




In-Memory Computing Spark

National Tsing Hua University
2018, Fall Semester



Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
From UC Berkeley

SPARK: LOW LATENCY, MASSIVELY PARALLEL PROCESSING FRAMEWORK



Outline

- Motivation: Iterative and Interactive Computation
- Scala: Functional Programming
- Spark: Data & Computation Management

Motivation & Objectives

■ Motivation:

- Data reuse is frequent in iterative and interactive data processing
- MapReduce only support acyclic workflow

■ Objectives:

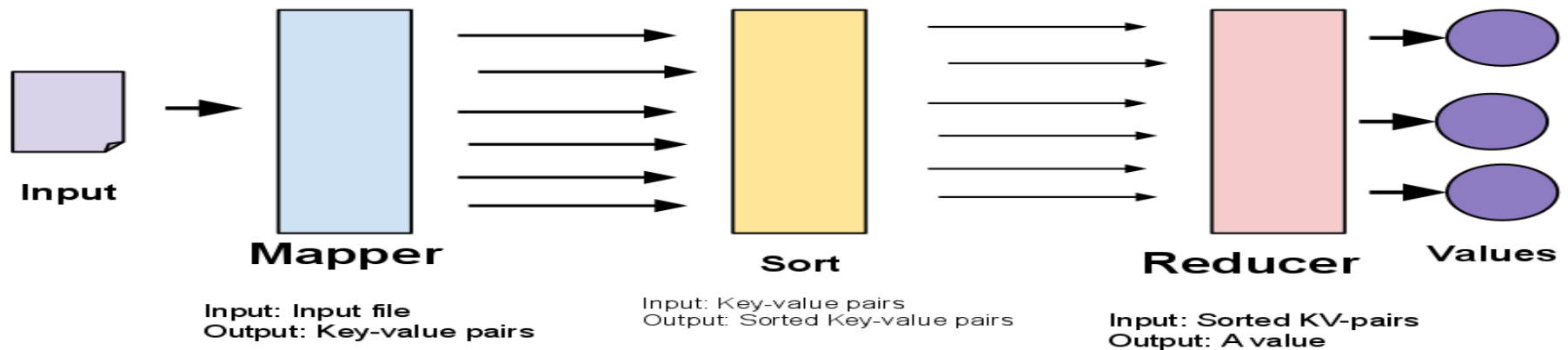
- Utilize **DSM** (Distributed Shared Memory) in data processing to enable **in-memory computing**
- Allow users to explicitly **cache dataset in memory across machines** and reuse it in multiple MapReduce-like parallel operations.
- Retain the **scalability and fault tolerance** property like MapReduce

Approach Overview

- Spark introduces an data abstraction called **Resilient Distributed Datasets (RDDs)**:
 - RDD is a *read-only collection* of objects partitioned across a set of machines
 - RDD **can be rebuilt** if a partition is lost using the “*lineage*” technique
- Spark is integrated into a general programming language called “**scala**”
 - Pure-bred **O.O language**: every variable/dataset is an object and every operation is a method-call
 - Seamless **Java interpreter**
 - Programmer specify operations to **transform dataset**
 - Operations are **parallelized and executed by Spark**

Limitation of MapReduce

- Simple but limited programming model



- Can only apply two computation functions in a job: Map&Reduce
 - ➔ More complex work must use **multiple** jobs
- The input and output of a job must store into a FS
 - ➔ FS(disk) is the only device to provide **data persistency**

Iterative Data Processing

■ Definition:

- A mathematical procedure that generates a sequence of improving **approximate solutions** for a class of problems
- The procedure iterates until converge or reach some termination criteria

■ Property of computation

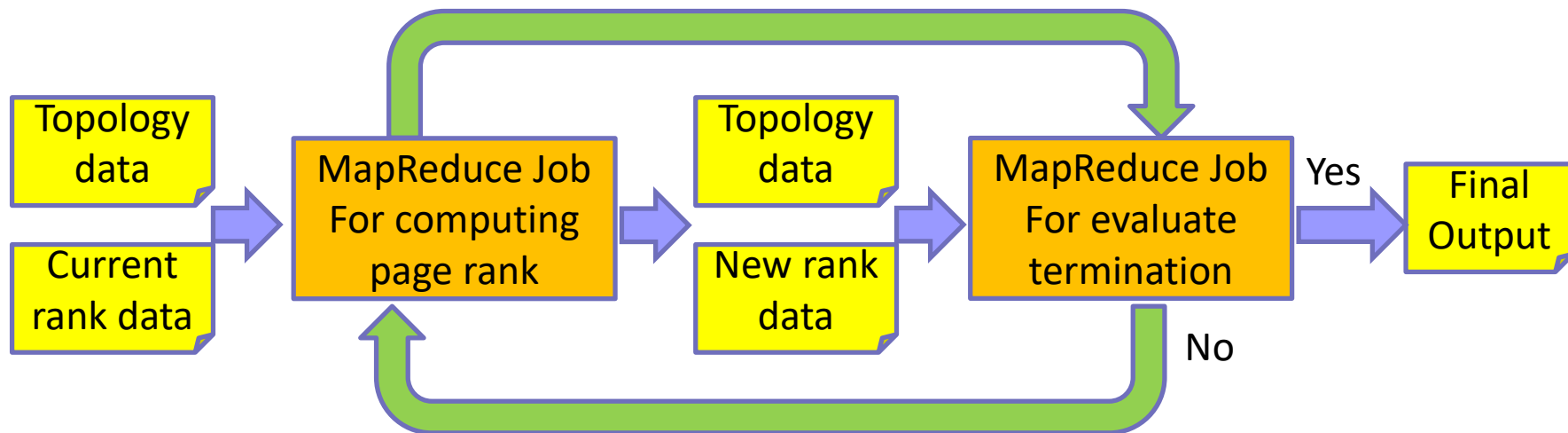
- Termination criteria must be evaluated after each iteration
- The output from an iteration will be the input of the next iteration
- Invariant data must re-load and re-processing in the loop

