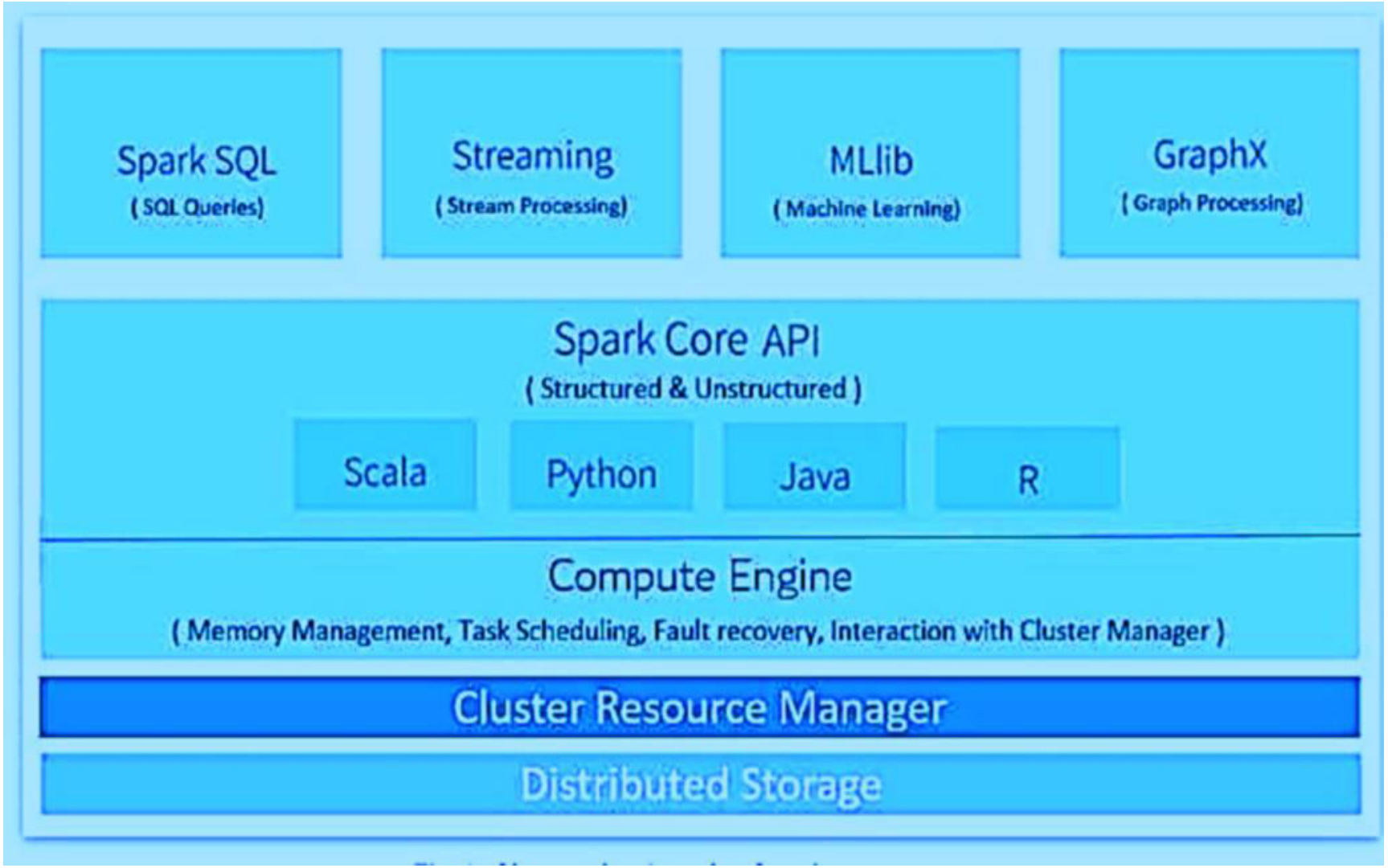
Overview of lecture – Spark Introduction

- Spark Ecosystem
- Spark Data processing
- Spark Architecture and working details
- Lazy Execution
- Spark Scheduling

Spark Ecosystem

BIG DATA

Spark Ecosystem



Storage and Cluster Manager

- Apache Spark is a distributed processing engine.
- it doesn't come with an inbuilt cluster resource manager and a distributed storage system.
- But, it allows us to use any compatible cluster manager and storage solution.
- Apache YARN, Mesos, and even Kubernetes can be used as a cluster manager
- HDFS, Amazon S3, Google Cloud storage, Cassandra File system etc., can be used for the storage system

Spark Core

- Apache Spark core contains two main components.
- Spark Compute engine
- Spark Core APIs
- The compute engine provides basic functionalities like memory management, task scheduling, fault recovery and most importantly interacting with the cluster manager and storage system.
- Spark compute engine executes and manages the Spark jobs

- The second part of Spark Core is core API. Spark core consists of two types of APIs.

  1.Structured API

  2.Unstructured API

- The Structured APIs consists of data frames and data sets.
- They are designed and optimized to work with structured data.
- The Unstructured APIs are the lower level APIs including RDDs, Accumulators and Broadcast variables. These core APIs are available in Scala, Python, Java, and R.

PES UNIVERSITY ONLINE

Libraries and DSL

1.Spark SQL

- For working with structured data
- View datasets as relational tables
- Define a schema of columns for a dataset
- Perform SQL queries
- Supports many sources of data
- Hive tables, Parquet and JSON

2. Spark Streaming

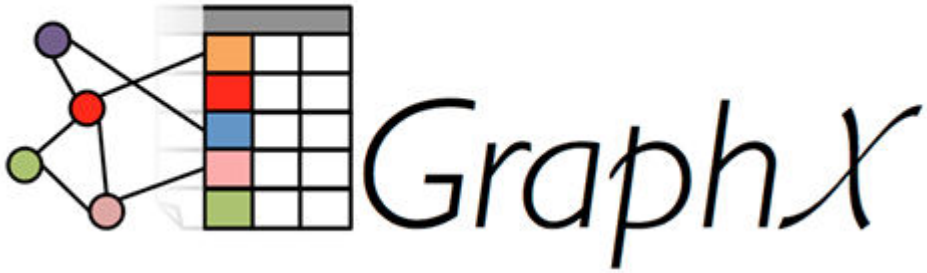
- Data analysis of streaming data
- e.g. log files generated by production web servers
- Aimed at high-throughput and fault-tolerant stream processing
- Dstream: stream of datasets that contain data from certain intervals
- Usually applied to time series data; intervals are time windows (1 second; 1 minute etc)
- Joins on unlimited streams are impossible so Spark Streaming joins within a time window, or a static dataset with a stream

3.MLlib

- MLlib is a machine learning library that contains high-quality algorithms.
- Basic statistics
- Classification (Naïve Bayes, decision tress, LR)
- Clustering (k-means, Gaussian mixture, …)
- All the methods are designed to scale out across a cluster.

GraphX - It comes with a library of typical graph algorithms.

- Graph Processing Library
- Defines a graph abstraction
- Directed multi-graph
- Properties attached to each edge and vertex
- RDDs for edges and vertices
- Provides various operators for manipulating graphs (e.g. subgraph and mapVertices)



Spark Data processing

BIG DATA

Spark data processing

Spark allows four different styles of data analysis and processing.

Batch:

This mode is used for manipulating large datasets, typically performing large map-reduce jobs

Streaming:

This mode is used to process incoming information in near real time

Iterative:

This mode is for machine learning algorithms such as a gradient descent where the data is accessed repetitively in order to reach convergence

Interactive:

This mode is used for data exploration as large chunks of data are in memory and due to the very quick response time of Spark

Spark Architecture and working details

BIG DATA

Spark Architecture

Spark Applications consist of a *driver* process and a set of *executor* processes.

Driver process is responsible for

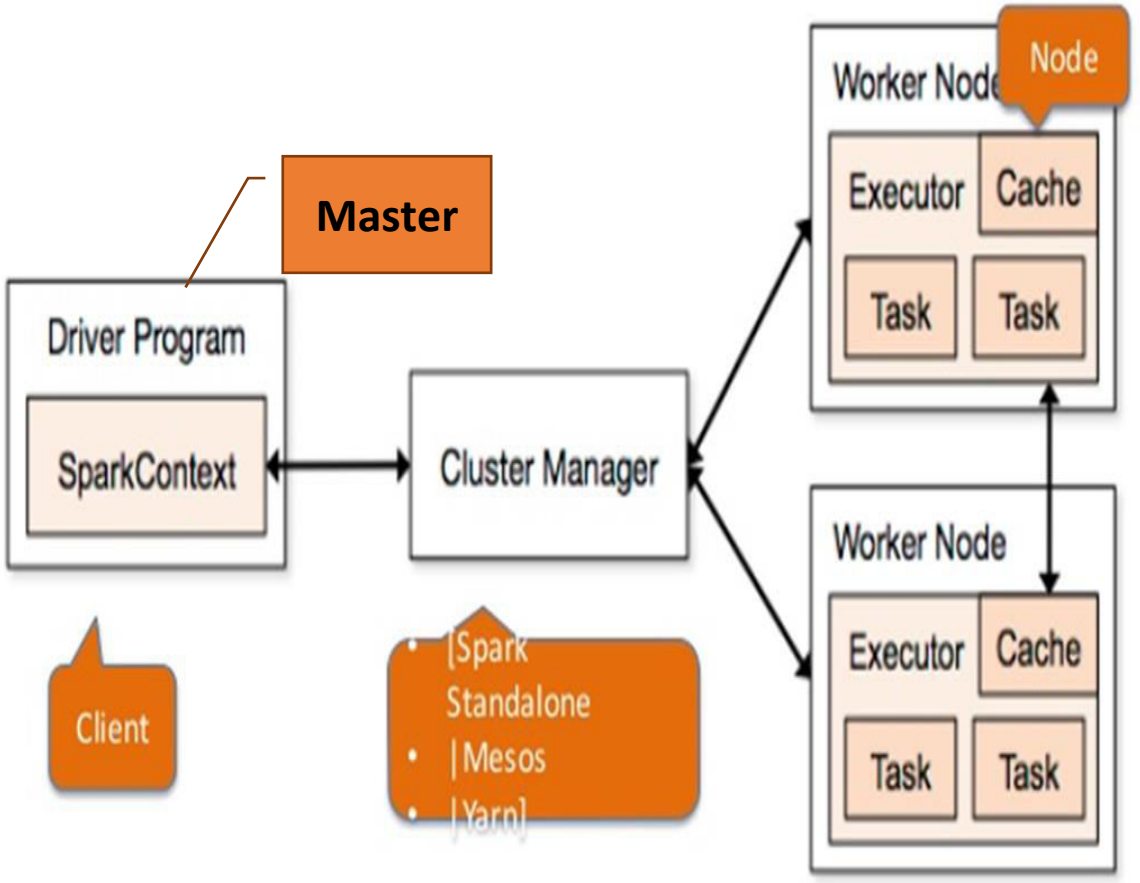
- maintaining information about the Spark Application;
- responding to a user's program or input;
- analyzing, distributing, and scheduling work across.

The **executors** are responsible for actually carrying out the work. It is responsible for

- executing code assigned to it by the driver
- reporting the state of the computation on that executor back to the driver node.

cluster manager controls physical machines and allocates resources to Spark Applications.

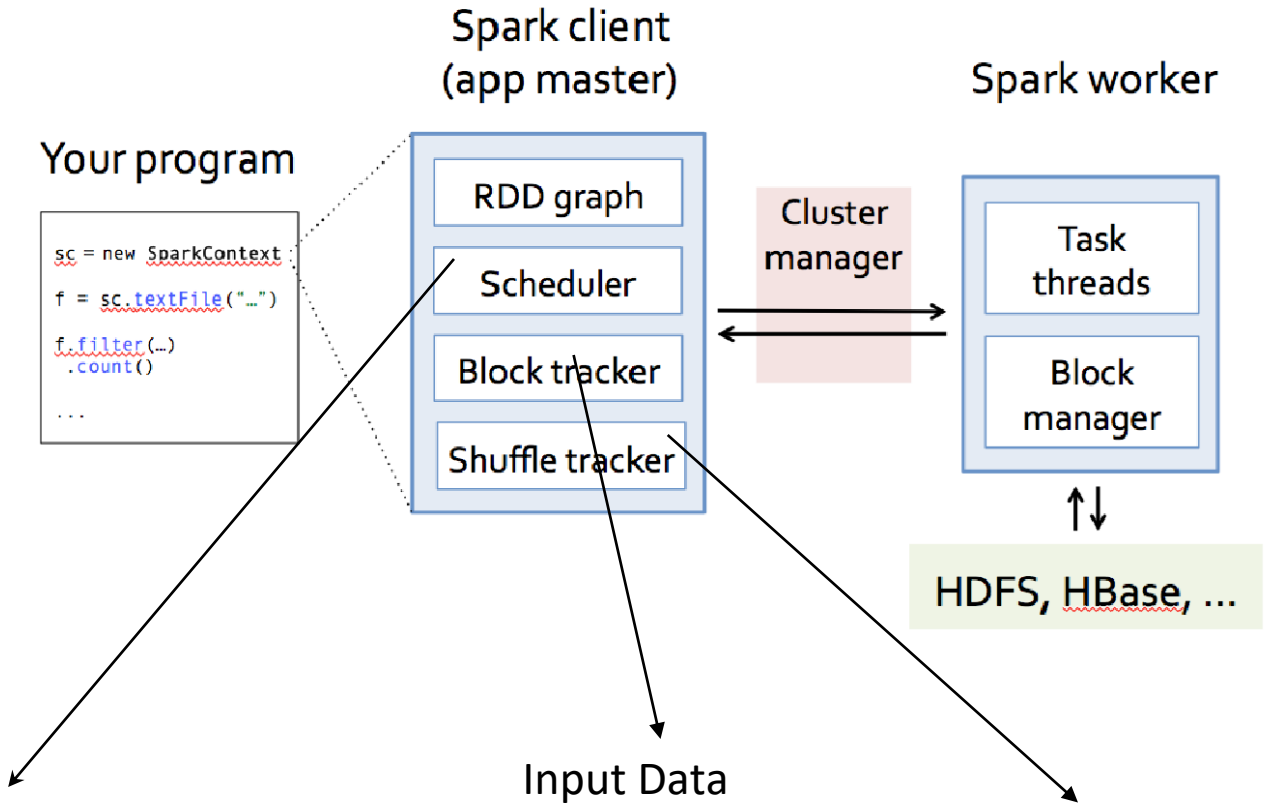
This can be one of three core cluster managers: Spark's standalone cluster manager, YARN, or Mesos.



https://spark.apache.org/docs/latest/cluster-overview.html

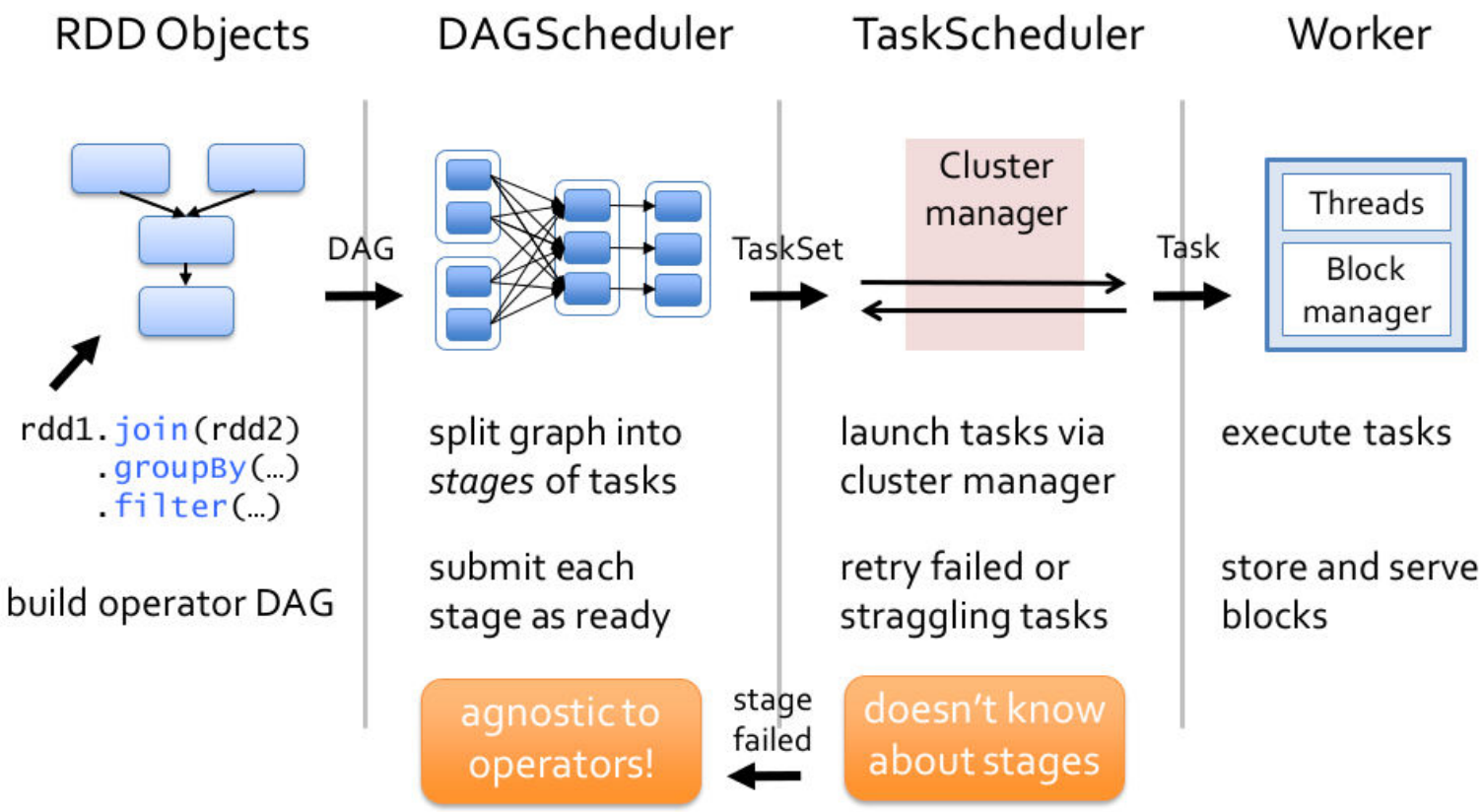
BIG DATA

Spark Architecture



BIG DATA

Spark Working details



Lazy Execution in Spark

Lazy Execution

- In Hadoop, when we submit a job the master starts executing it
- In Spark, when does the master start executing the job?
    - Spark uses a technique called Lazy execution

BIG DATA

Lazy Execution

- Remember that we defined Spark operations into *transformations* and *actions*
- The spark driver does not execute anything till it encounters an *action*
- *Transformations* are only noted for purpose of lineage.

Advantages of Lazy Evaluation in Spark Transformation

a. Increases Manageability

By lazy evaluation, users can organize their Apache Spark program into smaller operations. It reduces the number of passes on data by grouping operations.

b. Saves Computation and increases Speed

Spark Lazy Evaluation plays a key role in saving calculation overhead. Since only necessary values get compute. It saves the trip between driver and cluster, thus speeds up the process.

c. Reduces Complexities

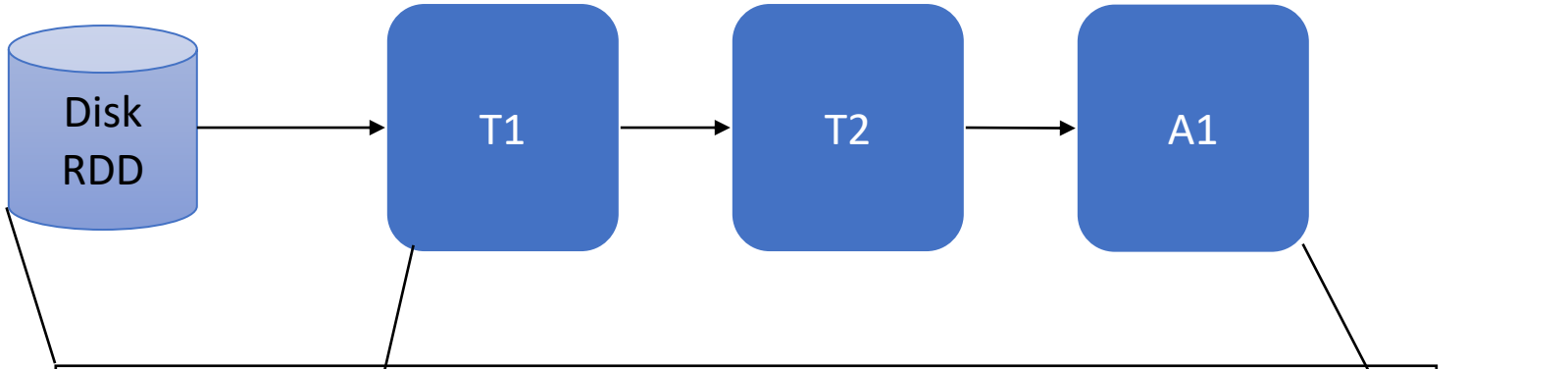
The two main complexities of any operation are time and space complexity. Using Apache Spark lazy evaluation we can overcome both. Since we do not execute every operation, Hence, the time gets saved. It let us work with an infinite data structure. The action is triggered only when the data is required, it reduces overhead.

d. Optimization

It provides optimization by reducing the number of queries. Learn more about Apache Spark Optimization.

