General Purpose Computing Systems II: In-memory Computation & Spark

Shiow-yang Wu (吳秀陽)

CSIE, NDHU, Taiwan, ROC

Lecture material is mostly home-grown, partly taken with permission and courtesy from Professor Shih-Wei Liao of NTU.

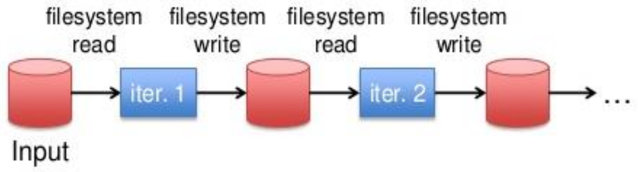
Outline

- Introduction
  - Motivation
  - Solution: In-memory computation
- Challenges
  - Designing a shared data abstraction with
    - Scalability
    - Data locality
    - Fault tolerance
- Resilient Distributed Datasets(RDD)
  - Design policy
  - Programming model
  - Implementation of RDD

Problems with Hadoop MapReduce

- When doing iterative computation
  - Bad performance due to **replication & disk I/O**
  - Even worse: **Communication overheads** in the distributed file system

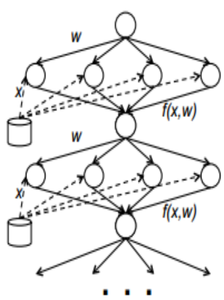
Iterative:



CSIE59830 Big Data Systems In-memory Computation & Spark 3

Problems with Hadoop MapReduce

- MapReduce greatly simplified big data analysis
- But as soon as it got popular, users wanted more:
  - More **complex**, **iterative** multi-pass analytics (e.g. ML, graph)
  - More **interactive** ad-hoc queries
- Requires intensive disk I/O
  - Intermediate data is always written to local disk make poor performance
  - solution: Apache Spark's in-memory computing



CSIE59830 Big Data Systems In-memory Computation & Spark 4

Solution: Keep the Data in Memory

- Apache Spark's in-memory computing
  - 10-100X faster than disk

Iterative:

- Sharing at memory speed

The Working Set Idea

- Peter Denning, "The Working Set Model for Program Behavior", *Communications of the ACM*, May 1968.
  - http://dl.acm.org/citation.cfm?id=363141
- **Idea**: conventional programs generally exhibit a high degree of locality, returning to the same data over and over again.
- Operating system, virtual memory system, compiler, and micro architecture are designed around this assumption!
- Exploiting this observation makes programs run 100X faster than simply using plain old main memory in the obvious way.

Spark

- Exploit the working set idea
- Fast and expressive cluster computing system interoperable with Apache Hadoop
- Improves efficiency through:
  - In-memory computing primitives
  - General computation graphs

→ Up to 100× faster (2-10× on disk)

- Improves usability through:
  - Rich APIs in Scala, Java, Python
  - Interactive shell

→ Often 2-10× less code

- The processing engine of BDAS (Berkeley Data Analytics Stack)

Spark History

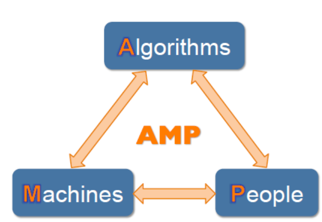
- 2008 – Yahoo! Hadoop team collaboration w Berkeley AMP/RAD Lab begins
- 2009 – Spark example built for Nexus(a common substrate for cluster computing) -> Mesos(a distributed systems kernel)
- 2011 – "Spark is 2 years ahead of anything at Google" – Conviva(a company for online video optimization and analytics) seeing good results w Spark
- 2012 – Yahoo! working with Spark/Shark(now Spark SQL)
- 2013 – donated to Apache

Spark History

- 2014 – became a Top-Level Apache Project
- 2014(Nov) – Databricks(company) set a new world record in sorting using Spark
- 2015 – 1000+ contributors, one of most active Apache projects, one of most active open source big data projects
- 2016 – Spark 2.0 released, Spark SQL one of the best Big Data SQL engines, new Structured Streaming APIs
- 2017 – a unified engine for big data processing

Berkeley Data Analytics Stack(BDAS)

- An open source software stack that integrates software components being built by the Berkeley AMPLab to make sense of Big Data
- The AMPLab was launched at Jan 2011
- Goal: Next Generation of Analytics Data Stack for Industry & Research
  - Berkeley Data Analytics Stack (BDAS)
  - Release as Open Source

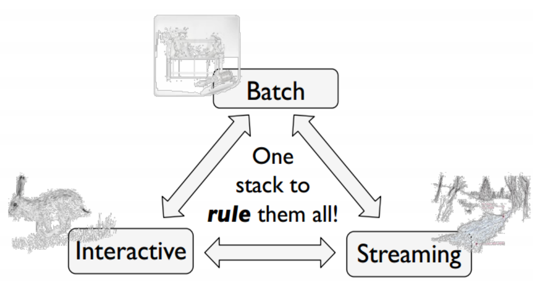


The Berkeley AMPLab

- Funding & Sponsor
  - Government
  - Industry

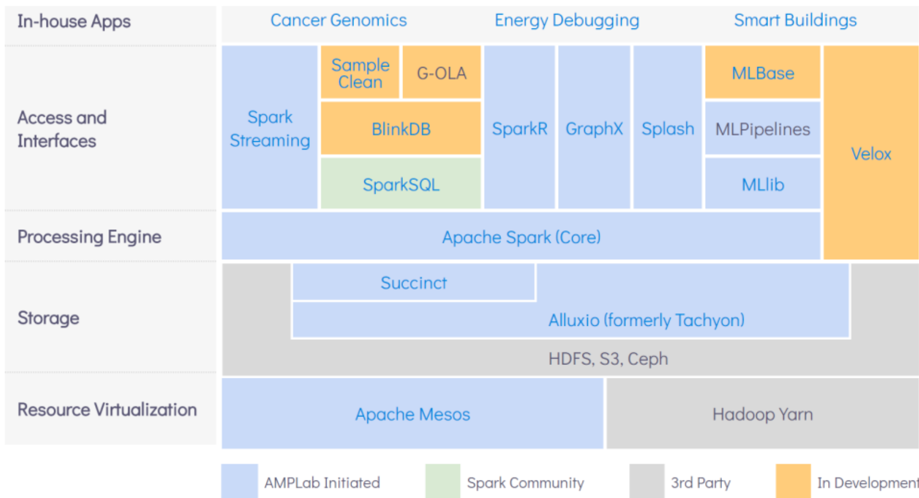
Berkeley Data Analytics Stack

- **Goals**:



  - Easy to combine batch, streaming, and interactive computations
  - Easy to develop sophisticated algorithms
  - Compatible with existing open source ecosystem (Hadoop/HDFS)

Berkeley Data Analytics Stack(BDAS)



BDAS Main Components

- Three main components:
  - **Mesos**: a distributed systems kernel and resource manager that provides efficient resource isolation and sharing across distributed applications, or *frameworks*
  - **Alluxio(Tachyon)**: memory-centric distributed storage system enabling reliable data sharing at memory-speed across cluster frameworks
  - **Spark**: a cluster computing engine that aims to make specified computing(data analytics, ad-hoc) *fast*

