COMP9313: Big Data Management

Lecturer: Xin Cao

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

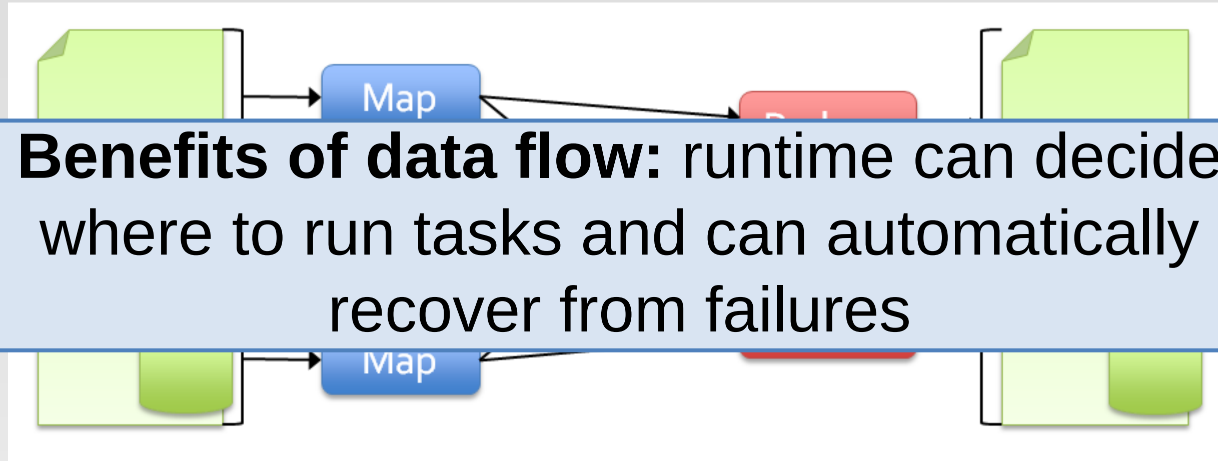
Chapter 4.1: Spark I

Part 1: Spark Introduction

Limitations of MapReduce

- MapReduce greatly simplified big data analysis on large, unreliable clusters. It is great at one-pass computation.
- But as soon as it got popular, users wanted more:
  - More **complex**, multi-pass analytics (e.g. ML, graph)
  - More **interactive** ad-hoc queries
  - More **real-time** stream processing
- All 3 need faster **data sharing** across parallel jobs
  - One reaction: specialized models for some of these apps, e.g.,
    - Pregel (graph processing)
    - Storm (stream processing)

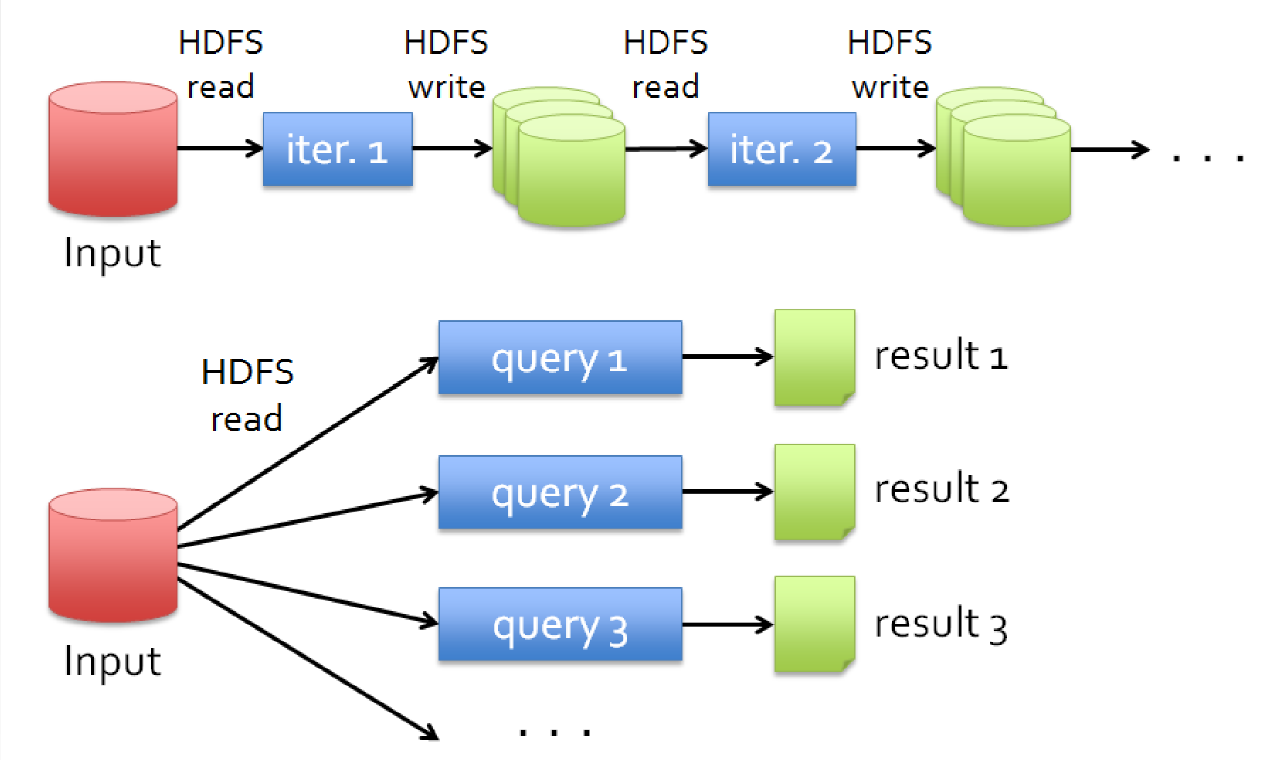
Limitations of MapReduce



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

- As a general programming model:
  - It is more suitable for one-pass computation on a large dataset
  - Hard to compose and nest multiple operations
  - No means of expressing iterative operations
- As implemented in Hadoop
  - All datasets are read from disk, then stored back on to disk
  - All data is (usually) triple-replicated for reliability
  - Not easy to write MapReduce programs using Java

Data Sharing in MapReduce



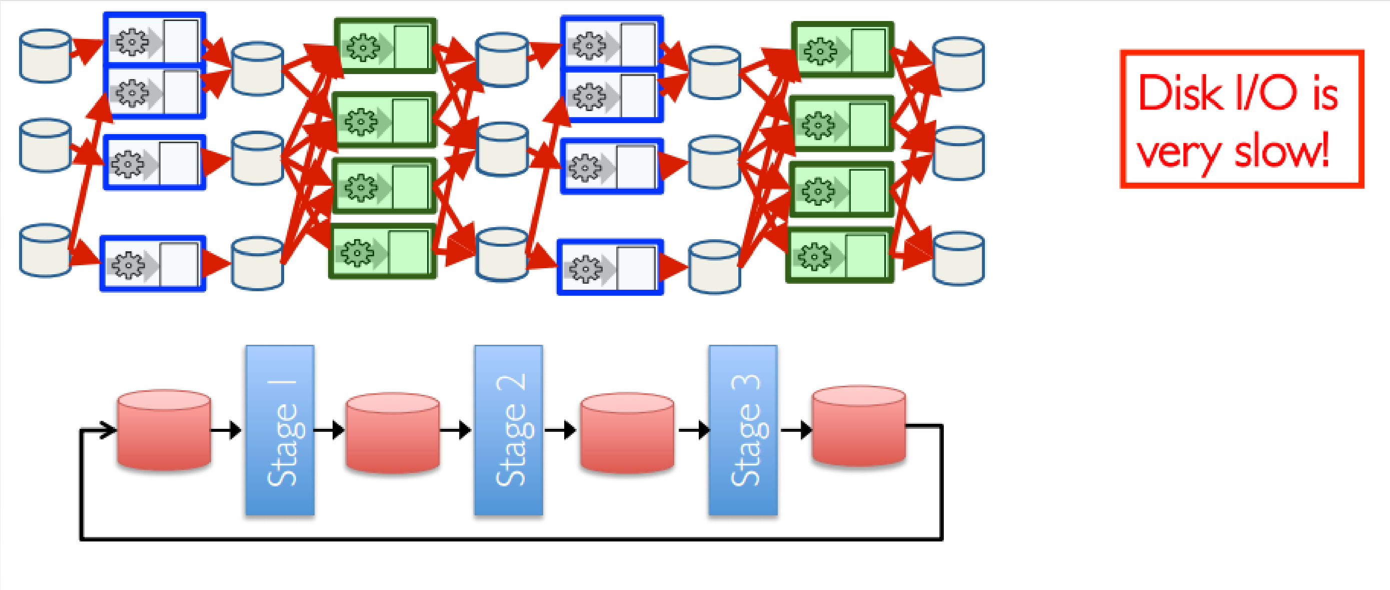
Slow due to replication, serialization, and disk IO