■ Applications:

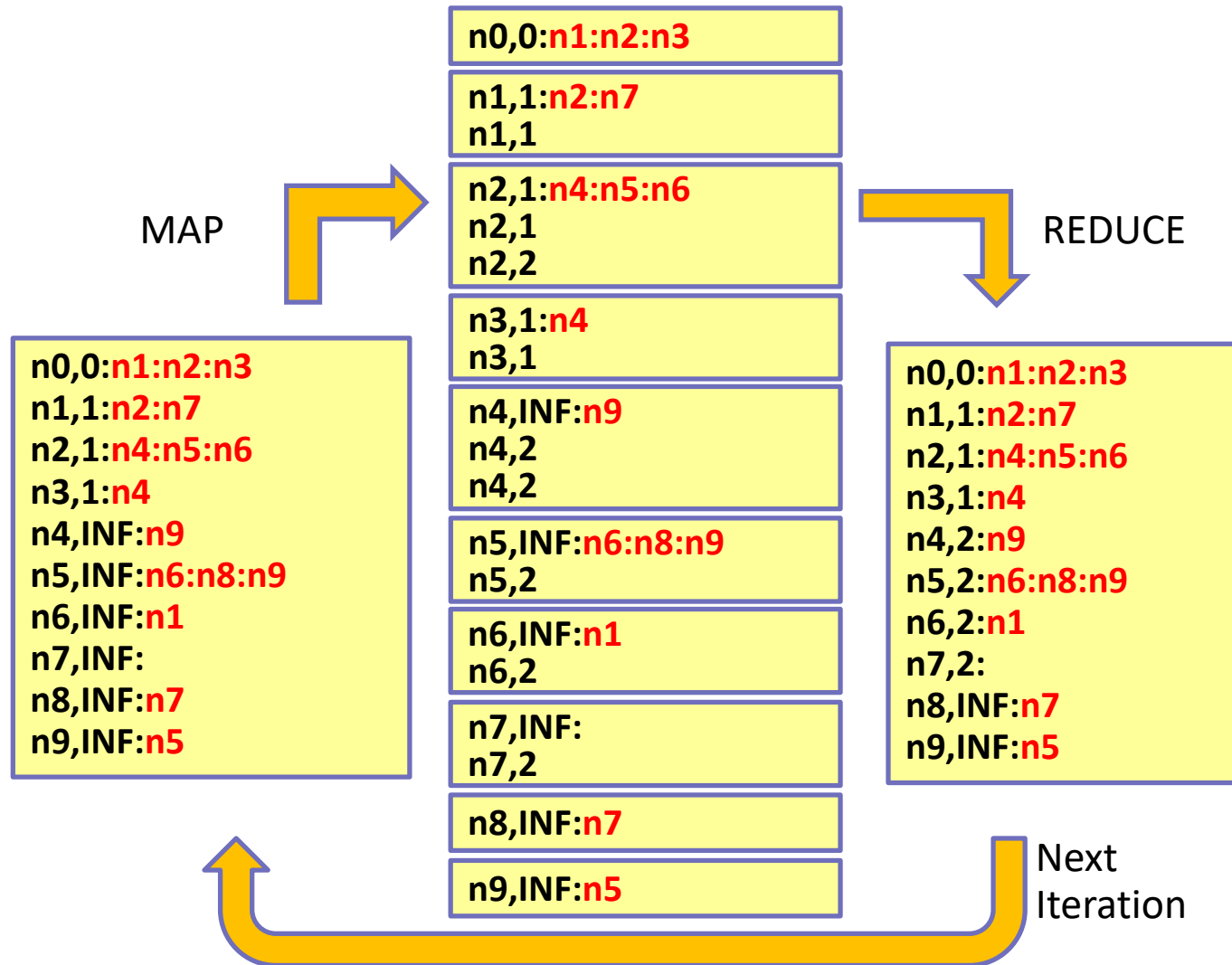
- Machine learning algorithm: K-mean
- Graph algorithm: Page-rank
- Approximation algorithm

Iterative Method in MapReduce

- Each iteration is submitted as an independent job
 - hard to ask the scheduler to manage and guarantee the performance of the whole application
- Termination criteria is evaluated after each iteration
- Data is written and read out disk after each iteration
- The invariant data is repeatedly store/load/transfer

