BIG DATA

UNIT - 3

In Memory Computation

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

Hadoop Challenges

- No iterative support from language
- Must do manually

SCALA

- SCAlable LAnguage
- Object-oriented, compiled into Java bytecode (runs on JVM)

hello.scala $\xrightarrow{\text{compiled}}$ scala.class

- Can reference Java libraries
- Blends OO and FP (functional programming)
- Strongly statically typed

What's wrong with Java?

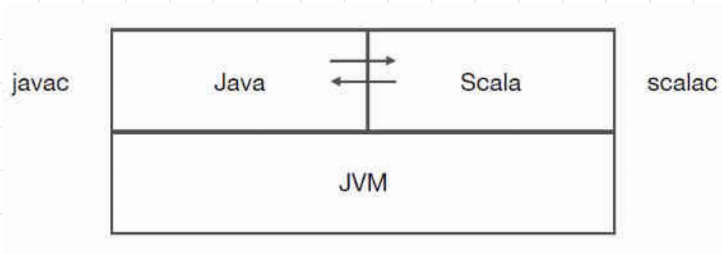
- Verbose (boilerplate)
- Not designed to be very concurrent (before Java 5+)

What's right with Java?

- Popular
- OO
- Strongly typed
- Library of classes — large
- JVM — platform independent

Java vs Scala

- Almost completely interoperable



SCALA

```scala
// Declaring variables
var x: Int = 7   // explicit type
var x = 7        // type inferred
val y = "hi"     // read-only (constant)

// Functions
def square(x: Int): Int = x*x

def square(x: Int): Int = {
    x*x
}

def announce(text: String) = {
    println(text)
}
```

JAVA

```java
class Test {
    public static void main(String[] args) {
        int x = 7;

        final String y = "hi";
    }
}

class Example {
    int square(int x) {
        return x*x;
    }

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

- no return keyword
- features similar to JS & python

Major Differences

1. Minimal verbosity

2. Referential Transparency
   - type inferencing in Scala
   - compiler checks type of subexpressions, atomic values

3. Concurrency
   - Actor model
   - Akka — open-source framework for Actor-based concurrency

4. Functional Programming
   - Higher order functions that can return another function
   - Nested functions

1. Minimal Verbosity

- Getters & setters (note: ppt link does not work; objectmentor has been moved to http://cleancoder.com/)

- Java

```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 String getFirstName() { return this.firstName; }

    public void setLastName(String lastName) { this.lastName = lastName; }
    public String getLastName() { return this.lastName; }

    public void setAge(int age) { this.age = age; }
    public int getAge() { return this.age; }
}
```

- Scala (automatic)

```
class Person(var firstName: String, var lastName: String, var age: Int)
```

2. Type Inferencing

- Java is statically typed
  - type errors caught by compiler

- Ruby & Python do not require declared types
  - harder to debug
  - not type safe

- Scala is statically typed but it uses type inferencing
  - type errors caught by compiler
  - https://docs.scala-lang.org/tour/type-inference.html

```
val collegeName = "PES University"      // const reference
def squareOf(x: Int) : x * x            // def method that cannot be reassigned
```

Consistency

- Java: every value is a type, except primitive types (int, bool) for efficiency reasons

- Scala: every value is an object; compiler turns into primitives for efficiency

- Java has operators & methods with different syntaxes

- In Scala, operators are methods and either syntax can be used

3. Concurrency

- Concurrency vs parallelism
  - Concurrency creates illusion of parallelism; can execute multiple threads on the same core
  - Concurrency achieved through context switching
  - Parallelism requires multiple cores to run multiple computations simultaneously

- Fine-grained concurrency: frequent interactions between threads working together
  - difficult to implement right
  - requires locks on shared resources

- Coarse-grained concurrency: infrequent interactions between largely independent sequential processes
  - easier to get right
  - map-reduce
  - not at cycle level

- Java 5 & 6 — reasonable support for Fine-grained concurrency

- Scala has access to the Java API

- Scala also has Actors for coarse-grained parallelism
  - Sending messages using send ! Abstraction

Map $\xrightarrow{\text{message}}$ Reduce

4. Functional Programming

- Problem with concurrency: acquire locks

- If prog language does not allow modification of variables, locks not required

- Functional programming languages use only immutable data (eg: ML, OCaml, Haskell, lisp)

- Difficult to learn

- Scala is an impure functional language — can program functionally but not forced upon you

- Features

(a) Immutable
  - functional operations create new structures and do not modify existing structures

(b) Program implicitly captures data flow

(c) Order of operations unimportant

(d) Functions
  - are objects
  - arguments
  - can be returned
  - can operate on collections

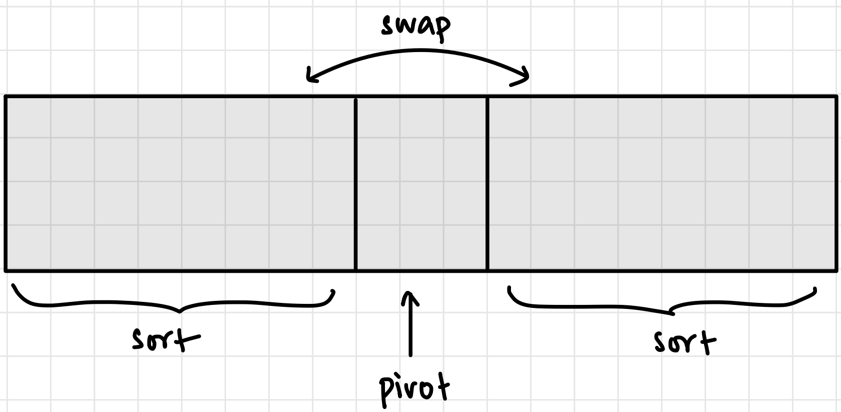
- Quicksort in Scala – Java Style

```scala
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)
}
```

pivot in the middle

explicitly iterate

determine when to swap



- Quicksort in Scala – functional programming

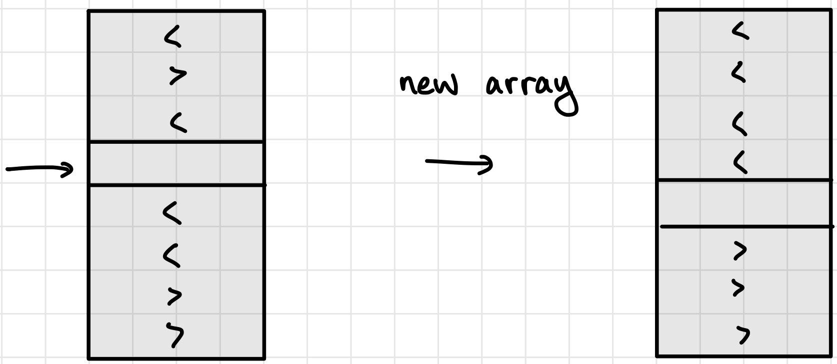
```scala
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
           xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
}
```

pick a pivot

sort values smaller than pivot

sort values greater than pivot

concatenate result



Q: Does this sort array in ascending order or descending order?