Spark Stack

Spark SQL & DataFrames	MLlib	GraphX	Spark Streaming

Spark Core

YARN	Mesos	Standalone

Hadoop HDFS	Cassandra	HBase	S3

CSIE59830 Big Data Systems MapReduce & Hadoop 15

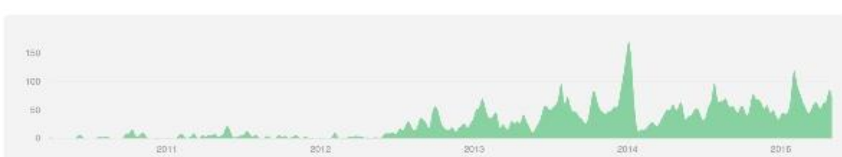
The Spark Community

The Spark Community

Mar 28, 2010 – May 7, 2015

Contributions to master, excluding merge commits

Contributions: Commits



CSIE59830 Big Data Systems In-memory Computation & Spark 16

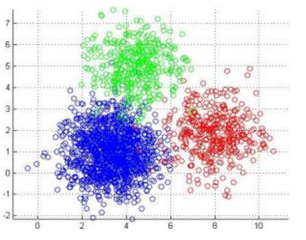
Key Advantages of Spark

- Speed - up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk
- Ease of Use - Write applications quickly in Java, Scala, Python, R.
- Generality - Combine SQL, streaming, and complex analytics.
- Runs Everywhere - Spark runs on Hadoop, Mesos, Kubernetes, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

Additional Goals of Spark

- Low latency (interactive) queries on historical data: enable faster decisions
  - E.g., identify why a site is slow and fix it
- Iterative Analytics: Graph Processing , Machine Learning
  - E.g., PageRank, MaxFlow, K-Means

Max flow in road network

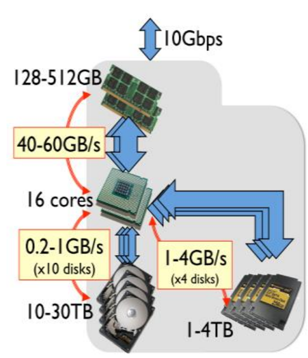


The Working Set Idea in Spark

- The user should identify which datasets they want to access.
- Load datasets into memory, and use them multiple times.
- Keep newly created data in memory until explicitly told to store it.
- Master-Worker architecture: Master (driver) contains the main algorithmic logic, and the workers simply keep data in memory and apply functions to the distributed data.
- The master knows where data is located, so it can exploit locality.
- The driver is written in a functional programming language (Scala) which can be easily parallized.

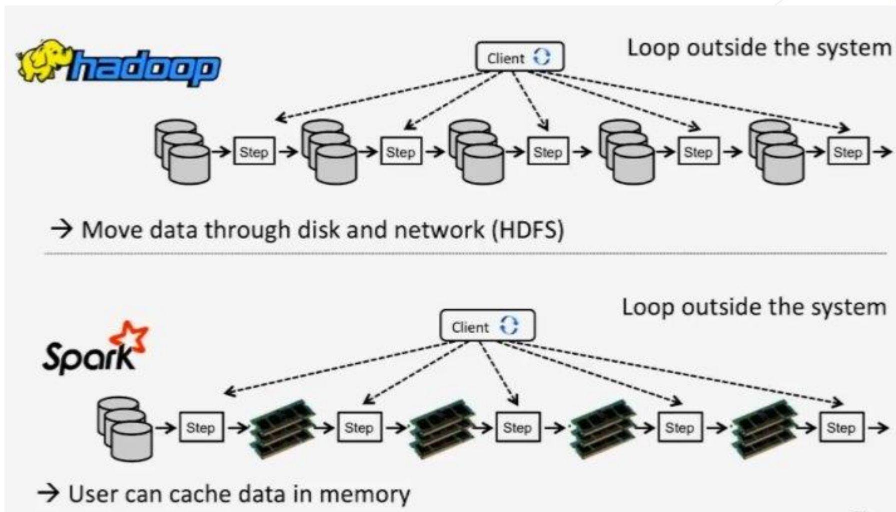
Approach

- Aggressive use of **Memory**
  - Memory transfer rate >> Disk transfer rate
  - Memory density (capacity) still grows with Moore's Law
    - RAM/SSD hybrid memories
  - Many datasets already fit into memory
    - The inputs of over 90% of jobs in Facebook, Yahoo!, and Bing clusters fit into memory
    - E.g., 1TB = 1 billion records @ 1 KB each



High end datacenter node

Spark vs Hadoop



Approach

- Trade between **result accuracy** and **response times**
- Why?
  - In-memory processing does not guarantee interactive query processing
  - E.g., ~10's sec just to scan 512 GB RAM!
  - Gap between memory capacity and transfer rate increasing
- Trade between response time, quality, and cost

Challenges for In-Memory Computation

- Provide distributed memory abstractions for clusters to support apps with working sets
- Retain the attractive properties of MapReduce:
  - **Fault tolerance** (for crashes & stragglers)
  - Data locality
  - Scalability

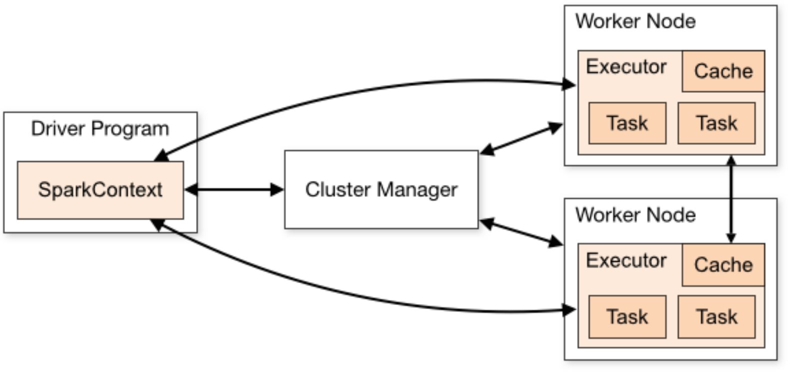
Challenge: Fault Tolerance

- Existing in-memory storage systems have interfaces based on *fine-grained* updates
  - Read/Write to cells
  - E.g. Database, key-value store, distributed memory
- Requires replicating data or logs for fault tolerance
  - Very **inefficient & expensive** under **Big Data**

Challenge:
How to design a distributed memory abstraction that is both ***fault-tolerant*** and ***efficient***?
Solution: Augment data flow model with **RDD**

Spark: Runtime Architecture

- **Driver**: Spark program
- **Worker**: Compute and store distributed data



Resilient Distributed Datasets (RDD)

- **Distributed data abstraction**
  - for in-memory computation on large cluster
- **Read-only**, partitioned records
  - Only way to "write" is to **create** a new RDD
  - Partitions are scattered over the cluster
- Only **coarse-grained** operations are allowed
  - map, join, filter ...
  - operate on the whole dataset

The reasons of the designs will be discussed later.