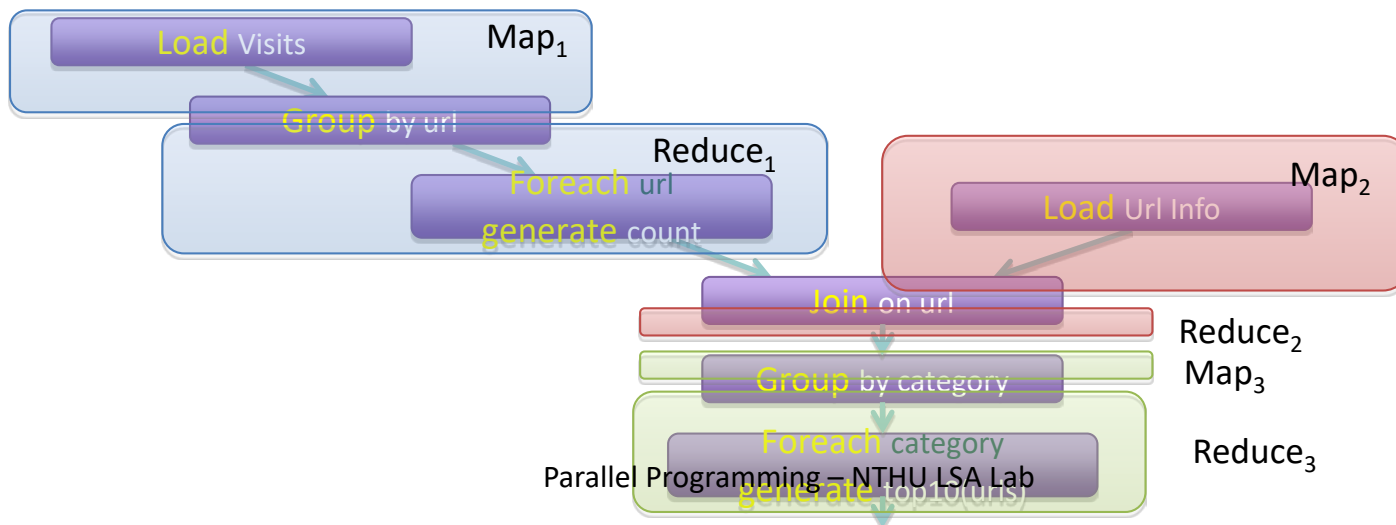


Iterative Method in MapReduce



Interactive Processing

- Definition: computation involving the exchange of information between a user and the computer
- Property:
 - Require short response time → disk is too slow
 - Repeatedly process on the same set of data → redundant I/O
 - Complex data-flow → can be specified by one job
- Example:
 - Ad-Hoc query processing, ex: Pig, Hive

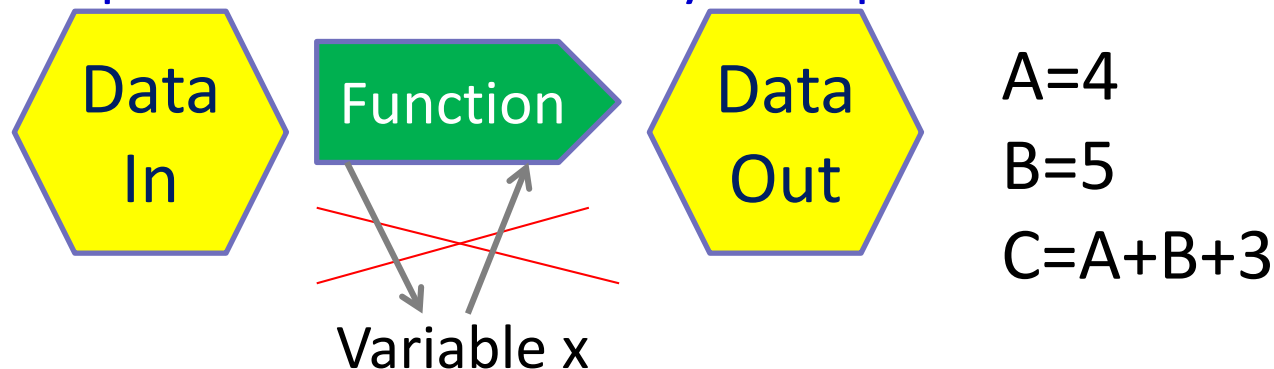


Functional Programming

- Immutable data + function = functional programming
 - Theoretical foundation based on Alonzo Church's **Lambda Calculus**.
 - A style of building the structure and elements of computer programs—that treats **computation as the evaluation of mathematical functions** and **avoids changing-state and mutable data**
 - It is **referential transparency**: the output value of a function **depends only on the arguments** that are input to the function.
- Languages:
 - Haskell, Erlang, Scala, Lisp, Scheme, F#

Referential Transparency

- An expression is said to be referentially transparent if it can be replaced with its value without changing the behavior of a program (in other words, yielding **a program that has the same effects and output on the same input**).
- While in mathematics all function applications are referentially transparent, in programming this is not always the case.
- If all functions involved in the expression are **pure functions**, then the expression is referentially transparent.



Imperative vs. Functional Programming

- **imperative (procedure) programming** is a programming paradigm that describes computation in terms of **statements that change a program state**.

Procedure programming	Functional programming
Everything is done in a specific order	Order of evaluation is usually undefined
Execution of a routine may have side effects	<ul style="list-style-type: none">• Must be stateless. i.e. No operation can have side effects• Always returns the same output for a given input
Tends to emphasize implementing solutions in a linear fashion	Good fit for parallel execution

About SCALA

- Scala = “Scalable Language”
- High-level language for the JVM
 - Combine **object oriented** and **functional programming** with a powerful static type system and expressive syntax
- Interoperates with Java
 - Can use any Java class
 - Can be called from Java code
- Upsurge in adoption since 2010 to handle massive parallel data processing problem