```scala
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 <)))
  }
}
```

→ ascending

Q: 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 {
        sort(xs filter (pivot <)),
        xs filter (pivot ==),
        sort(xs filter (pivot >))
    }
}
```

can parallelise filtering tasks

Functional Programming & Functions

list remains 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
```

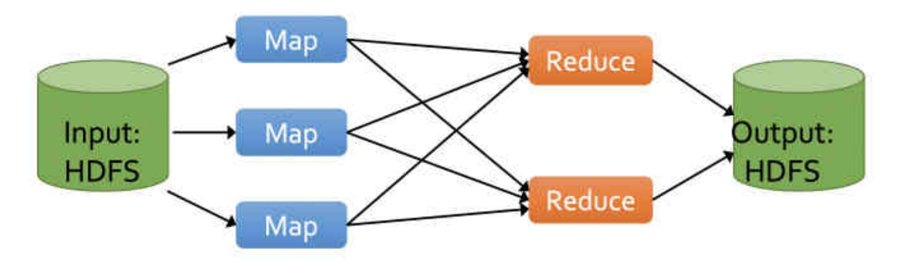
Functional Programming & Big Data

- Independent parallel operations
  - map() in FP

- Parallel operations to be consolidated
  - aggregation() of FP

SPARK

- Most cluster prog models: DAG (Directed Acyclic Graph)



- Advantages of Hadoop
  1. Input HDFS ⟶ output HDFS
  2. User specified no. of M/R
  3. Handle failures

- Issues of Hadoop (think: page rank)
  1. Iterative
  2. Every iteration requires write to disk
  3. O/P of reducer ⟶ input of mapper (in & out of disk)

- Look: Haloop

In-Memory computation

- Word-count program in Scala

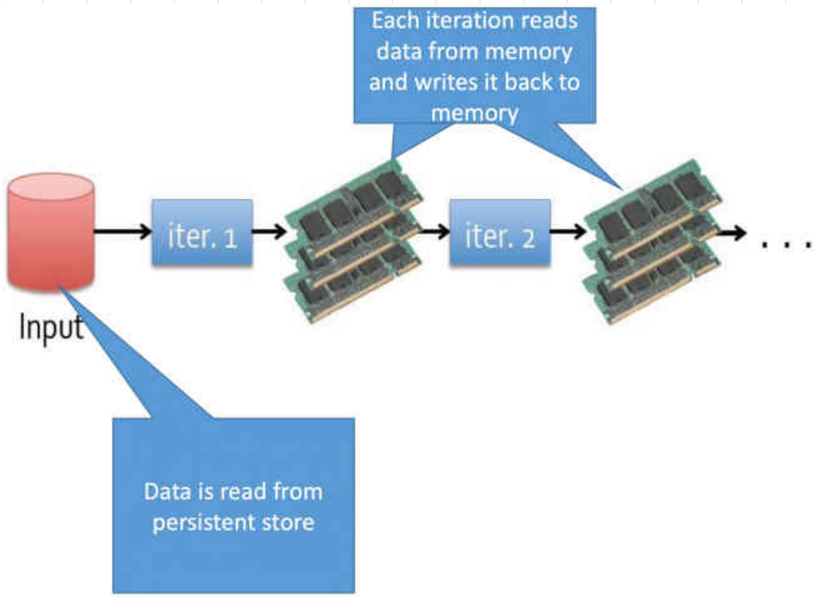
```
val lines = scala.io.Source.fromFile("textfile.txt").getLines
val words = lines.flatMap(line => line.split(" ")).toIterable
val counts = words.groupBy(identity).map(words =>
    words._1 -> words._2.size)
val top10 = counts.toArray.sortBy(_._2).reverse.take(10)
println(top10.mkString("\n"))
```

→ compute word count

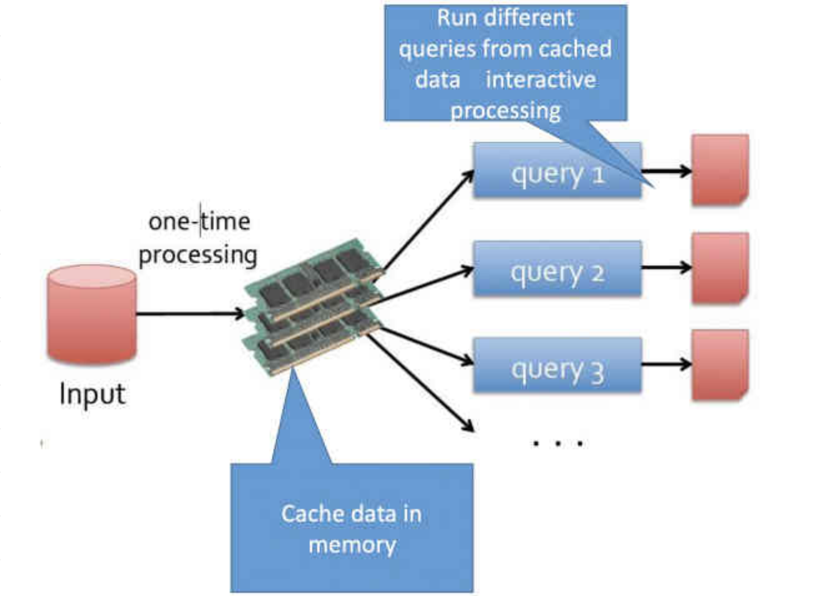
each operation creates data value that can be kept in memory & reused

Iterative Processing in Memory

- Think: page rank



- Caching



Problems

1. Too large for RAM ; how to deal with overflows
2. How to split across DRAM of entire cluster
3. How to handle failures (power failures)

DISTRIBUTED DATASET

- Flume: import logs into HDFS (error logs, activity logs etc)

Example Log Processing

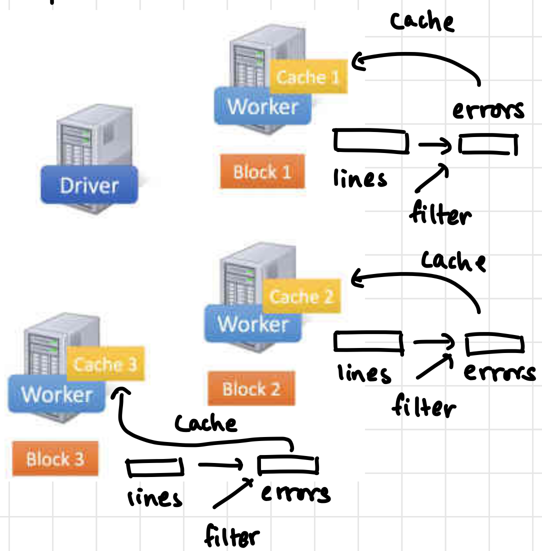
- Load from log into memory
- Search for patterns

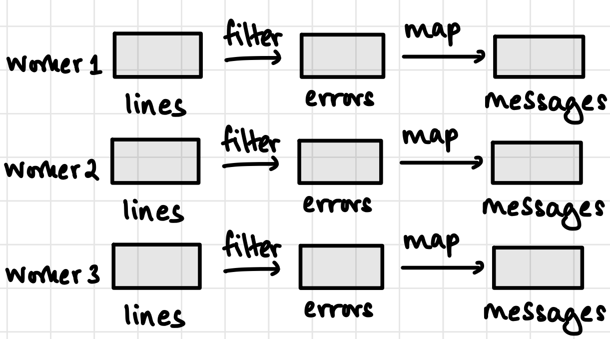
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(startsWithERROR())
messages = errors.map(split("\t"),2)
cachedMsgs = messages.cache()

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

Distribute the Computation

- lines: virtual DS with 3 parts (distributed)
- filter performed separately on each partition (parallelly)