BIG DATA

Lazy Execution



lines = spark.textFile("hdfs://...")
// Doesn't read file. Notes that RDD is to be created

Transformation say *map*
Doesn't do transformation immediately

Action say *count.* Triggers execution: all transformations done block by block on same server followed by count

BIG DATA

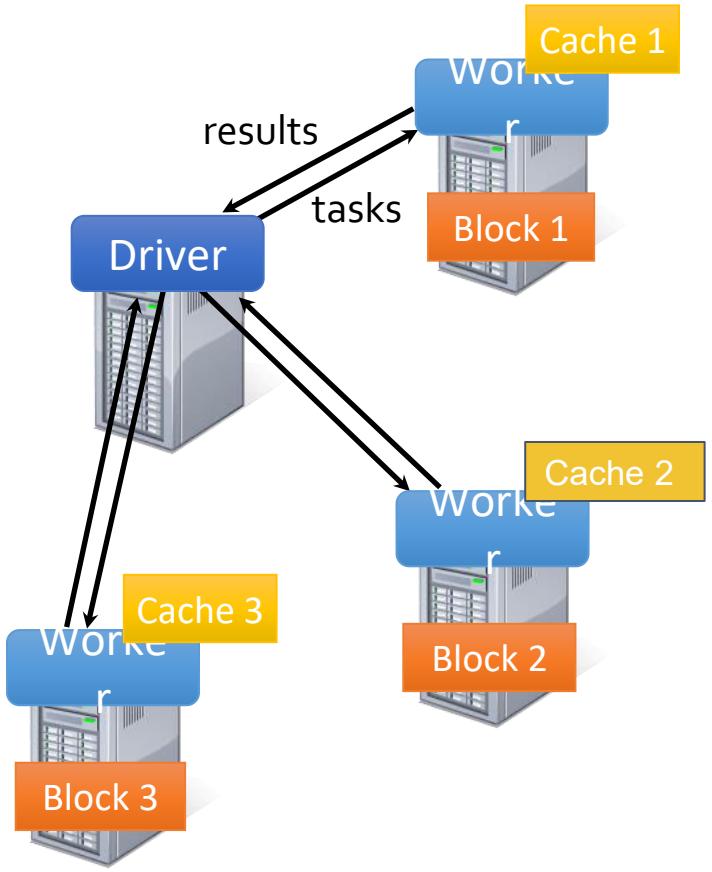
Spark Working with Log Mining Example

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(startsWithERROR)
messages = errors.map(split("\t"),2)
cachedMsgs = messages.cache()

cachedMsgs.filter(containsfoo()).count()
cachedMsgs.filter(containsbar()).count()
```

Base RDD

Transformed RDD

Action



Spark Scheduling

BIG DATA

Page Rank example in Spark

```
lines = textfile (“urls.txt”))

links = lines.map (lambda urls:
urls.split()).groupByKey().cache()

ranks = links.map(lambda url_neighbors:
(url_neighbors[0], 1.0))

 for iteration in range(MAXITER)):

        contribs = links.join(ranks).flatMap(
lambda url_neighbors_rank: computeContribs
(url_neighbors_rank)

        ranks =
contribs.reduceByKey(add).mapValues(lambd
a rank: rank * 0.85 + 0.15)
```

```
def computeContribs (url_neighbors_rank):
"""Calculates URL contributions to the rank of other URLs.
"""
num_neighbors = len (url_neighbors_rank) – 2

rank = url_neighbors_rank [len (url_neighbors_rank) – 1]

for i in range (1, num_neighbors):
        yield (url_neighbors_rank[i], rank /
num_neighbors)
```

1. Start each page with a rank of 1
2. On each iteration, update each page’s rank to

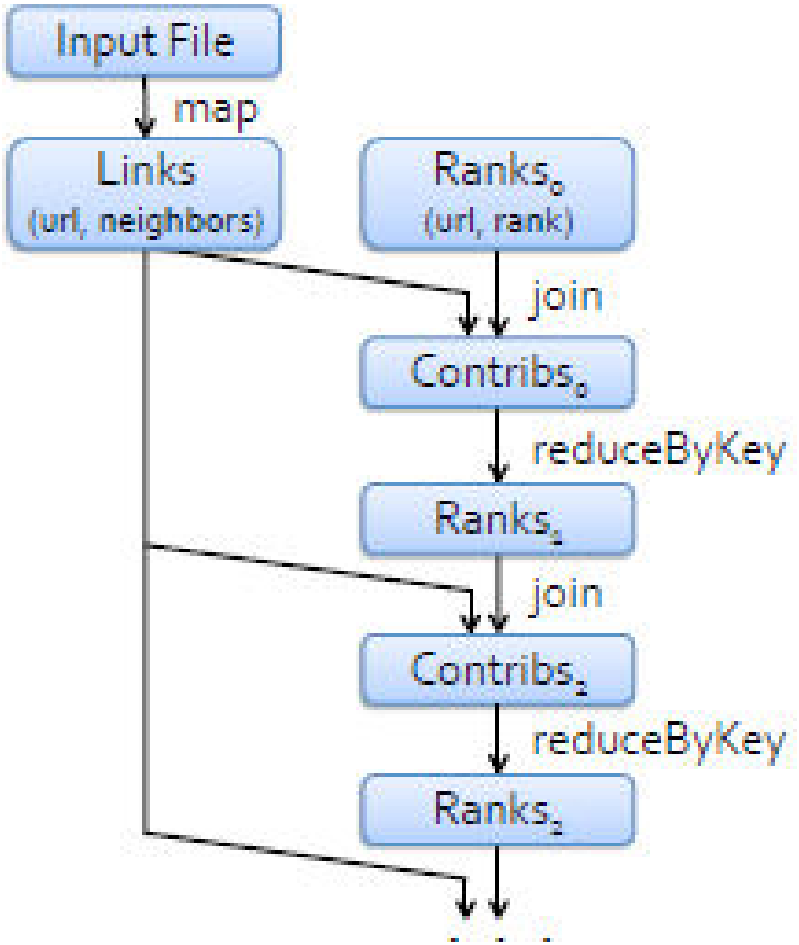
$$\Sigma_{i \in neighbors} rank_i / |neighbors_i|$$

DAG representation

- The Spark Driver will first convert this program into a DAG representation
- What does the DAG representation contain?
  - Each RDD is a node in the graph and
  - all transformations/actions on the RDD as edges

BIG DATA

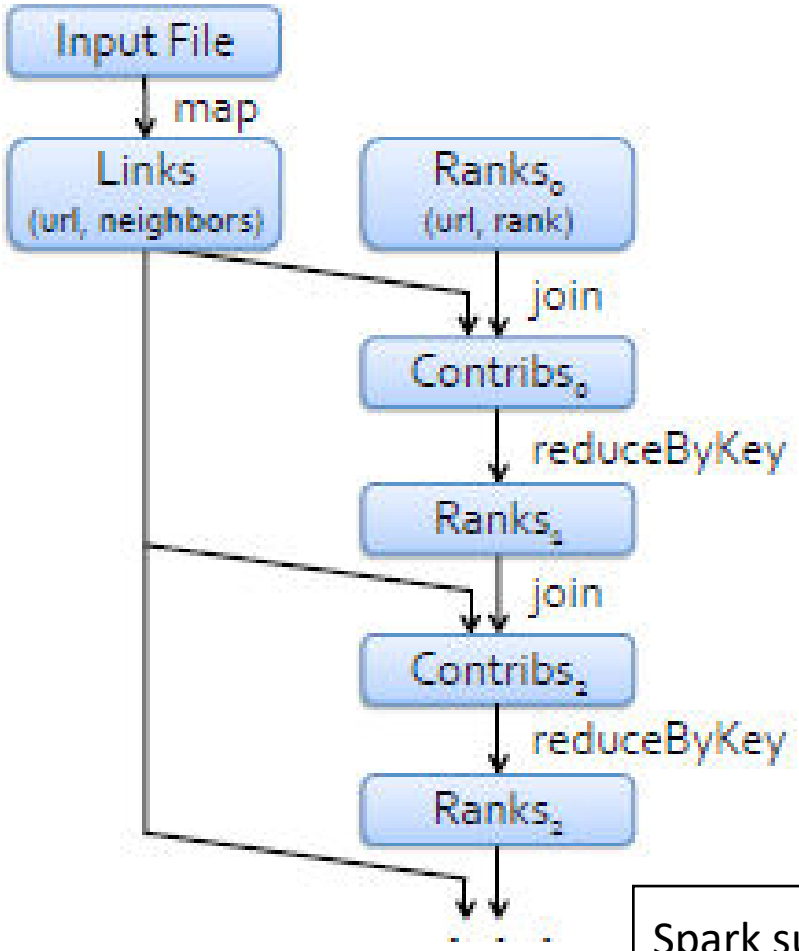
DAG representation of Page Rank




```
lines = textfile ("urls.txt"))

links = lines.map (lambda urls:
urls.split()).groupByKey().cache()

ranks = links.map(lambda url_neighbors:
(url_neighbors[0], 1.0))

 for iteration in range(MAXITER)):

        contribs =
links.join(ranks).flatMap( lambda
url_neighbors_rank: computeContribs

(url_neighbors_rank)

        ranks =
contribs.reduceByKey(add).mapValues(la
mbda rank: rank * 0.85 + 0.15)
```

Ranks and Links are spread across multiple nodes. How does Spark ensure join works properly? Hint: Think about how join works.

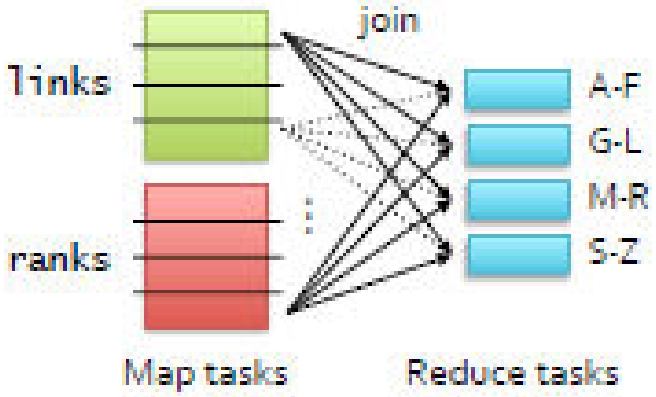
DAG representation of Page Rank



links and ranks are repeatedly joined

Each join requires a full shuffle over the network

» Hash both onto same nodes

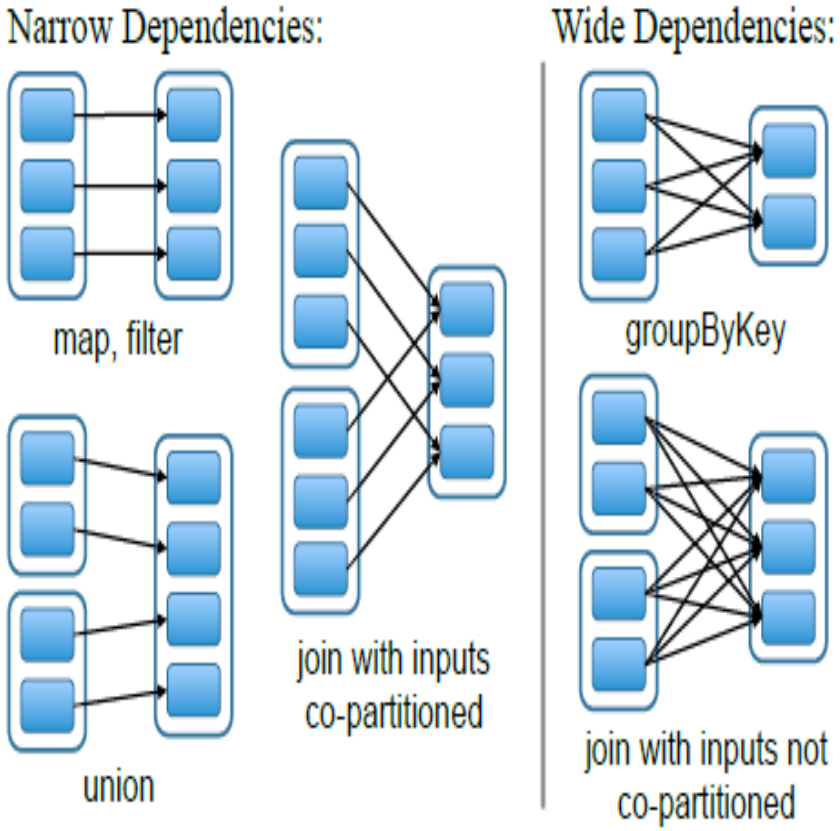


Spark supports two types of partitioning – hash and range

- **narrow dependencies**
  - **where each partition of** the parent RDD is used by at most one partition of the child RDD
  - Does not need a shuffle; pipeline operations
  - Shuffle: movement of data from one node to another

- **wide dependencies**
  - where multiple child partitions may depend on it.
  - May need a shuffle

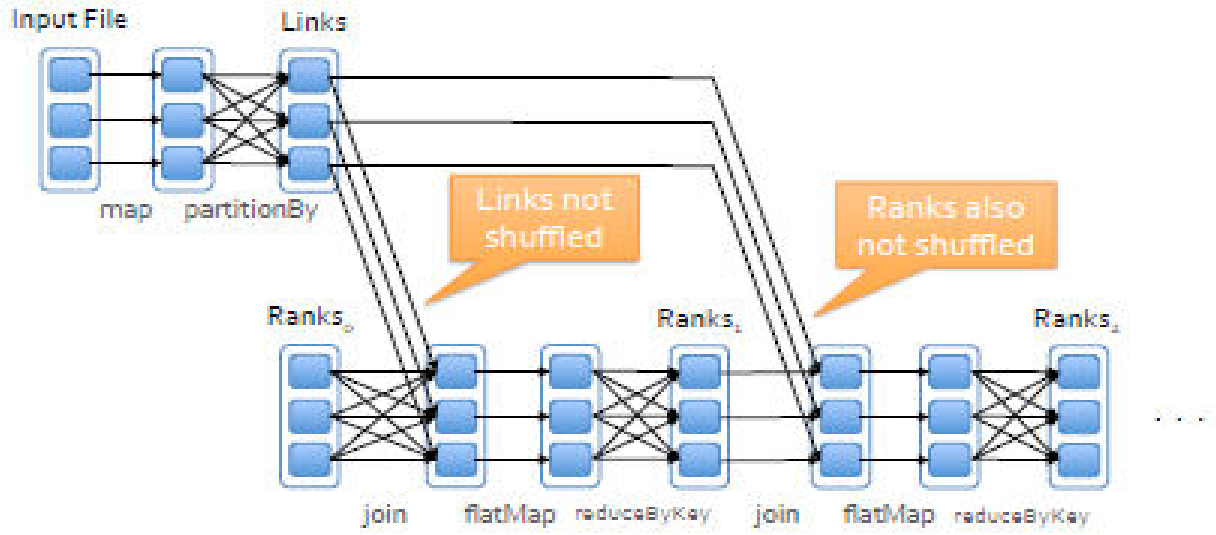
- Copartition - technique to make sure that both inputs to a join are partitioned using same function



BIG DATA

Lineage and Optimizing Placement

- links & ranks repeatedly joined
- Can copartition them (e.g.hash both on URL) to avoid shuffles
- Spark supports two types of partitioning: hash and range




```
lines = textfile ("urls.txt"))

links = lines.map (lambda urls:
urls.split()).groupByKey().cache()

ranks = links.map(lambda url_neighbors:
(url_neighbors[0], 1.0))

 for iteration in range(MAXITER)):

        contribs = links.join(ranks).flatMap(
lambda url_neighbors_rank: computeContribs
(url_neighbors_rank)

        ranks =
contribs.reduceByKey(add).mapValues(lambda
rank: rank * 0.85 + 0.15)
```

BIG DATA

Narrow and Wide Dependencies

Narrow Dependencies

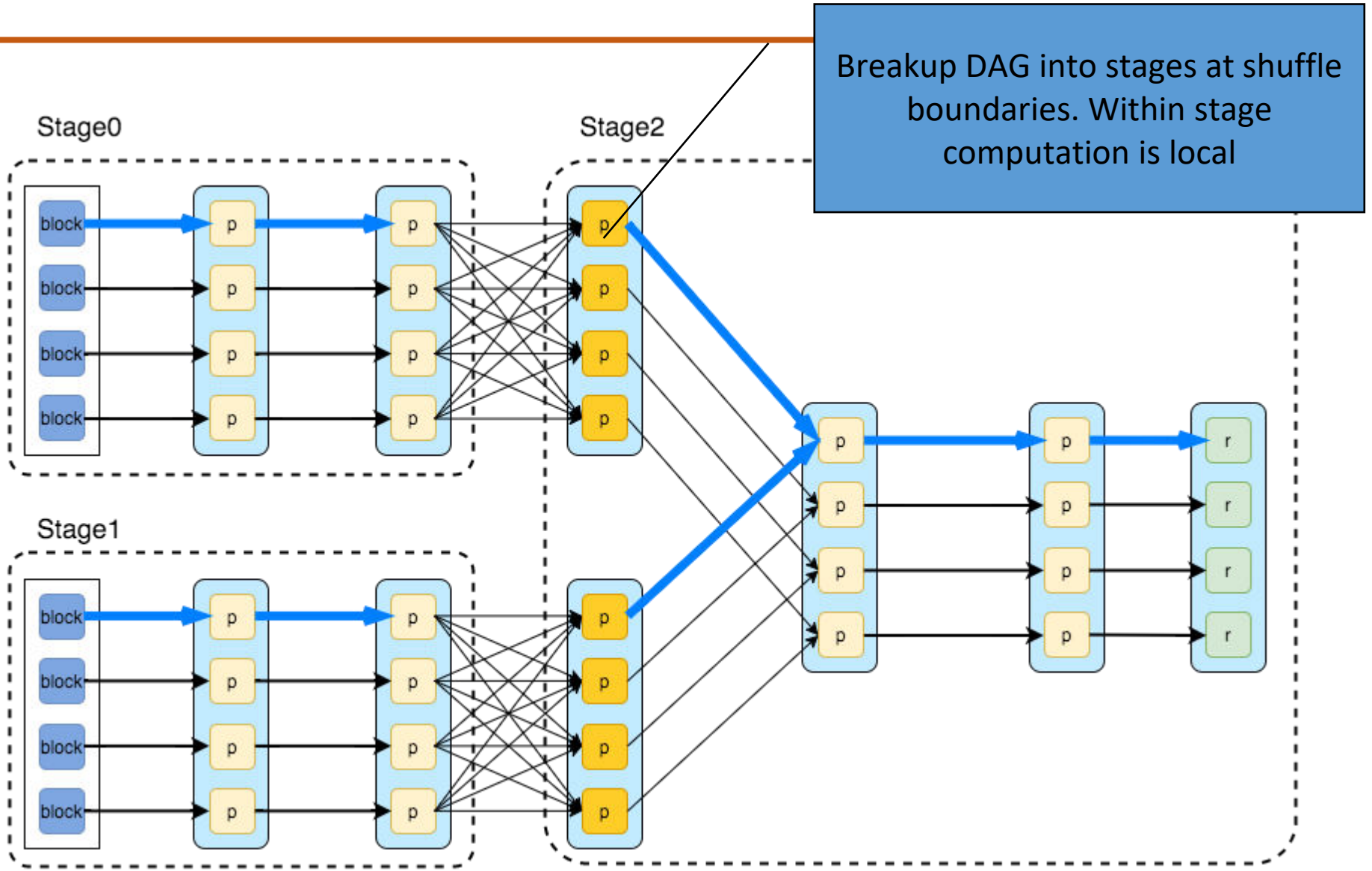
- Map
- FlatMap
- MapPartitions
- Filter
- Sample
- Union

Wide Dependencies

- Intersection
- Distinct
- ReduceByKey
- GroupByKey
- Join
- Cartesian
- Repartition
- Coalesce

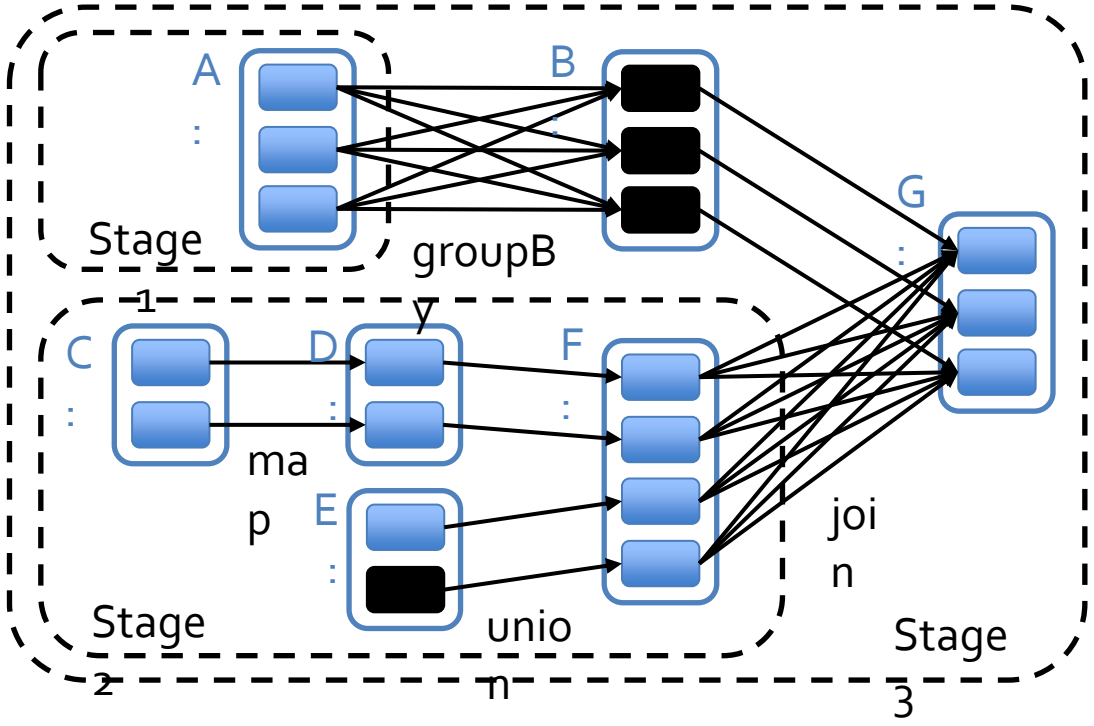
BIG DATA

Scheduling



Task Assignment

- scheduler assigns tasks to machines based on data locality using delay scheduling
  - if a task needs to process a partition that is available in
  - memory on a node, then send it to that node
  - otherwise, a task processes a partition for which the containing RDD provides preferred locations (e.g., an HDFS file), then send it to those



THANK YOU

Prof. J.Ruby Dinakar
Dept. of Computer Science and Engineering
rubydinakar@pes.edu

BIG DATA

RDD

Prof. J.Ruby Dinakar

Department of Computer Science and Engineering

Acknowledgements:

Significant information in the slide deck presented through the Unit 3 of the course have been created by **Dr. K V Subramaniam's** and would like to acknowledge and thank him for the same. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for classroom presentation only.

RDDs - details

References

Learning Spark Lightning-Fast Data Analysis- Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia
https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

What is an RDD

- RDD is partitioned, locality aware, distributed collections
  - RDDs are immutable. (Why it's necessary?)

- RDDs are data structures that either
  - Point to the source (HDFS)
  - Apply some transformations to the parent RDDs to generate new data elements

- Computations on RDDs
  - Lazily evaluated lineage DAGs composed of chained RDDs

Why the RDD abstraction?

- Support operations other than map and reduce
- Support in memory computation
- Arbitrary composition of such operators
- Simplify scheduling

How to capture dependencies generically?

Representing RDDs

- Set of partitions (“splits”)
  - Much like Hadoop. Each RDD associated with a input partitions
- List of dependencies on parent RDDs
  - Not there in Hadoop. This is new
- Function to compute a partition given parents
  - User defined code. (similar to map()/reduce() in Hadoop
- Optional preferred locations
  - For data locality
- Optional partitioning information (partitioner)
  - Advanced – for shuffle (see later)

BIG DATA

RDDs Interface

Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p, parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

https://medium.com/@shagun/resilient-distributed-datasets-97c28c3a9411

Examples of RDDs

- Hadoop RDD
  - Partitions – one per block
  - Dependencies – none
  - Compute (partition) – read corresponding block
  - Preferred locations – HDFS block location
  - Partitioner - none
- Filtered RDD (as in sample application)
  - Partitions – same as parent
  - Dependencies – 1-1 with parent
  - Compute – compute parent and filter it.
  - Preferred locations – ask parent (none)
  - Partitioner - none

Exercise

Based on the sample of the Filter RDD, can you work out what will be the partitions, compute, dependencies, preferred locations and partitioner for a joinRDD

- Filtered RDD (as in sample application)
  - Partitions – same as parent
  - Dependencies – 1-1 with parent
  - Compute – compute parent and filter it.
  - Preferred locations – ask parent (none)
  - Partitioner - none
-

Examples of RDDs

- Joined RDD
  - RDDPartitions – one per reduce task
  - Dependencies – shuffle on each parent
  - Compute (partition) – read and join shuffled data
  - Preferred locations – none
  - Partitioner – HashPartioner (num tasks)

Adding fault tolerance – The RDD

BIG DATA

Handling fault tolerance

Consider the following code:

```
                    Step1                    Step2
messages = textFile(...).filter(startsWithERROR())
                  .map(split("\t")(2))
                                             Step3
```

Step1: Read in the file to an in memory RDD

Step2: remove all lines that don't contain the term ERROR

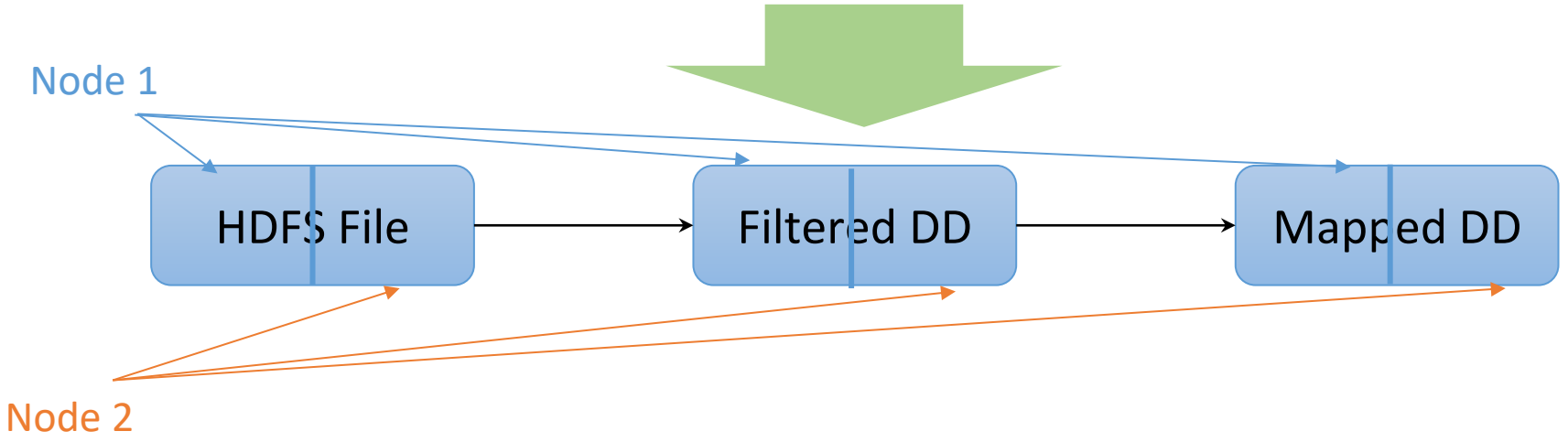
Step3: split the line

map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.

Handling fault tolerance

Ex:

```
messages = textFile(...).filter(startsWithERROR())
                        .map(split("\t")(2))
```



RDD Operations : Transformations and Actions

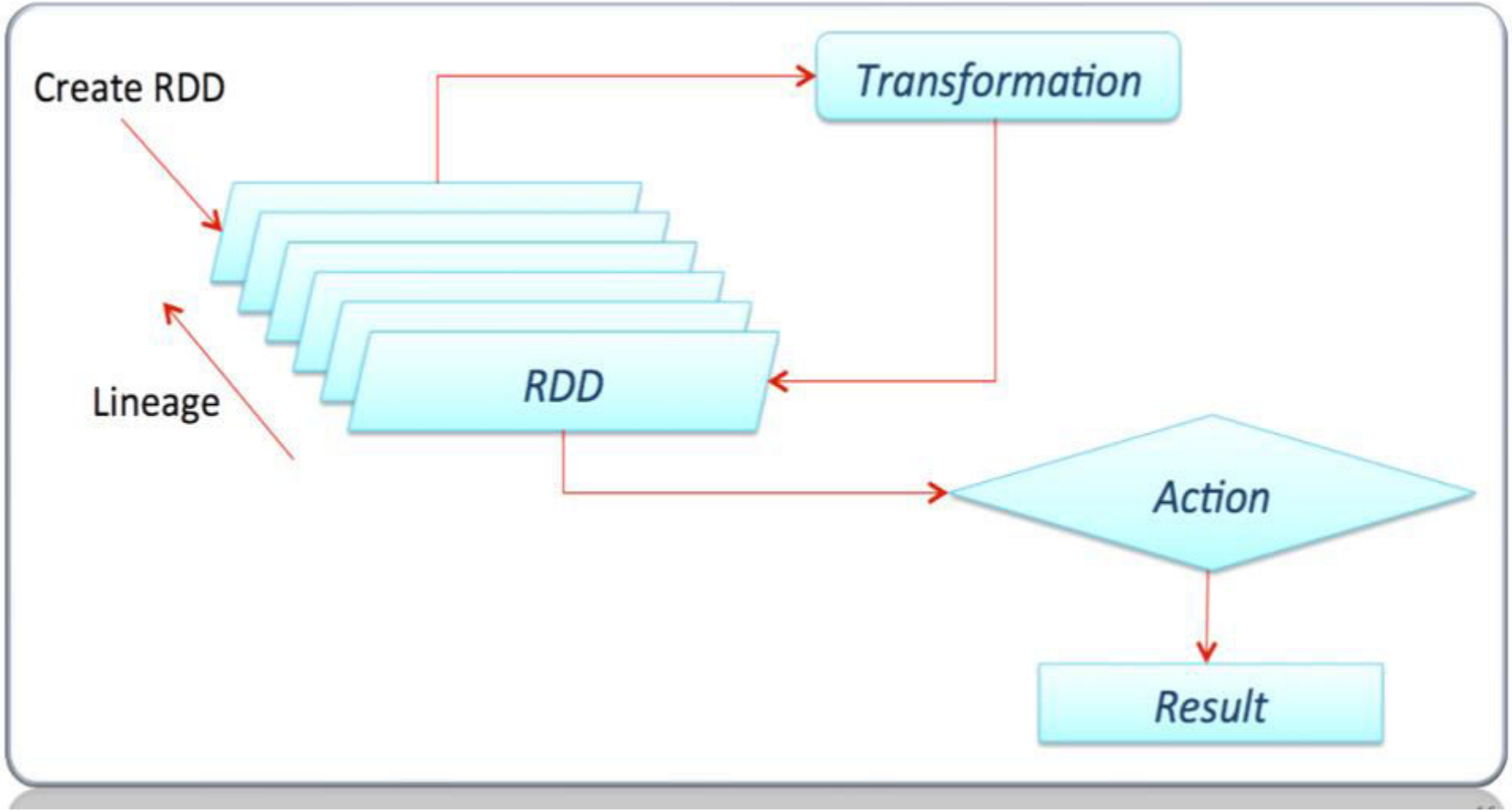
BIG DATA

Types of Operations

- Operations are of two types
  - ***Transformations***
  - ***Actions***

BIG DATA

Iterative Operations on Spark RDD



Transformation

- Are operations that create a *new dataset* from an existing dataset
- For example:
  - *map()* is a transformation
    - Each line on input RDD is passed through the *map()* function
    - result of *map()* function applied on each value is stored in the output RDD.
- Note it is similar to the Map of map-reduce, but is more generic.

Transformations

Sr.	Transformation and Meaning
1	**map(func)** Returns a new distributed dataset, formed by passing each element of the source through a function func.
2	**filter(func)** Returns a new dataset formed by selecting those elements of the source on which func returns true.
3	**flatMap(func)** Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
4	**mapPartitions(func)** Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.

Transformations

Sr.	Transformation and Meaning
5	**mapPartitionsWithIndex(func)** Similar to map Partitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
6	**sample(withReplacement, fraction, seed)** Sample a fraction of the data, with or without replacement, using a given random number generator seed.
7	**union(otherDataset)** Returns a new dataset that contains the union of the elements in the so dataset and the argument.

Transformations

Sr.	Transformation and Meaning
8	**intersection(otherDataset)** Returns a new RDD that contains the intersection of elements in the source dataset and the argument.
9	**distinct([numTasks]))** Returns a new dataset that contains the distinct elements of the source dataset.
10	**groupByKey([numTasks])** When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.

Transformations

Sr.	Transformation and Meaning
11	**reduceByKey(func, [numTasks])** When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V, V) => V.
12	**aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])** When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different from the input value type, while avoiding unnecessary allocations.
13	**sortByKey([ascending], [numTasks])** When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument.

Transformations

Sr.	Transformation and Meaning
17	**pipe(command, [envVars])** Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
18	**coalesce(numPartitions)** Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
19	**repartition(numPartitions)** Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
20	**epartitionAndSortWithinPartitions(partitioner)** Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Actions

- Are operations that return a value

- For example:
  - Reduce() is an action
  - Aggregates all elements of a RDD to produce a result.

Actions

Sr.	Actions and Meaning
1	**reduce(func)** Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
2	**collect()** Returns all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
3	**count()** Returns the number of elements in the dataset.
4	**first()** Returns the first element of the dataset (similar to take (1)).

Actions

Sr.	Actions and Meaning
5	**take(n)** Returns an array with the first n elements of the dataset.
6	**takeSample (withReplacement,num, [seed])** Returns an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
7	**takeOrdered(n, [ordering])** Returns the first n elements of the RDD using either their natural order or a custom comparator.
8	**saveAsTextFile(path)** Writes the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark calls toString on each element to convert it to a line of text in the file.

Actions

Sr.	Actions and Meaning
9	**saveAsSequenceFile(path) (Java and Scala)** Writes the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
10	**saveAsObjectFile(path) (Java and Scala)** Writes the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
11	**countByKey()** Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
12	**foreach(func)** Runs a function func on each element of the dataset. This is usually, done for side effects such as updating an Accumulator or interacting with external storage systems.

RDD Operations : Working with key-value pairs

BIG DATA

Key Value Pairs

- Consider our earlier operation of map/reduce using Spark
- Worked on datasets with only single values
- Let's consider how to represent *<key, value>* pairs
- Spark provides
  - Separate RDDs called pair RDDs for this
  - Separate operations to function on Pair RDDs

BIG DATA

Pair RDD Transformations

Table 4-1. Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})

Function name	Purpose	Example	Result
`reduceByKey(func)`	Combine values with the same key.	`rdd.reduceByKey( (x, y) => x + y)`	`{(1, 2), (3, 10)}`
`groupByKey()`	Group values with the same key.	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)`	Combine values with the same key using a different result type.	See Examples 4-12 through 4-14.	

BIG DATA

Pair RDD Transformations

`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key.	`rdd.mapValues(x => x+1)`	`{(1, 3), (3, 5), (3, 7)}`
`flatMapValues(func)`	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	`rdd.flatMapValues(x => (x to 5)`	`{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}`
`keys()`	Return an RDD of just the keys.	`rdd.keys()`	`{1, 3, 3}`
`values()`	Return an RDD of just the values.	`rdd.values()`	`{2, 4, 6}`

Pair RDD Actions

countByKey(k, V) □ returns a HashMap of (k, Int) key value pairs with count of each key

Persistence

- Spark can persist (or cache) a dataset in memory across operations
- Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster
- The cache is fault-tolerant:
- if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

BIG DATA

Persistence

transformation	description
`MEMORY_ONLY`	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
`MEMORY_AND_DISK`	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
`MEMORY_ONLY_SER`	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
`MEMORY_AND_DISK_SER`	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
`DISK_ONLY`	Store the RDD partitions only on disk.
`MEMORY_ONLY_2`, `MEMORY_AND_DISK_2`, etc	Same as the levels above, but replicate each partition on two cluster nodes.

PES UNIVERSITY ONLINE

Scala:

```
val f = sc.textFile("README.md")
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).collect.foreach(println)
```

Python:

```
from operator import add
f = sc.textFile("README.md")
w = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).cache()
w.reduceByKey(add).collect()
```

DryadLINQ, FlumeJava

- Similar “distributed collection” API, but cannot reuse datasets efficiently *across* queries

- Relational databases
  - Lineage/provenance, logical logging, materialized views

GraphLab, Piccolo, BigTable, RAMCloud

- Fine-grained writes similar to distributed shared memory

- Iterative MapReduce (e.g. Twister, HaLoop)
  - Implicit data sharing for a fixed computation pattern

- Caching systems (e.g. Nectar)
  - Store data in files, no explicit control over what is cached

Shared Variables

- Spark contains two different types of shared variables – one is broadcast variables and second is accumulators.
- Broadcast variables – used to efficiently, distribute large values.
- Accumulators – used to aggregate the information of particular collection.

BIG DATA

Shared Variables

- 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. They can be used, for example, to give every node, a copy of a large input dataset, in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

- Spark actions are executed through a set of stages, separated by distributed “shuffle” operations. Spark automatically broadcasts the common data needed by tasks within each stage.

- The data broadcasted this way is cached in serialized form and is deserialized before running each task. This means that explicitly creating broadcast variables, is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

BIG DATA

Shared Variables

- Accumulators
- Spark Accumulators are shared variables which are only “added” through an associative and commutative operation and are used to perform counters (Similar to Map-reduce counters) or sum operations

- Spark by default supports to create an accumulators of any numeric type and provide a capability to add custom accumulator types

THANK YOU

Prof. J.Ruby Dinakar
Dept. of Computer Science and Engineering
rubydinakar@pes.edu

BIG DATA

RDD Operations : Transformations and Actions

Prof. J.Ruby Dinakar
Department of Computer Science and Engineering

Acknowledgements:
Significant information in the slide deck presented through the Unit 3 of the course have been created by **Dr. K V Subramaniam's** and would like to acknowledge and thank him for the same. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for classroom presentation only.

RDD Operations : Transformations and Actions

Types of Operations

Spark Operations = 
TRANSFORMATIONS

+


ACTIONS

BIG DATA

Types of Operations

 = easy  = medium

Essential Core & Intermediate Spark Operations

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

BIG DATA

Transformation (Recap)

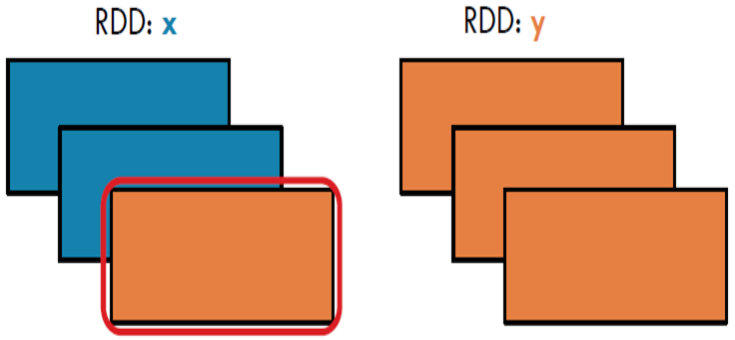
- Are operations that create a *new dataset* from an existing dataset
- For example:
  - *map()* is a transformation
    - Each line on input RDD is passed through the *map()* function
    - result of *map()* function applied on each value is stored in the output RDD.
- Note it is similar to the Map of map-reduce, but is more generic.

map(func)

The map function iterates over every line in RDD and split into new RDD.

Using map() transformation we take in any function, and that function is applied to every element of RDD.

In the map, we have the flexibility that the input and the return type of RDD may differ from each other. For example, we can have input RDD type as String, after applying the map() function the return RDD can be Boolean.




```
x = sc.parallelize(["b", "a", "c"])
y = x.map(lambda z: (z, 1))
print(x.collect())
print(y.collect())
```

A lambda function is a small anonymous function. It can take any number of arguments, but can only have one expression.

collect() is an action used to retrieve all the elements of the dataset

```
x: ['b', 'a', 'c']

y: [('b', 1), ('a', 1), ('c', 1)]
```

flatMap()

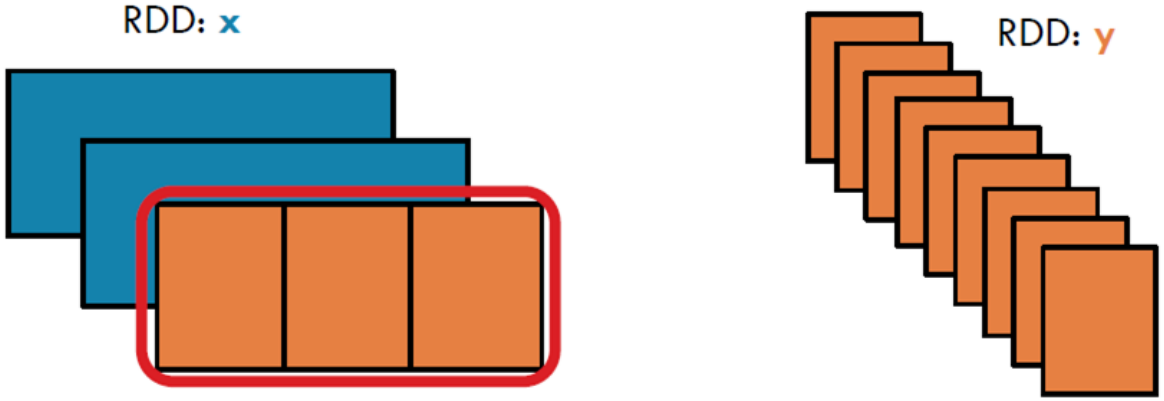
With the help of flatMap() function, to each input element, we have many elements in an output RDD. The most simple use of flatMap() is to split each input string into words.

Map and flatMap are similar in the way that they take a line from input RDD and apply a function on that line.

The key difference between map() and flatMap() is map() returns only one element, while flatMap() can return a list of elements.

BIG DATA

Transformations - flatMap



Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results

```
x = sc.parallelize([1,2,3])
y = x.flatMap(lambda x: (x, x*100, 42))
print(x.collect())
print(y.collect())
```

```
x: [1, 2, 3]

y: [1, 100, 42, 2, 200, 42, 3, 300, 42]
```

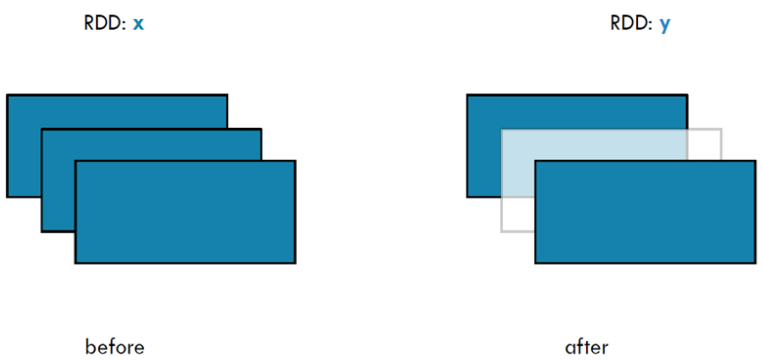
BIG DATA

Transformations - filter

filter(func)

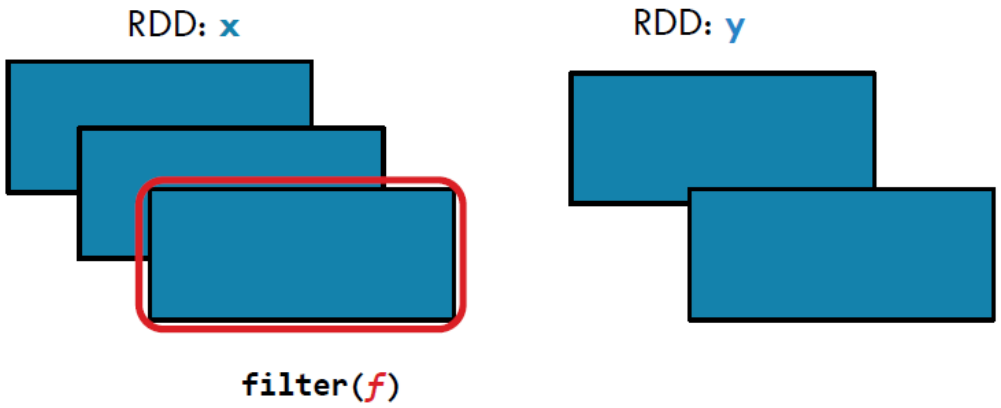
Spark RDD filter() function returns a new RDD, containing only the elements that meet a predicate. It is a narrow operation because it does not shuffle data from one partition to many partitions.

For example, Suppose RDD contains first five natural numbers (1, 2, 3, 4, and 5) and the predicate is check for an even number. The resulting RDD after the filter will contain only the even numbers i.e., 2 and 4.



BIG DATA

Transformations - filter



Return a new RDD containing only the elements that satisfy a predicate

```
x = sc.parallelize([1,2,3])
y = x.filter(lambda x: x%2 == 1) #keep odd values
print(x.collect())
print(y.collect())
```

```
x:  [1, 2, 3]

y:  [1, 3]
```

BIG DATA

Transformations - groupBy

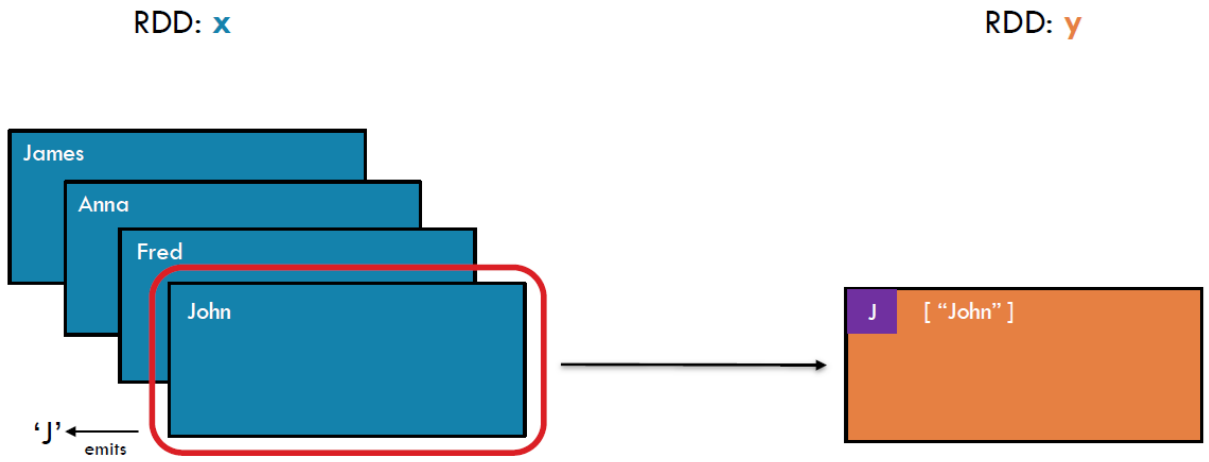
- Group the data in the original RDD. Create pairs where the key is the output of a user function, and the value is all items for which the function yields this key.

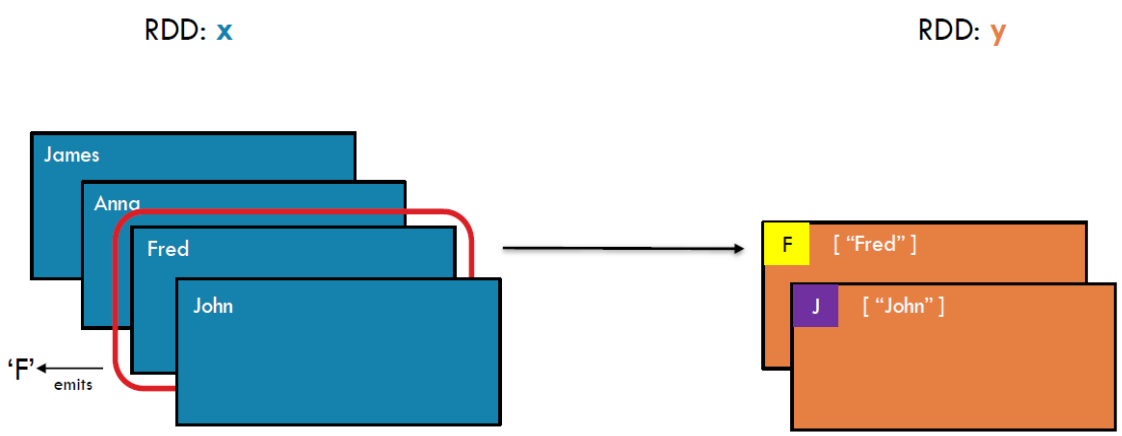
```
x = sc.parallelize(['John', 'Fred', 'Anna', 'James'])
y = x.groupBy(lambda w: w[0])
print [(k, list(v)) for (k, v) in y.collect()]
```

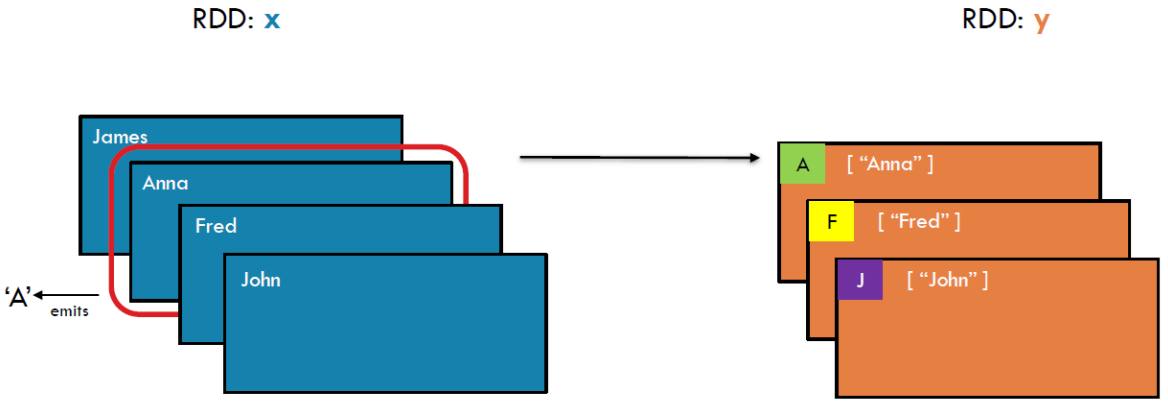
```
x: ['John', 'Fred', 'Anna', 'James']