- Complex apps, streaming, and interactive queries all need one thing that MapReduce lacks:

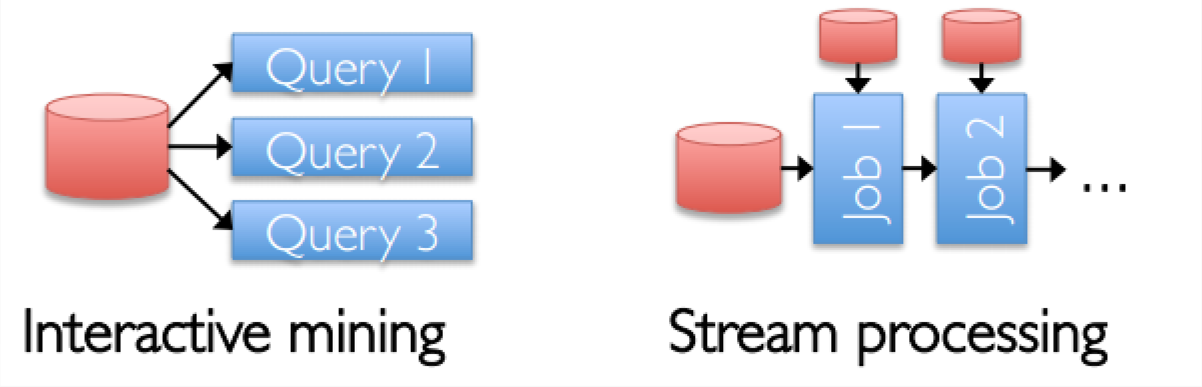
Efficient primitives for **data sharing**