```
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()
```



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

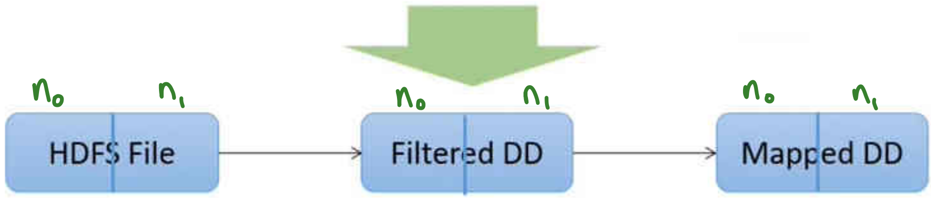
Step2: remove all lines that don't contain the term ERROR

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.

- What if n_1 crashes after filter

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



Resilient Distributed Dataset (RDD)

- Add lineage information to the concept of a distributed dataset
- Ability to recreate in case of failure
- Keep track of operations performed on RDD and to create RDD
- Types of operations support

RDD Operations

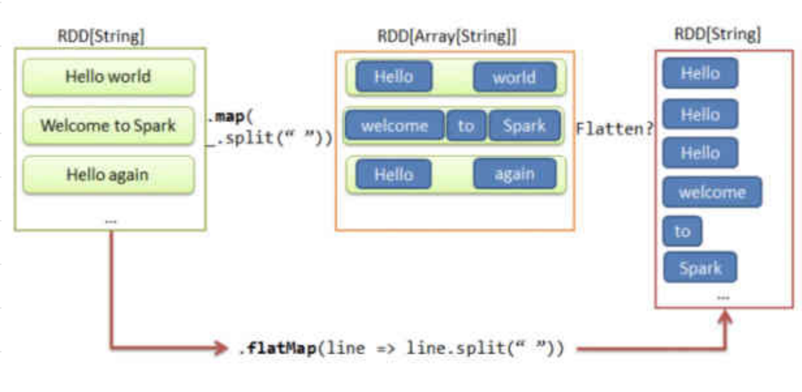
1. Transformations
2. Actions

Transformation

- Create new dataset from existing dataset
- Eg: map() in scala

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	{2, 3, 4, 4}
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	{1, 2, 3, 2, 3, 3, 3}
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	{2, 3, 3}
`distinct()`	Remove duplicates.	`rdd.distinct()`	{1, 2, 3}
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic



Q: $l = \{47, 39, 22, 25, 36\}$

l.filter(x => (x%2) == 1)

$47 \% 2 == 1 \longrightarrow$ true
$39 \% 2 == 1 \longrightarrow$ true
$22 \% 2 == 1 \longrightarrow$ false
$25 \% 2 == 1 \longrightarrow$ true
$36 \% 2 == 1 \longrightarrow$ false

filtered = $\{47, 39, 25\}$

Q: n_0 $\{47, 39, 22\}$ n_1 $\{25, 36\}$

l.filter(x => (x%2) == 1)

n_0 $\{47, 39\}$ n_1 $\{25\}$

Flatmap

```
["line one", "line two"]
        |
        ↓ split each line
[["line", "one"], ["line", "two"]]
        |
        ↓ combine
["line", "one", "line", "two"]
```

Transformations on 2 RDDs

Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
union()	Produce an RDD containing elements from both RDDs.	rdd.union(other)	{1, 2, 3, 3, 4, 5}
intersection()	RDD containing only elements found in both RDDs.	rdd.intersection(other)	{3}
subtract()	Remove the contents of one RDD (e.g., remove training data).	rdd.subtract(other)	{1, 2}
cartesian()	Cartesian product with the other RDD.	rdd.cartesian(other)	{(1, 3), (1, 4), … (3,5)}

check if repeat is mistake

Actions

- Operations that return a value
- Eg: reduce()

Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	4
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return `num` elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9

at master

at master

RDD Operations on Key-Value pairs

- Spark: pair RDDs

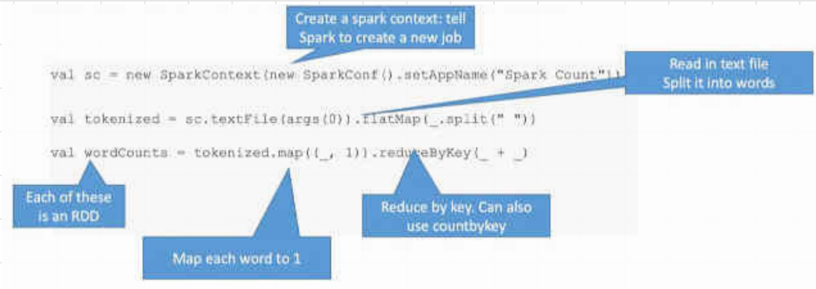
Table 4-1. Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})

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.	
`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}`

needs partition function

countByKey: distributed action
returns hashmap of (k, Int) pairs with count of each key

WORD COUNT IN SPARK

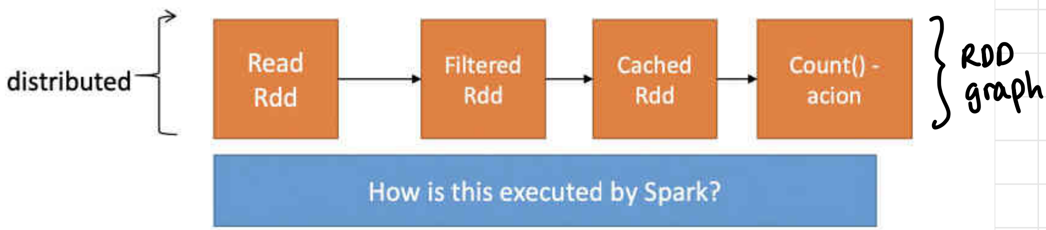


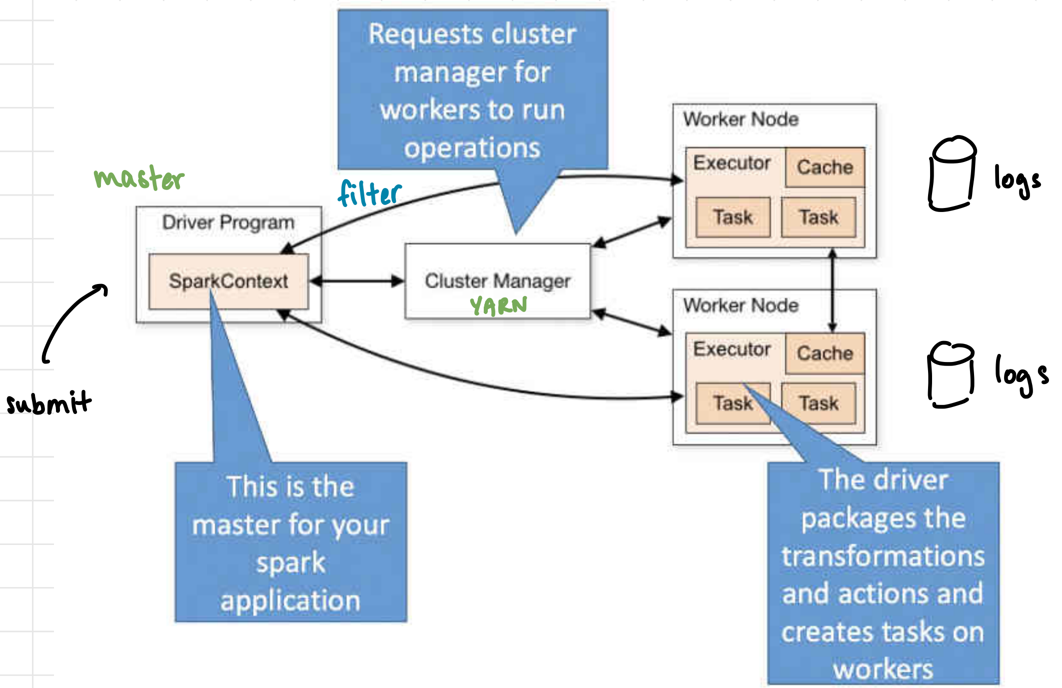
Spark Architecture

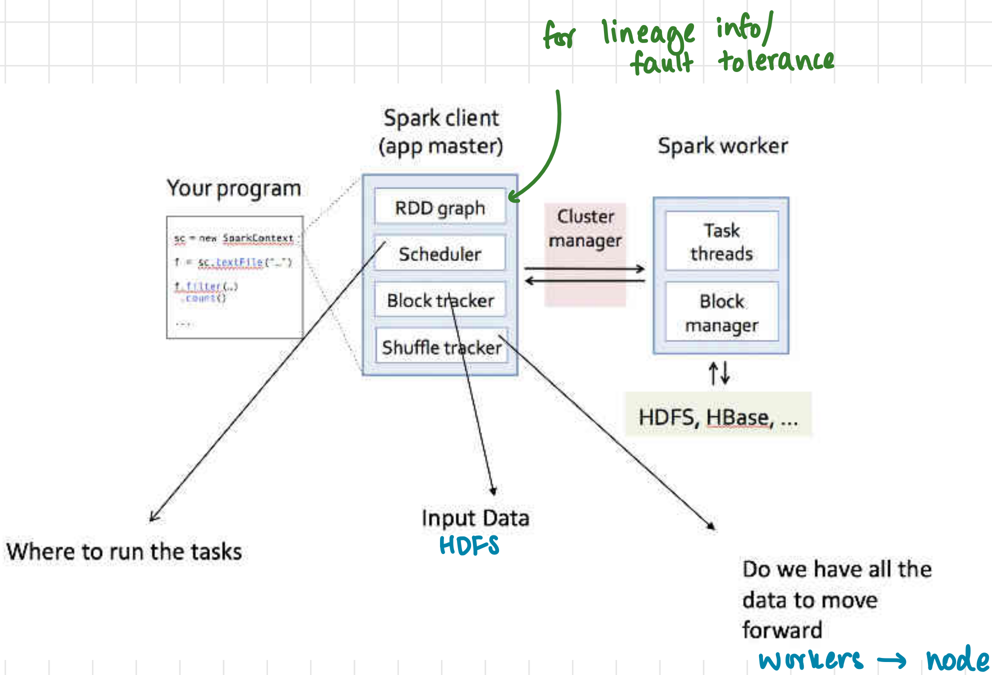
- Eg: log mining