<http://www.scala-lang.org/>

https://twitter.github.io/scala_school/

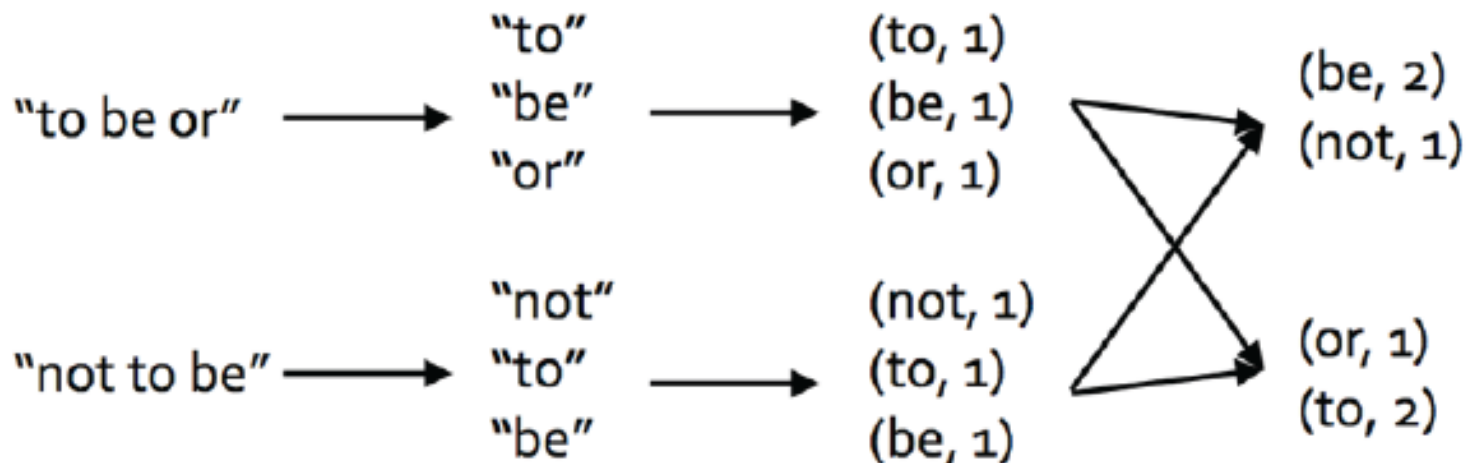
Functional methods on collections

- There are a lot of methods on Scala collections, just [google Scala Seq](http://www.scala-lang.org/api/2.10.4/index.html#scala.collection.Seq) or <http://www.scala-lang.org/api/2.10.4/index.html#scala.collection.Seq>

Method on Seq[T]	Explanation
map(f: T=>U): Seq[U]	Each element is result of f
flatMap(f: T=>Seq[U]): Seq[U]	One to many mapping
filter(f: T=>Boolean): Seq[T]	Keep elements passing f
exists(f: T=>Boolean): Boolean	True if one element passes f
forall(f: T=>Boolean): Boolean	True if all elements pass f
reduce(f: (T,T) => T): T	Merge elements using f
groupBy(f: T=>K): Map[K,List[T]]	Group elements by f
sortBy(f: T=>K): Seq[T]	Sort elements

Word Count Example

- `val lines = sc.textFile("hamlet.txt")!`
- `val counts = lines.flatMap(line => line.split(" ")).
map(word => (word, 1)).
reduceByKey(_ + _)`



Spark: RDDs

Resilient distributed datasets (RDDs)

- **Immutable, partitioned collections** of objects
- **Created through parallel *transformations*** (map, filter, groupBy, join, ...) on data in stable storage
- Can be ***cached*** for efficient reuse
- RDDs are lazy and ephemeral. That is, partitions of a dataset are **materialized (i.e. computed)** on demand when they are used in a parallel operation

Actions on RDDs

- Count, reduce, collect, save, ...

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

```
cachedMsgs. filter(_. contains("foo")). count
cachedMsgs. filter(_. contains("bar")). count
. . .
```

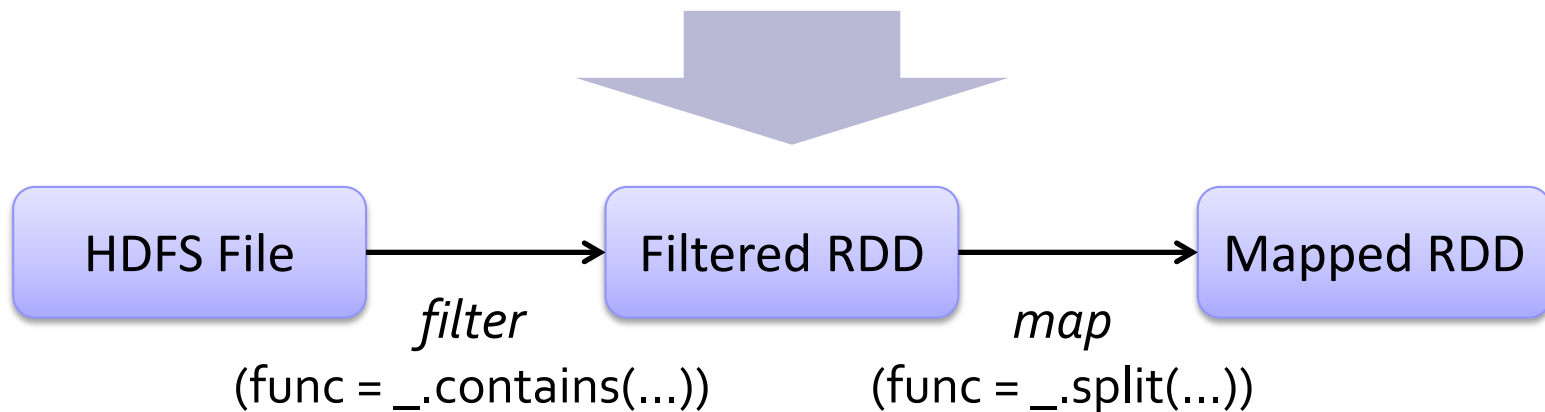
The diagram illustrates the Spark architecture components and their interactions:

- Driver:** The central component that coordinates the execution. It sends **tasks** to the workers and receives **results** back.
- Workers:** Multiple worker nodes that execute tasks. Each worker has its own **Cache** (Cache 1, Cache 2, Cache 3) and stores data in **Blocks** (Block 1, Block 2, Block 3).
- Transformed RDD:** A callout points to the Driver, indicating that the RDD is transformed into tasks.
- Action:** A callout points to the Driver, indicating that the action is executed on the worker.