y: [('A',['Anna']),('J',['John','James']),('F',['Fred'])]
```

BIG DATA

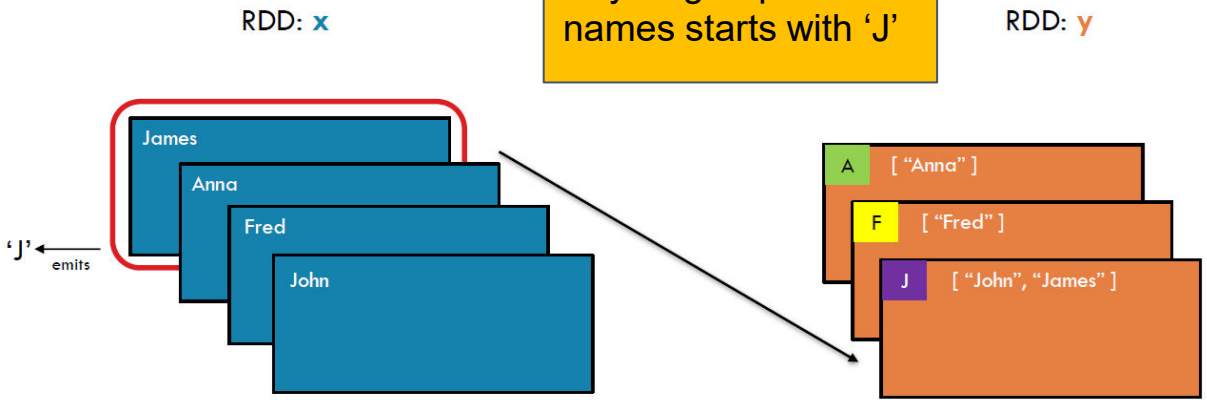
Transformations – groupBy







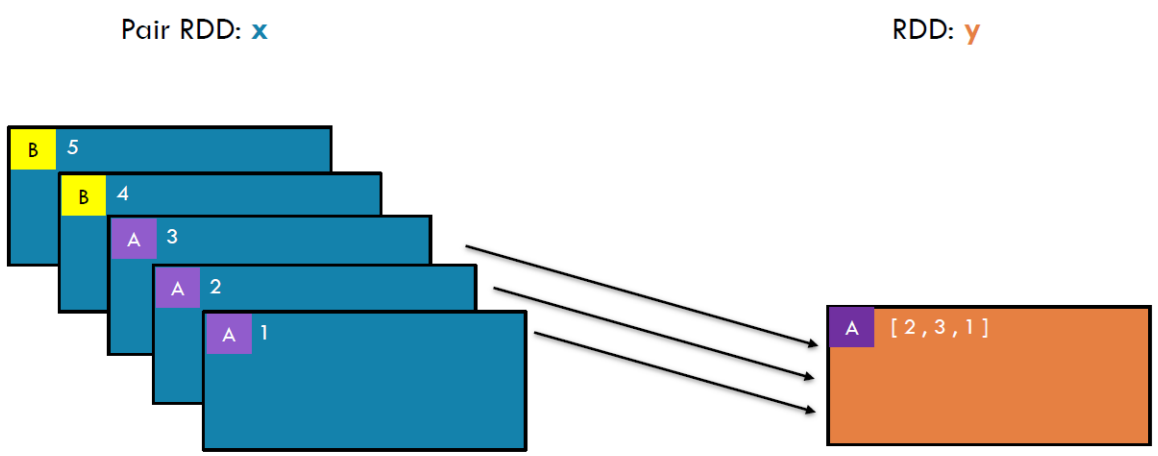
Key 'J' groups the names starts with 'J'

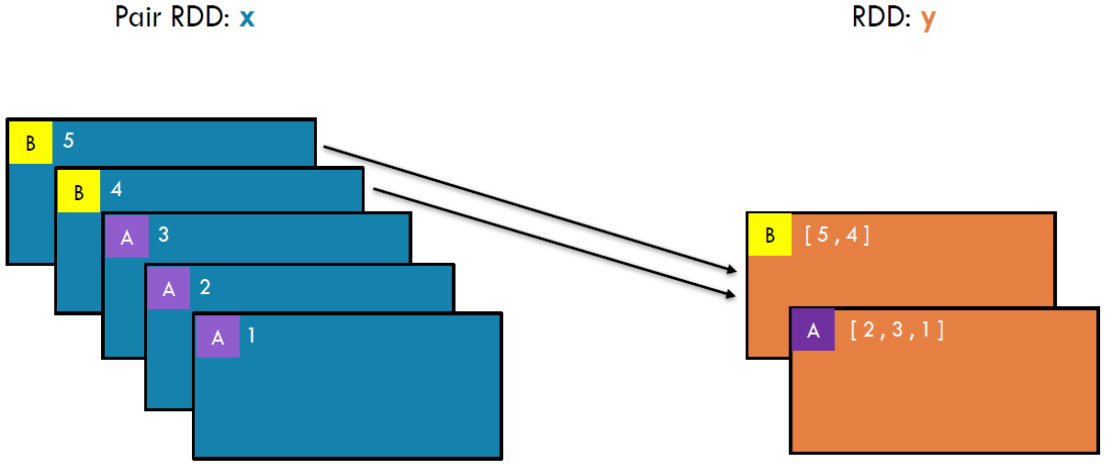


BIG DATA

Transformations - groupByKey

- Group the values for each key in the original RDD. Create a new pair where the original key corresponds to this collected group of values.



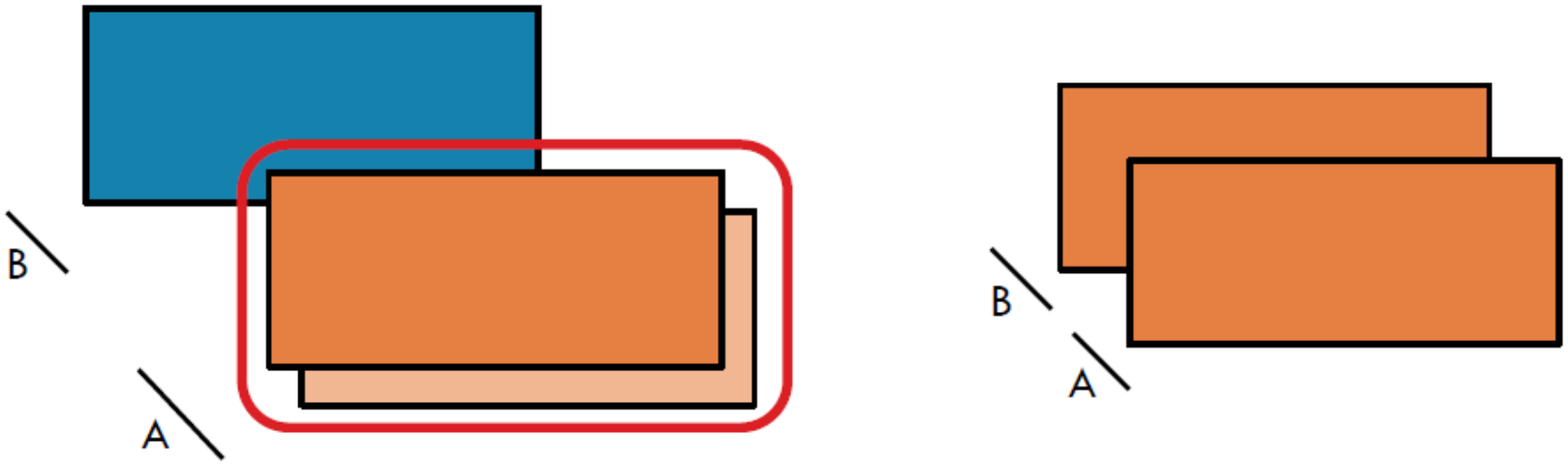



```
x = sc.parallelize([('B',5),('B',4),('A',3),('A',2),('A',1)])
y = x.groupByKey()
print(x.collect())
print(list((j[0], list(j[1])) for j in y.collect()))
```

```
x: [('B', 5),('B', 4),('A', 3),('A', 2),('A', 1)]

y: [('A', [2, 3, 1]),('B',[5, 4])]
```

BIG DATA

Transformations - Map Partitions



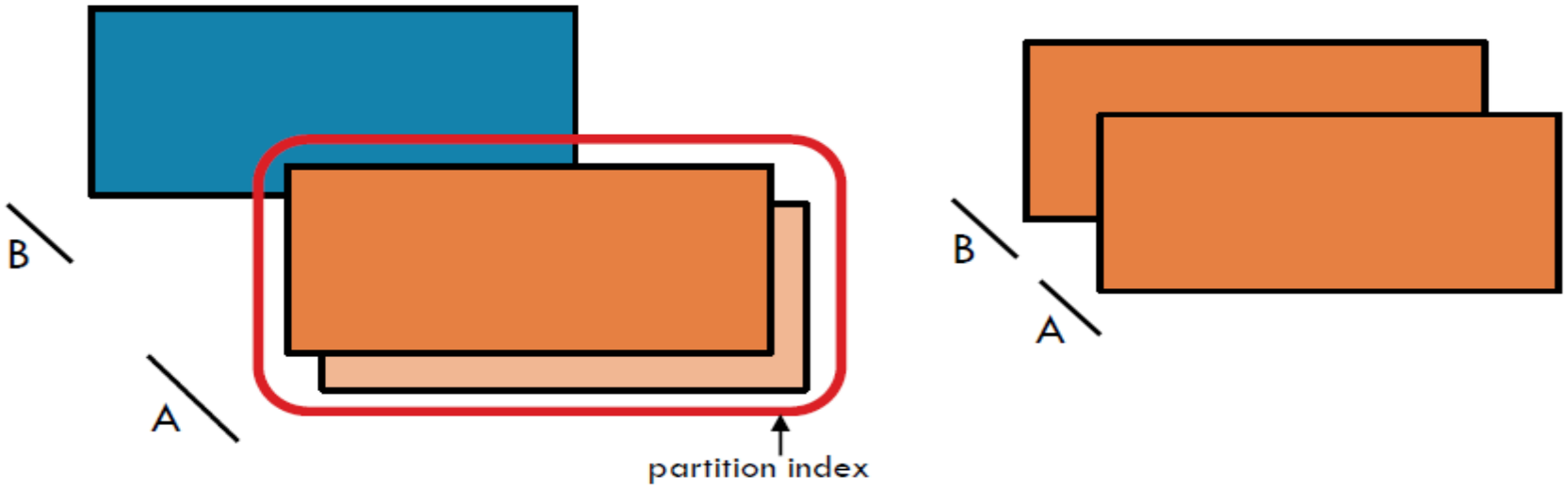
Return a new RDD by applying a function to each partition of this RDD

```
x = sc.parallelize([1,2,3], 2)
def f(iterator): yield sum(iterator); yield 42
y = x.mapPartitions(f)
# glom() flattens elements on the same partition
print(x.glom().collect())
print(y.glom().collect())
```

```
x: [[1], [2, 3]]

y: [[1, 42], [5, 42]]
```

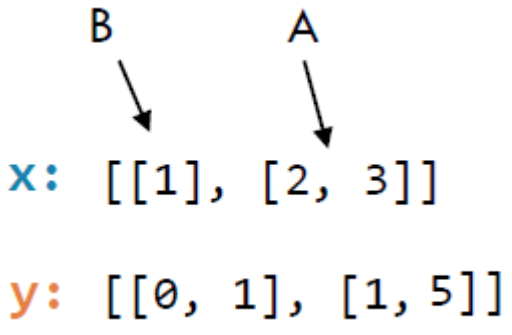
BIG DATA

Transformations - Map Partitions with Index



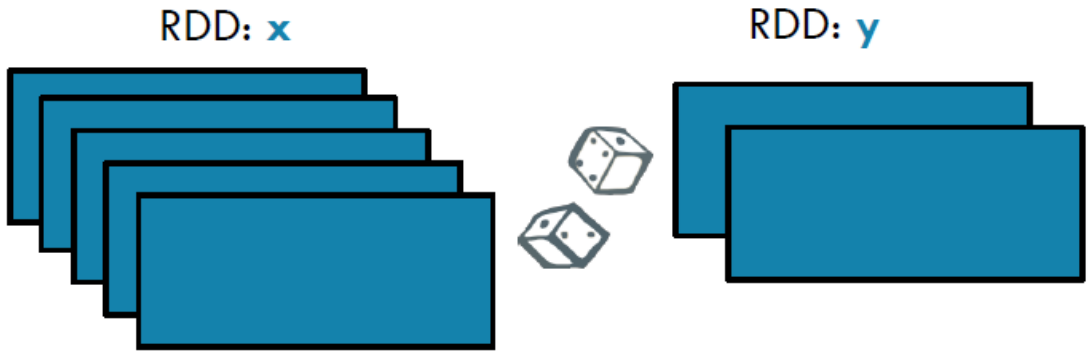
Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition

```
x = sc.parallelize([1,2,3], 2)
def f(partitionIndex, iterator): yield
(partitionIndex, sum(iterator))
y = x.mapPartitionsWithIndex(f)
# glom() flattens elements on the same partition
print(x.glom().collect())
print(y.glom().collect())
```



BIG DATA

Transformations - Sample



sample(*withReplacement, fraction, seed=None*)

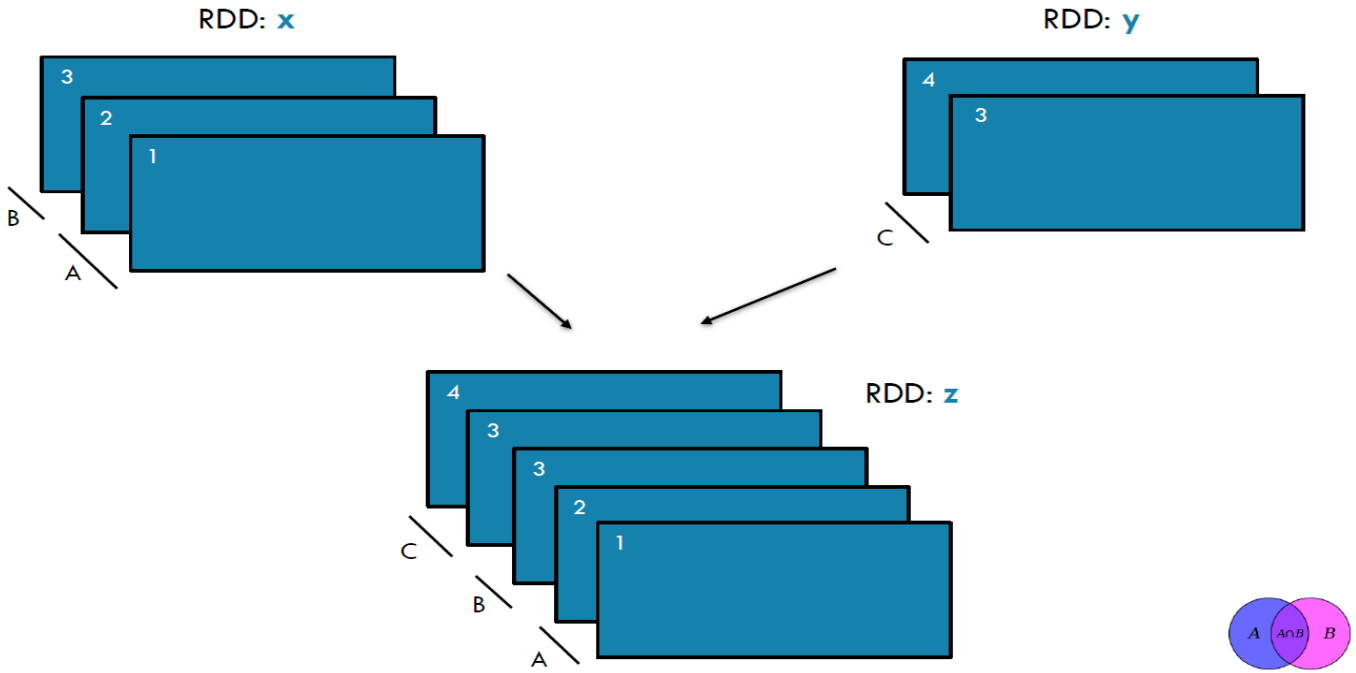
Return a new RDD containing a statistical sample of the original RDD

```
x = sc.parallelize([1, 2, 3, 4, 5])
y = x.sample(False, 0.4, 42)
print(x.collect())
print(y.collect())
```

```
x: [1, 2, 3, 4, 5]
y: [1, 3]
```

BIG DATA

Transformations - Union



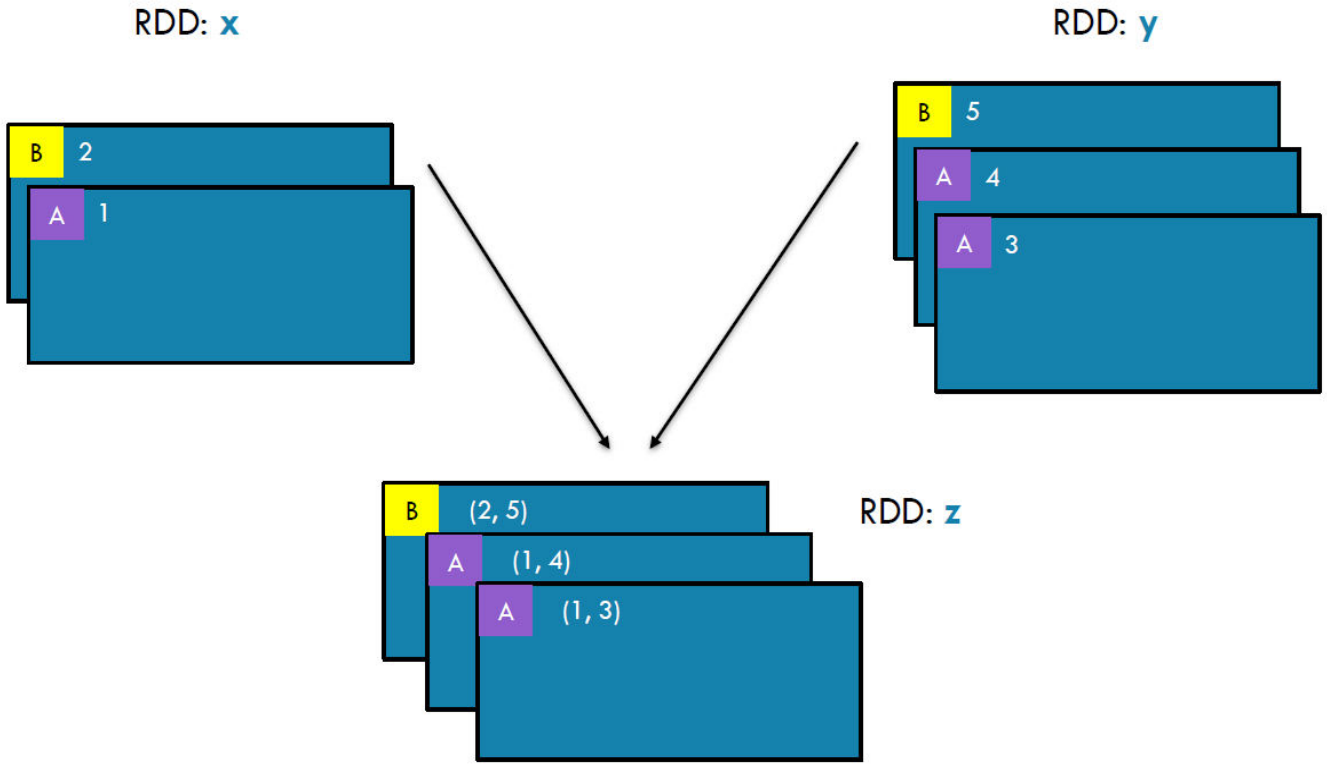
Return a new RDD containing all items from two original RDDs. Duplicates are not culled.

```
x = sc.parallelize([1,2,3], 2)
y = sc.parallelize([3,4], 1)
z = x.union(y)
print(z.glom().collect())
```

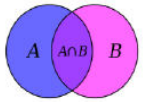
```
x: [1, 2, 3]
y: [3, 4]
z: [[1], [2, 3], [3, 4]]
```

BIG DATA

Transformations - join



Return a new RDD containing all pairs of elements having the same key in the original RDDs

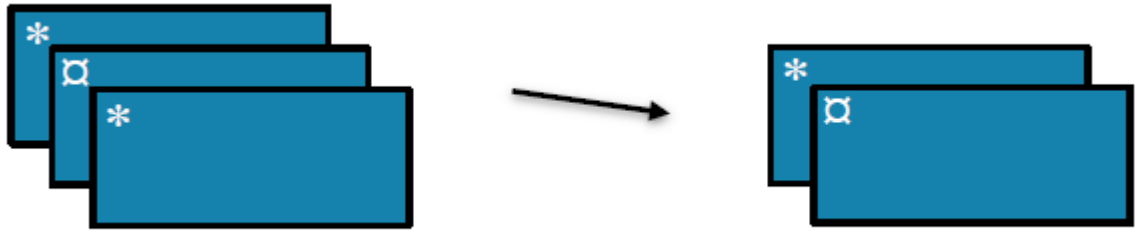



```
x = sc.parallelize([("a", 1), ("b", 2)])
y = sc.parallelize([("a", 3), ("a", 4), ("b", 5)])
z = x.join(y)
print(z.collect())
```

```
x: [("a", 1), ("b", 2)]

y: [("a", 3), ("a", 4), ("b", 5)]

z: [('a', (1, 3)), ('a', (1, 4)), ('b', (2, 5))]
```

BIG DATA

Transformations - distinct



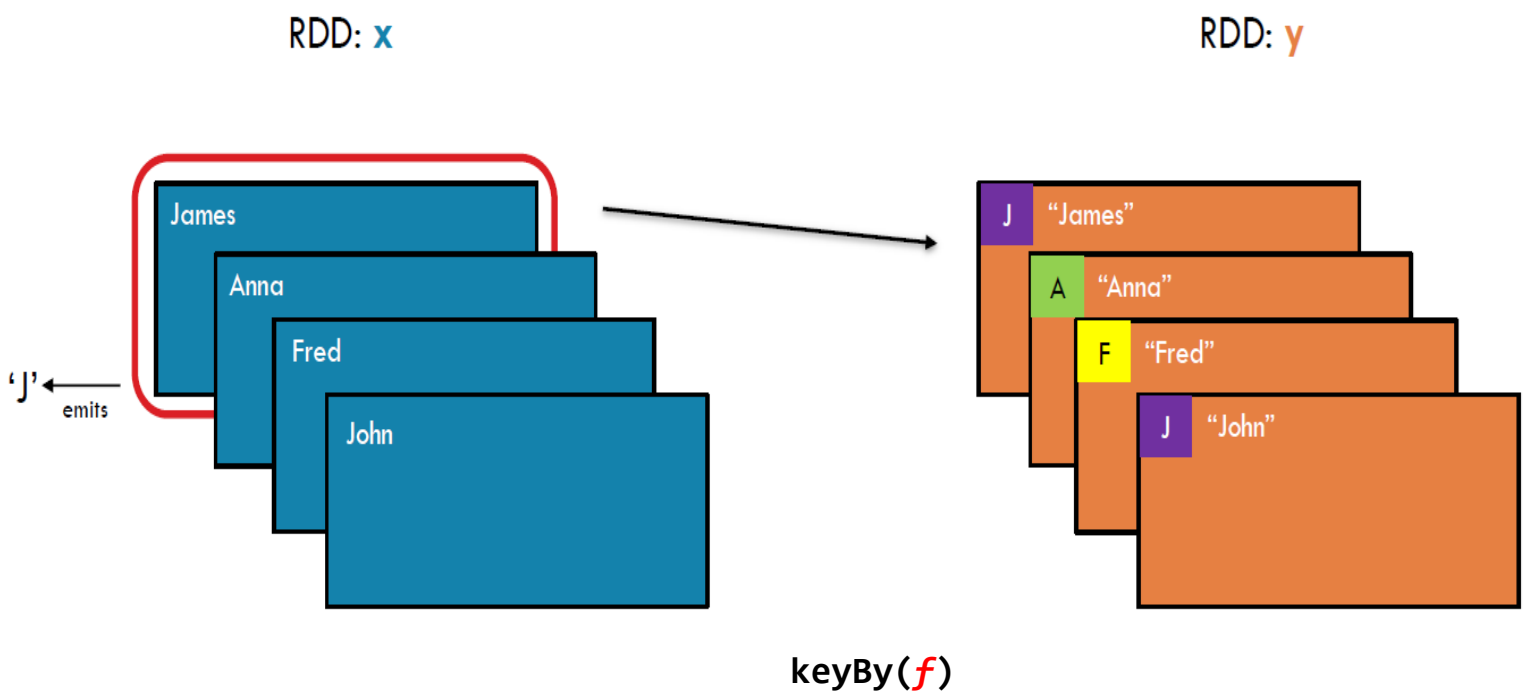
Return a new RDD containing distinct items from the original RDD (omitting all duplicates)

```
x = sc.parallelize([1,2,3,3,4])
y = x.distinct()
print(y.collect())
```

```
x: [1, 2, 3, 3, 4]

y: [1, 2, 3, 4]
```

BIG DATA

Transformations - keyBy



Create a Pair RDD, forming one pair for each item in the original RDD. The pair's key is calculated from the value via a user-supplied function.