```
val sc = new SparkContext("spark://...", "MyJob", home,
    jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
    an RDD

errors.cache()

errors.count() // This is an action
```

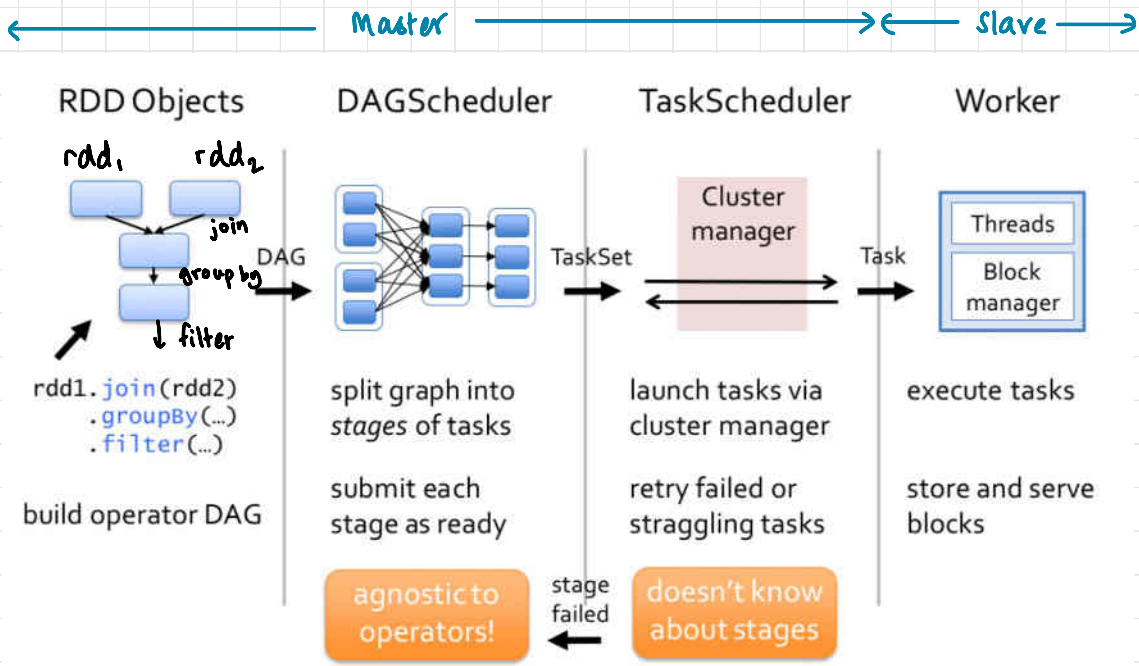
cache: in order to replicate if fails, reduce processing time, retain for further use (prevent garbage collect)




Requests cluster manager for workers to run operations
master
filter
Driver Program
SparkContext
Cluster Manager
YARN
Worker Node
Executor
Cache
Task
Task
Worker Node
Executor
Cache
Task
Task
logs
logs
submit
This is the master for your spark application
The driver packages the transformations and actions and creates tasks on workers

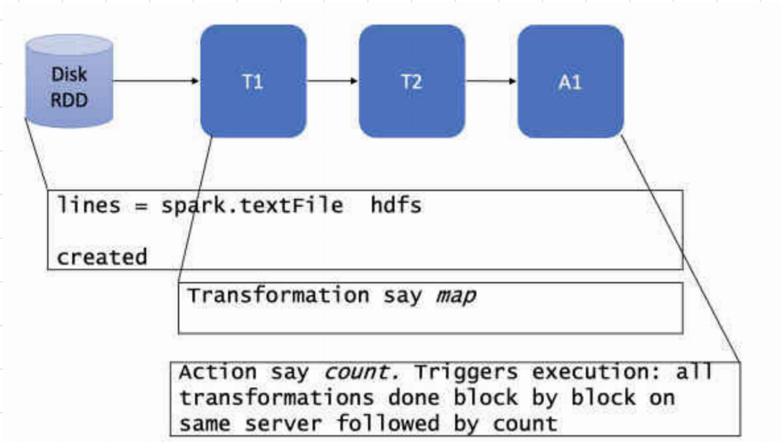

for lineage info/ fault tolerance
Spark client (app master)
Spark worker
Your program
sc = new SparkContext
f = sc.textFile("…")
f.filter(..)
.count()
...
RDD graph
Scheduler
Block tracker
Shuffle tracker
Cluster manager
Task threads
Block manager
HDFS, HBase, …
Where to run the tasks
Input Data
HDFS
Do we have all the data to move forward
workers → node

Spark Working Details



Lazy Execution

- Spark driver: no execution when encounters a transformation
- Transformations only noted for lineage
- Executes only when action encountered



Q: Why lazy execution?

- Clubbing transformations reduce net traffic
- Bring data into memory once
- Optimisations can be performed

Spark Log Mining Example

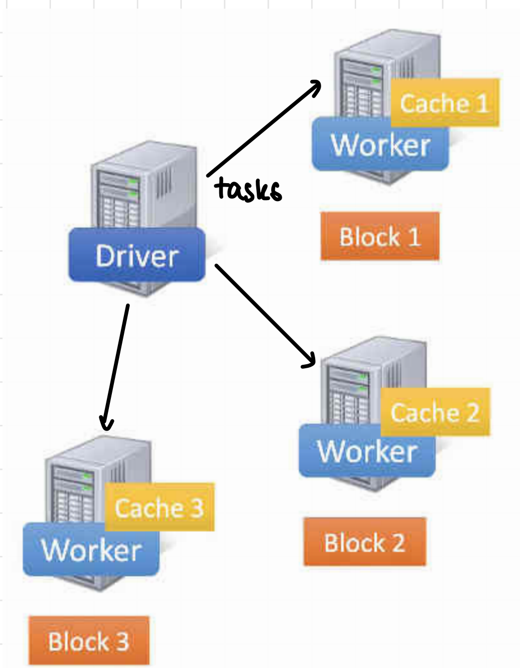
parallel