Resilient Distributed Datasets (RDD)

- **Lazy Evaluation**
  - Two types of perations on RDD: Transformations & Actions
  - **Transformations**: create a new dataset from an existing one
    - do not compute right away but add this record to **Lineage**
    - only computed when an action requires a result
  - **Actions**: return a value after a computation on the dataset
    - It would execute all operation of the **Lineage**

RDD Operations

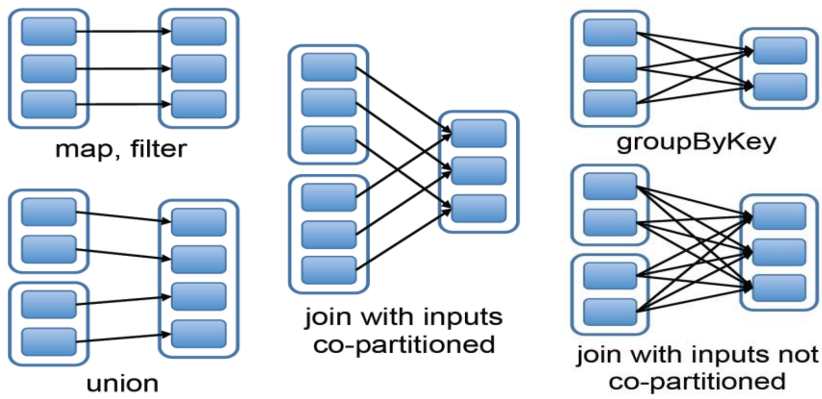
Transformations
- Create a new dataset from an existing one.
- Lazy in nature, executed only when some action is performed.
- Example
  - Map(func)
  - Filter(func)
  - Distinct()

Actions
- Returns a value or exports data after performing a computation.
- Example:
  - Count()
  - Reduce(func)
  - Collect
  - Take()

Persistence
- Caching dataset in-memory for future operations
- store on disk or RAM or mixed
- Example:
  - Persist()
  - Cache()

Transformations

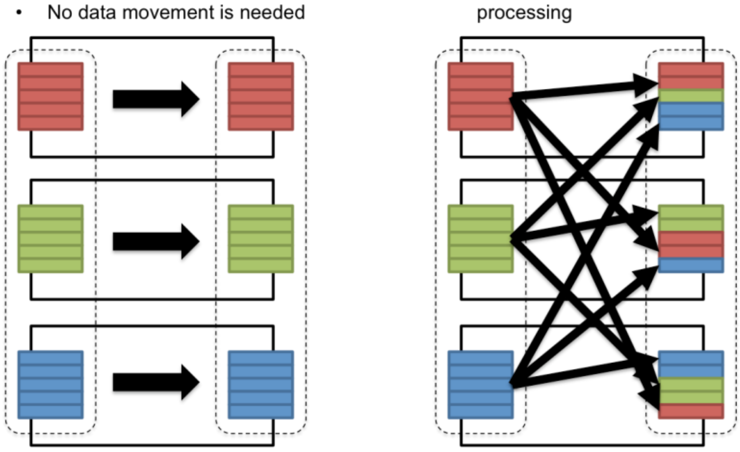
- Immutable data



Narrow vs Wide Transformations

Narrow transformation
- Input and output stays in same partition
- No data movement is needed

Wide transformation
- Input from other partitions are required
- Data shuffling is needed before processing



Operations on RDD

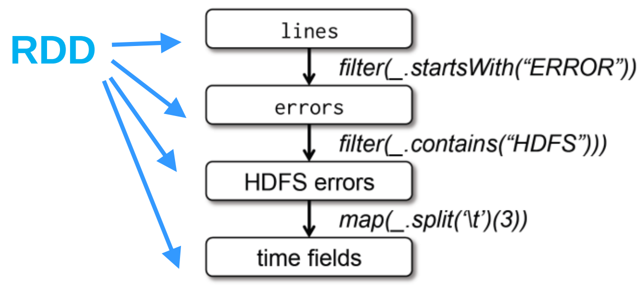
Map **Transformations** (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross …
Reduce **Actions** (return a result to driver program)	collect reduce count save …	

Generality of RDDs

- Spark's combination of data flow with RDDs unifies many proposed cluster programming models
  - *General data flow models:* MapReduce, Dryad, SQL
  - *Specialized models for stateful apps:* Pregel (BSP), HaLoop (iterative MR), Continuous Bulk Processing
- Instead of specialized APIs for one type of app, give user first-class control of distributed datasets

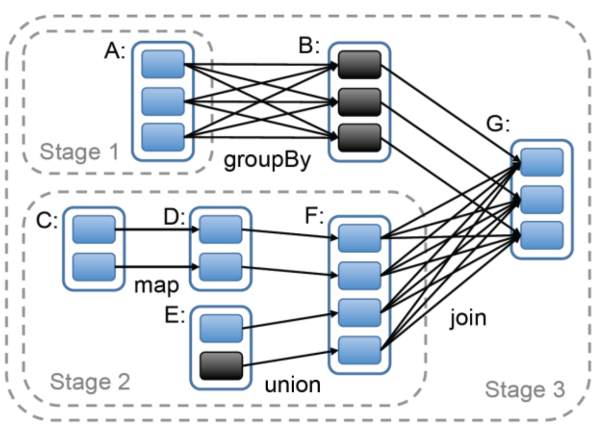
Lineage

- Records of operations to the data(RDDs)
  - Similar to logs
- Maintained by the master node
  - Centralized metadata



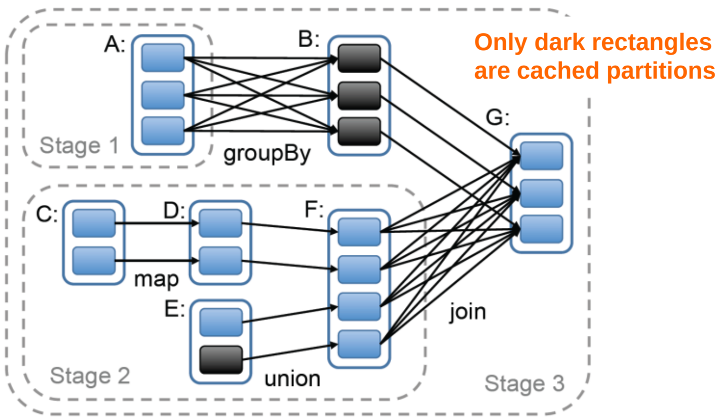
Lineage: Progress of Computation

- Each RDD consists of partitions
  - Detailed lineage structure is a DAG



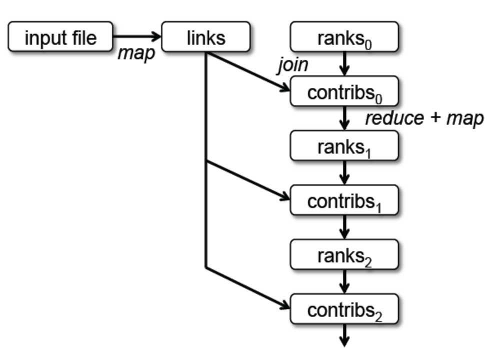
Lineage: Lazy Evaluation

- Partitions of RDDs are not necessarily in RAM
  - Only cached partitions are in preserved