```
x = sc.parallelize(['John', 'Fred', 'Anna', 'James'])
y = x.keyBy(lambda w: w[0])
print y.collect()
```

```
x: ['John', 'Fred', 'Anna', 'James']

y: [('J','John'),('F','Fred'),('A','Anna'),('J','James')]
```

Actions (Recap)

- Are operations that return a value

- For example:
  - Reduce() is an action

  - Aggregates all elements of a RDD to produce a result.

Actions

take(n)

The action take(n) returns n number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection. We cannot presume the order of the elements.

For example, consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “take (4)” will give result { 2, 2, 3, 4}

count()

Action count() returns the number of elements in RDD.

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “rdd.count()” will give the result 8.

countByValue()

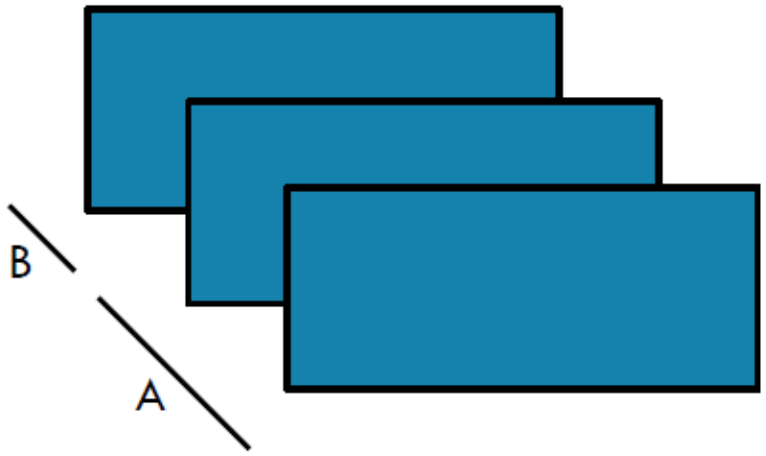
The countByValue() returns, many times each element occur in RDD.
For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD "rdd.countByValue()" will give the result {(1,1), (2,2), (3,1), (4,1), (5,2), (6,1)}

top()

If ordering is present in our RDD, then we can extract top elements from our RDD using top(). Action top() use default ordering of data.

BIG DATA

Actions - GetNum Partitions



getNumPartitions()

Return the number of partitions in RDD


```
x = sc.parallelize([1,2,3], 2)
y = x.getNumPartitions()
print(x.glom().collect())
print(y)
```

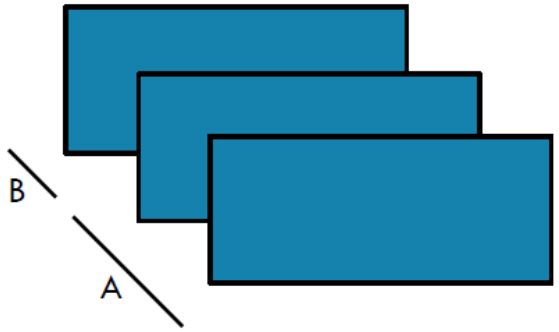
```
x: [[1], [2, 3]]

y: 2
```

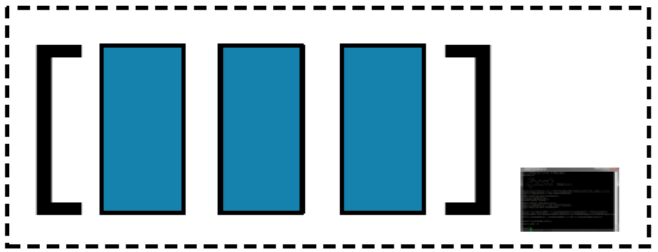
BIG DATA

Actions - collect

collect()

The action collect() is the common and simplest operation that returns our entire RDDs content to driver program. The application of collect() is unit testing where the entire RDD is expected to fit in memory. As a result, it makes easy to compare the result of RDD with the expected result. Action Collect() had a constraint that all the data should fit in the machine, and copies to the driver.





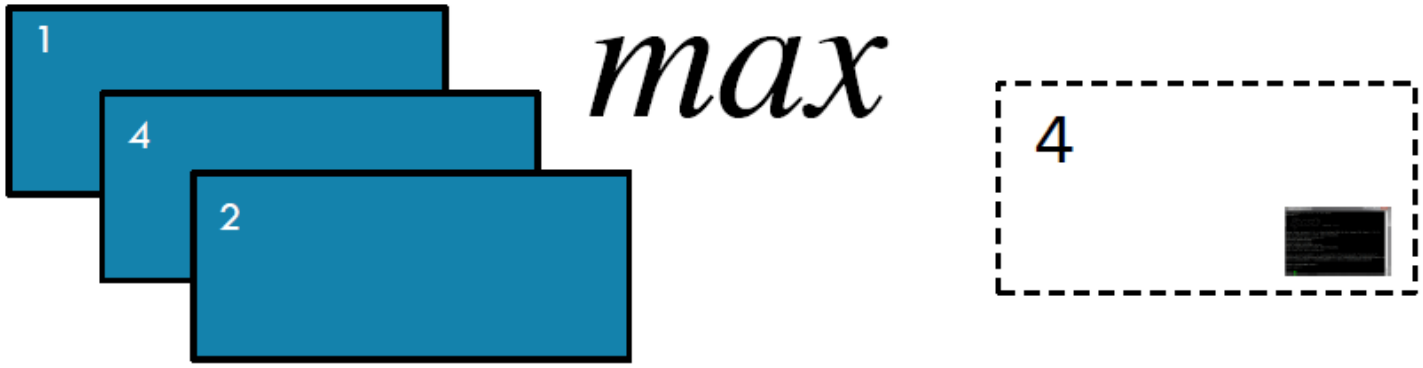
Return all items in the RDD to the driver in a single list

```
x = sc.parallelize([1,2,3], 2)
y = x.collect()
print(x.glom().collect())
print(y)
```

```
x: [[1], [2, 3]]
y: [1, 2, 3]
```

BIG DATA

Actions - max



max()

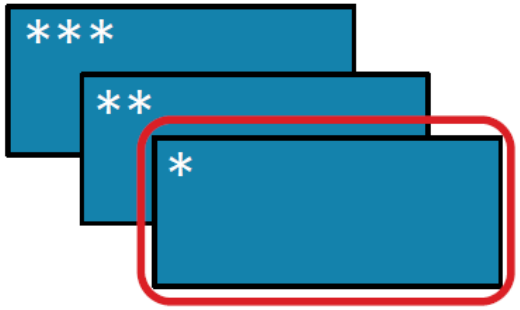
Return the maximum item in the RDD

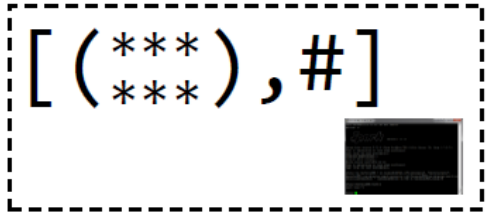
```
x = sc.parallelize([2,4,1])
y = x.max()
print(x.collect())
print(y)
```

```
x: [2, 4, 1]

y: 4
```

BIG DATA

Actions - Aggregate

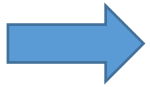




aggregate(identity, *seqOp*, *combOp*)

Aggregate all the elements of the RDD by:
- applying a user function to combine elements with user-supplied objects,
- then combining those user-defined results via a second user function,
- and finally returning a result to the driver.

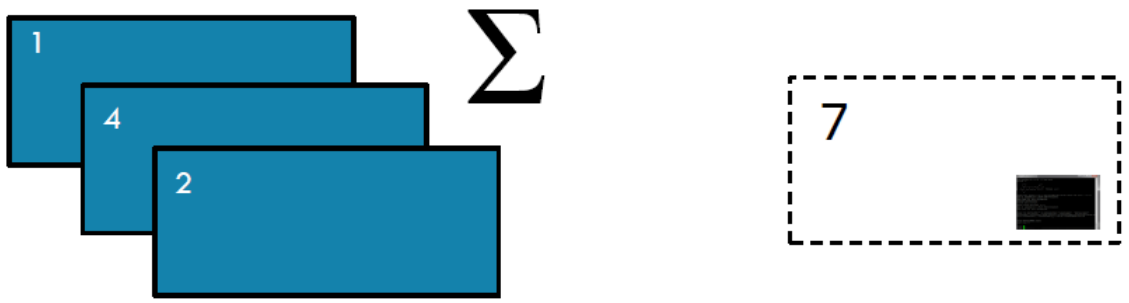
```
seqOp = lambda data, item: (data[0] + [item], data[1] + item)
combOp = lambda d1, d2: (d1[0] + d2[0], d1[1] + d2[1])
x = sc.parallelize([1,2,3,4])
y = x.aggregate(([], 0), seqOp, combOp)
print(y)
```



```
x: [1, 2, 3, 4]

y: ([1, 2, 3, 4], 10)
```

BIG DATA

Actions - sum



sum()

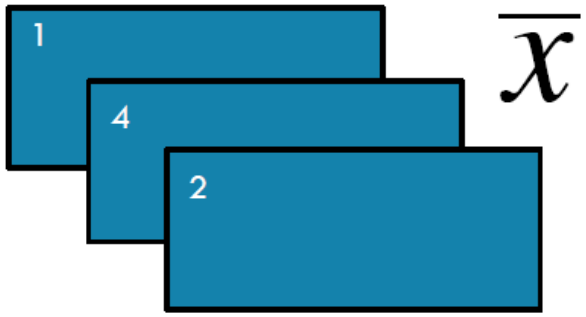
Return the sum of the items in the RDD

```
x = sc.parallelize([2,4,1])
y = x.sum()
print(x.collect())
print(y)
```

```
x: [2, 4, 1]

y: 7
```

BIG DATA

Actions - mean




```
2.3333333
```

mean()

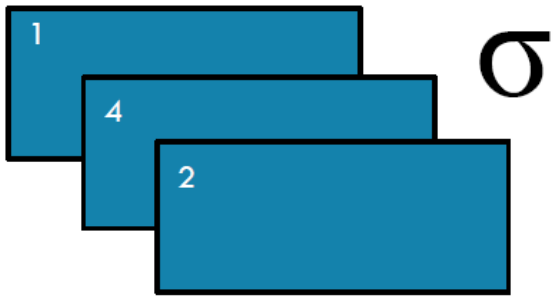
Return the mean of the items in the RDD

```
x = sc.parallelize([2,4,1])
y = x.mean()
print(x.collect())
print(y)
```

```
x: [2, 4, 1]

y: 2.3333333
```

BIG DATA

Actions - stddev



1.2472191

stdev()

Return the standard deviation of the items in the RDD

```
x = sc.parallelize([2,4,1])
y = x.stdev()
print(x.collect())
print(y)
```

```
x: [2, 4, 1]

y: 1.2472191
```

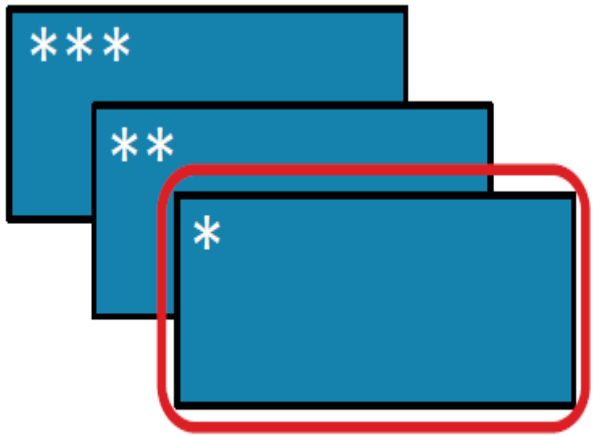
Actions - reduce

reduce()

The reduce() function takes the two elements as input from the RDD and then produces the output of the same type as that of the input elements. The simple forms of such function are an addition. We can add the elements of RDD, count the number of words. It accepts commutative and associative operations as an argument.

BIG DATA

Actions - reduce



reduce(*f*)

Aggregate all the elements of the RDD by applying a user function pairwise to elements and partial results, and returns a result to the driver

```
x = sc.parallelize([1,2,3,4])
y = x.reduce(lambda a,b: a+b)
print(x.collect())
print(y)
```

```
x:  [1, 2, 3, 4]

y:  10
```

Actions - fold

fold()

The signature of the fold() is like reduce(). Besides, it takes “zero value” as input, which is used for the initial call on each partition. But, the condition with zero value is that it should be the identity element of that operation.

The key difference between fold() and reduce() is that, reduce() throws an exception for empty collection, but fold() is defined for empty collection.

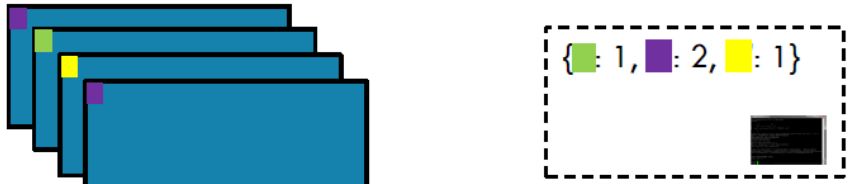
For example, zero is an identity for addition; one is identity element for multiplication.

The return type of fold() is same as that of the element of RDD we are operating on.

For example, rdd.fold(0)((x, y) => x + y).

BIG DATA

Actions - countByKey



countByKey()

Return a map of keys and counts of their occurrences in the RDD

```
x = sc.parallelize([('J', 'James'), ('F','Fred'),
('A','Anna'), ('J','John')])
y = x.countByKey()
print(y)
```

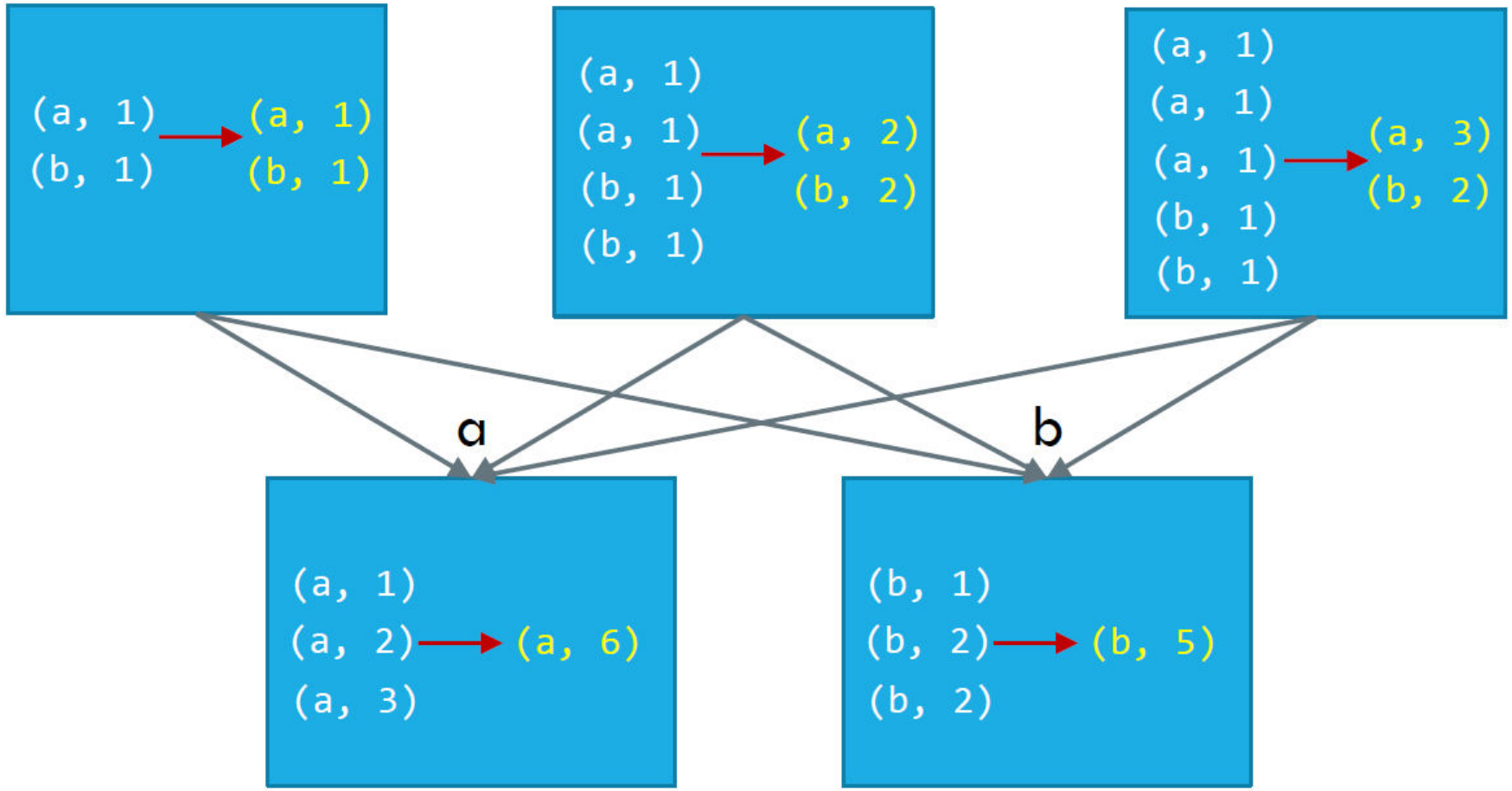
```
x: [('J', 'James'), ('F','Fred'),
    ('A','Anna'), ('J','John')]

y: {'A': 1, 'J': 2, 'F': 1}
```

reduceByKey VS groupByKey


```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))
val wordCountsWithReduce = wordPairsRDD
.reduceByKey(_ + _)
.collect()
val wordCountsWithGroup = wordPairsRDD
.groupByKey()
.map(t => (t._1, t._2.sum))
.collect()
```

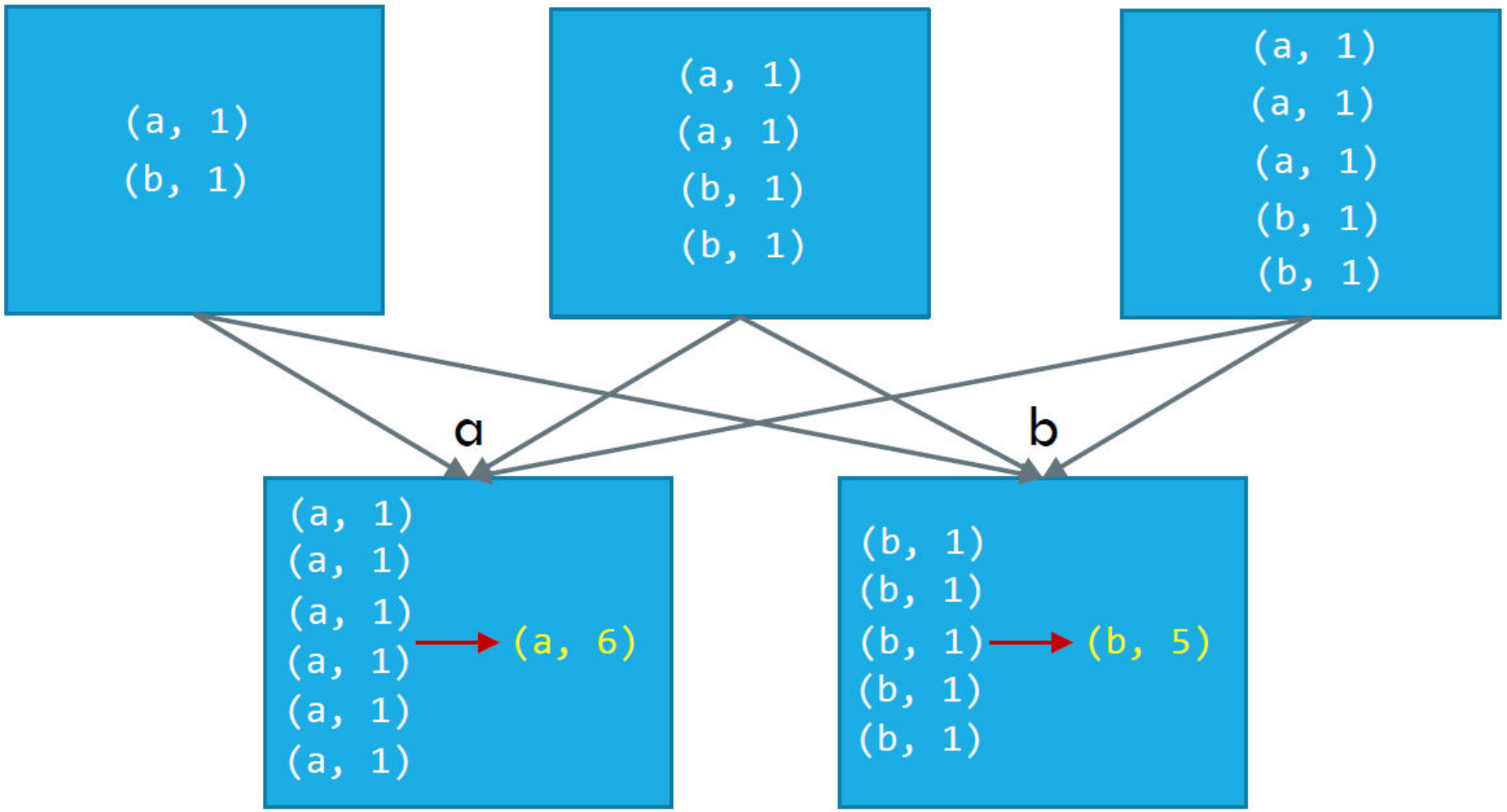
BIG DATA

reduceByKey



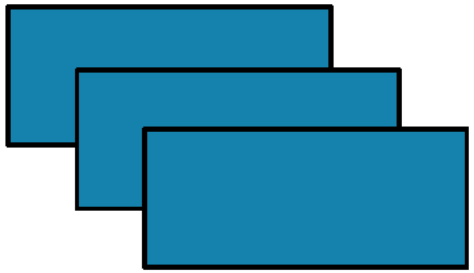
BIG DATA

groupByKey



BIG DATA

Actions -saveAsTextFile



saveAsTextFile(*path, compressionCodecClass=None*)

Save the RDD to the filesystem indicated in the path

```
dbutils.fs.rm("/temp/demo", True)
x = sc.parallelize([2,4,1])
x.saveAsTextFile("/temp/demo")
y = sc.textFile("/temp/demo")
print(y.collect())
```