RDD Fault Tolerance

RDDs maintain *lineage* information that can be used to reconstruct lost partitions

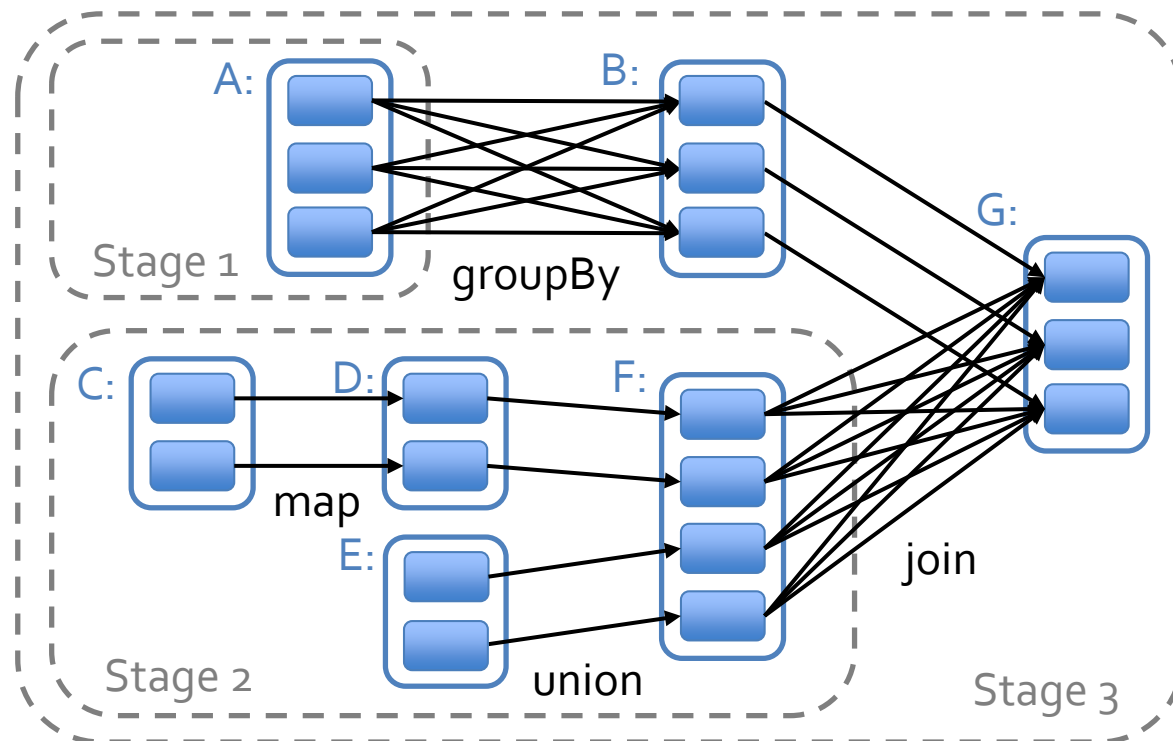
Ex: `messages = textFile(...).filter(_.startsWith("ERROR")).map(_.split(' \t')(2))`



*If lineage is too long, it could cause stack overflow (out of memory) problem during execution