Fault Tolerance using Lineage

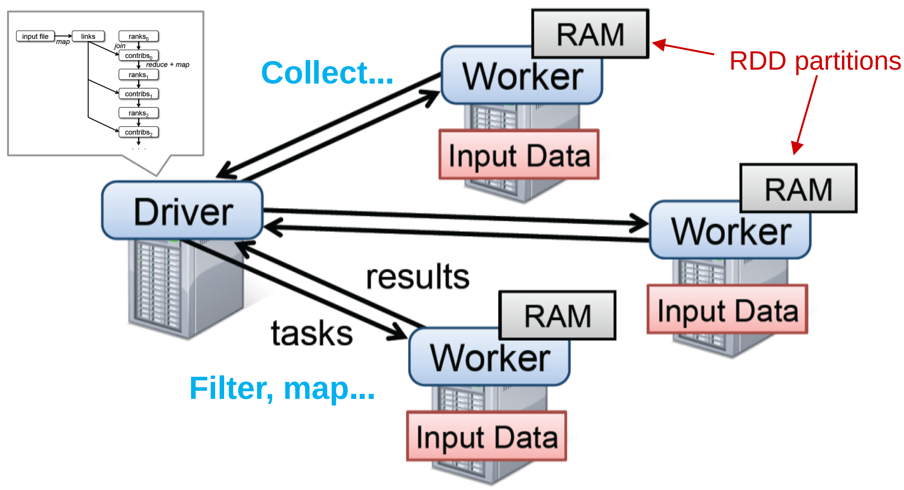
- RDD can only be created (written) from
  - Static Storage
  - Other RDDs
- Only coarse-grained opeations



➡ *Less information to maintain*

➡ *Lost partitions can be re-computed efficiently*

Spark: Runtime Architecture



Other Issue: Dealing with Stragglers

- Speculative Execution
  - Observe the process of the tasks of a job
  - Launch duplicates of those tasks that are slower
  - It then becomes a race between the original and the speculative copies

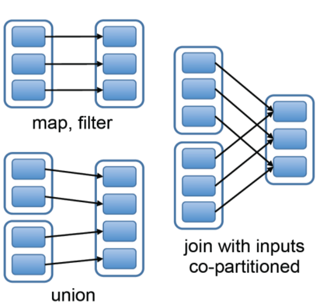
RDDs vs. DSM

Aspect	RDDs	Distr. Shared Mem.
Reads	Bulk or fine-grained	Fine-grained
Writes	Bulk transformations	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Other Issue: Dependency

Narrow Dependency:

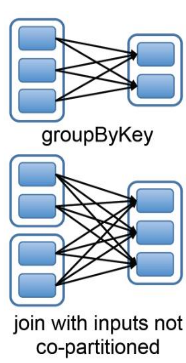
- 1/N-to-1



- Execution can be pipelined
- Faster to recompute

Wide Dependency:

- N-to-N



Other Issue: Memory Management

- Problem:
  - Some RDDs (partitions) are too large to store in some worker's memory
  - These RDDs are costly to re-compute
- Solution: Use hard disks
  - Swap RDDs out under LRU eviction policy
  - Users can set persistence priority to RDDs

Other Issue: Optimization

- Persistence
  - Users can indicate which RDDs they will reuse => save them in memory rather than recomputed
- Partitioning
  - Utilize data locality to optimize transformations
  - Similar to the *partition function* in MapReduce when mapping
  - e.g. partition URLs by domain name

Programming Model

- Resilient Distributed Datasets
  - HDFS files, “parallelized” Scala collections
  - Can be transformed with map and filter
  - *Can be cached across parallel operations*
- Parallel operations
  - Foreach, reduce, collect
- Shared variables
  - Accumulators (add-only)
  - Broadcast variables (read-only)

Resilient Distributed Datasets

- In Spark, RDD is represented by a **Scala object**. There are **four** ways to construct RDD:
  - From file in a shared filesystem, such as HDFS.
  - Scala collection (e.g., an array)
  - Transforming existing RDD
  - Changing the persistence of existing RDD, RDD by default are *lazy* and *ephemeral*(短暫的)
    - cache: hint that the data need to be cache after the first time
    - save: save the dataset to distributed file system (HDFS)

Parallel Operations

- Several parallel operations can be performed on RDD
  - **reduce**: combines dataset elements using an associative function to produce a result at the driver program.
  - **collect**: sends all elements of the dataset to the driver program.
  - **foreach**: Passes each element through a user provided function.

Functional Programming and Stateless

- Using ***Scala***, a functional programming language which runs on JVM
- Recall from the MapReduce session: **stateless properties** of functional programming language is good for parallelization
- That's why RDDs must be built from these semantics.

Example: Log Error Counting

- To count the lines containing errors in a large log file stored in HDFS

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val ones = errs.map(_ => 1)
val count = ones.reduce(_+_)
```

_ means "the default thing that should go here."

- Both `errs` and `ones` are lazy RDDs that are never materialized. Can be made persistent by

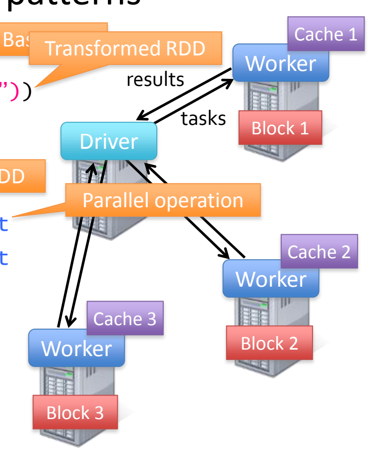
```
val cachedErrs = errs.cache()
```


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

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



Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

RDDs Revisited

- An RDD is an immutable, partitioned, logical collection of records
  - Need not be materialized, but rather contains information to rebuild a dataset from stable storage
- Partitioning can be based on a key in each record (using hash or range partitioning)
- Built using bulk transformations on other RDDs
- Can be cached for future reuse

RDD Fault Tolerance

- RDDs maintain ***lineage*** information that can be used to reconstruct the exact lost partitions
- Ex:

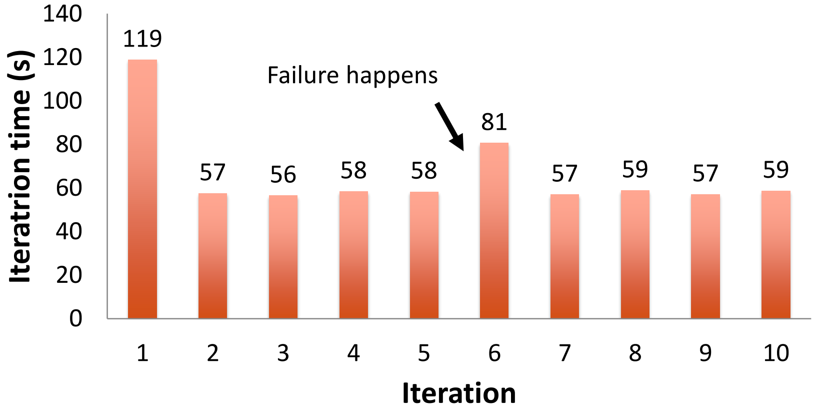
```
cachedMsgs = textFile(...).filter(_.contains("error"))
                          .map(_.split('\t')(2))
                          .cache()
```

HdfsRDD path: hdfs://... ← FilteredRDD func: contains(...) ← MappedRDD func: split(...) ← CachedRDD

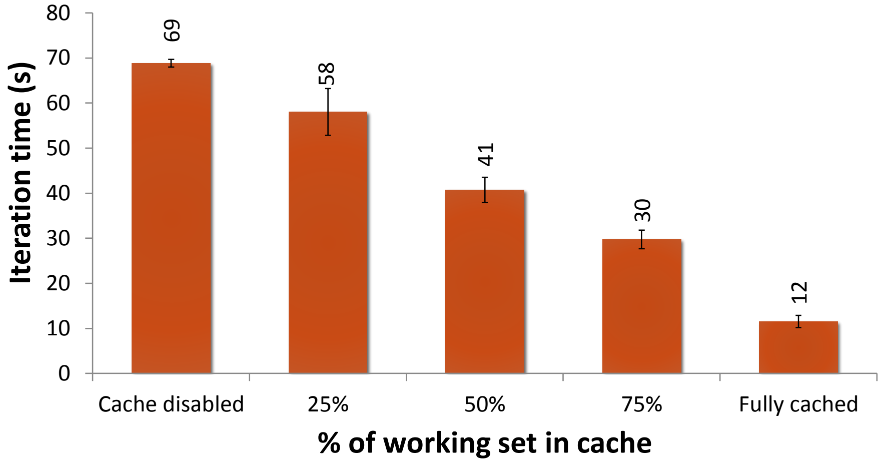
Benefits of RDD Model

- Consistency is easy due to immutability
- Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)
- Locality-aware scheduling of tasks on partitions
- High performance with in-mem computation
- Despite being restricted, model seems applicable to a broad variety of applications

Fault Recovery Test



Behavior with Increasing Cache



Spark in Java and Scala

Java API:

```
JavaRDD<String> lines = spark.textFile(…);

errors = lines.filter(
  new Function<String, Boolean>() {
    public Boolean call(String s) {
      return s.contains("ERROR");
    }
});

errors.count()
```

Scala API:

```
val lines = spark.textFile(…)

errors = lines.filter(
       s => s.contains("ERROR"))
// can also write
// filter(_.contains("ERROR"))

errors.count
```

Which Language to Use?

- Standalone programs can be written in any, but console is only Python & Scala
- **Python developers:** can stay with Python for both
- **Java developers:** consider using Scala for console (to learn the API)
- Performance: Java/Scala will be faster (statically typed), but Python can do well for numerical work with NumPy

Scala Cheat Sheet

Variables:

```
var x: Int = 7
var x = 7      // type inferred

val y = "hi"   // read-only
```

Functions:

```
def square(x: Int): Int = x*x

def square(x: Int): Int = {
  x*x    // last line returned
}
```

Collections and closures:

```
val nums = Array(1, 2, 3)

nums.map((x: Int) => x + 2) // => Array(3, 4, 5)

nums.map(x => x + 2)  // => same
nums.map(_ + 2)       // => same

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

Java interop:

```
import java.net.URL

new URL("http://cnn.com").openStream()
```

More details: scala-lang.org

Learning Spark

- Easiest way: Spark interpreter (spark-shell or pyspark)
  - Special Scala and Python consoles for cluster user
- Runs in local mode on 1 thread by default, but can control with MASTER environment var:

```
$ MASTER=local ./spark-shell                  # local, 1 thread
$ MASTER=local[2] ./spark-shell               # local, 2 threads
$ MASTER=spark://host:port ./spark-shell      # Spark standalone cluster
```


First Step: SparkContext

- Main entry point to Spark functionality
- Created for you in Spark shells as variable **sc**
- In standalone programs, you'd make your own with

Cluster URL, or local / local[N]

App name

```
val sc = new SparkContext(master, appName,
                          [sparkHome], [jars])  // or
val sc = new SparkContext(conf)
```

Spark install path on cluster

List of JARs with app code (to ship)

Creating RDDs

```
// Turn a local collection into an RDD
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
// sc.parallelize(Array(1, 2, 3, 4))

// Load text file from local FS, HDFS, or S3
val distFile = sc.textFile("data.txt")
// sc.textFile("directory/*.txt")
// sc.textFile("hdfs://namenode:9000/path/file")

// Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```


Basic Transformations

```
val nums = Array(1, 2, 3)

// Pass each element through a function
val squares = nums.map(x => x*x)   // => {1, 4, 9}

// Keep elements passing a predicate
val even = nums.filter(x => x % 2 == 0)       // => {4}

// Map each element to zero or more others
nums.flatMap(x => 0 to x-1)  // => {0, 0, 1, 0, 1, 2}
```

Sequence of numbers
0, 1, …, x-1

Basic Actions (in Python)

```
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2)   # => [1, 2]

# Count number of elements
nums.count()   # => 3

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```


Working with Key-Value Pairs

- Spark's "distributed reduce" transformations act on RDDs of *key-value pairs*
- Python:
```
pair = (a, b)
pair[0] # => a
pair[1] # => b
```
- Scala:
```
val pair = (a, b)
pair._1 // => a
pair._2 // => b
```
- Java:
```
Tuple2 pair = new Tuple2(a, b);  // scala.Tuple2
pair._1 // => a
pair._2 // => b
```

Some Key-Value Operations

```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])

pets.reduceByKey(lambda x, y: x + y)
# => {(cat, 3), (dog, 1)}

pets.groupByKey()
# => {(cat, Seq(1, 2)), (dog, Seq(1)}

pets.sortByKey()
# => {(cat, 1), (cat, 2), (dog, 1)}
```

reduceByKey also automatically implements combiners on the map side

Spark for MapReduce

- MapReduce data flow can be expressed using RDD transformations

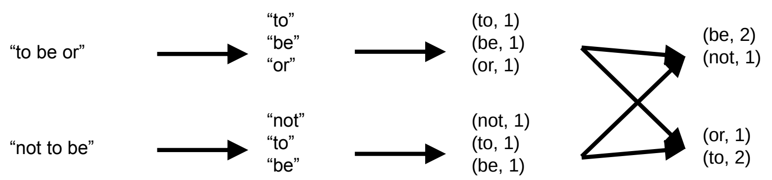
```
res = data.flatMap(rec => myMapFunc(rec))
          .groupByKey()
          .map((key, vals) => myReduceFunc(key, vals))
```

Or with combiners:

```
res = data.flatMap(rec => myMapFunc(rec))
          .reduceByKey(myCombiner)
          .map((key, val) => myReduceFunc(key, val))
```

Example: WordCount

```
// Create RDD from HDFS
file = spark.textFile("hdfs://...")
Counts = file.flatMap(line => line.split(" "))
             .map(word => (word, 1))
             .reduceByKey(_ + _)
// The "map" and "reduce" imply parallelism
```



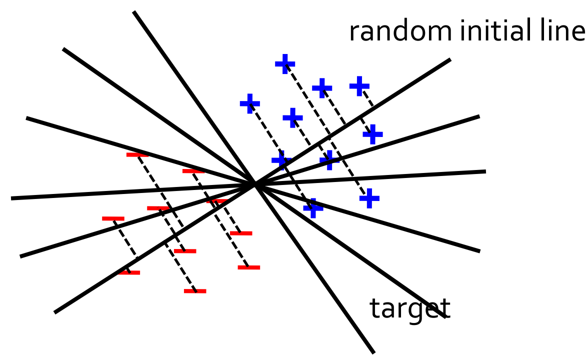
WordCount Complete App

```
import spark.SparkContext
import spark.SparkContext._

object WordCount {
  def main(args: Array[String]) {
    val sc = new SparkContext("local", "WordCount",
                              args(0), Seq(args(1)))
    val lines = sc.textFile(args(2))
    lines.flatMap(_.split(" "))
         .map(word => (word, 1))
         .reduceByKey(_ + _)
         .saveAsTextFile(args(3))
  }
}
```

Example: Logistic Regression

- Goal: find best line separating two sets of points



Example: Logistic Regression

- An iterative classification algorithm to find a hyperplane *w* that best separates two sets of points.
- Popular binary classifier in machine learning
- Gradient Descent
  - ***ITERATIVELY*** minimizes the error by computing the gradient over ***all data points***
  - Computing among data points: parallelization
  - But the iterative instrinsic is another bottleneck

Logistic Regression Algorithm

```
w = random(D) // D-dimensional vector
for i from 1 to ITERATIONS do {
  //Compute gradient
  g = 0  // D-dimensional zero vector
  for every data point (yn, xn) do {
    // xn is a vector, yn is +1 or -1
    g += yn * xn / (1 + exp(yn * w * xn))
  }
  w -= LEARNING_RATE * g
}
```

Very big!!!!!!