```
x: [2, 4, 1]

y: [u'2', u'4', u'1']
```

THANK YOU

Prof. J.Ruby Dinakar
Dept. of Computer Science and Engineering
rubydinakar@pes.edu

BIG DATA

PySpark Hands on

Prof. J.Ruby Dinakar
Department of Computer Science and Engineering

Acknowledgements:
Significant information in the slide deck presented through the Unit 3 of the course have been created by **Dr. K V Subramaniam's** and would like to acknowledge and thank him for the same. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for classroom presentation only.

BIG DATA

Overview of lecture

- What is PySpark
- Installation
- PySpark Architecture
- Word count with PySpark and Scala

What is PySpark

BIG DATA

What is PySpark

- PySpark is a Python API
- released by the Apache Spark community to support Python with Spark.
- easily integrate and work with RDDs in Python programming language.
- It has numerous features to work with huge datasets.
- Python has a rich library set.
- majority of data scientists and analytics experts use Python nowadays.

BIG DATA

Features of PySpark

Real-time computations: Because of the in-memory processing in the PySpark framework, it shows low latency.

Polyglot: The PySpark framework is compatible with various languages such as Scala, Java, Python, and R, which makes it one of the most preferable frameworks for processing huge datasets.

Caching and disk persistence: This framework provides powerful caching and great disk persistence.

Fast processing: The PySpark framework is way faster than other traditional frameworks for Big Data processing.

Works well with RDDs: Python programming language is dynamically typed, which helps when working with RDDs.

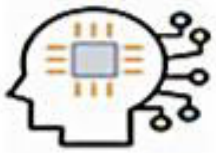
BIG DATA

Spark with Python vs Spark with Scala

Criteria	Python with Spark	Scala with Spark
Performance Speed 	Python is comparatively slower than Scala when used with Spark, but programmers can do much more with Python than with Scala as Python provides an easier interface	Spark is written in Scala, so it integrates well with Scala. It is faster than Python
Learning Curve 	Python is known for its easy syntax and is a high-level language easier to learn. It is also highly productive even with its simple syntax	Scala has an arcane syntax making it hard to learn, but once you get a hold of it you will see that it has its own benefits
Data Science Libraries 	In Python API, you don't have to worry about the visualizations or Data Science libraries. You can easily port the core parts of R to Python as well	Scala lacks proper Data Science libraries and tools, and it does not have proper tools for visualization

BIG DATA

Spark with Python vs Spark with Scala

Readability of Code 	Readability, maintenance, and familiarity of code are better in Python API	In Scala API, it is easy to make internal changes since Spark is written in Scala
Complexity 	Python API has an easy, simple, and comprehensive interface	Scala, in fact, produces verbose output, and hence it is considered a complex language
Machine Learning Libraries 	Python is preferred for implementing Machine Learning algorithms	Scala is preferred when you have to implement Data Engineer technologies rather than Machine Learning

BIG DATA

Interactive and Batch processing

- Supports the pyspark shell for interactive processing
- And regular batch processing jobs can be run by writing pyspark scripts

PySpark configuration

BIG DATA

Downloading and Setting up pyspark

- Download and untar the spark tar file from the spark repository

```
# tar -xvf Downloads/spark-3.0.1-bin-hadoop2.7.tgz
```

- It will create a directory `spark-3.0.1-bin-hadoop2.7`.
  Before starting PySpark, you need to set the following environments to set the Spark path and the Py4j path.

- Pyspark is bundled along with spark

- However, it requires some configuration
  - Setup of proper paths.

BIG DATA

Pyspark configuration

- Needs environment variables to be setup
- First modify .bashrc to include

```
export SPARK_HOME = /home/hadoop/ spark-3.0.1-bin-hadoop2.7
export PATH = $PATH:/home/hadoop/ spark-3.0.1-bin-hadoop2.7 /bin
export PYTHONPATH = $SPARK_HOME/python:$SPARK_HOME/python/lib/py4j-0.10.4-src.zip:$PYTHONPATH
export PATH = $SPARK_HOME/python:$PATH
```

Then run so that the env variables are setup

Or, to set the above environments globally, put them in the .bashrc file. Then run the following command for the environments to work.

```
# source .bashrc
```

Starting pyspark

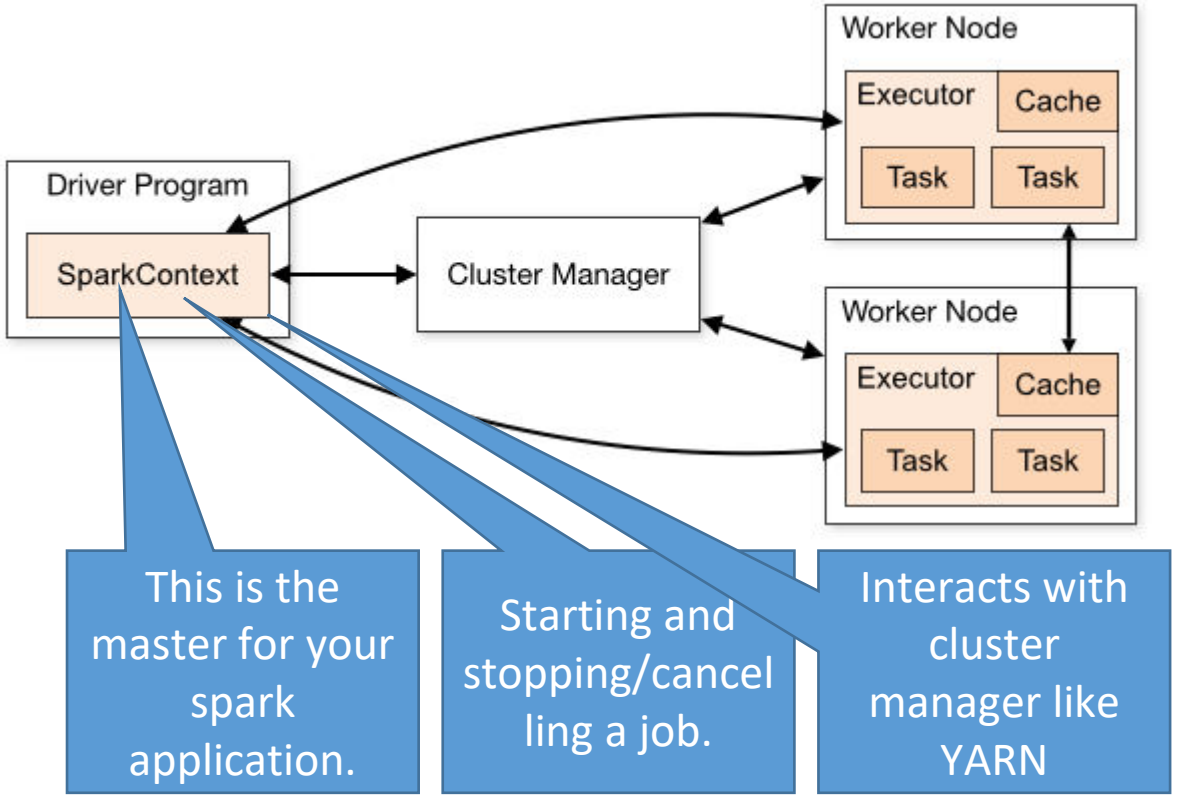
- Needs environment variables to be setup
- First modify .bashrc to include

- Now that we have all the environments set, let us go to Spark directory and invoke PySpark shell by running the following command –

```
./bin/pyspark
```

BIG DATA

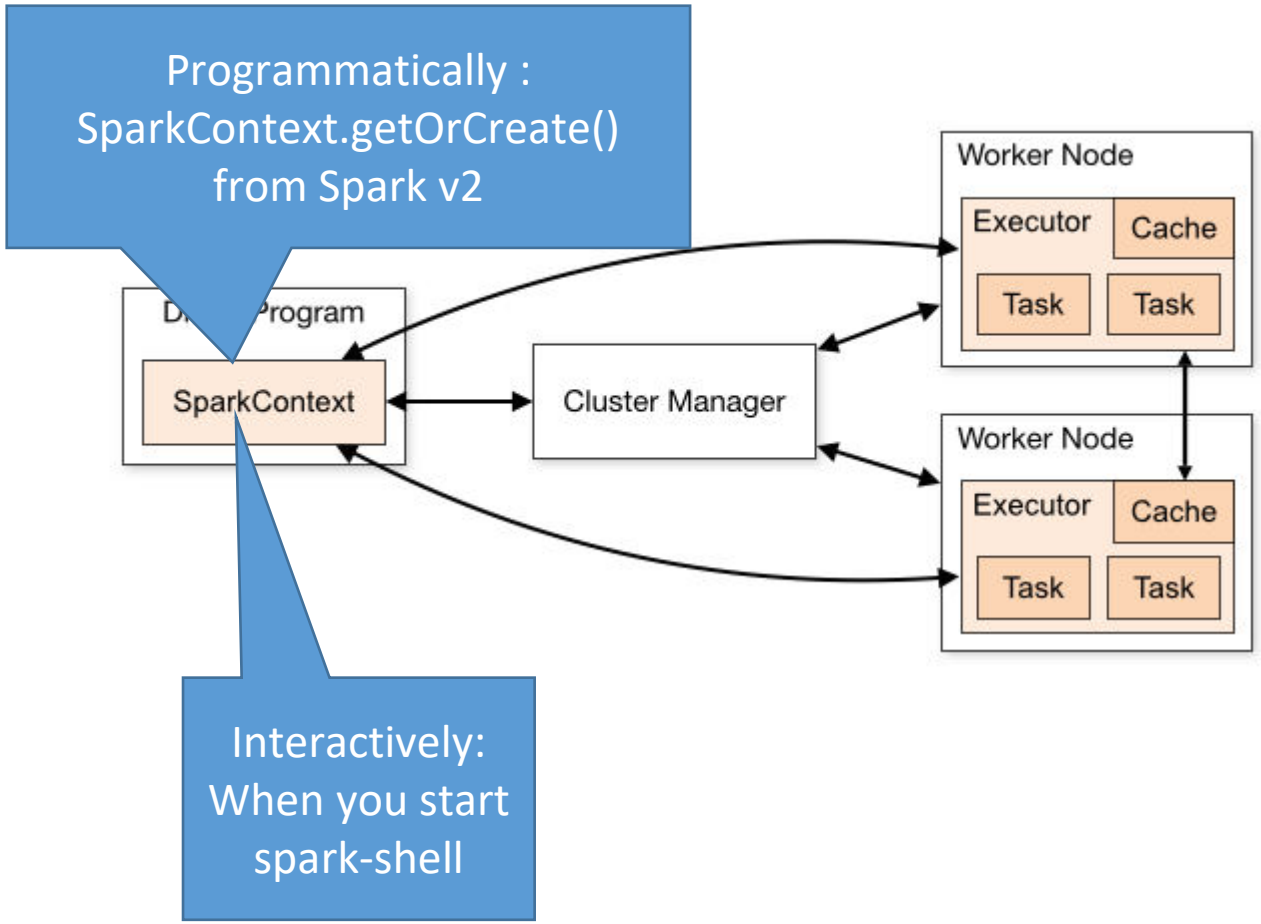
Recall: The Spark Context



https://spark.apache.org/docs/latest/cluster-overview.html

BIG DATA

When is Spark Context created?



There is one SparkContext per JVM

https://spark.apache.org/docs/latest/cluster-overview.html

Pyspark architecture

Pyspark architecture

- So if the spark context is maintained per JVM
- How does pyspark take care of the SparkContext?
  - Who creates and maintains this?

py4j

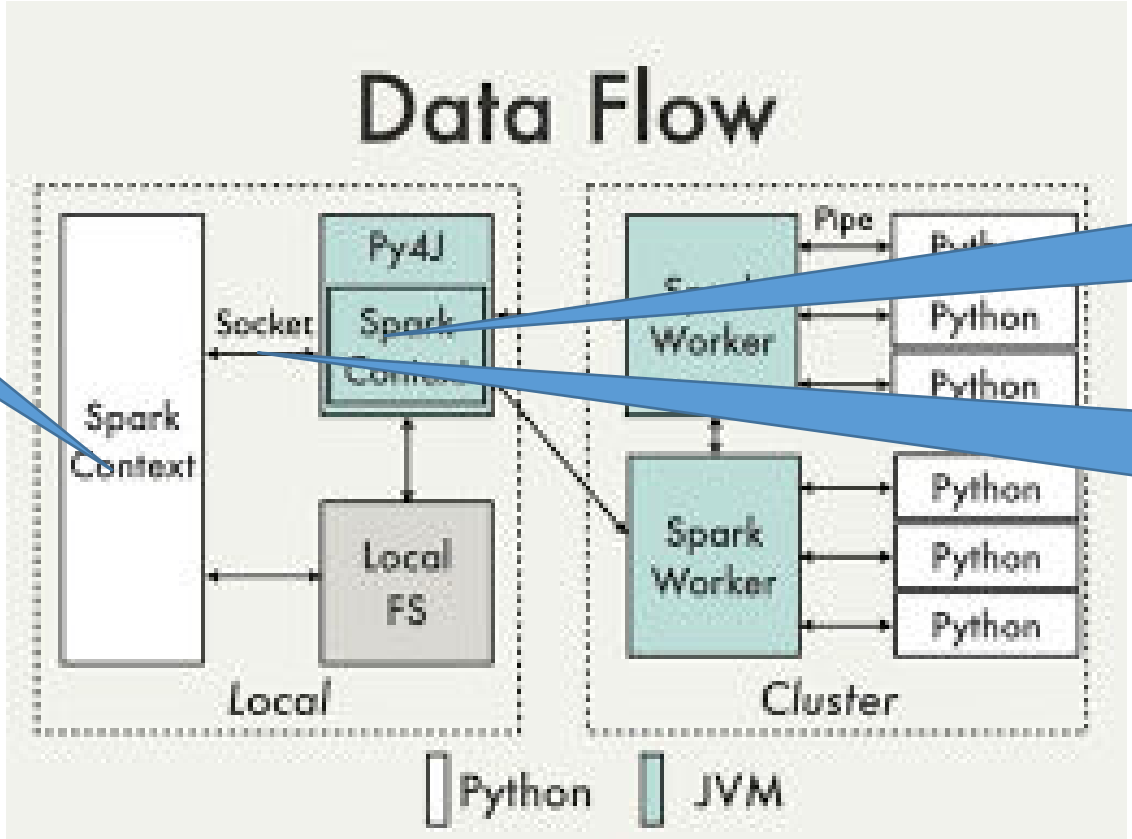
- Acts like a bridge between python and java
- Allows python interpreter to access Java objects instantiated within the JVM
- Can invoke methods on the Java objects as if they were python methods

https://www.py4j.org/

BIG DATA

Pyspark architecture

Python driver



Java SparkContext used to drive the application

Data has to be serialized using cPickle serializer

https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals

Lifecycle of a Spark Program

- Create some input RDDs from external data or parallelize a collection in your driver program.
- Lazily transform them to define new RDDs using transformations like filter() or map()
- Ask Spark to cache() any intermediate RDDs that will need to be reused.
- Launch actions such as count() and collect() to kick off a parallel computation, which is then optimized and executed by Spark.

SparkContext

- Main entry point for Spark functionality.
- A SparkContext represents the connection to a Spark cluster, and it can be used to create RDD and broadcast variables on that cluster.
- When you create a new SparkContext, at least the master and app name should be set, either through the named parameters or through conf.

Create RDD

Two ways we can create the RDDs:

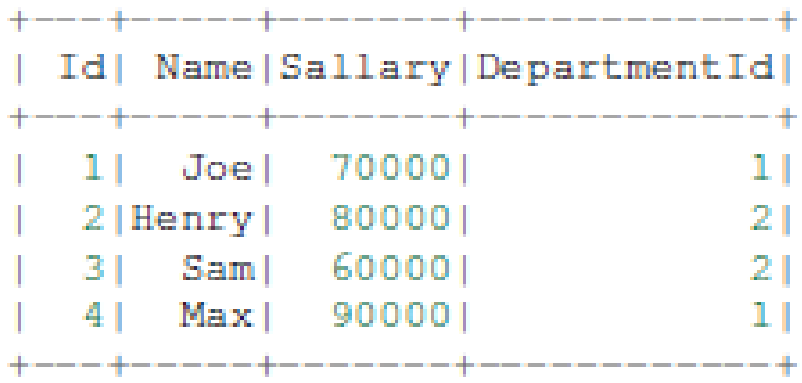
- loading an external dataset
- distributing a set of collection of objects.

Using parallelize()

```
from pyspark.sql import SparkSession

spark = SparkSession \

.builder \

.appName("Python Spark create RDD example") \

.config("spark.some.config.option", "some-value") \

.getOrCreate()

myData = spark.sparkContext.parallelize([(1,2), (3,4), (5,6), (7,8), (9,10)])

myData.collect()
```

Using createDataFrame()

```
from pyspark.sql import SparkSession
spark = SparkSession \
.builder \
.appName("Python Spark create RDD example") \
.config("spark.some.config.option", "some-value") \
.getOrCreate()
Employee = spark.createDataFrame([
('1', 'Joe', '70000', '1'),
('2', 'Henry', '80000', '2'),
('3', 'Sam', '60000', '2'),
('4', 'Max', '90000', '1')],
['Id', 'Name', 'Sallary','DepartmentId']
)
```



Id	Name	Sallary	DepartmentId
1	Joe	70000	1
2	Henry	80000	2
3	Sam	60000	2
4	Max	90000	1

Simplifying tasks for programmers - DataFrames

BIG DATA

Need for dataframes

- Data RDDs are completely opaque to Spark
  - Meaning Spark cannot parse these values
- Is there some way to make Spark understand the format, so that we can do processing more easily
  - Like sql type queries
- Consider data like on the right that we need to run a query on?

USN	Name	Marks
45	Vkoli	11
10	Stendul	43
195	Abachpan	28

BIG DATA

What is a dataframe

- Introduced in 2015
- Inspired by Dataframes in R and Pandas in python
- Distributed collection of data into named columns
- An abstraction built over RDD that allows
  - Schema to be defined on a RDD
- Also has an optimizer built in for queries

USN	Name	Marks
45	Vkoli	11
10	Stendul	43
195	Abachpan	28

BIG DATA

Creating dataframes

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

df = sqlContext.jsonFile("pes/students.json")

# let us display the contents
df.show()
## USN  name  marks
## 045  Vkoli   11
## 010  Stendul  43
## 195  Abachpan  28

Df = rdd.toDF("USN", "name")
```

Dataframes can be created from existing RDDs, HIVE tables other Data sources.

This example is creating from a JSON file

Alternatively, from an existing RDD by naming the columns

BIG DATA

Using a dataframe


```
# Print the schema in a tree format
df.printSchema()
## root
## |-- usn: long (nullable = true)
## |-- name: string (nullable = true)
## |-- marks: long (nullable = true)

# Select only the "name" column
df.select("name").show()
## name
## VKoli
## STendul
## ABachpan

# Select everybody, but increment the marks by 1
df.select("name", df.marks + 1).show()
## name    (marks + 1)
## Vkoli       12
## STendul     44
## ABachpan    29
```

Exercise

- Consider a case where you have data in a CSV file that consists of <pan number, date, tax_paid> and you wanted to find out the total tax paid by each individual pan holder
  - How will you do it in Spark?
  - How will you do it with Spark Data frames

Task Assignment

- Consider a case where you have data in a CSV file that consists of <pan number, date, tax_paid> and you wanted to find out the total tax paid by each individual pan holder
  - How will you do it in Spark?
  - How will you do it with Spark Data frames

Using Dataframes

```
Df = rdd.toDF("pan number", "date", "taxpaid")
Df.select("pan number", "tax paid").groupBy("pan number").sum()
```

Note that this is done using the name of the column rather than by splitting the data which we would do if used Spark.

PySpark Demo

THANK YOU

Prof. J.Ruby Dinakar
Dept. of Computer Science and Engineering
rubydinakar@pes.edu

BIG DATA

Spark SQL

Prof. J.Ruby Dinakar
Department of Computer Science and Engineering

Spark SQL

BIG DATA

Structured Data Processing

A common use case in big-data is to process structured or semi-structured data

- In Spark RDD, all functions and objects are black-boxes.
- Any structure of the data has to be part of the functions which includes:
  - **Parsing**
  - **Conversion**
  - **Processing**

BIG DATA

Structured data processing

- Pig/Pig Latin

Builds on Hadoop

Converts SQL-like programs to MapReduce

- Hive/HiveQL

Supports SQL-like queries

- Shark (Hive on Spark)

Translates HiveQL queries to RDD programs

Initial attempt to support SQL on Spark

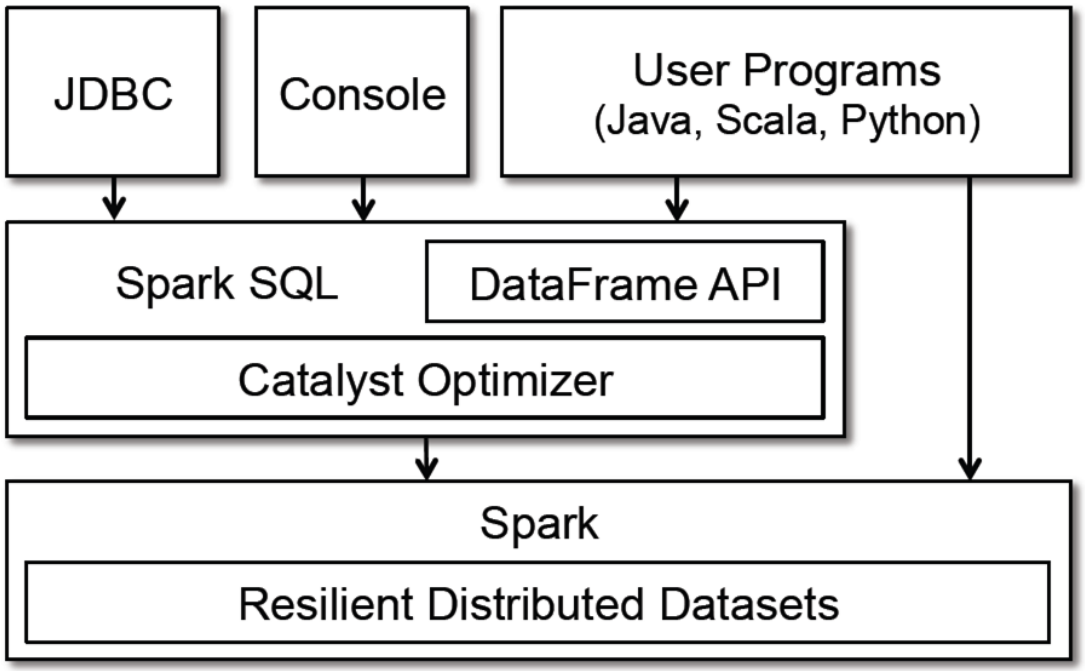
Redesigned to consider Spark query model

- Supports all the popular relational operators
- Can be intermixed with RDD operations
- Uses the Dataframe API as an enhancement to the RDD API

Dataframe = RDD + schema

BIG DATA

Programming interface



Spark SQL runs as a library on top of Spark.
Spark SQL **components** are
DataFrame API and catalyst optimizer.
Application written in Java and other languages can access the database by creating a connection using JDBC/ODBC APIs.
DataFrame API is integrated into Spark's supported programming languages.
A **DataFrame** is a distributed collection of data organized into named columns. It is equivalent to a relational table in SQL used for storing data into tables.
Catalyst optimizer automatically finds out the most efficient plan to execute data operations specified in the user's program.
It "translates" transformations used to build the Dataset to physical plan of execution. It's similar to DAG scheduler which creates plan of execution for RDD.

Data Model

- Spark SQL uses a nested data model based on Hive for tables and DataFrames.
- Supports both primitive SQL types (boolean, integer, double, decimal, string, data, timestamp) and complex types (structs, arrays, maps, and unions); also user defined types.
- First class support for complex data types

Built-in operations in SprkSQL

Filter (Selection)

- Select (Projection)
- Join
- Group By (Aggregation)
- Load/Store in various formats
- Cache
- Conversion between RDD (back and forth)

BIG DATA

DataFrame Operations

Users can perform relational operations on DataFrames using a domain-specific language (DSL) similar to R data frames and Python Pandas

Support all common relational operations (select, where, join, groupBy) via a DSL

Operators take expression objects

Operators build up an abstract syntax tree (AST), which is then optimized by Catalyst.