Data Sharing in MapReduce

- Iterative jobs involve a lot of disk I/O for each repetition

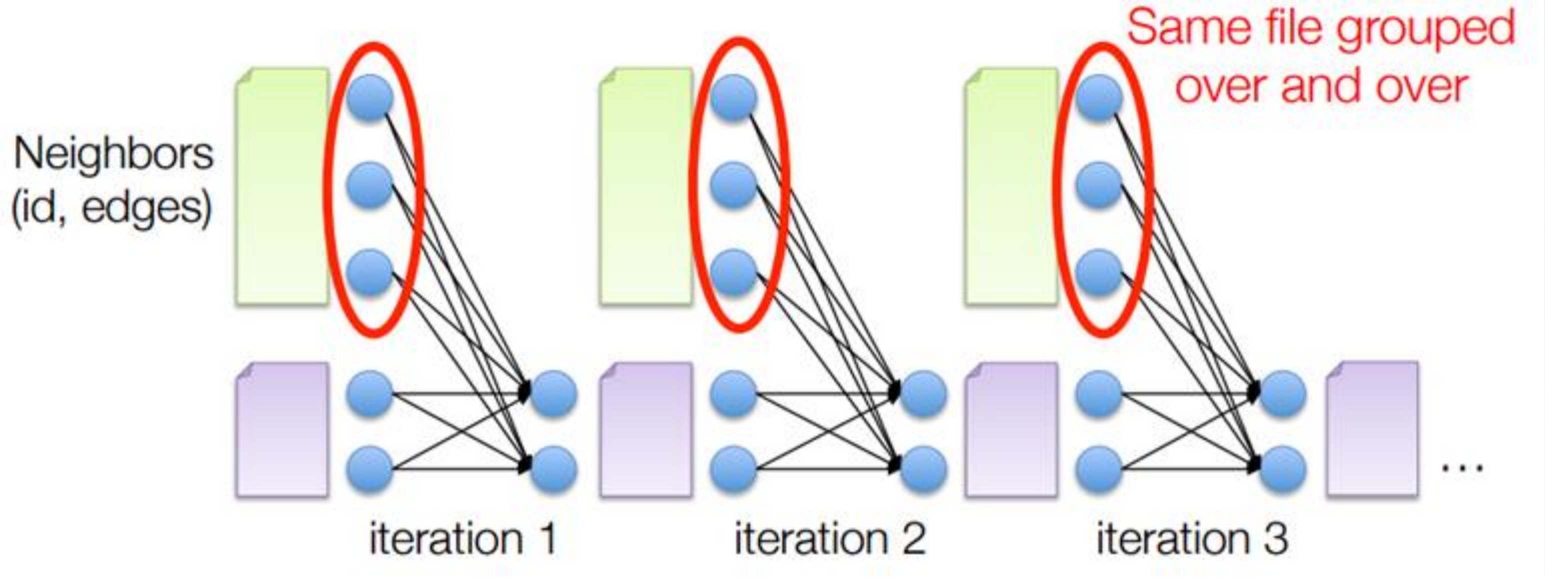


- Interactive queries and online processing involves lots of disk I/O



Example: Shortest Path

- Repeatedly send graph structure from mapper to reducer



Hardware for Big Data

Lots of hard drives

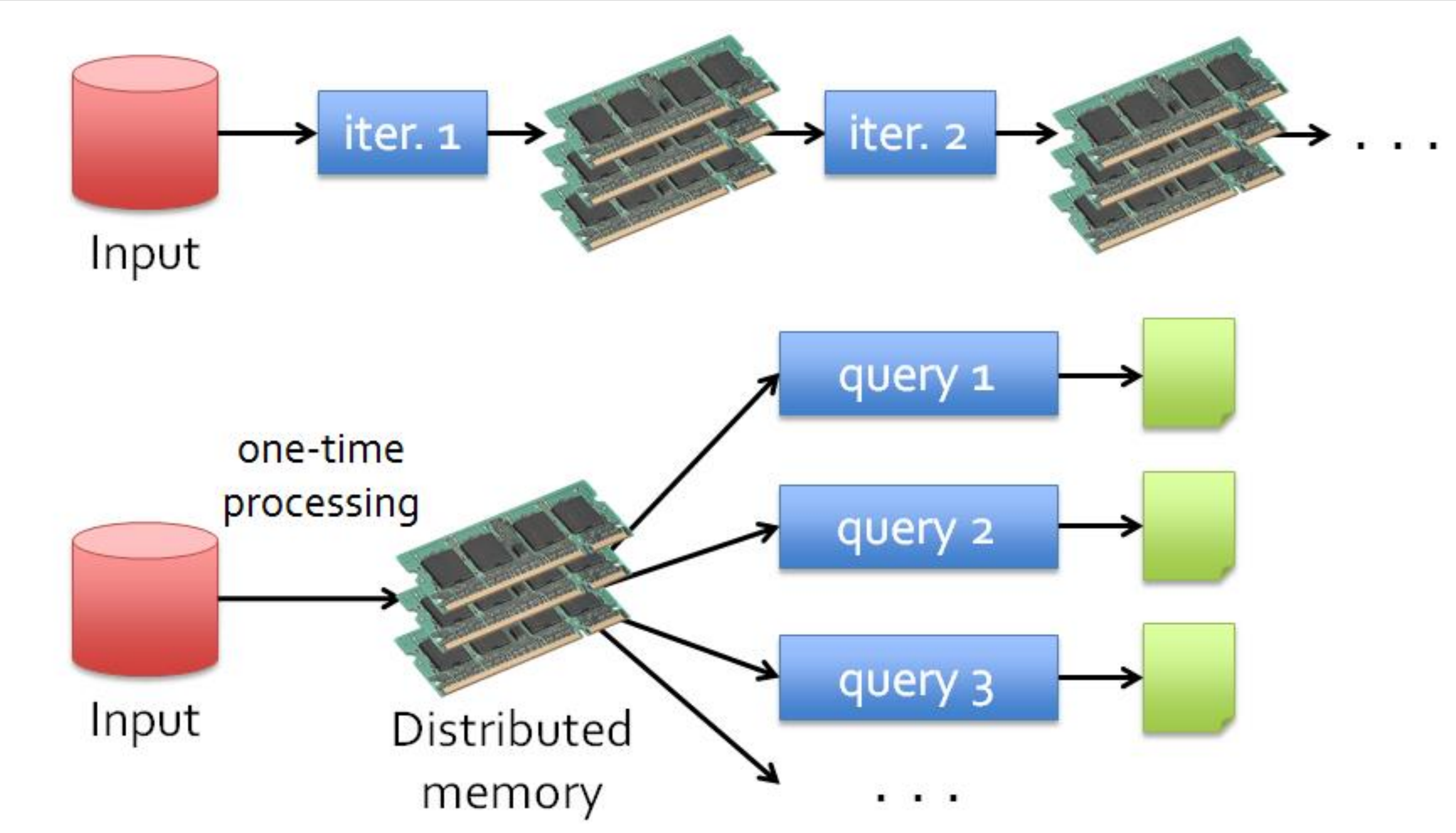
Lots of CPUs

And lots of memory!

Goals of Spark

- Keep more data in-memory to improve the performance!
- Extend the MapReduce model to better support two common classes of analytics apps:
  - Iterative algorithms (machine learning, graphs)
  - Interactive data mining
- Enhance programmability:
  - Integrate into Scala programming language
  - Allow interactive use from Scala interpreter

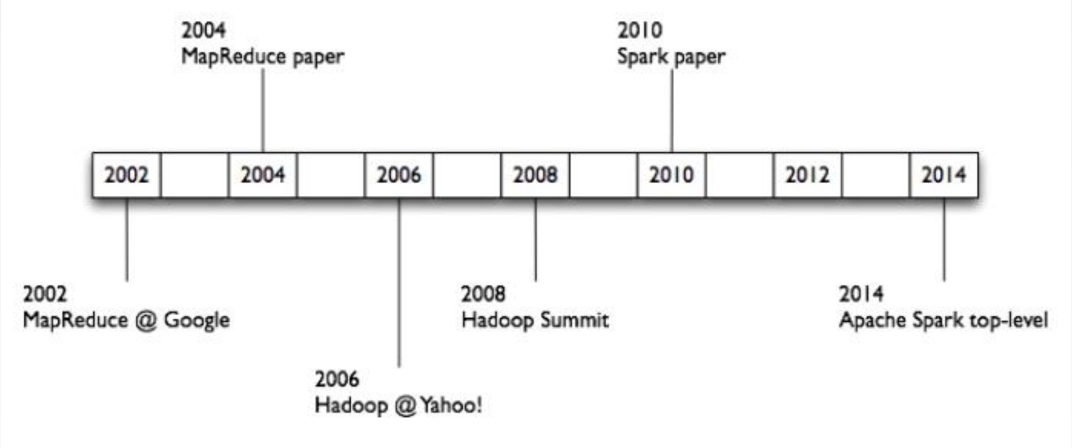
Data Sharing in Spark Using RDD



10-100× faster than network and disk

What is Spark

- One popular answer to "What's beyond MapReduce?"
- Open-source engine for large-scale distributed data processing
  - Supports generalized dataflows
  - Written in Scala, with bindings in Java, Python, and R
- Brief history:
  - Developed at UC Berkeley AMPLab in 2009
  - Open-sourced in 2010
  - Became top-level Apache project in February 2014
  - Commercial support provided by DataBricks



What is Spark

- Fast and expressive cluster computing system interoperable with Apache Hadoop
- Improves efficiency through:
  - **In-memory** computing primitives → Up to 100 × faster (10 × on disk)
  - General computation graphs
- Improves usability through:
  - Rich APIs in Scala, Java, Python
  - Interactive shell → Often 5 × less code

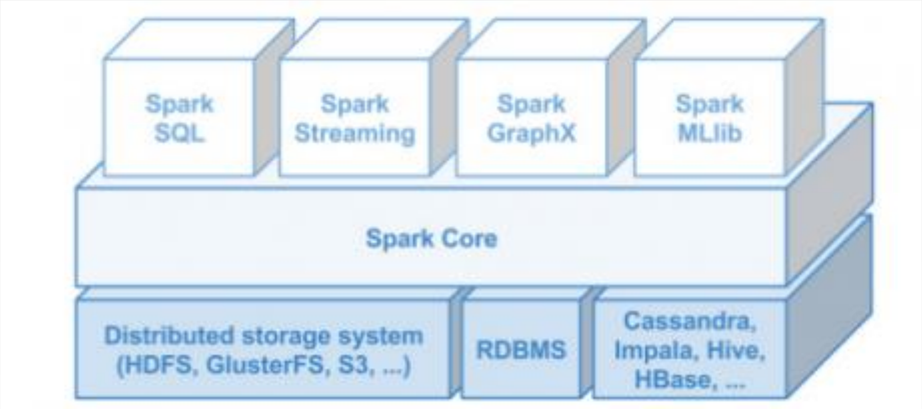
- **Spark is not**
  - a modified version of Hadoop
  - dependent on Hadoop because it has its own cluster management
  - Spark uses Hadoop for storage purpose only

What is Spark

- Spark's design philosophy centers around four key characteristics:
  - Speed
    - Its internal implementation benefits immensely from the performance improvement of CPUs and memory
    - Spark builds its query computations as a directed acyclic graph
    - It has a physical execution engine which generates compact code for execution
  - Ease of use
    - RDD, DataFrames, and Datasets
  - Modularity
    - Spark operations can be applied across many types of workloads and expressed in any of the supported programming languages: Scala, Java, Python, SQL, and R.
  - Extensibility
    - Spark focuses on its fast, parallel computation engine rather than on storage