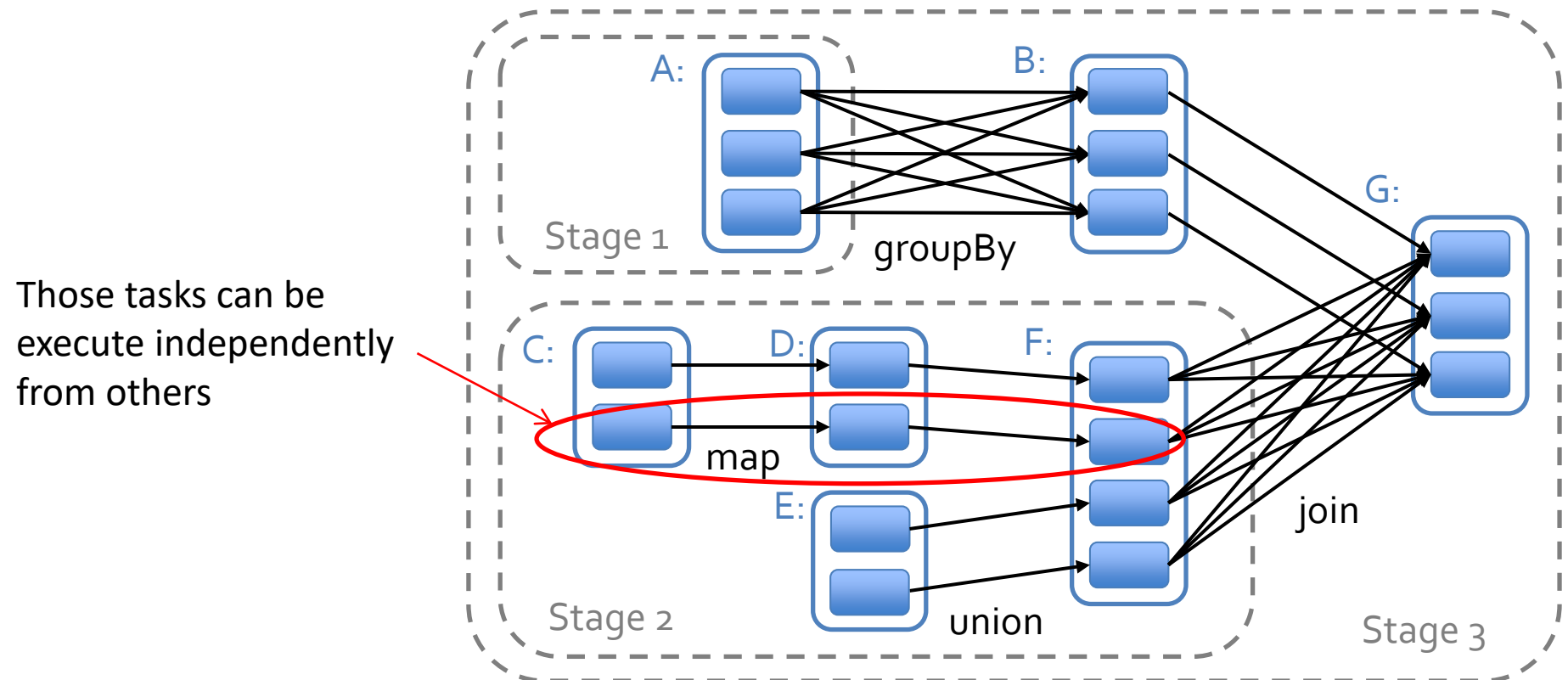
Spark Scheduler

- Whenever a user runs an action (e.g., count or save) on an RDD, the scheduler examines that RDD's lineage graph to build a DAG of stages to execute



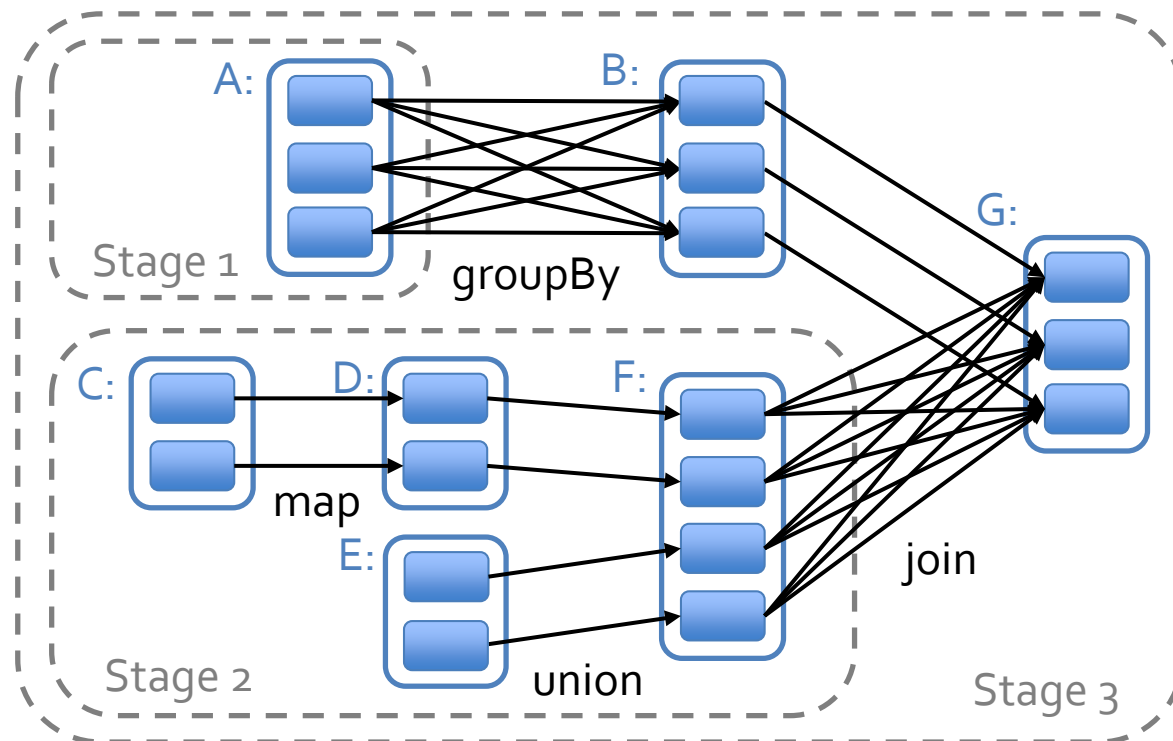
Spark Scheduler

- Each stage contains as many pipelined transformations with **narrow dependencies** as possible