```
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

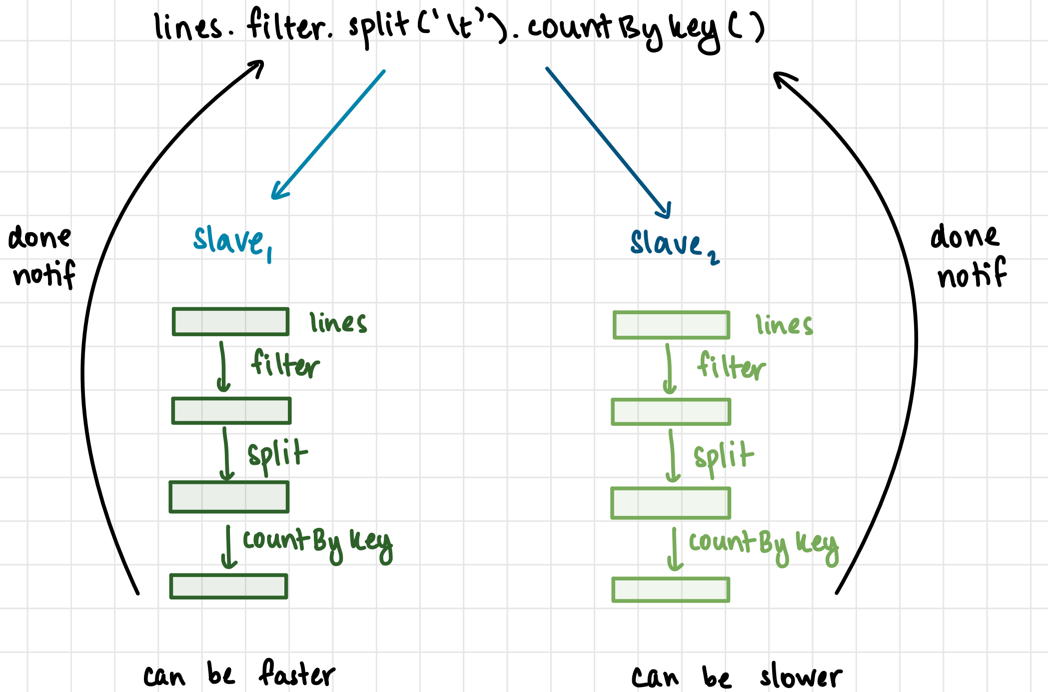
transformed RDD

action



RDDs- Details

- Partitioned, locality aware, distributed collection
- RDDs are immutable (no partial state with multiple threads)
- RDDs are DS that either
  - point to data source (HDFS)
  - apply a transformation to parent RDDs to generate new elements
- Computations on RDDs
  - lazily evaluated lineage DAGs composed of chained RDDs

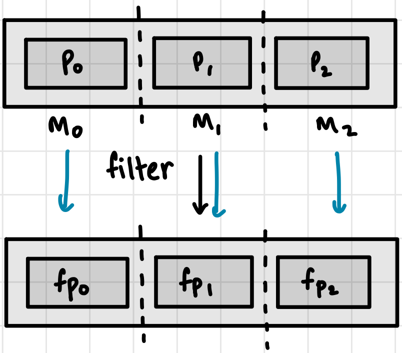


Why RDD Abstraction?

- Support operations other than MR
- Support in-memory computation
- Arbitrary composition of such operators
- Simplify scheduling (order of generation of RDDs)

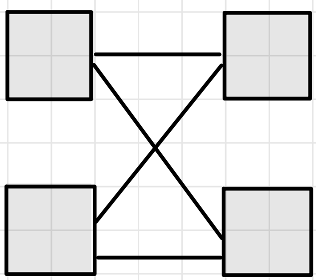
Representing RDDs

- Splits — set of partitions (machines)
  - like Hadoop, each RDD associated with input partition



filter 1:1 dependency

MR:



- List of dependencies on parent RDDs
- Function to compute partitions given parents
- Optional preferred locations
- Optional partitioning info (partitioner for shuffle)

RDDs Interface

Operation	Meanning
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

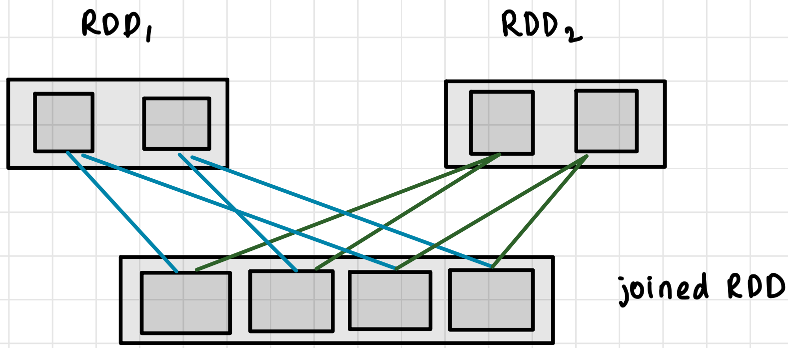
Hadoop RDD (map)

Partitions – one per block
Dependencies – none
Compute (partition) – read corresponding block
Preferred locations – HDFS block location
Partitioner – none

Filtered RDD

Partitions – same as parent
Dependencies – 1-1 with parent
Compute – compute parent and filter it
Preferred locations – ask parent (none)
Partitioner – none

Join RDD

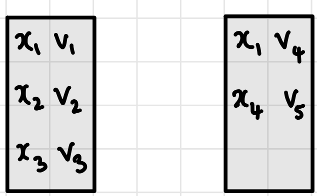


Partitions – one per reduce task
Dependencies – many to one
Compute – read and join shuffled data
Preferred locations – none
Partitioner – HashPartitioner

ReduceByKey RDD

- Transformation

Partitions – one per key
Dependencies – many to many (on all parent nodes)
Compute – reduce data and send
Preferred locations – subset of parent partitions involved
Partitioner – hash