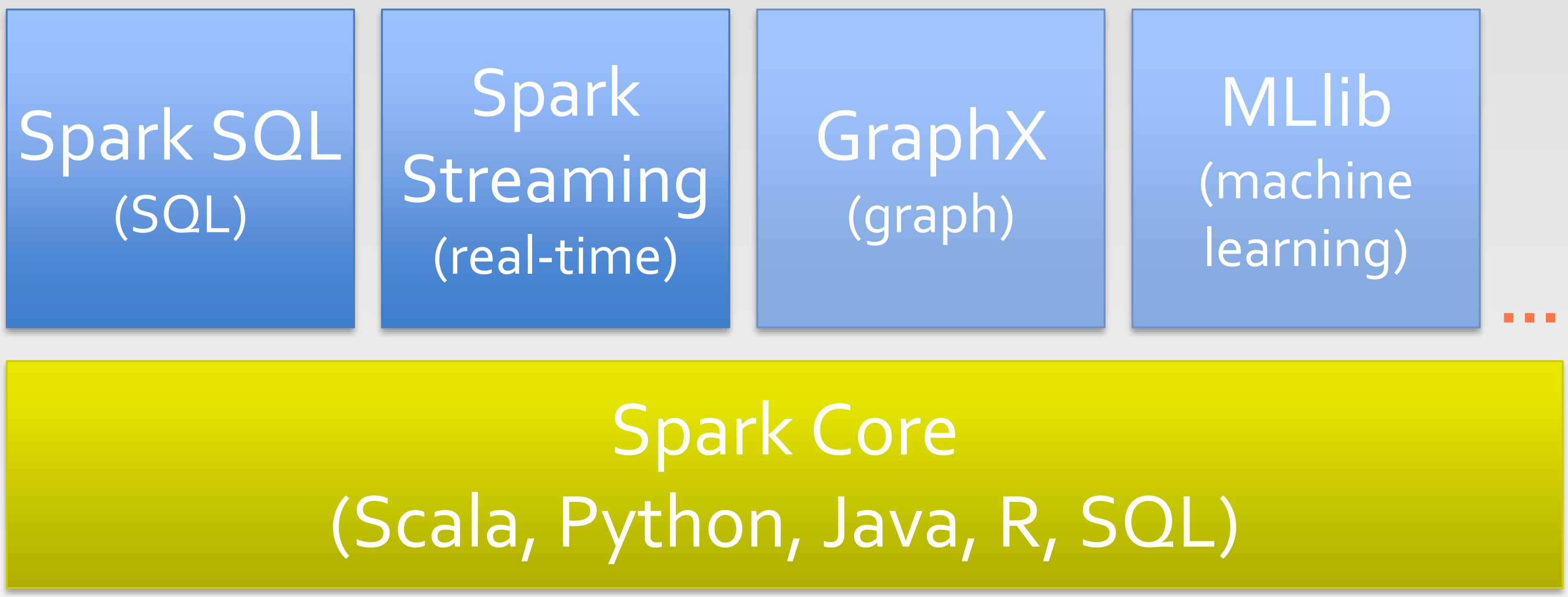
Data Sources

- Local Files
  - file:///opt/httpd/logs/access_log
- S3
- Hadoop Distributed Filesystem
  - Regular files, sequence files, any other Hadoop InputFormat
- HBase, Cassandra, etc.



What is Spark

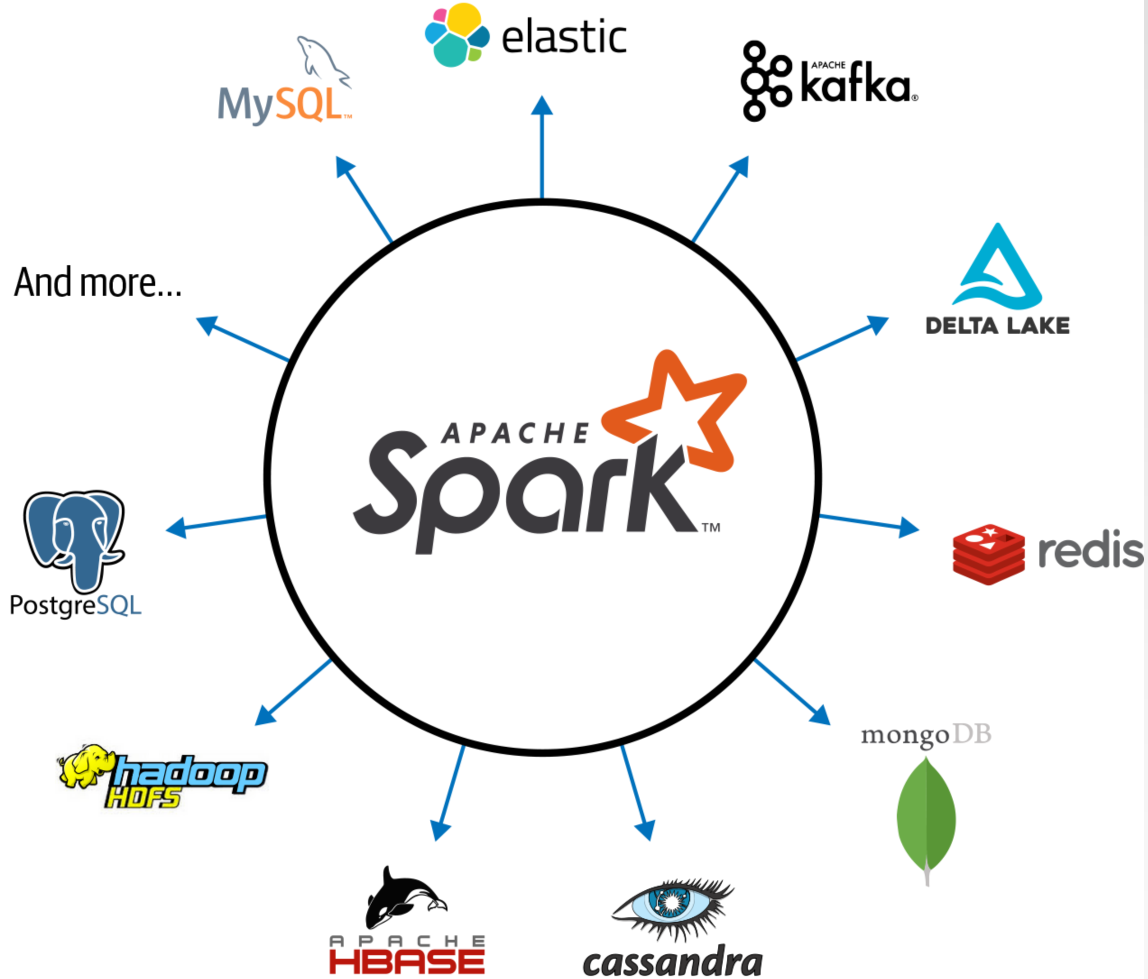
- Spark is the basis of a wide set of projects in the Berkeley Data Analytics Stack (BDAS)



  - Spark SQL (SQL on Spark)
  - Spark Streaming (stream processing)
  - GraphX (graph processing)
  - MLlib (machine learning library)

Spark's Ecosystem of Connectors

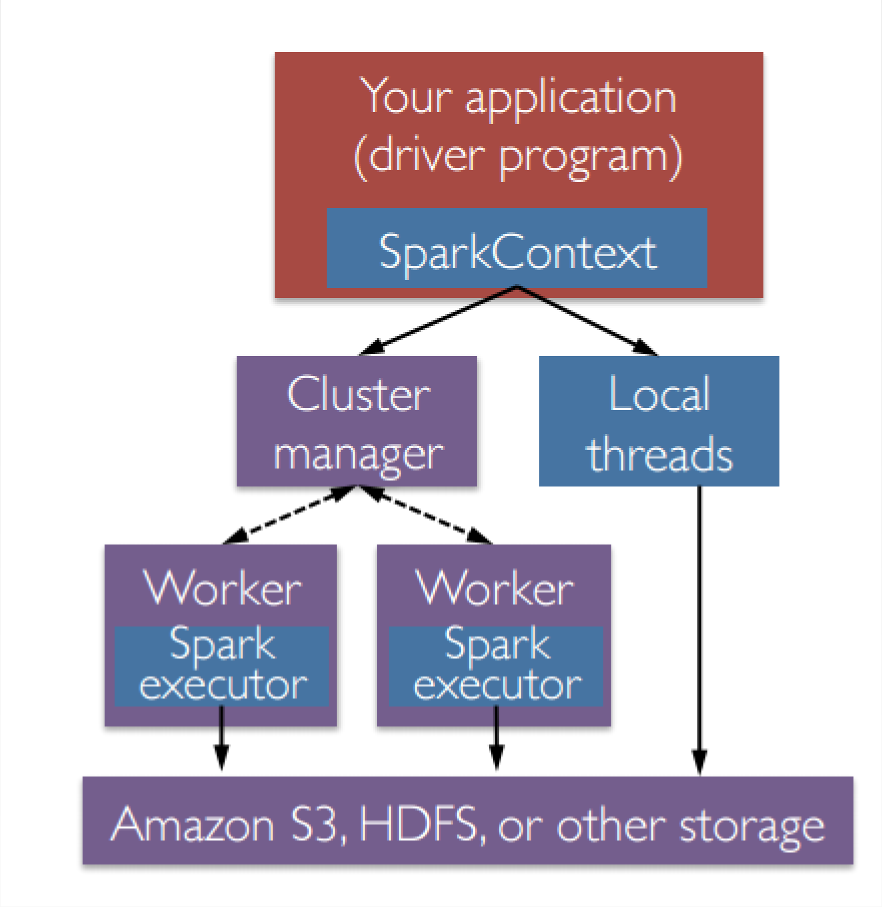
- The community of Spark developers maintains a list of third-party Spark packages as part of the growing ecosystem