Serial Version

```
// Read points from a text file
val points = readData(...)
// Initialize w to a random D-dimensional vector
var w = Vector.random(D)
// Run multiple iterations to update w
for (i <- 1 to ITERATIONS) {
  var gradient = Vector.zeros(D)
  for (p <- points) {
    val s = (1/(1 + exp(-p.y*(w dot p.x)))-1) * p.y
    gradient += s * p.x
  }
  w -= gradient
}
```

Spark Version

```
// Read points from a text file and cache them
val points =
     spark.textFile(...).map(parsePoint).cache()
// Initialize w to a random D-dimensional vector
var w = Vector.random(D)
// Run multiple iterations to update w
for (i <- 1 to ITERATIONS) {
  var gradient = spark.accumulator(new Vector(D))
  for (p <- points) { // Run in parallel
    val s = (1/(1 + exp(-p.y*(w dot p.x)))-1)*p.y
    gradient += s * p.x
  )
  w -= LEARNING_RATE * gradient
}
```

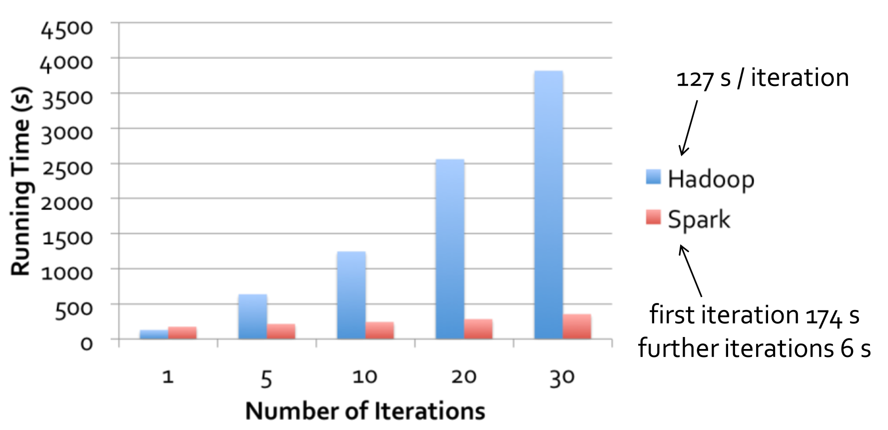

Spark: FP Version

```
// Read points from a text file and cache them
val points =
     spark.textFile(...).map(parsePoint).cache()
// Initialize w to a random D-dimensional vector
var w = Vector.random(D)
// Run multiple iterations to update w
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1/(1 + exp(-p.y*(w dot p.x)))-1) * p.y * p.x
  ).reduce(_ + _)
  w -= LEARNING_RATE * gradient
}
```

Some Spark Features

- `for(p <- points){body}` is equivalent to `points.foreach(p => {body})` and therefore is an invocation of the Spark's parallel foreach operation
- Accumulator allows results of tasks running on clusters to be accumulated using operators like +=
- Only the driver program can read the accumulator's value

Logistic Regression Performance

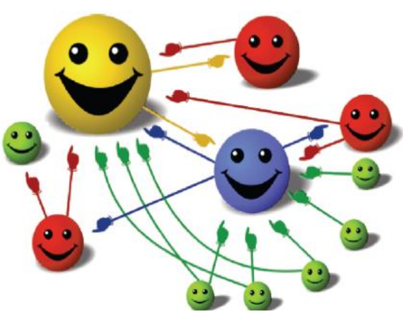


Example: PageRank

- Use PageRank as a Spark example
- Good example of a more complex algorithm
  - Multiple stages of map & reduce
- Benefits from Spark's in-memory computation
  - Multiple iterations over the same data

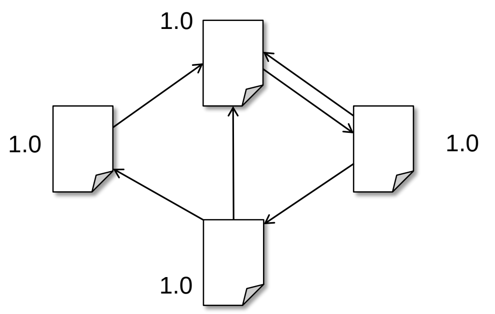
Basic Idea

- Give pages ranks based on links to them
  - Links from mamy pages -> high rank
  - Links from a high rank page -> high rank



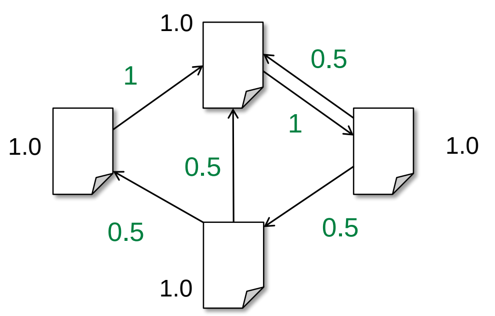
PageRank Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p/|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 085 \times contribs$



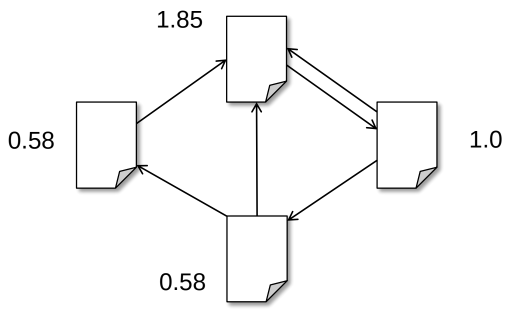
PageRank Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p/|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 085 \times contribs$



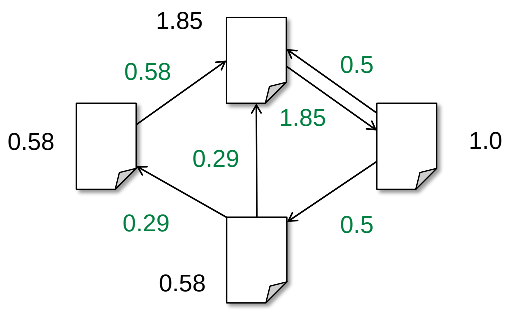
PageRank Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p / |neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 085 \times contribs$



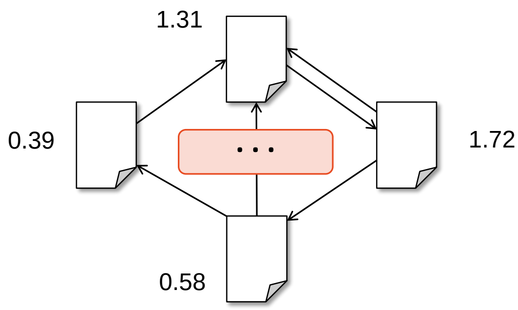
PageRank Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p / |neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 085 \times contribs$