Spark Scheduling

Page Rank in Spark

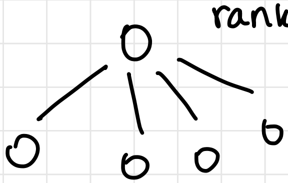
urls $\longrightarrow$ link

(source) (dest)

```
lines = textfile
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)
```

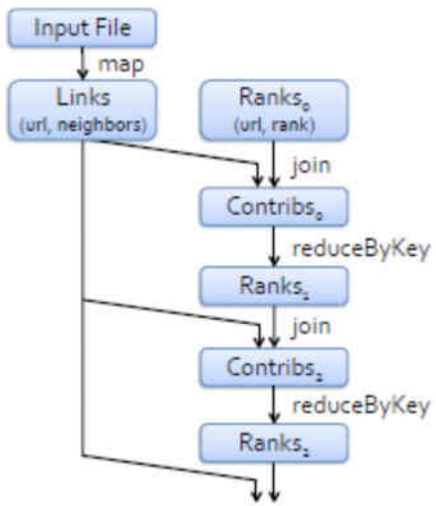
```
def computeContribs (url_neighbors_rank):

     """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)
```

rank/neighbours



DAG Representation

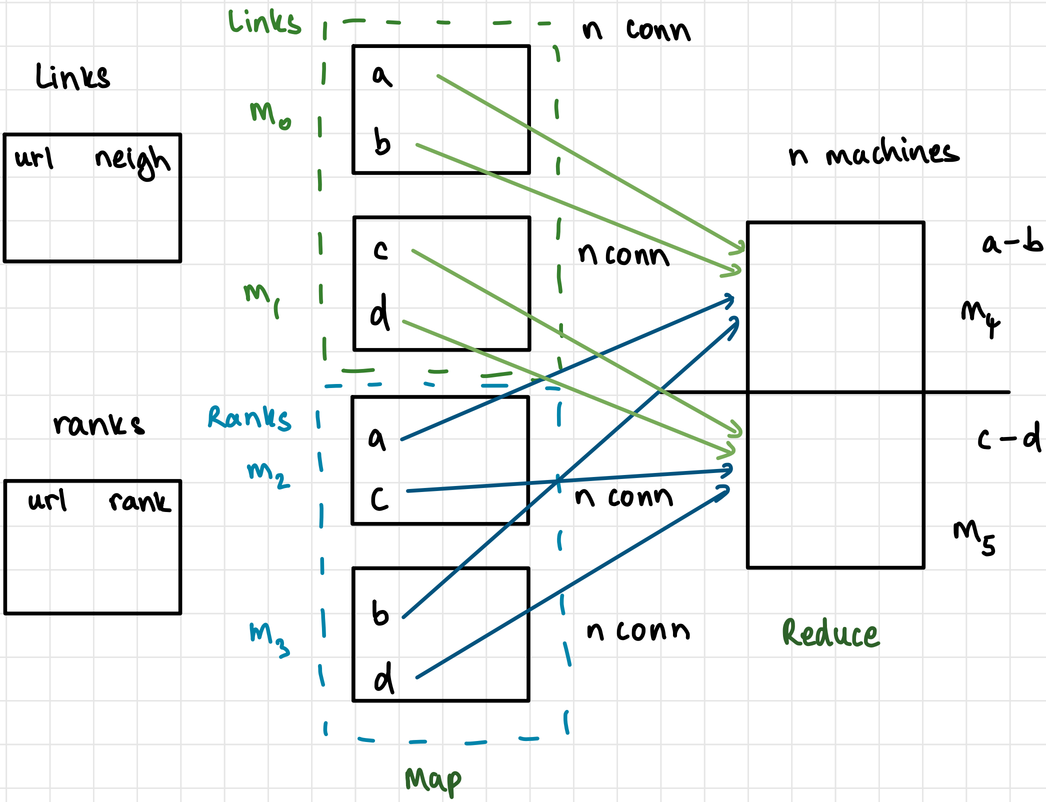
```
lines = textfile("urls.txt")
links = lines.map (lambda urls: urls.split()).groupByKey().cache()

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)
```



Note: ranks & links spread across multiple nodes. How does spark ensure join works properly?

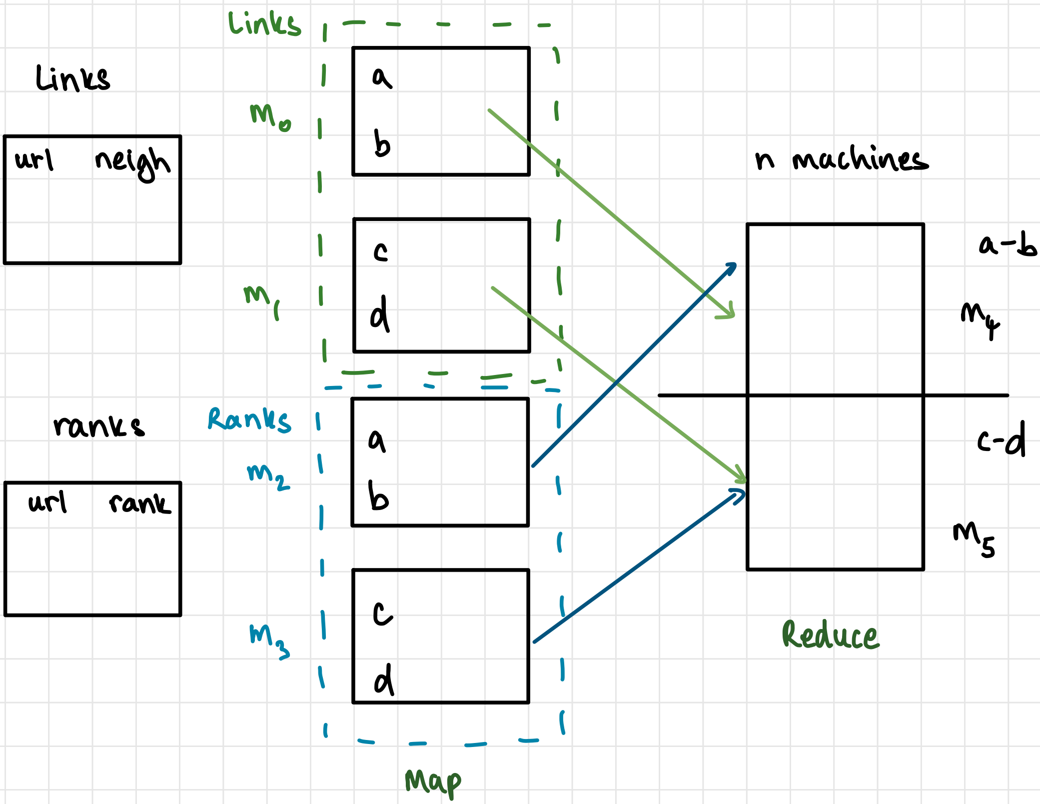
Links join Ranks

- wide dependency



- Requires full shuffle over network
- Each worker depends on all parents
- Make more efficient?
  - copartition
- If Links & Ranks partitioned with same function