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

This code computes the number of female employees in each department. Here, employees is a DataFrame, and employees("deptId") is an expression representing the deptId column.

Alternatively, register as temp SQL table and perform traditional SQL query strings

```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

Advantages over Relational Query Languages

Holistic optimization across functions composed in different languages.

Control structures (e.g. if, for)

Logical plan analyzed eagerly - identify code errors associated with data schema issues on the fly.

BIG DATA

Querying Native Datasets

- Infer column names and types directly from data objects (via reflection in Java and Scala and data sampling in Python, which is dynamically typed)

```
case class User(name: String, age: Int)
```

  - in the above example Spark SQL automatically detects the names ("name" and "age") and data types (string and int) of the columns.

Columnar storage with *hot* columns cached in memory

- Native objects accessed in-place to avoid expensive data format transformation(extracts only the fields used in the query)
- Benefits:
- Run relational operations on existing Spark programs.
- Combine RDDs with external structured data

RDD[String] → (User Defined Function) → RDD[User] → (toDF method) → DataFrame

BIG DATA

User-Defined Functions (UDFs)

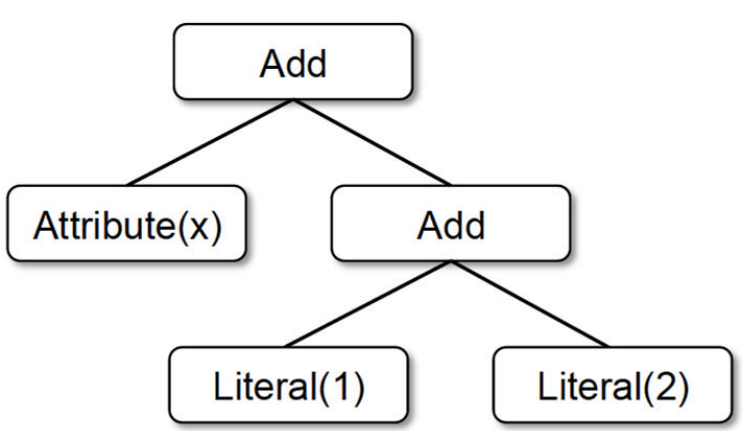
- Extend the limited functions of the framework and reuse this function on several DataFrames.
- Allows inline registration of UDFs
- Compare with Pig, which requires the UDF to be written in a Java package that's loaded into the Pig script.
- Can be defined on simple data types or entire tables.
- UDFs available to other interfaces after registration

```
val model: LogisticRegressionModel = ...

ctx.udf.register("predict",
  (x: Float, y: Float) => model.predict(Vector(x, y)))

ctx.sql("SELECT predict(age, weight) FROM users")
```

BIG DATA

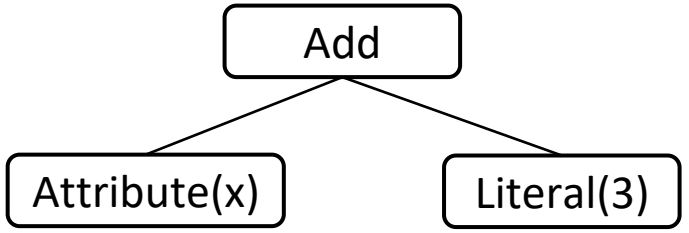
Catalyst

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```



x + (1 + 2)

Add(Attribute(x), Add(Literal(1), Literal(2)))



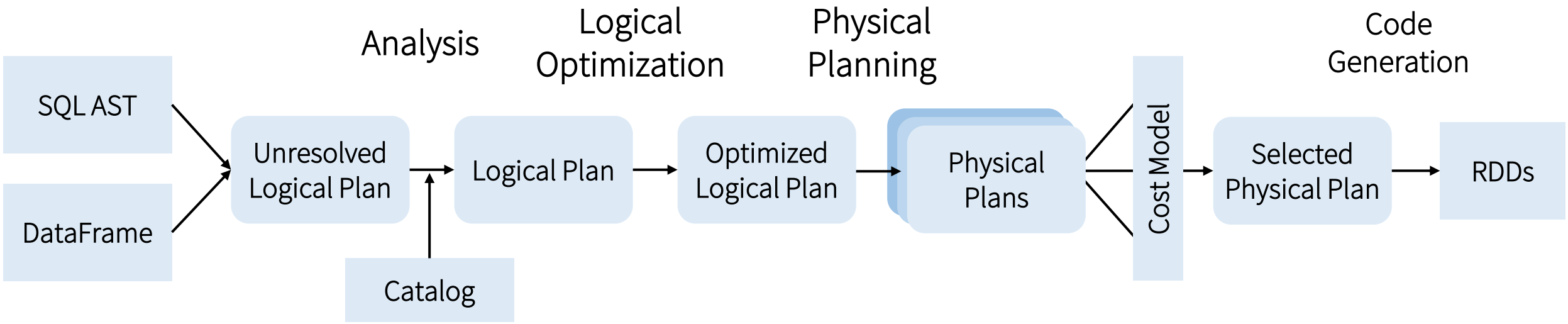
x + 3

The main data type in Catalyst is a tree composed of node objects.
Each node has a node type and zero or more children.
New node types are defined in Scala as subclasses of the TreeNode class.
These objects are immutable and can be manipulated using transformations

Catalyst Rules

- Trees can be manipulated using rules, which are functions from a tree to another tree.
- Pattern matching functions that transform subtrees into specific structures.
- Partial function—skip over subtrees that do not match no need to modify existing rules when adding new types of operators.
- Multiple patterns in the same transform call.
- May take multiple batches to reach a fixed point.
- transform can contain arbitrary Scala code.

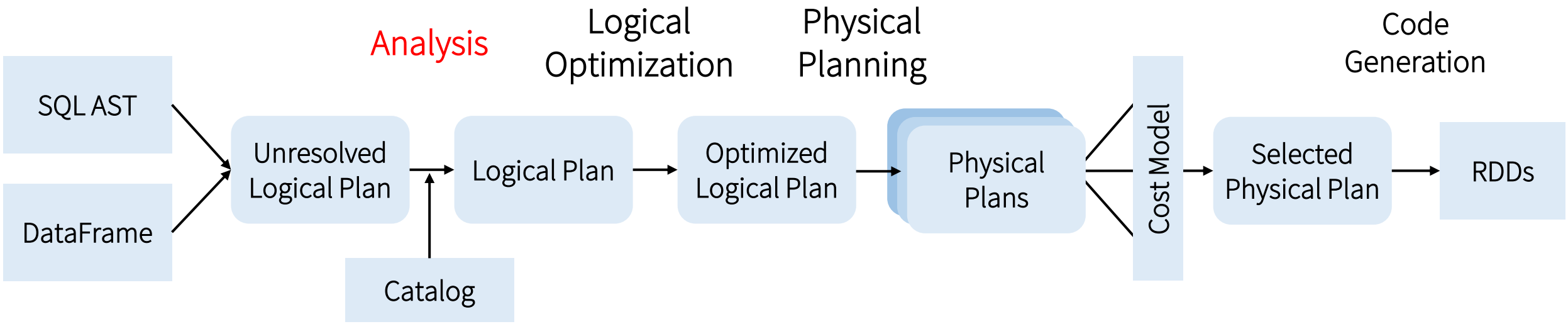
Plan Optimization & Execution



Data Frames and SQL share the same optimization/execution pipeline

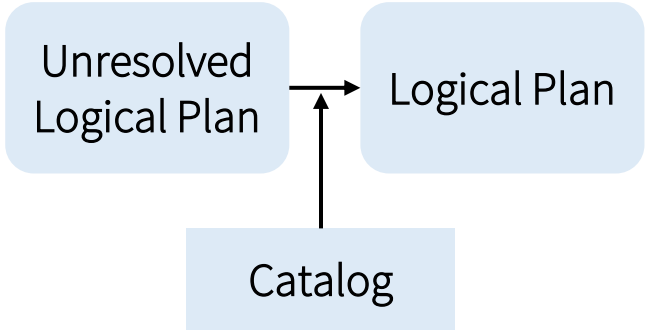
The SQL queries of Spark application will be converted to Dataframe APIs
Logical Plan is converted to an Optimized Logic plan and then to one or more Physical Plans

Plan Optimization & Execution



Plan Optimization & Execution - Analysis

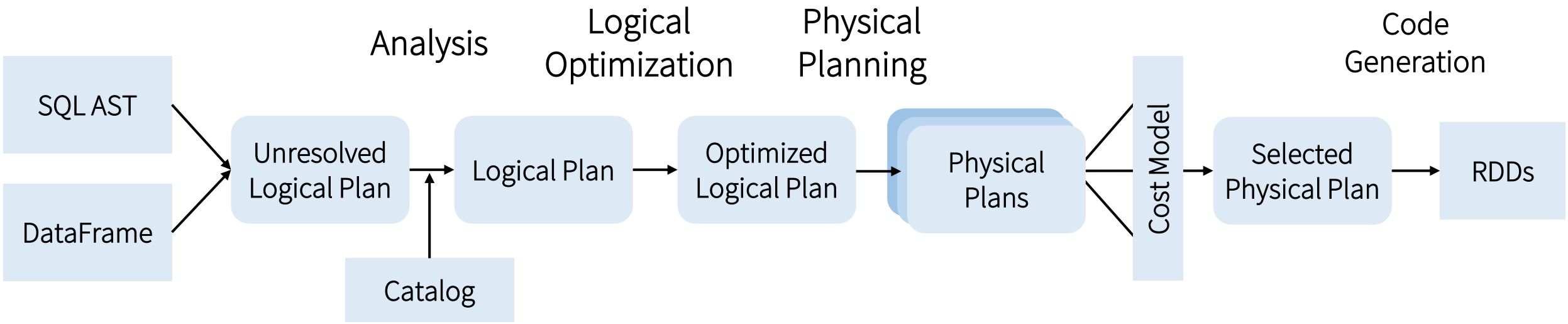
Analysis



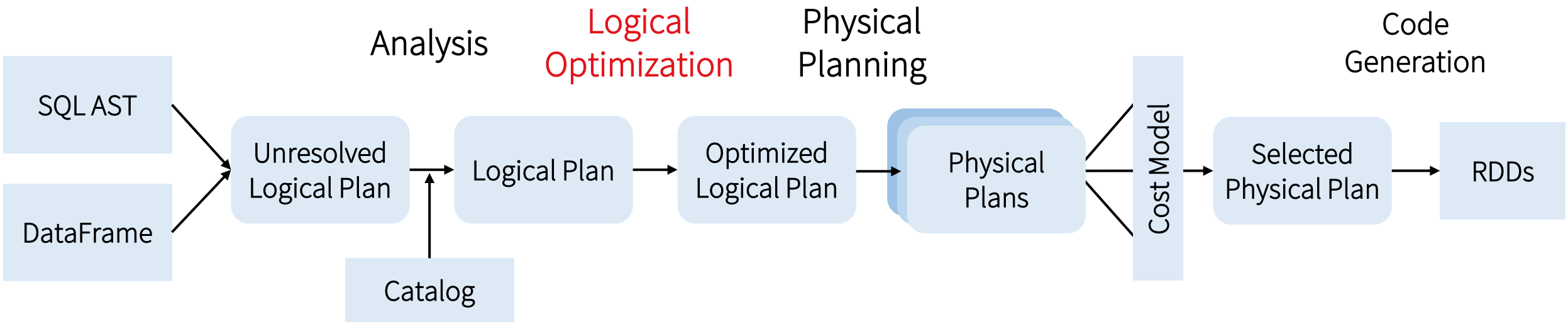
SELECT *col* FROM *sales*

- An attribute is unresolved if its type is not known or it's not matched to an input table.
- To resolve attributes:
- Look up relations by name from the catalog.
- Map named attributes to the input provided given operator's children.
- UniqueID for references to the same value
- Propagate and coerce types through expressions (e.g. 1 + col)

Plan Optimization & Execution

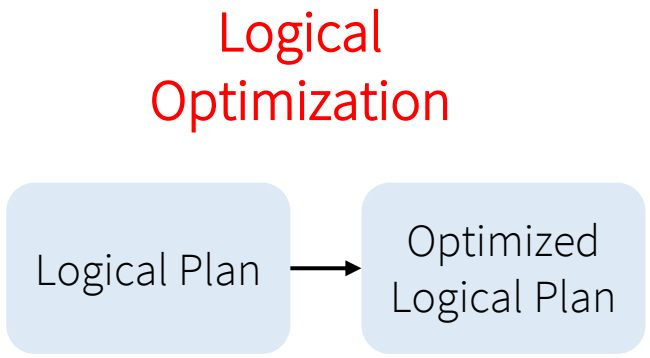


Plan Optimization & Execution



Spark Catalyst's optimizer is responsible for generating an optimized logical plan from the analyzed logical plan. Optimization is done by applying rules in batches. Each operation is represented as a TreeNode in Spark SQL When an analyzed plan goes through the optimizer , the tree is transformed to a new tree repeatedly by applying a set of optimization rules

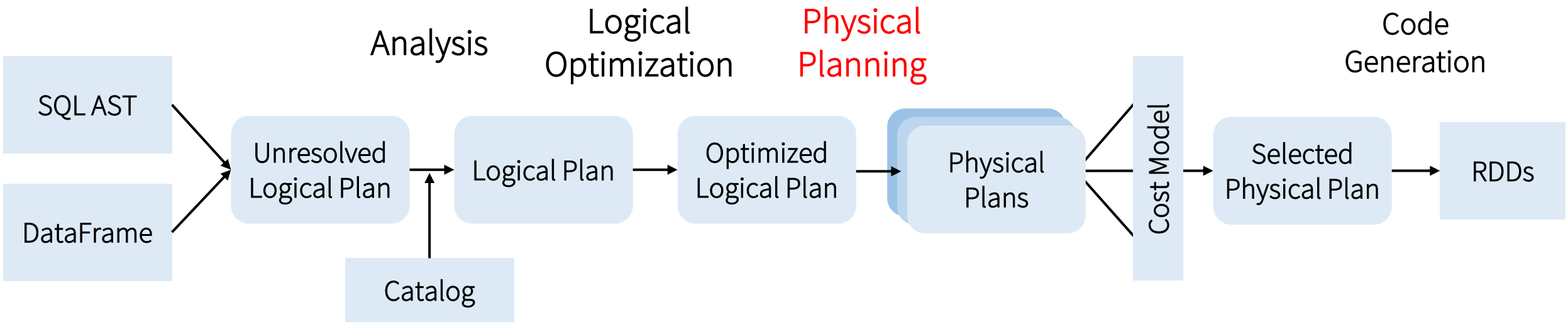
Plan Optimization & Execution – Logical Optimization



- Applies standard rule-based optimization (constant folding, predicate-pushdown, projection pruning, null propagation, boolean expression simplification, etc)
- 800LOC

```
object DecimalAggregates extends Rule[LogicalPlan] {
  /** Maximum number of decimal digits in a Long */
  val MAX_LONG_DIGITS = 18

  def apply(plan: LogicalPlan): LogicalPlan = {
    plan transformAllExpressions {
      case Sum(e @ DecimalType.Expression(prec, scale))
              if prec + 10 <= MAX_LONG_DIGITS =>
        MakeDecimal(Sum(LongValue(e)), prec + 10, scale)
    }
}
```

Plan Optimization & Execution – Logical Optimization

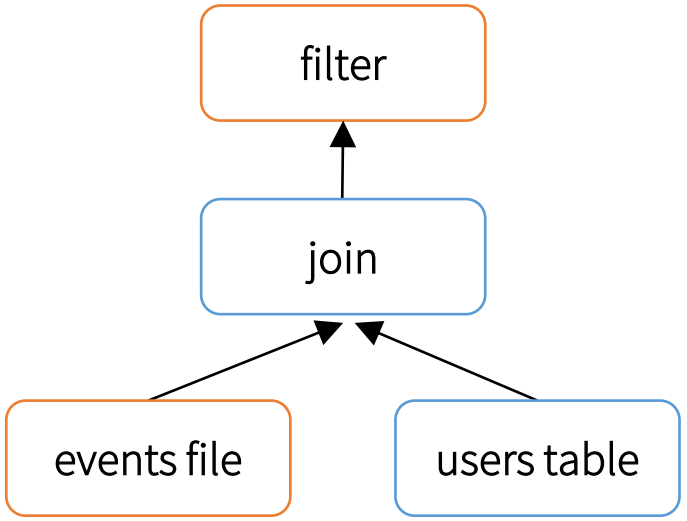


- e.g. Pipeline projections and filters into a single map

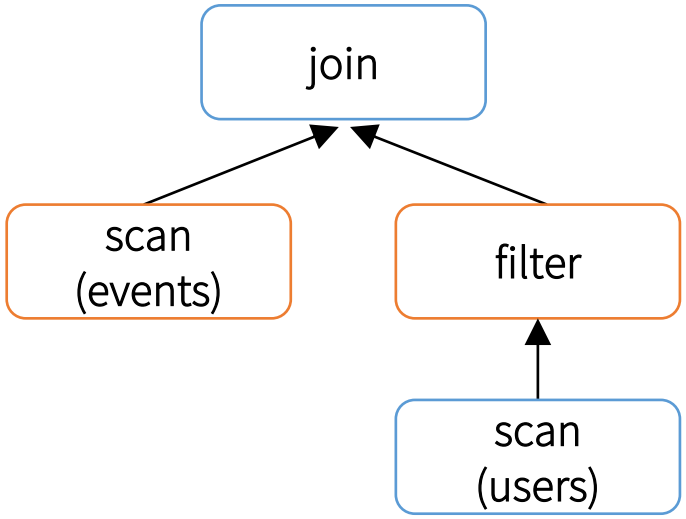
Plan Optimization & Execution – Logical Optimization

```
def add_demographics(events):
    u = sqlCtx.table("users")                    # Load partitioned Hive table
    events \
      .join(u, events.user_id == u.user_id) \    # Join on user_id
      .withColumn("city", zipToCity(df.zip))     # Run udf to add city column

events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "Melbourne").select(events.timestamp).collect()
```

Logical Plan

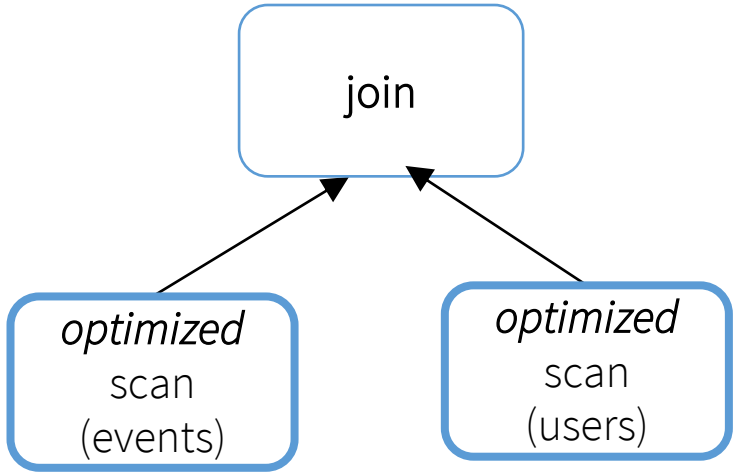


Physical Plan



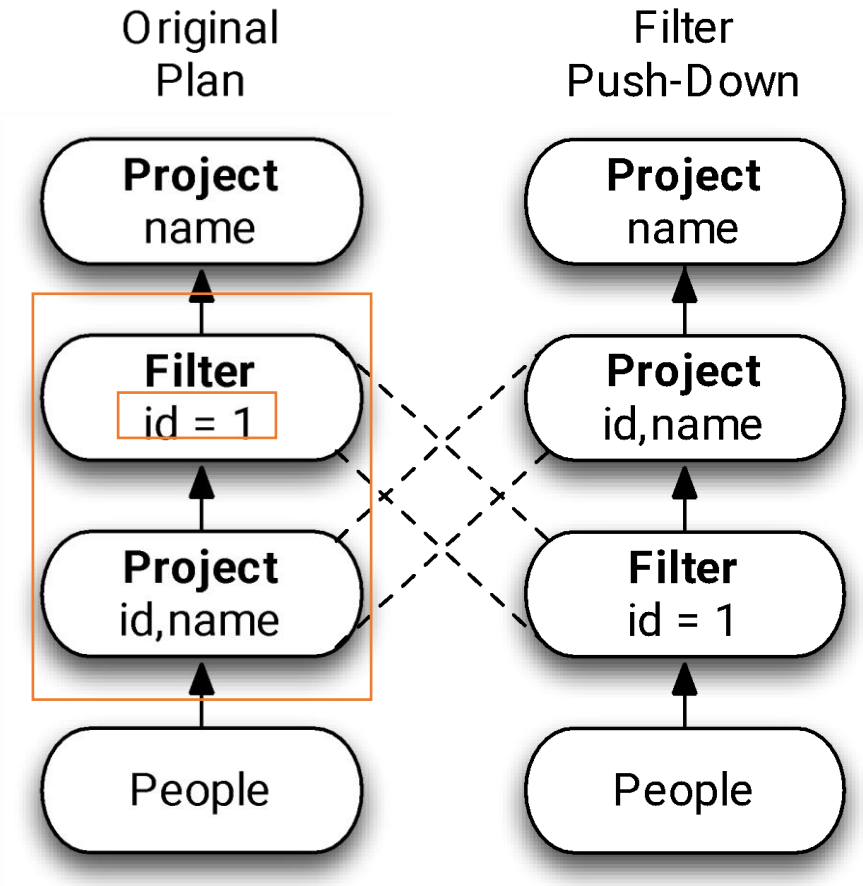
Physical Plan

with Predicate Pushdown
and Column Pruning

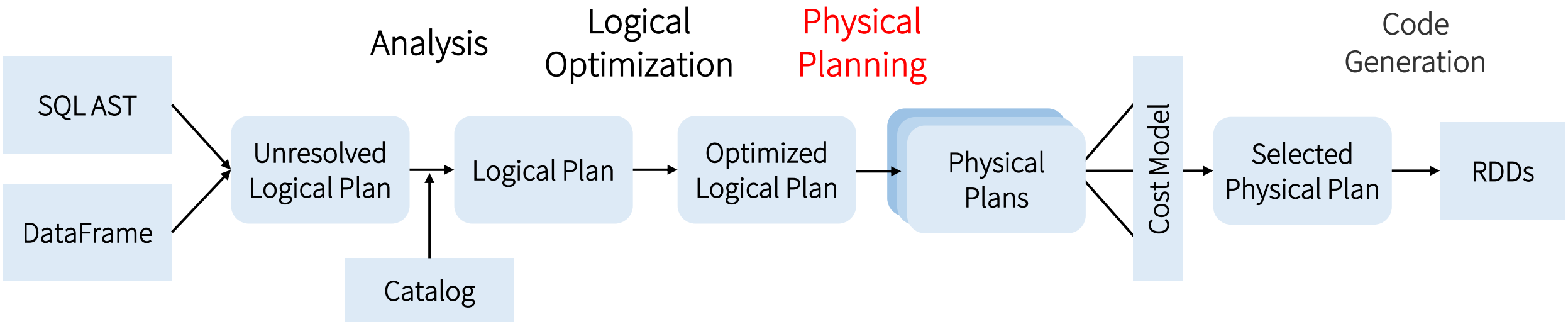


An Example Catalyst Transformation

- Find filters on top of projections.
- Check that the filter can be evaluated without the result of the project.
- If so, switch the operators.



Plan Optimization & Execution - Physical Planning

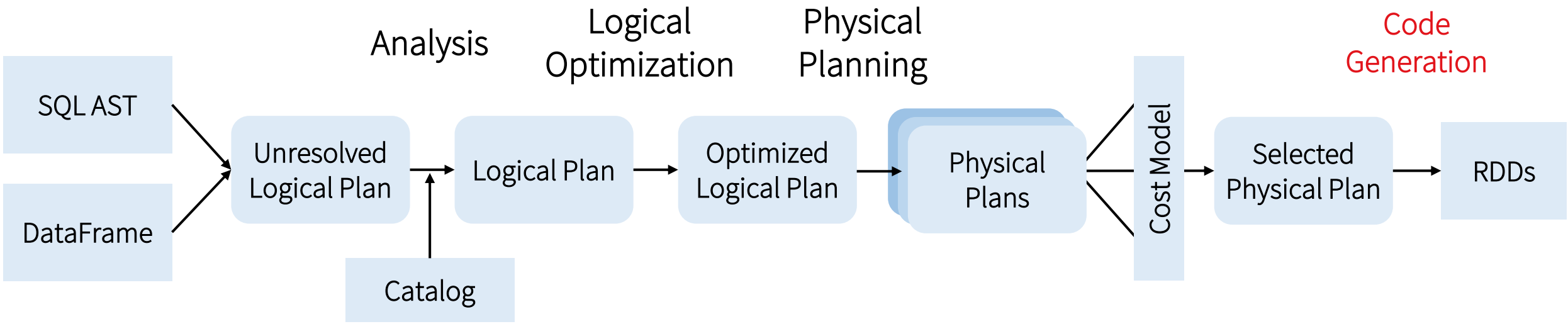


Physical plans are the ones that can actually be executed on a cluster. They actually translate optimized logical plans into RDD operations to be executed on the data source

A generated Optimized Logical Plan is passed through a series of Spark strategies to produce one or more Physical plans

Spark uses cost based optimization (CBO) to select the best physical plan based on the data source (i.e. table sizes)

Plan Optimization & Execution - Physical Planning



Plan Optimization & Execution – Code Generation

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case Add(left, right) =>
    q"${compile(left)} + ${compile(right)}"
}
```