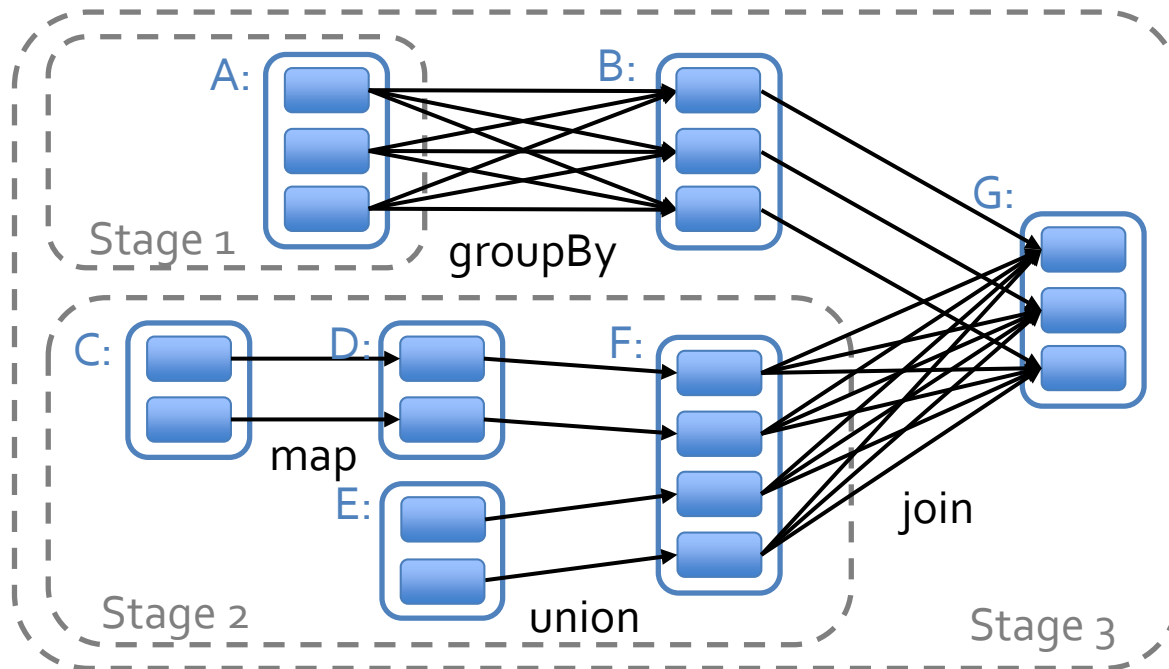
Spark Scheduler

- The scheduler assigns tasks to machines based on data **locality of the cached data (not the file in disks)**



Spark Scheduler

- If a task fails, it is re-run on another node as long as its stage's parents are still available.



Performance Comparison

- HadoopBinMem: A Hadoop deployment that **converts the input data into a low-overhead binary format** in the first iteration to eliminate text parsing in later ones, and **stores it in an in-memory HDFS instance**.
- Spark beats HadoopBinMem by 20X
 - Overhead of HDFS
 - Deserialization cost to convert binary record to usable in-memory Java objects

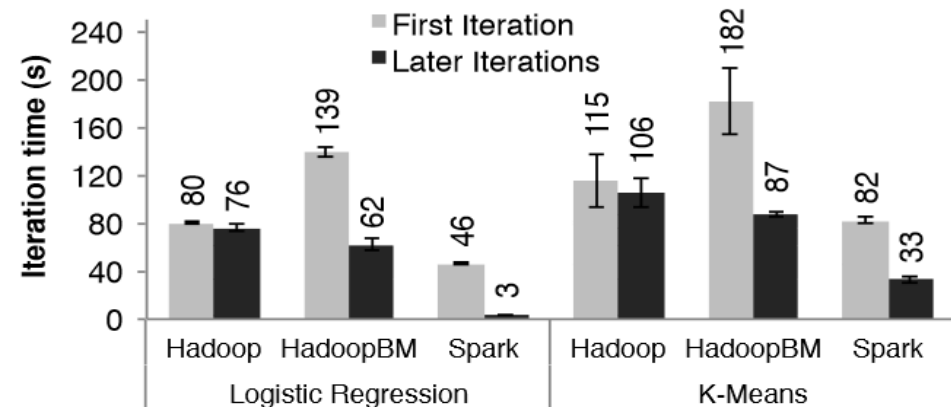


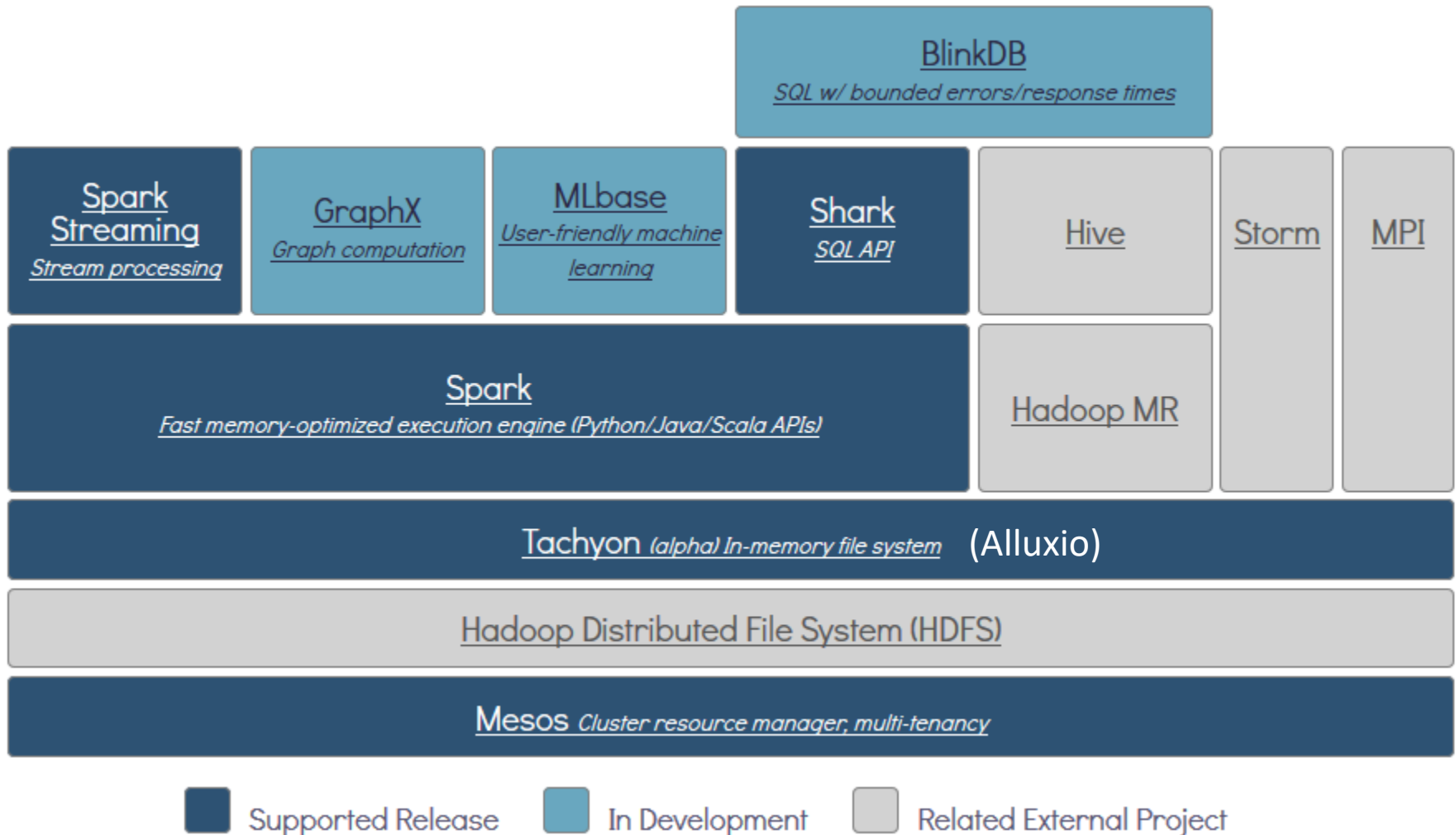
Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