Spark Ideas

- Expressive computing system, not limited to map-reduce model
- Facilitate system memory
  - avoid saving intermediate results to disk
  - cache data for repetitive queries (e.g. for machine learning)
- Layer an in-memory system on top of Hadoop.
- Achieve fault-tolerance by re-execution instead of replication

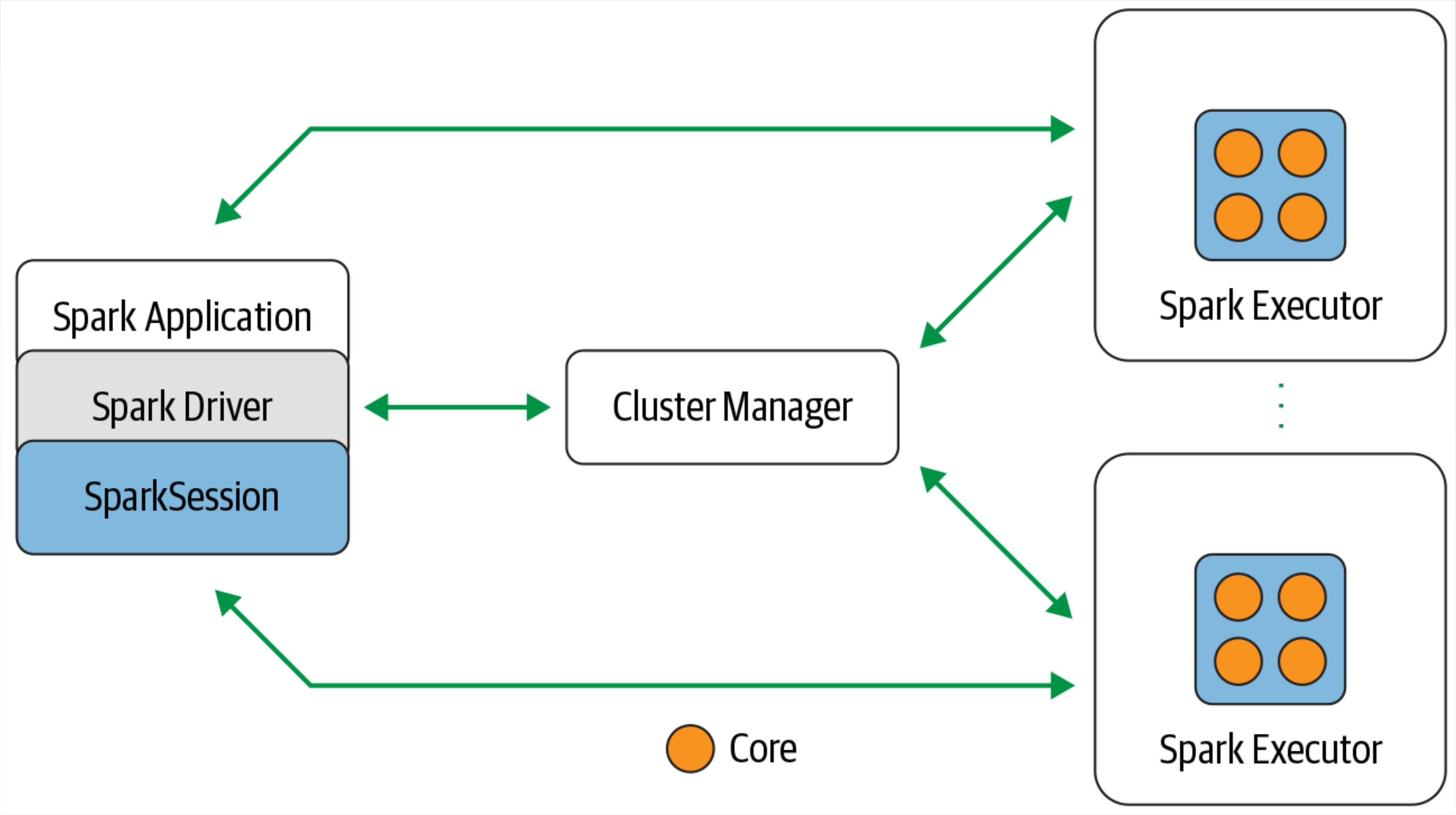
Spark Workflow (Spark 1.x)



- A Spark program first creates a SparkContext object
  - Tells Spark how and where to access a cluster
  - Define RDDs
  - Connect to several types of cluster managers (e.g., YARN, Mesos, or its own manager)
- Cluster manager:
  - Allocate resources across applications
- Spark executor:
  - Run computations
  - Access data storage

Spark Workflow (Spark 3.x)

- A Spark application consists of a driver program that is responsible for orchestrating parallel operations on the Spark cluster. The driver accesses the distributed components in the cluster—the Spark executors and cluster manager—through a SparkSession.



Spark Components (Spark 3.x)

- **Spark Driver**: part of the Spark application responsible for instantiating a SparkSession
  - Communicates with the cluster manager
  - Requests resources (CPU, memory, etc.) from the cluster manager for Spark's executors (JVMs)
  - Transforms all the Spark operations into DAG computations, schedules them, and distributes their execution as tasks across the Spark executors
  - Once the resources are allocated, it communicates directly with the executors.

Spark Components (Spark 3.x)