- This phase involves generating java bytecode to run on each machine
- Catalyst transforms a SQL tree into an abstract syntax tree (AST) for Scala code to eval expr and generate code
- 700LOC

Extensions

Data Sources

- must implement a createRelation function that takes a set of key-value params and returns a BaseRelation object.
- E.g. CSV, Avro, Parquet, JDBC

User-Defined Types (UDTs)

- Map user-defined types to structures composed of Catalyst's built-in types.

```
class PointUDT extends UserDefinedType[Point] {
  def dataType = StructType(Seq( // Our native structure
    StructField("x", DoubleType),
    StructField("y", DoubleType)
  ))
  def serialize(p: Point) = Row(p.x, p.y)
  def deserialize(r: Row) =
    Point(r.getDouble(0), r.getDouble(1))
}
```

Advanced Analytics Features

Schema Inference for Semi structured Data JSON

- Automatically infers schema from a set of records, in one pass or sample
- A tree of STRUCT types, each of which may contain atoms, arrays, or other STRUCTs.
- Find the most appropriate type for a field based on all data observed in that column. Determine array element types in the same way.
- Merge schemata of single records in one reduce operation.
- Same trick for Python typing

```
{
  "text": "This is a tweet about #Spark",
  "tags": ["#Spark"],
  "loc": {"lat": 45.1, "long": 90}
}

{
  "text": "This is another tweet",
  "tags": [],
  "loc": {"lat": 39, "long": 88.5}
}

{
  "text": "A #tweet without #location",
  "tags": ["#tweet", "#location"]
}
```

```
text STRING NOT NULL,
tags ARRAY<STRING NOT NULL> NOT NULL,
loc STRUCT<lat FLOAT NOT NULL, long FLOAT NOT NULL>
```

Demo Spark SQL

Demo

THANK YOU

Prof. J.Ruby Dinakar
Dept. of Computer Science and Engineering
rubydinakar@pes.edu

BIG DATA

Big Data Algorithm Complexity

K V Subramaniam
Computer Science and Engineering

BIG DATA

Overview of lecture

- Motivation – Algorithm complexity
- Communication Cost Complexity Model
- 3 Way joins with the communication cost complexity
- Key parameters
- Similarity join - analysis

Motivation – Algorithm Complexity

Why Study complexity?

- So far, we have looked at MapReduce algorithms
- However, for a particular problem, there could be many algorithms
- Which algorithm should we choose?
- This is why we study complexity of MapReduce
- We will actually study complexity of workflow systems
  - Generalization of MapReduce
  - Many important Big Data systems are workflow systems

Exercise

- Consider the following two problems
  - Matrix multiplication
  - Database query
- What would be the complexity of these algorithms when executing on a single node?
- What does complexity depend on?

Solution: Class Exercise

- Matrix multiplication
  - Expressed in terms of the bound on total #computations performed
- Database query
  - Complexity depends on disk read

Communication Cost Complexity

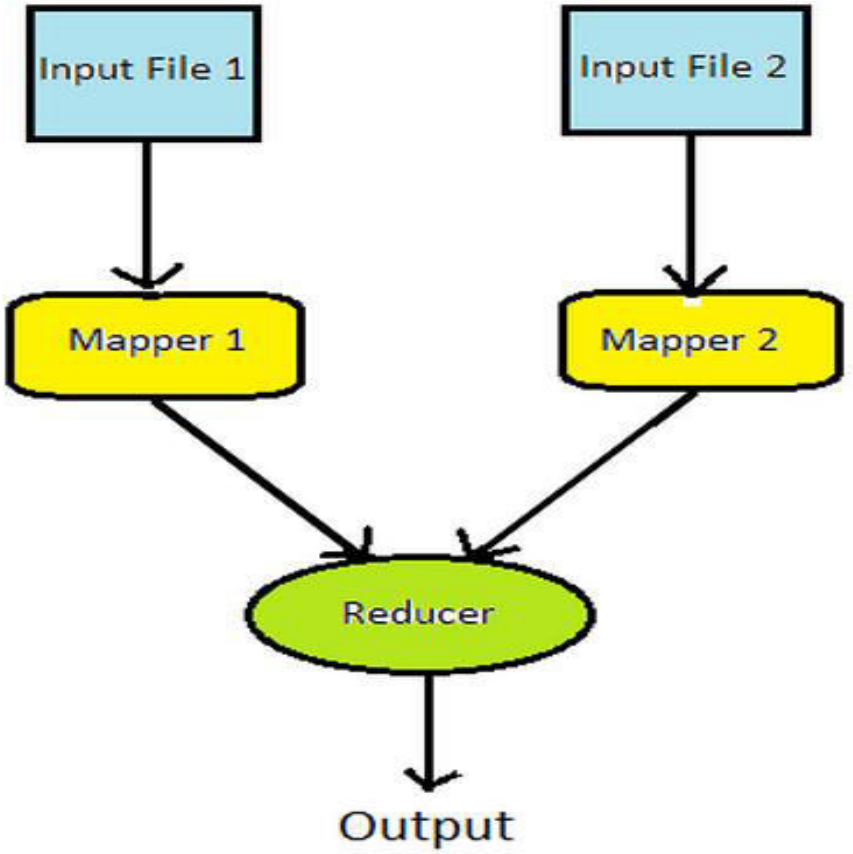
- Communication cost: size of input
- Why communication cost?
  - Algorithm tends to be linear in data
  - Network speed << CPU speed
  - Disk speed << CPU speed
  - Major time could be communication time

Communication Cost Complexity

- Why only input size?
  - Output is input to some other task
  - Final output is generally small by aggregation
    - Otherwise not human-readable

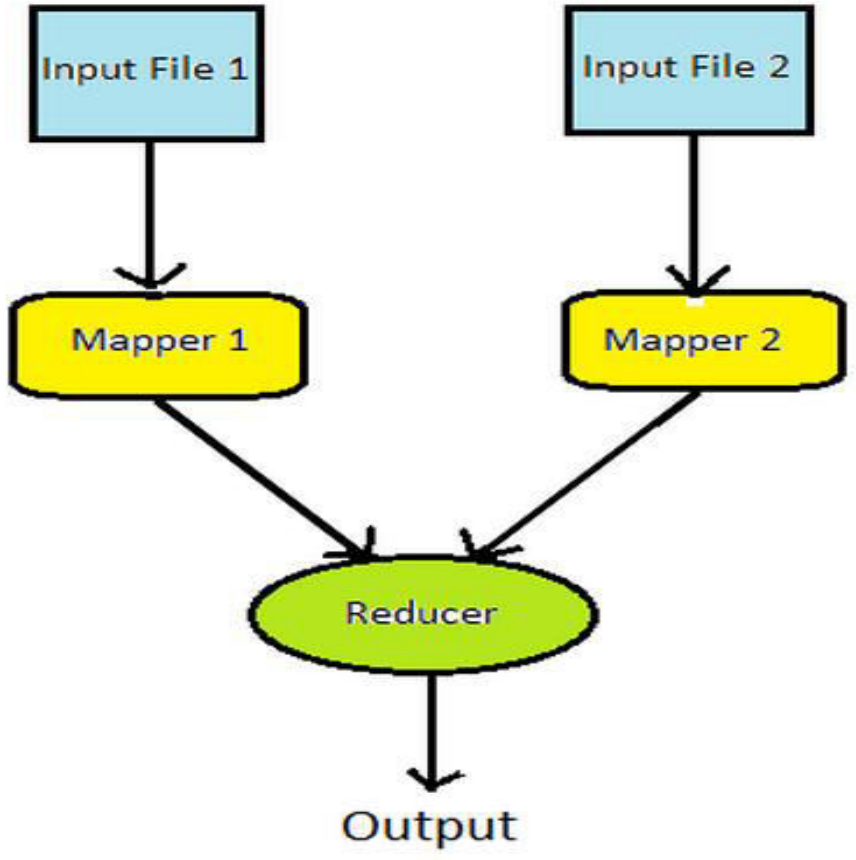
Natural Join: join R, S

- Mapper input complexity = *r+s*
  - Read data from disk
- Reducer input complexity = *r+s*
  - Network reads
- Total Complexity: *O(r+s)*



Exercise – Join complexity

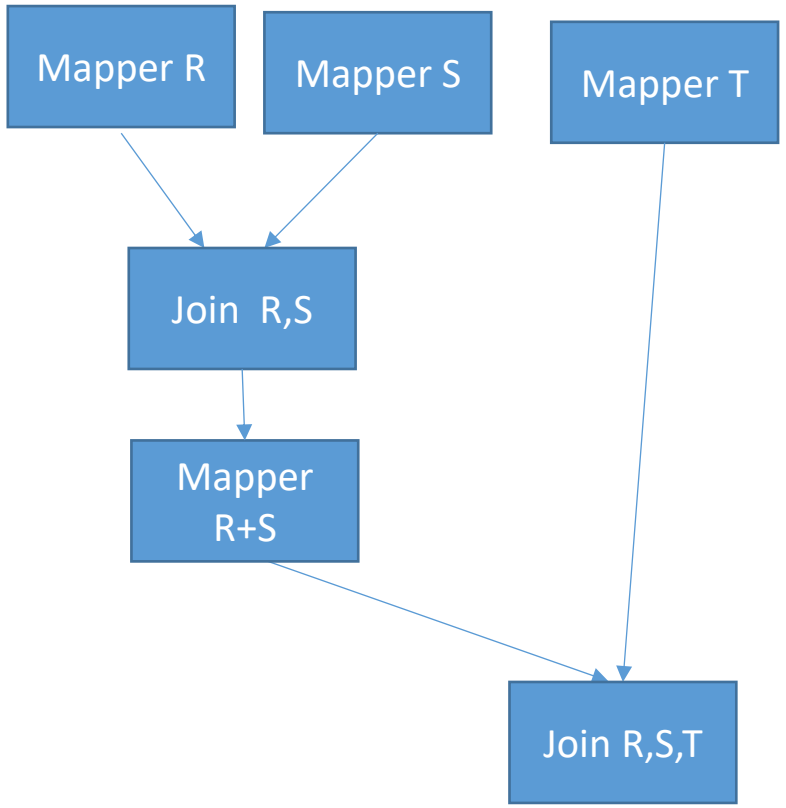
- Consider three relations R, S and T
- How will you perform a join using map reduce across the three?
- Estimate the complexity



3 way join complexity

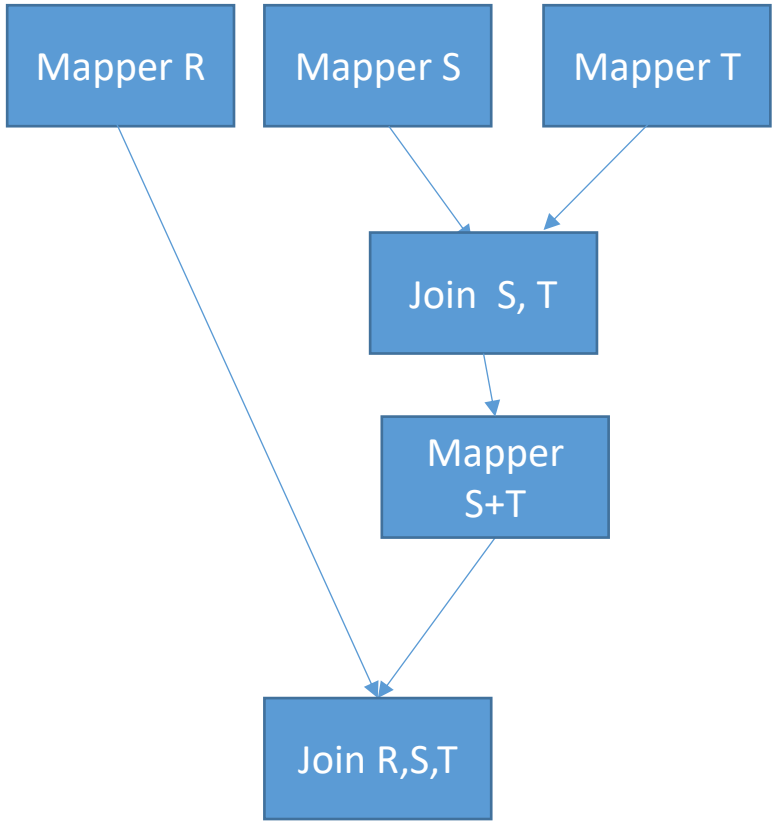
3-way Join: R, S, T

- 2 MapReduce phases
- Case 1: Join *R,S* and then join *T*
  - Input to Mapper 1: *r*
  - Input to Mapper 2: *s*
  - Input to Reducer 1: *r+s*
  - Let *p* be the probability of match between *r,s*
  - Input to Mapper 3: *prs*
  - Input to Mapper 4: *t*
  - Input to Reducer 2: *t+prs*
- Total Complexity: *O(r+s+t+prs)*



…3 Way Join R, S, T

- Case 1: Join *S,T* and then join *R*
  - Input to Mapper 1: *s*
  - Input to Mapper 2: *t*
  - Input to Reducer 1: *s+t*
  - Let *q* be the probability of match between *s,t*
  - Input to Mapper 3: *qst*
  - Input to Mapper 4: *r*
  - Input to Reducer 2: *r+qst*
- Total Complexity: *O(r+s+t+qst)*
  - Depends upon join order
  - If *p~q,* first join could be whichever of *rs, st, rt* is the smallest



Key performance parameters

Wall Clock Time

- Can make communication cost very low by executing all tasks on single CPU
- However, program may run slowly
- Need to consider *wall clock time*
  - Time taken for entire job to finish
- Dividing jobs could increase communication but reduce wall clock tme
  - Need to trade off communication time, wall clock time

Parameters: Wall Clock Time vs Communication cost

- Reducer size *q*
  - Max # of values that can have the same key
    - Not the number of reducers
  - If *q* is small, there can be more reducers
    - Suppose the number of Map outputs is *T*
    - Max number of reducers = *T/q*
    - This will reduce the wall clock time
    - But increase the communication cost
- Replication rate *r*
  - *r = (#key value pairs output by Mapper) / (# input records to Mapper)*
  - Average communication cost from Map tasks to Reduce tasks

Similarity Joins

Wall Clock Example: Similarity Join Between Images

- Assume we have a database of 1 million images
- Each image is 1 MB
- Total DB size = 1 TB
- Assume there is a function *s(x,y)* which determines how similar two images *x,y* are
  - *s(x,y) = s(y,x)*
- Problem: output all pairs *x,y* such that *s(x,y) > t*

Naïve Algorithm for example

- Assume each image P_i has an index i
- Mapper
  - Reads in (i,P_i)
  - Generates all pairs $(\{i,j\},\{P_i,P_j\})$
- Reducer
  - Reads $(\{i,j\},\{P_i,P_j\})$
  - Computes $s(P_i,Pj)$

Exercise

- What is the
  - Communication cost of the naïve algorithm?
  - Parallelism of the naïve algorithm?
  - What is the replication rate for the algorithm?

Exercise - Solution

- What is the
  - Communication cost of the naïve algorithm?
  - Parallelism of the naïve algorithm?
- Algorithm doesn't work
  - Data to be transmitted = 1,000,000 x 999,999 x 1,000,000 bytes = 10^{18}
  - Communication cost is ~n^2 where n is the number of images (extremely high)
  - However, potential parallelism is very high
    - Reducer size is very high → since 10^{18} key value pairs are generated, each can be processed in parallel by a different reducer
    - Replication rate -> each piece of data copied 999,999 times, so huge amount of data copied.

The other extreme

- Do everything on one node
- Mapper
  - Reads in (i, P_i)
  - Generates all pairs $(\{i,j\},\{P_i, P_j\})$
- Reducer (runs on same node as mapper)
  - Reads $(\{i,j\},\{P_i, P_j\})$
  - Computes $s(P_i, Pj)$
- No communication cost
- Very low parallelism (wall clock time high)

Summary

Send one pair to each reducer

- High communication cost (bad)
- High parallelism (good)

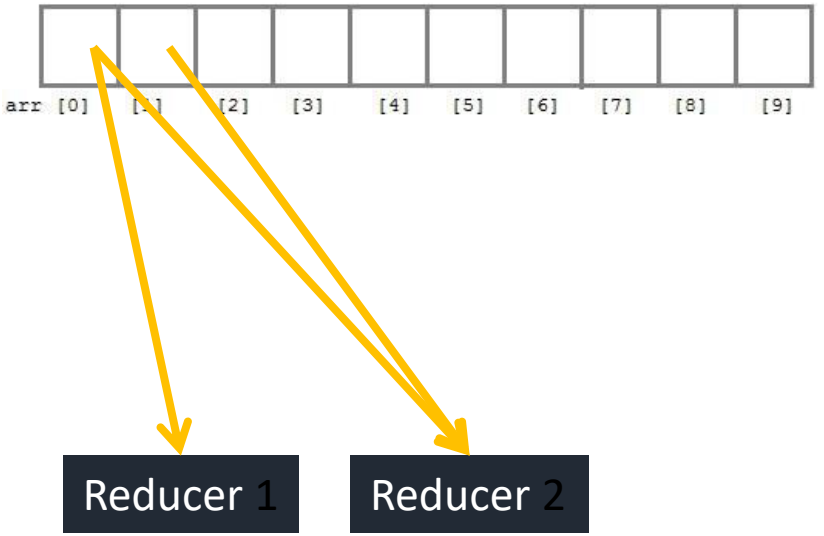
Do everything on one node

- Low parallelism (bad)
- Low communication cost (good)

Can we get something in between?

Group-based algorithm

- Overview
  - Group images
  - Read in two groups of images in a single reducer
  - Store them in memory
  - Compare all pairs from those groups
  - Output results



Example: 100 Groups

- Suppose the groups are G0, … G99
- Group G0 is sent to nodes 0, 1, ..., 98
  - Why?
- Group G0 has to be compared with 99 other groups
- Group G1 is sent to?
  - ~~0, 1, …, 98~~
- Group G1 is sent to 0, 99, 100, ...196 (0+98 other nodes)
- Group G2 is sent to?
- Group G2 is sent to 1, 99, 197, 198, ... 293 (1, 99 +97 other nodes)
- Group G3 is sent to 2, 100, 197, 294, ...389 (2, 100, 197, +96 other nodes)

In General

- If there are *g* groups
- The number of images per group *m=n/g*
- Each group is sent to *g-1* servers
- Total number of messages = *g(g-1)*
- **Total data = *mg(g-1) = n(g-1) ~ ng***
- **Parallelism = no of nodes = *(g-1) + (g-2) + (g-3) + … = g(g-1)/2 =O(g^2)***
- Another way = no of nodes = $^nC_2$ *=g(g-1)/2*

Exercise

- Suppose we have groups of 100
  - How many groups are there?
  - How many nodes is each group sent to?
- What is the
    - Communication cost of the algorithm?
    - Parallelism of the algorithm?

…Group Based Algorithm…

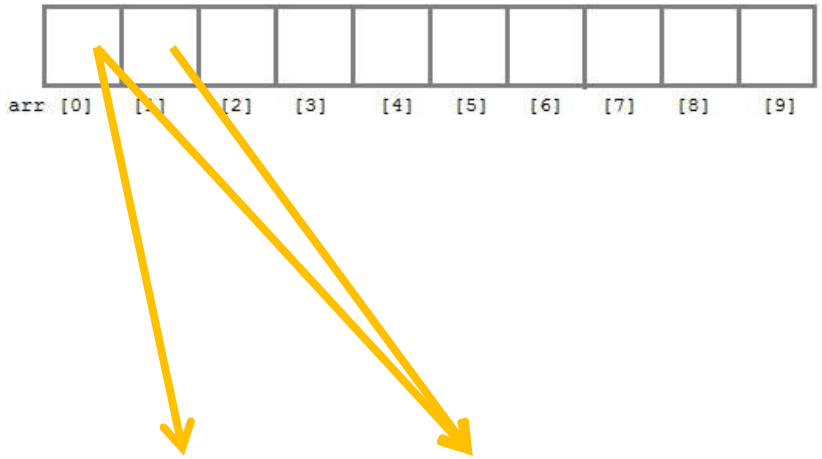
- Naïve: approximately 1,000,000x1,000,000 images = 10^{12} images, parallelism = 10^{12}
- Example: group images into groups of100
  - Communication cost: 1,000,000 images x 100 groups = 10^8 images
  - Parallelism = 10^4 (100 groups compared with 99 other groups ~ 10^4)
- Example: group images into groups of1000
  - Communication cost = 1,000,000 x 1000 = 10^9 images
  - Parallelism = 10^6
- Trade-off: increasing group size
  - Increases communication complexity (more reads)
  - Reduces wall clock time (increases parallelism)

…Group Based Algorithm

- On next slide
- Note
  - Group-based algorithm is discussed in book as if mapper sends pictures to reducer
  - In previous slides, we have discussed as if reducer reads in pictures from disk
  - From communication cost complexity both are the same
  - Performance: probably better to read from disk

Group Based algorithm

- Overview
  - Group images
  - Read in two groups of images in a single reducer
  - Store them in memory
  - Compare all pairs from those groups
  - Output results
- Complexity
  - Communication cost ~*ng* where *g* is the number of groups
  - Can still get good parallelism



Grouping images..

- Group images into *g* groups
- Mapper
  - Input: (i, P_i)
  - Find *u* = group to which image *i* belongs
  - Output *g-1* key-value pairs $(\{u,v\}, i, P_i)$ for all *v != u*
- Reducer:
  - There is one reducer for each unique key {*u,v*}
  - Use the input to store images for groups *u,v*
  - Compare all pairs of images in groups *u,v*
  - If *v=u+1,* then compare all pairs of images in group *u*

Actual Values

- If group size is 1000
- Need ~2GB to store 2000 images in memory
- Need ~500,000 reducers
- If we have a 10,000 node cluster
  - Can do computation in 50 passes
  - Speedup of 10,000

Summary

- There could be many MapReduce algorithms to implement a particular functionality
  - Matrix multiplication
  - Multi-way joins
- There are two factors to consider when selecting an algorithm
  - Communication complexity: volume of data that is input to the different phases of the algorithm
  - Wall clock time: Total elapsed time for algorithm
    - Depends upon parallelism
  - Frequently there is a trade-off between these factors
    - Group size

References

- Mining of Massive Datasets
  - Rajaraman et. Al.
  - Chapter 2.4-2.5

THANK YOU

K V Subramaniam, Usha Devi

Dept. of Computer Science and Engineering

subramaniamkv@pes.edu

ushadevibg@pes.edu