Frameworks Built on Spark

- Pregel on Spark (Bagel)
 - Google message passing model for graph computation
 - 200 lines of code
- Hive on Spark (Shark)
 - 3000 lines of code
 - Compatible with Apache Hive
 - ML operators in Scala
- Spark Streaming
 - Small batch streaming
- SparkNet
 - Neural network



BDAS: the Berkeley Data Analytics Stack



Summary of Parallel Computing

Single-Core Era

Enabled by:
Moore's Law
Voltage Scaling

Constraint by:
Power
Complexity

Assembly → C/C++ → Java ...

Heterogeneous Systems Era

Enabled by:
Abundant data
parallelism
Power efficient GPUs

Constraint by:
Programming
models
Comm. overhead

Shader → CUDA → OpenCL ...

Muti-Core Era

Enabled by:
Moore's Law
SMP

Constraint by:
Power
Parallel SW
Scalability

Pthread → OpenMP ...

Distributed System Era

Enabled by:
Networking

Constraint by:
Synchronization
Comm. overhead

MPI → MapReduce ...



Backup Slides

SCALA: Function Definition

```
def add(a:Int, b:Int): Int = a+ b
```

```
val m:Int =add(1,2)
```

```
Println(m)
```

■ Scala is a “statically typed language”

- We define “add” to be a function which accepts two parameters of type Int and return a value of type Int.
- “m” is defined as a variable of type Int.

SCALA: Function Definition


```
def func(a:Int, b:Int): Int = {  
    a + 1  
    b - 2  
    a*b  
}  
  
val p:Int fun(1,2)  
println(m)
```

- There is no explicit “return” statement! The value of the last expression in the body is automatically returned.

SCALA: Type Inference

```
def add(a:Int, b:Int) = a+ b  
val m =add(1,2)  
Println(m)
```

```
def add(a, b) = a+ b  
val m =add(1,2)  
Println(m)
```



Scala does NOT infer type
of function parameters

- The return type of the function and the type of variable “m” is not specified. Scala “**infers**” that automatically because it is a statically typed language.

SCALA: Expression Oriented Programming

```
val i = 3
```

```
val p = if(i > 0) -1 else -2
```

```
val q = if(true) "hello" else "world"
```

```
println(p)
```

```
println(q)
```

- Unlike languages like C/JAVA, almost **everything in Scala is an “expression” that returns a value!**
- Rather than programming with “statements”, we program with “expressions”

SCALA: Functions return functions

```
def fun():Int => Int = {  
    def sqr(x: Int): Int= x*x  
    sqr  
}  
val f = func();  
println(f(10));
```

- “def fun():Int => Int “ says “fun” is a function which does not take any argument and returns a function which maps an Int to an Int.

SCALA: Lazy val's

```
def hello() = {  
    println("hello")  
    10  
}  
lazy val a = hello()
```

- “hello” is NOT printed by the program because the expression which assigns a value to a “lazy” val is executed only when that lazy val is used somewhere in the code!

Basic Data Structures

■ List: List of elements

➤ `List(1, 1, 2) → {1, 1, 2}`

■ Set: Sets have no duplicates

➤ `Set(1, 1, 2) → {1, 2}`

■ Map(Tuple):

➤ Groups together simple logical collections of items

➤ Values have accessors that are named by their position and is 1-based rather than 0-based.

➤ E.g.: `val hostPort = ("localhost", 80)`

`hostPort._1 // localhost`

`hostPort._2 // 80`

SCALA: Data collections

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

list.map(x=>x + 2)                       // return new list (3,4,5)
list.map( __ + 2)                       // same

list.filter(x=> x%2 == 1)                // return new list (1,3)
list.filter( __ %2 == 1)                // same

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

- All of these leave the list unchanged as it is immutable