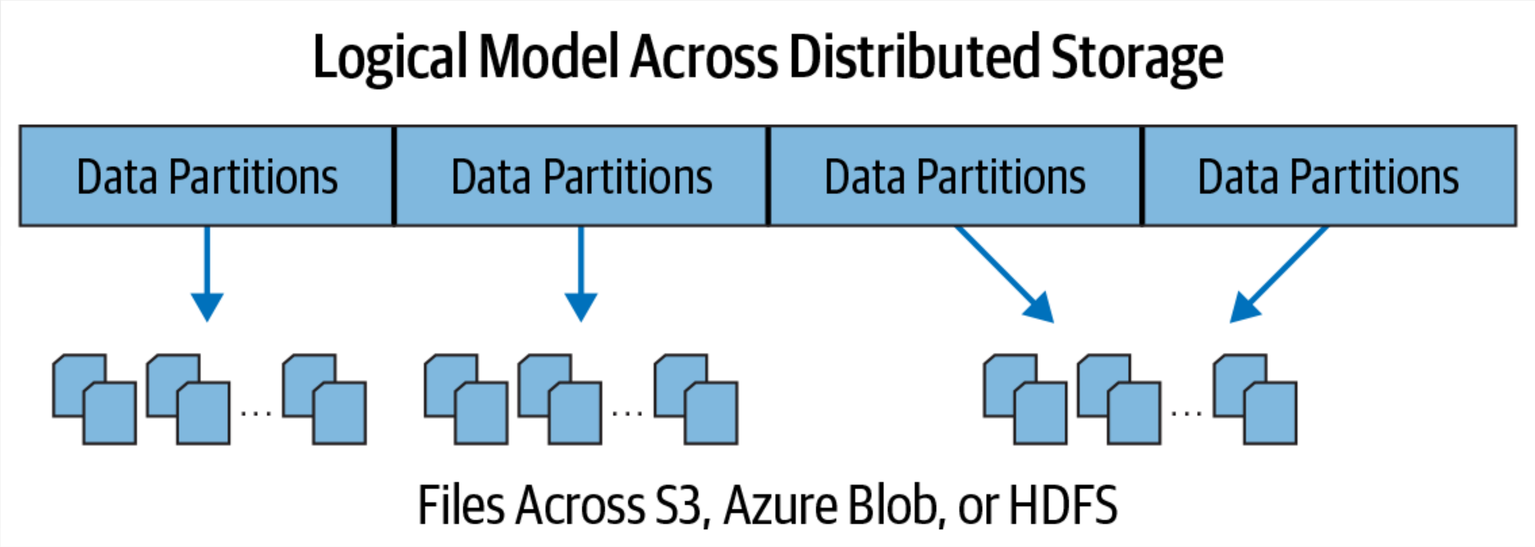
- Since Spark 2.x, the **SparkSession** became a unified conduit to all Spark operations and data (it subsumes previous entry points to Spark like the SparkContext)
- SparkSession provides a single unified entry point to all of Spark's functionality
  - Create JVM runtime parameters
  - Define DataFrames and Datasets
  - Read from Data Sources
  - Access catalog metadata
  - Issue Spark SQL queries

Spark Components (Spark 3.x)

- Cluster manager
  - Responsible for managing and allocating resources for the cluster of nodes on which your Spark application runs.
  - Support four cluster managers: the built-in standalone cluster manager, Apache Hadoop YARN, Apache Mesos, and Kubernetes.
- Spark executor
  - Runs on each worker node in the cluster.
  - Communicate with the driver program and is responsible for executing tasks on the workers.
  - In most deployments modes, only a single executor runs per node.

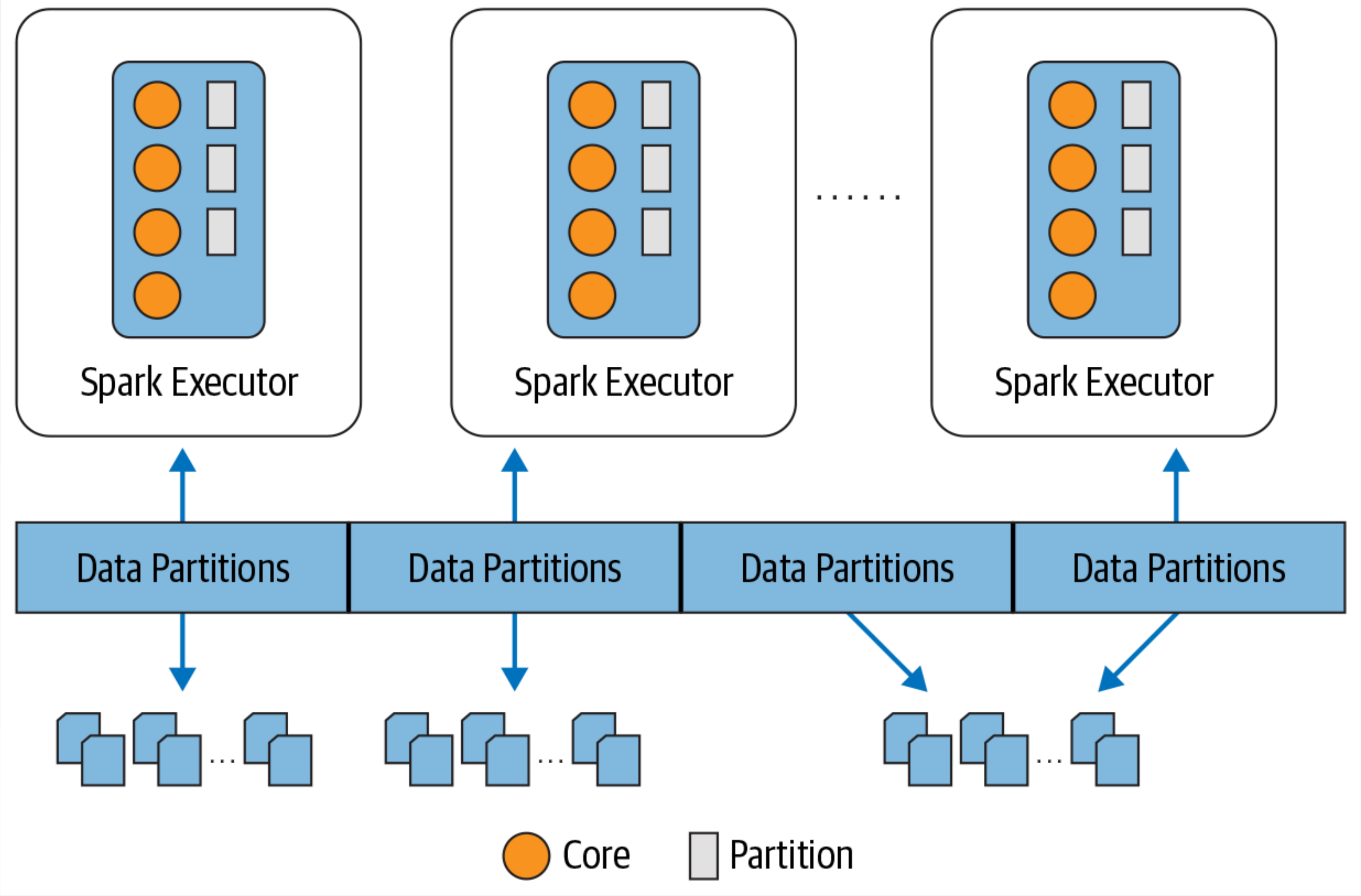
Distributed Data and Partitions

- Actual physical data is distributed across storage as partitions residing in either HDFS or other cloud storage.
- The data is distributed as partitions across the physical cluster
- Spark treats each partition as a high-level logical data abstraction in memory.
- Each Spark executor is preferably allocated a task that requires it to read the partition closest to it in the network, observing data locality.