- Narrow dependency

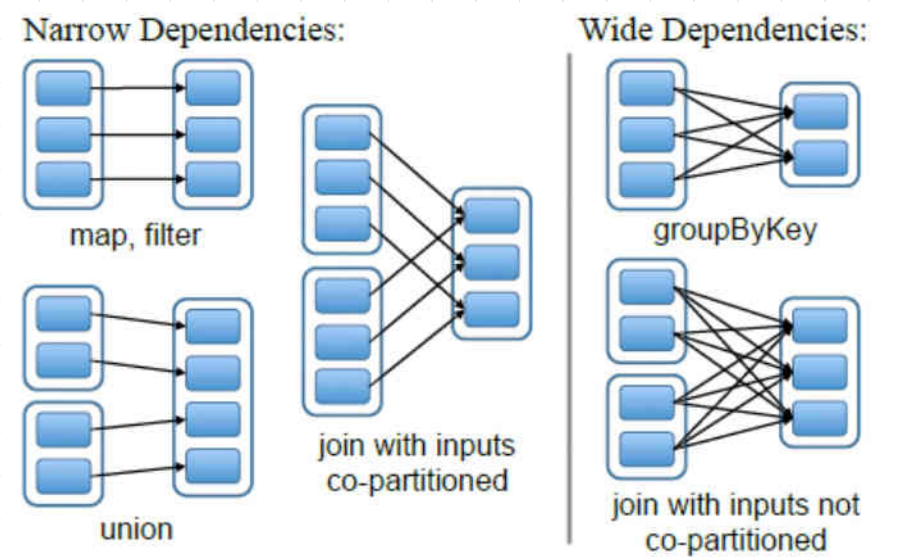


Wide & Narrow Partitioning

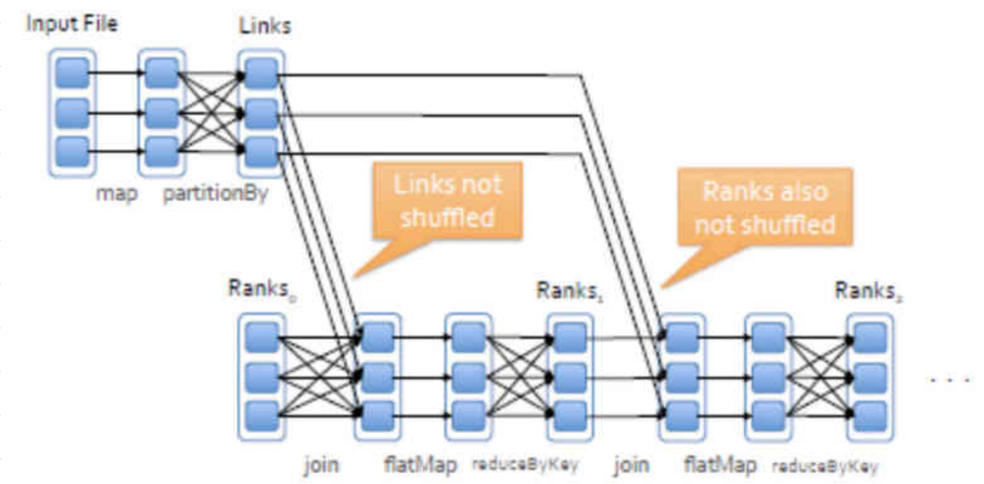
1. Narrow
   - each partition of parent RDD used by at most one partition of child RDD
   - No shuffle ; pipeline operations

2. Wide
   - multiple child partitions may depend
   - shuffle

Copartition: both join inputs partitioned with same function



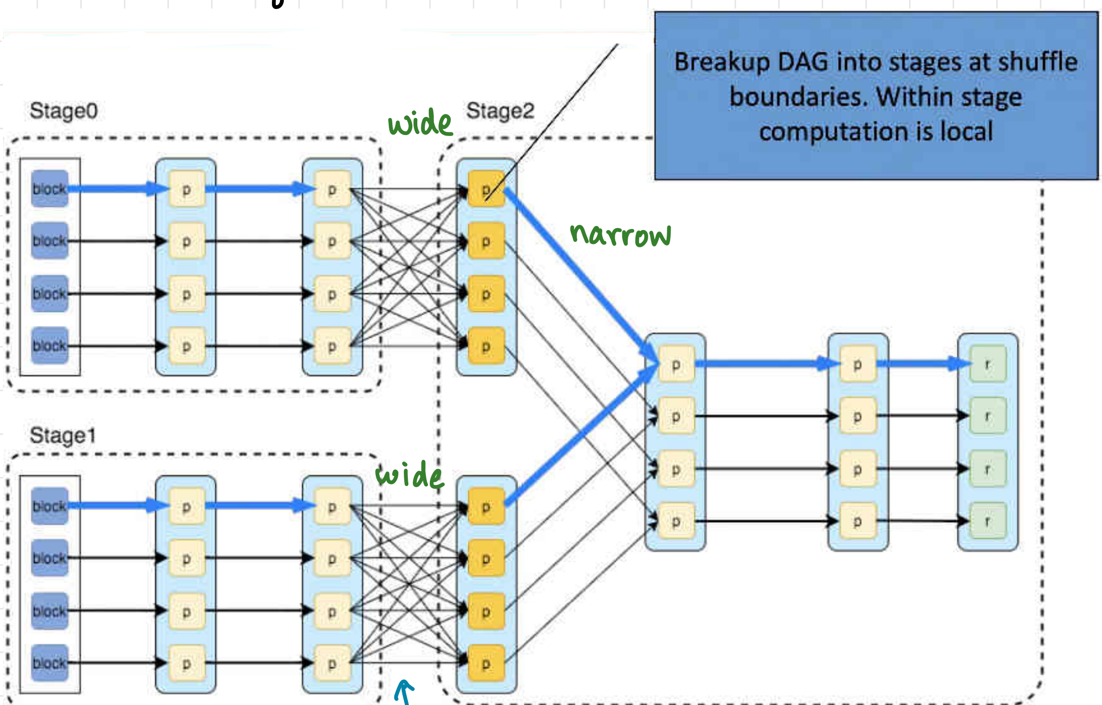
Lineage & Optimising Placement



Narrow	Wide
Map	Intersection
FlatMap	Distinct
MapPartitions	ReduceByKey
Filter	GroupByKey
Sample	Join
Union	Cartesian
	Repartition
	Coalesce

Scheduling

- subgraphs separated by wide partition
- within stages: local



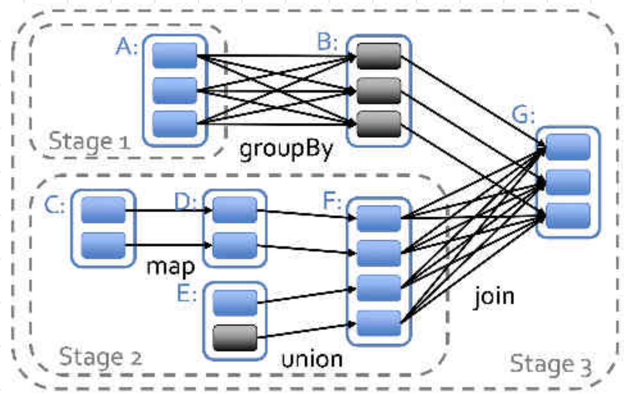
subgraph

stage boundary: wide dependency

- S_0 and S_1 can be scheduled in parallel
- S_2 only after S_0 & S_1 complete

TASK ASSIGNMENT

- Scheduler assigns tasks to machines based on data locality using delay scheduling



- If partition to be processed available in memory on a node, sent to that node
- Otherwise, task processes partition for which the containing RDD provides preferred location and sends to those