PageRank Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p / |neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 085 \times contribs$

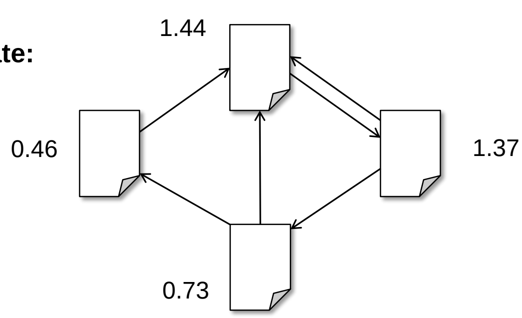


CSIE59830 Big Data Systems In-memory Computation & Spark 81

PageRank Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p / |neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 085 \times contribs$

Final state:

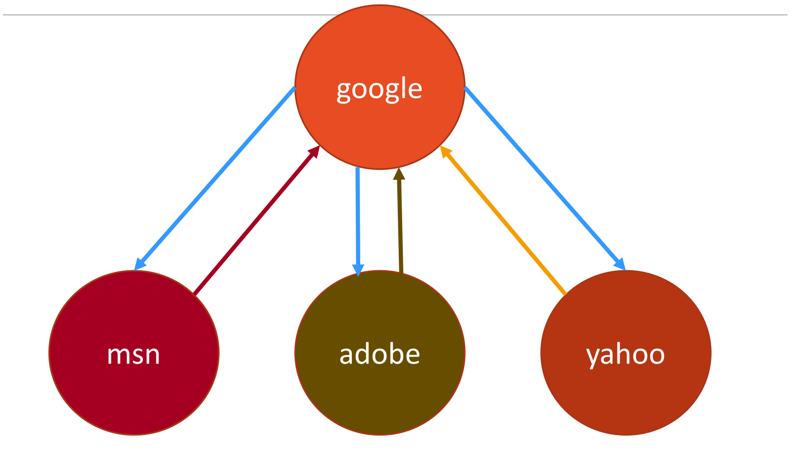


CSIE59830 Big Data Systems In-memory Computation & Spark 82

Spark Program (Scala)

```
val links = // RDD of (url, neighbors) pairs
var ranks = // RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _)
                  .mapValues(0.15 + 0.85 * _)
}
ranks.saveAsTextFile(...)
```


PageRank Example



Spark Program

links-RDD
(google,[Ljava.lang.String;@1771f11)
(yahoo,[Ljava.lang.String;@19897e4)
(msn,[Ljava.lang.String;@11c228f)
(adobe,[Ljava.lang.String;@20f065)

ranks-RDD
(google,0.25)
(yahoo,0.25)
(msn,0.25)
(adobe,0.25)

join

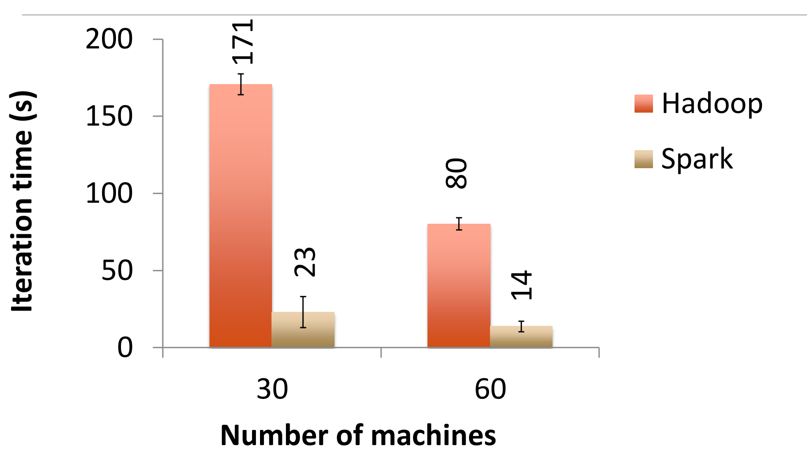
links.join(ranks)-RDD
(google,([Ljava.lang.String;@df1177,0.25))
(msn,([Ljava.lang.String;@f3b9d3,0.25))
(adobe,([Ljava.lang.String;@12cd143,0.25))
(yahoo,([Ljava.lang.String;@15e8963,0.25))

flat Map

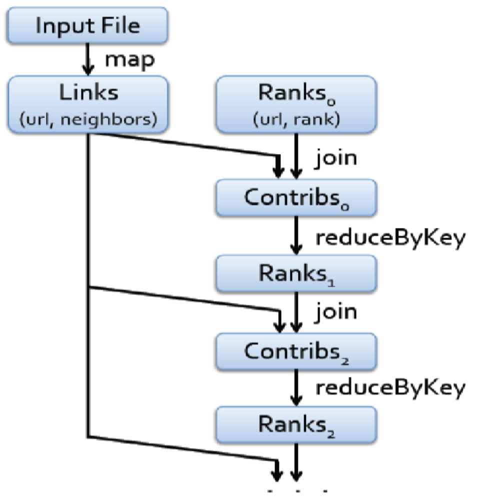
contribs-RDD
(yahoo,0.08333333333333333)
(msn,0.08333333333333333)
(adobe,0.08333333333333333)
(google,0.25)
(google,0.25)
(google,0.25)

reduceByKey

PageRank Performance



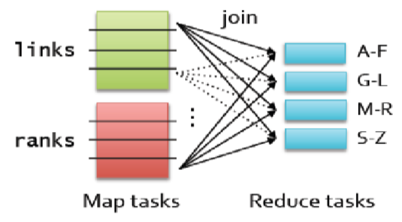
Spark Execution



links and ranks are repeatedly joined

Each join requires a full shuffle over the network

» Hash both onto same nodes



Solution: Controlled Partitioning

- Network bandwidth is ~100× as expensive as memory bandwidth
- *Pre-partition* the **links RDD** -

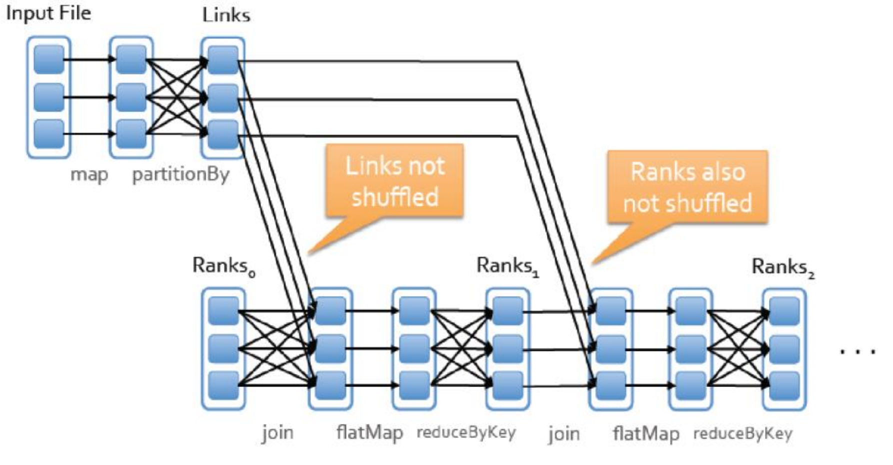
  so that links for URLs with the same hash code are on the same node