Distributed Data and Partitions

- Each executor's core is assigned its own data partition to work on

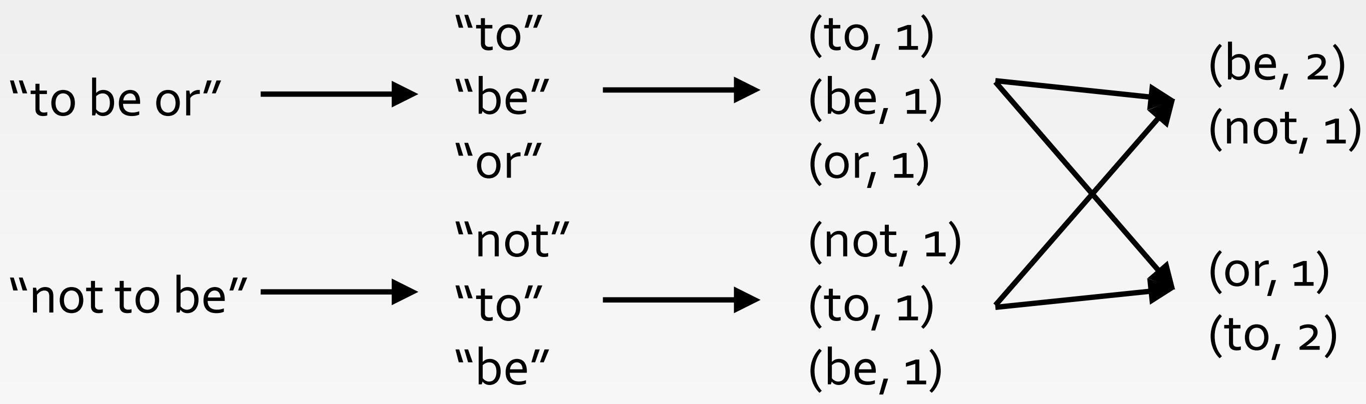


Word Count in Spark (RDD API)

```
val file = sc.textFile("hdfs://...", 4)

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

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



Part 2: Scala Introduction

Scala (Scalable language)

- Scala is a *general-purpose programming language* designed to express common programming patterns in a concise, elegant, and type-safe way
- Scala supports both Object Oriented Programming and Functional Programming
- Scala is Practical
  - Can be used as drop-in replacement for Java
    - Mixed Scala/Java projects
  - Use existing Java libraries
  - Use existing Java tools (Ant, Maven, JUnit, etc…)
  - Decent IDE Support (NetBeans, IntelliJ, Eclipse)

Why Scala

- Scala supports object-oriented programming. Conceptually, every value is an object and every operation is a method-call. The language supports advanced component architectures through classes and traits
- Scala is also a functional language. Supports functions, immutable data structures and preference for immutability over mutation
- Seamlessly integrated with Java
- Being used heavily for Big data, e.g., Spark, Kafka, etc.

Scala Basic Syntax

- When considering a Scala program, it can be defined as a collection of objects that communicate via invoking each other’s methods.
- **Object** − same as in Java
- **Class** − same as in Java
- **Methods** − same as in Java
- **Fields** − Each object has its unique set of instant variables, which are called fields. An object's state is created by the values assigned to these fields.
- **Traits** − Like Java Interface. A trait encapsulates method and field definitions, which can then be reused by mixing them into classes.
- **Closure** − A **closure** is a function, whose return value depends on the value of one or more variables declared outside this function.

closure = function + enviroment

Object-Oriented Programming in Scala

- Scala is object-oriented, and is based on Java's model
- An `object` is a singleton object (there is only one of it)
  - Variables and methods in an `object` are somewhat similar to Java's `static` variables and methods
  - Reference to an `object`'s variables and methods have the syntax ***ObjectName*.*methodOrVariableName***
  - The name of an `object` should be capitalized
- A `class` may take parameters, and may describe any number of objects
  - The class body *is* the constructor, but you can have additional constructors
  - With correct use of `val` and `var`, Scala provides getters and setters for class parameters

Scala is Statically Typed

- You don't have to specify a type in most cases
- Type Inference

```
val sum = 1 + 2 + 3
val nums = List(1, 2, 3)
val map = Map("abc" -> List(1,2,3))
```

Explicit Types

```
val sum: Int = 1 + 2 + 3
val nums: List[Int] = List(1, 2, 3)
val map: Map[String, List[Int]] = ...
```

Scala is High level

```
// Java – Check if string has uppercase character
boolean hasUpperCase = false;
for(int i = 0; i < name.length(); i++) {
    if(Character.isUpperCase(name.charAt(i))) {
        hasUpperCase = true;
         break;
    }
}

// Scala
val hasUpperCase = name.exists(_.isUpper)
```

Scala is Concise

```java
// Java
public class Person {
  private String name;
  private int age;
  public Person(String name, Int age) {
    this.name = name;
     this.age = age;
  }
  public String getName() {               // name getter
    return name;
  }
  public int getAge() {                   // age getter
    return age;
  }
  public void setName(String name) {      // name setter
    this.name = name;
  }
  public void setAge(int age) {           // age setter
    this.age = age;
  }
}
```

```scala
// Scala
class Person(var name: String, private var _age: Int) {
  def age = _age              // Getter for age
  def age_=(newAge:Int) {   // Setter for age
    println("Changing age to: "+newAge)
    _age = newAge
  }
}
```

Variables and Values

- Variables: values stored can be changed

```
var foo = "foo"
foo = "bar"  // okay
```

- Values: immutable variable

```
val foo = "foo"
foo = "bar" // nope
```

Scala is Pure Object Oriented

```
// Every value is an object
1.toString
// Every operation is a method call
1 + 2 + 3  →  (1).+(2).+(3)
// Can omit . and ( )
"abc" charAt 1  →  "abc".charAt(1)
// Classes (and abstract classes) like Java
abstract class Language(val name:String) {
  override def toString = name
}
// Example implementations
class Scala extends Language("Scala")
// Anonymous class
val scala = new Language("Scala") { /* empty */ }
```

Scala Traits

```
// Like interfaces in Java
trait JVM {
  // But allow implementation
  override def toString = super.toString+" runs on JVM" }
trait Static {
  override def toString = super.toString+" is Static" }

// Traits are stackable
class Scala extends Language with JVM with Static {
  val name = "Scala"
}
println(new Scala) → "Scala runs on JVM is Static"
```

Scala is Functional

- First-Class Functions. Functions are treated like objects:
  - passing functions as arguments to other functions
  - returning functions as the values from other functions
  - assigning functions to variables or storing them in data structures

```
// Lightweight anonymous functions
(x:Int) => x + 1

// Calling the anonymous function
val plusOne = (x:Int) => x + 1
plusOne(5)  →  6
```

Scala is Functional

- Closures: a function whose return value depends on the value of one or more variables declared outside this function.

```
// plusFoo can reference any values/variables in scope
var foo = 1
val plusFoo = (x:Int) => x + foo

plusFoo(5)   → 6

// Changing foo changes the return value of plusFoo
foo = 5
plusFoo(5)   → 10
```

Scala is Functional

- Higher Order Functions
  - A function that does at least one of the following:
    - takes one or more functions as arguments
    - returns a function as its result

```
val plusOne = (x:Int) => x + 1
val nums = List(1,2,3)
// map takes a function: Int => T
nums.map(plusOne)       →  List(2,3,4)
// Inline Anonymous
nums.map(x => x + 1)    →  List(2,3,4)
// Short form
nums.map(_ + 1)         →  List(2,3,4)
```

More Examples on Higher Order Functions

```
val nums = List(1,2,3,4)
// A few more examples for List class
nums.exists(_ == 2)          →  true
nums.find(_ == 2)            →  Some(2)
nums.indexWhere(_ == 2)      →  1

// functions as parameters, apply f to the value “1”
def call(f: Int => Int) = f(1)

call(plusOne)      →  2
call(x => x + 1)   →  2
call(_ + 1)        →  2
```

More Examples on Higher Order Functions