Dataframes

- RDDs opaque to Spark → cannot parse
- Spark must understand format

- Distributed collection of data into named columns (abstraction over RDDs)

```
from pyspark.sql import SQLContext

sqlContext = SQLContext(sc)

# DataFrames can be created from existing RDDs,
# HIVE tables or other data sources.
# Here, from JSON file
df = sqlContext.jsonFile("pes/students.json")

# Display the contents
df.show()

## USN  name   marks
## 045  Vkoli    11
## 010  Stendul   43
## 195  Abachpan28


# Alternatively, from an existing RDD by naming
# the columns
Df = rdd.toDF("USN", "name")
```

Using 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 age by 1
df.select("name", df.marks + 1).show()
## name      (marks + 1)
## Vkoli        12
## STendul      44
## ABachpan     29
```

Q: 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

(a) How will you do it in Spark?

(b) How will you do it with Spark Data frames?

(a) reduceByKey((x,y) => x+y) key = pan-number

(b) select sum(tax-paid) from df group by pan-number (pseudocode)

```
df = rdd.toDF("pan_number", "date", "tax paid")

df.select("pan_number", "tax_paid").groupBy("pan_number").sum()
```

Other Tools

- Spark always preferred

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

Big Data Algorithm Complexity

Q: Complexity of matrix multiplication on single node?

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} 1 = O(n^3)$$ — complexity depends on no of computations

Q: Complexity of database query?

hashtable — amortised $O(1)$
bst / rbt — $O(\log n)$
} in-memory

complexity depends on disk reads

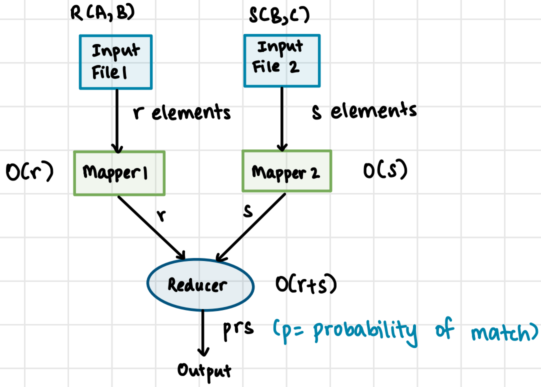
Cost of BD Algorithms

- Algorithms tend to be $O(n)$ (map-reduce)
- Network speed << CPU speed
- Disk speed << CPU speed
- Majorly impacted by communication time

Communication Cost

- Depends on input size
- Final o/p usually smaller by aggregation

Q: Complexity of natural join of R, S

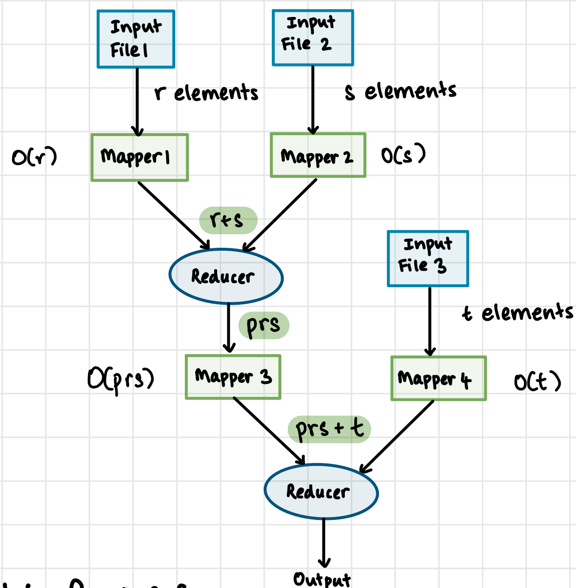


Mapper I/P complexity: r+s
Reducer I/P complexity: r+s

Total complexity: O(r+s)

Q: Complexity of natural join of R, S, T

Case 1:



rs: complexity of cartesian prod

Let p = probability of match for R & S

total cost = O(r+s+t+prs)

Case 2: join S, T and then R

Let q = prob of join of S, T

Total cost: $O(r+s+t+qst)$

If $p \sim q$, join $\min(rs, st, rt)$

Wall Clock Time

- Time taken for entire job to finish
- Linux: `time ls /bin`

  real: wall clock time

  user: time CPU spends executing user code

  sys: time CPU spends executing system (OS) code

Trade-off — Parallelism

- Dividing tasks: reduce wall clock time, increase communication time

- Reducer size q (not no. of reducers)
  - no of unique values with same key

- No of map O/Ps T

- q small, more reducers
  - max reducers T/q
  - reduce WCT, increase CT

- Replication rate r
  - r = (no of kv pairs in mapper o/p) / (no of input records to mapper)
  - $\propto$ avg CT from M tasks to R tasks

Wall Clock Example – SIMILARITY JOIN BETWEEN IMAGES

- DB of 10^6 images, 1 MB each (1 TB DB)
- Similarity function $s(x,y)$ on images x and y such that $s(x,y) = s(y,x)$
- Output all x, y ST $s(x,y) > t$

Naive Algorithm

- Each image P_i has index i
- Mapper
  - reads (i, P_i)
  - generates all possible pairs $(\{i,j\}, \{P_i, P_j\})$ $(i \neq j)$
- Reducer
  - reads $(\{i,j\}, \{P_i, P_j\})$
  - computes $s(P_i, P_j)$
- Communication cost of naive algorithm?
  - $O(n^2)$ where n = no. of images (to generate pairs)
- Parallelism of naive algorithm
  - potentially very high as reducer size very high & each can be processed in parallel
- Replication rate of naive algorithm?
  - O/P of mapper / I/P to mapper
  - $O(n)$

Alternate Solution (Low Communication Cost)

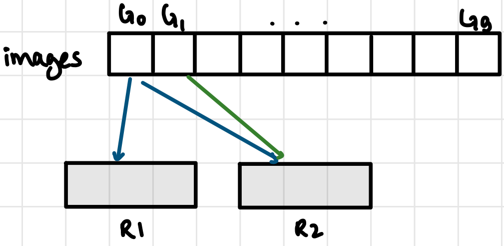
- Reducer runs on same node as mapper
- Very low parallelism

Summary - 2 Choices

1. One pair to each reducer
   - High communication cost (bad)
   - High parallelism (good)
2. Do everything on one node
   - Low parallelism (bad)
   - Low communication cost (good)
3. Something in between?

Group-Based Algorithm

- Group images



Suppose the groups are G0,

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,~~

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)

- No. of groups = g
- No. of images per group = $m = n/g$
- Each group sent to $g-1$ servers
- Total no. of messages = $g(g-1)$
- Total data = $mg(g-1) = n(g-1) \sim ng$ = CC
- Parallelism = no. of nodes = $(g-1) + (g-2) + (g-3) + \dots = \frac{g(g-1)}{2} = O(g^2)$
- Or, no. of nodes = ${}^nC_2 = \frac{g(g-1)}{2}$

Q: Suppose we have groups of 100
(a) How many groups are there?
(b) How many nodes is each group sent to?
What is the
(i) Communication cost of the algorithm?
(ii) Parallelism of the algorithm?

(a) 10^6 images, 10^2 per group = 10^4 groups

(b) Each group sent to $10^4 - 1$ nodes

(i) CC = ng = 10^6 images $\times$ 10^4 groups = 10^{10}

(ii) parallelism $\sim (10^4)^2 = 10^8$