Controlled Partitioning

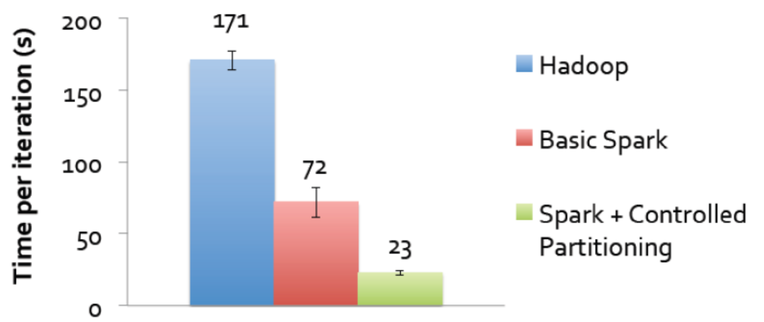
```
val ranks = // RDD of (url, rank) pairs
val links = sc.textFile(...).map(...)
               .partitionBy(new HashPartitioner(8))

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
   .mapValues(0.15 + 0.85 * _)
}
```


New Execution



PageRank Performance



why it helps so much:

links RDD is much bigger in bytes than **ranks RDD**

PageRank Demo

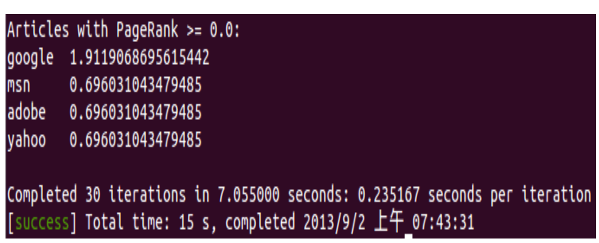
- Input data : simple.dat

google: yahoo msn adobe

yahoo: google

msn: google

adobe: google



```
Articles with PageRank >= 0.0:
google  1.9119068695615442
msn     0.696031043479485
adobe   0.696031043479485
yahoo   0.696031043479485

Completed 30 iterations in 7.055000 seconds: 0.235167 seconds per iteration
[success] Total time: 15 s, completed 2013/9/2 上午 07:43:31
```

- *numberIterations = 30, usePartitioner = false*

PageRank Demo

- *numberIterations = 30, usePartitioner = true*

```
Articles with PageRank >= 0.0:
google  1.9119068695615442
msn     0.696031043479485
adobe   0.696031043479485
yahoo   0.696031043479485

Completed 30 iterations in 3.048000 seconds: 0.101600 seconds per iteration
[success] Total time: 15 s, completed 2013/9/2 上午 07:45:08
```

- *numberIterations = 45, usePartitioner = false*

```
Articles with PageRank >= 0.0:
[error] (run-main) spark.SparkException: Job failed: ShuffleMapTask(3, 0) failed: ExceptionFailure(java.lang.StackOverflowError)
spark.SparkException: Job failed: ShuffleMapTask(3, 0) failed: ExceptionFailure(java.lang.StackOverflowError)
        at spark.scheduler.DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:642)
        at spark.scheduler.DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:640)
        at scala.collection.mutable.ResizableArray$class.foreach(ResizableArray.scala:60)
        at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:47)
        at spark.scheduler.DAGScheduler.abortStage(DAGScheduler.scala:640)
        at spark.scheduler.DAGScheduler.handleTaskCompletion(DAGScheduler.scala:601)
        at spark.scheduler.DAGScheduler.processEvent(DAGScheduler.scala:300)
        at spark.scheduler.DAGScheduler.spark$scheduler$DAGScheduler$$run(DAGScheduler.scala:364)
        at spark.scheduler.DAGScheduler$$anon$1.run(DAGScheduler.scala:107)
[trace] Stack trace suppressed: run last compile:run for the full output.
java.lang.RuntimeException: Nonzero exit code: 1
        at scala.sys.package$.error(package.scala:27)
[trace] Stack trace suppressed: run last compile:run for the full output.
[error] (compile:run) Nonzero exit code: 1
[error] Total time: 16 s, completed 2013/9/2 上午 07:41:44
```


PageRank Demo

- *numberIterations = 45, usePartitioner = true*

```
Articles with PageRank >= 0.0:
google  1.9195314510148316
msn     0.6934895163283893
adobe   0.6934895163283893
yahoo   0.6934895163283893

Completed 45 iterations in 4.366000 seconds: 0.097022 seconds per iteration
[success] Total time: 13 s, completed 2013/9/2 上午 07:37:45
```

Example: Alternating Least Squares (ALS)

- ALS is for collaborative filtering such as predicting u users' ratings for m movies based on their movie rating history.
- A user to a movie has a k-dim feature vector.
- A user's rating to a movie is the dot product of the user's feature vector with the movie's.
- Let M be a m × k matrix and U be a k × u matrix of feature vectors, the rating R can be represented as M × U

ALS Algorithm

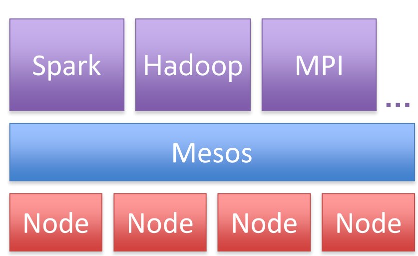
- ALS algorithm:
  1. Initialize M to a random value.
  2. Optimize U given M to minimize error on R.
  3. Optimize M given U to minimize error on R.
  4. Repeat steps 2 and 3 until convergence.
- All steps need R. It is helpful to make R a broadcast variable so that it does not re-sent to each node on each step.

ALS Program in Spark

```
val Rb = spark.broadcast(R)
for (i <- 1 to ITERATIONS) {
   U = spark.parallelize(0 until u)
                  .map(j => updateUser(j, Rb, M))
                  .collect()
   M = spark.parallelize(0 until m)
                  .map(j => updateUser(j, Rb, U))
                  .collect()
}
```


Spark Implementation Overview

- Spark runs on the Mesos cluster manager, letting it share resources with Hadoop & other apps
- Can read from any Hadoop input source (e.g. HDFS)



~6000 lines of Scala code thanks to building on Mesos

Language Integration

- Scala closures are Serializable Java objects
  - Serialize on driver, load & run on workers
- Not quite enough
  - Nested closures may reference entire outer scope
  - May pull in non-Serializable variables not used inside
  - Solution: bytecode analysis + reflection
- Shared variables implemented using custom serialized form (e.g. broadcast variable contains pointer to BitTorrent tracker)

Interactive Spark

- Modified Scala interpreter to allow Spark to be used interactively from the command line
- Required two changes:
  - Modified wrapper code generation so that each "line" typed has references to objects for its dependencies
  - Place generated classes in distributed filesystem
- Enables in-memory exploration of big data

Conclusion

- By making distributed datasets a first-class primitive, Spark provides a simple, efficient programming model for stateful data analytics
- RDDs provide:
  - Lineage info for fault recovery and debugging
  - Adjustable in-memory caching
  - Locality-aware parallel operations
- Spark can be the basis of a suite of batch and interactive data analysis tools

Assignment 2a

- Implement the PageRank algorithm with Spark and provide suitable input to test it.
- Given a set of house owners information in the format:

  OwnerID, HouseID, Zip, Value

  Write a Spark program to compute the average house value of each zip code.

Assignment 2b

- Write a Spark program to compute the inverted index of a set of documents. More specifically, given a set of (DocumentID, text) pairs, output a list of (word, (doc1, doc2, …)) pairs.

- Due date: 3 weeks