```
val basefunc = (x:Int) => ((y:Int) => x + y)
// interpreted by:
   basefunc(x){
        sumfunc(y){ return x+y;}
        return sumfunc;
   }

val closure1 = basefunc(1)   closure1(5) = ?
                                          6
val closure2 = basefunc(4)   closure2(5) = ?
                                          9
```

- basefunc returns a function, and closure1 and closure2 are of function type.
- While closure1 and closure2 refer to the same function basefunc, the associated environments differ, and the results are different

The Usage of "_" in Scala

- In anonymous functions, the "_" acts as a placeholder for parameters

nums.map(x => x + 1)

is equivalent to:

nums.map(_ + 1)

List(1,2,3,4,5).foreach(print(_))

is equivalent to:

List(1,2,3,4,5).foreach(a => print(a))

- You can use two or more underscores to refer different parameters.

val sum = List(1,2,3,4,5).reduceLeft(_+_)

is equivalent to:

val sum = List(1,2,3,4,5).reduceLeft((a, b) => a + b)

  - The reduceLeft method works by applying the function/operation you give it, and applying it to successive elements in the collection

Part 3: RDD Introduction

Challenge

- Existing Systems
  - Existing in-memory storage systems have interfaces based on fine-grained updates
    - Reads and writes to cells in a table
    - E.g., databases, key-value stores, distributed memory
  - Requires replicating data or logs across nodes for fault tolerance

    -> expensive!
    - 10-100x slower than memory write

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

Solution: Resilient Distributed Datasets

- *Resilient Distributed Datasets (RDDs)*
  - Distributed collections of objects that can be cached in memory across cluster
  - Manipulated through parallel operators
  - Automatically recomputed on failure based on lineage
- RDDs can express many parallel algorithms, and capture many current programming models
  - Data flow models: MapReduce, SQL, …
  - Specialized models for iterative apps: Pregel, …

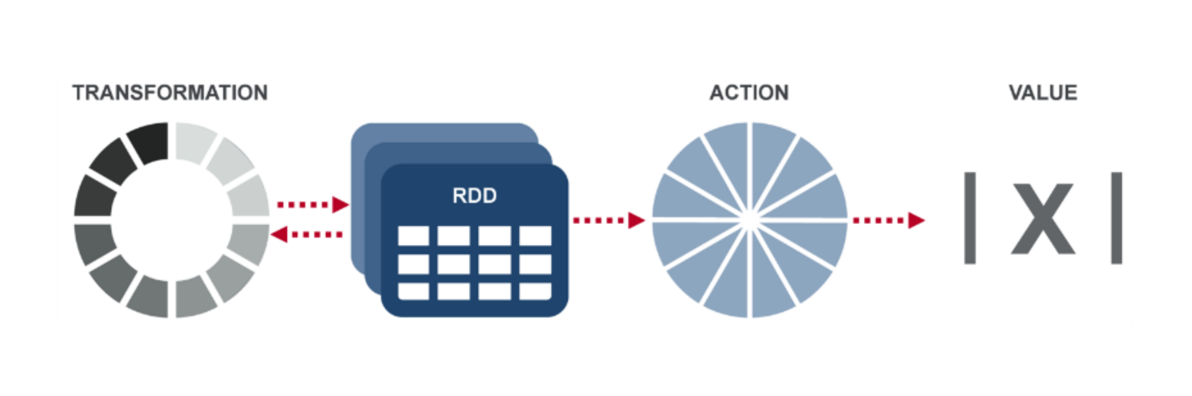
What is RDD

- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, et al. NSDI'12
  - RDD is a **distributed** memory abstraction that lets programmers perform **in-memory** computations on large clusters in a **fault-tolerant** manner.
- **Resilient**
  - Fault-tolerant, is able to recompute missing or damaged partitions due to node failures.
- **Distributed**
  - Data residing on multiple nodes in a cluster.
- **Dataset**
  - A collection of partitioned elements, e.g. tuples or other objects (that represent records of the data you work with).
- RDD is the primary data abstraction in Apache Spark and the core of Spark. It enables operations on collection of elements in parallel.

RDD Traits

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable** or **Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.
- **Typed**, i.e. values in a RDD have types, e.g. RDD[Long] or RDD[(Int, String)].
- **Partitioned**, i.e. the data inside a RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node).

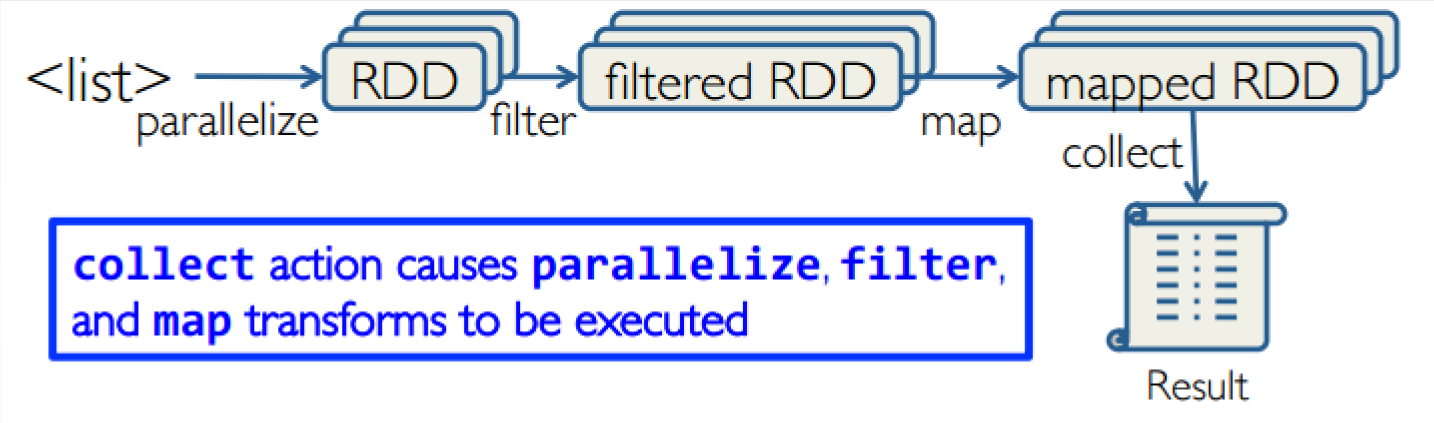
RDD Operations



- **Transformation:** returns a new RDD.
  - Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.
  - Transformation functions include *map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, join, etc.*
- **Action:** evaluates and returns a new value.
  - When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.
  - Action operations include *reduce, collect, count, first, take, countByKey, foreach, saveAsTextFile, etc.*

Working with RDDs

- Create an RDD from a data source
  - by parallelizing existing collections (lists or arrays)
  - by transforming an existing RDDs
  - from files in HDFS or any other storage system
- Apply transformations to an RDD: e.g., map, filter
- Apply actions to an RDD: e.g., collect, count



- Users can control two other aspects:
  - Persistence
  - Partitioning

Creating RDDs

- From HDFS, text files, Amazon S3, Apache HBase, SequenceFiles, any other Hadoop InputFormat
- Creating an RDD from a File
  - val inputfile = sc.textFile("...", 4)
    - RDD distributed in 4 partitions
    - Elements are lines of input
    - Lazy evaluation means no execution happens now

```
scala> val inputfile = sc.textFile("pg100.txt")
inputfile: org.apache.spark.rdd.RDD[String] = pg100.txt MapPartitionsRDD[17] at
textFile at <console>:24
```

- Turn a collection into an RDD
  - sc.parallelize([1, 2, 3]), creating from a Python list
  - sc.parallelize(Array(“hello”, “spark”)), creating from a Scala Array
- Creating an RDD from an existing Hadoop InputFormat
  - sc.hadoopFile(keyClass, valClass, inputFmt, conf)

Spark Transformations

- Create new datasets from an existing one
- Use lazy evaluation: results not computed right away – instead Spark remembers set of transformations applied to base dataset
  - Spark optimizes the required calculations
  - Spark recovers from failures
- Some transformation functions

Transformation	Description
`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
`distinct([`*numTasks*`]))`	return a new dataset that contains the distinct elements of the source dataset
`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)

Spark Actions

- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark
- Some action functions

Action	Description
`reduce(`*`func`*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(`*`n`*`)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(`*`n, key=func`*`)`	return n elements ordered in ascending order or as specified by the optional key function

- Example: words.collect().foreach(println)

References

- http://spark.apache.org/docs/latest/index.html
- http://www.scala-lang.org/documentation/
- http://www.scala-lang.org/docu/files/ScalaByExample.pdf
- A Brief Intro to Scala, by Tim Underwood.
- Learning Spark. 1st and 2nd Edition

End of Chapter 4.1