Introduction to Apache Spark

Slides from: Patrick Wendell - Databricks

What is Spark?

Fast and **Expressive** Cluster Computing
Engine Compatible with Apache Hadoop

Up to **10×** faster on disk,
100× in memory

2-5× less code

Efficient

- General execution graphs
- In-memory storage

Usable

- Rich APIs in Java, Scala, Python
- Interactive shell

Spark

Spark Programming Model

Key Concept: RDD's

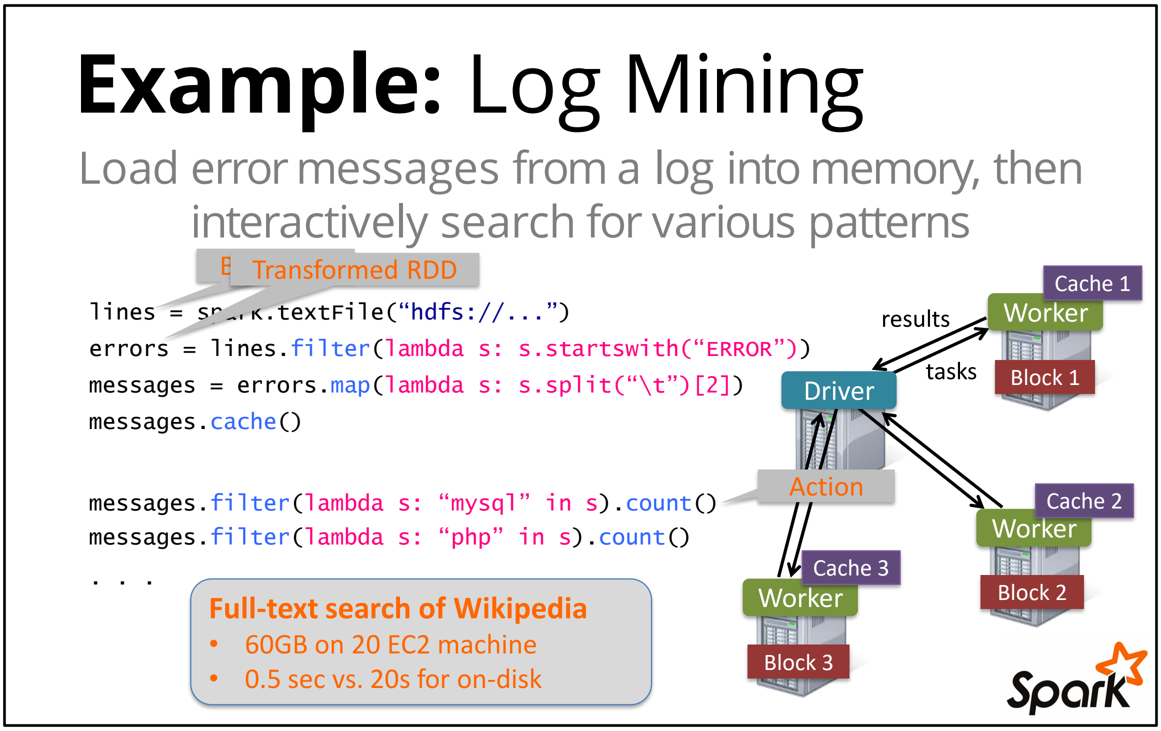
Write programs in terms of **operations** on **distributed datasets**

Resilient Distributed Datasets

- Collections of objects spread across a cluster, stored in RAM or on Disk
- Built through parallel transformations
- Automatically rebuilt on failure

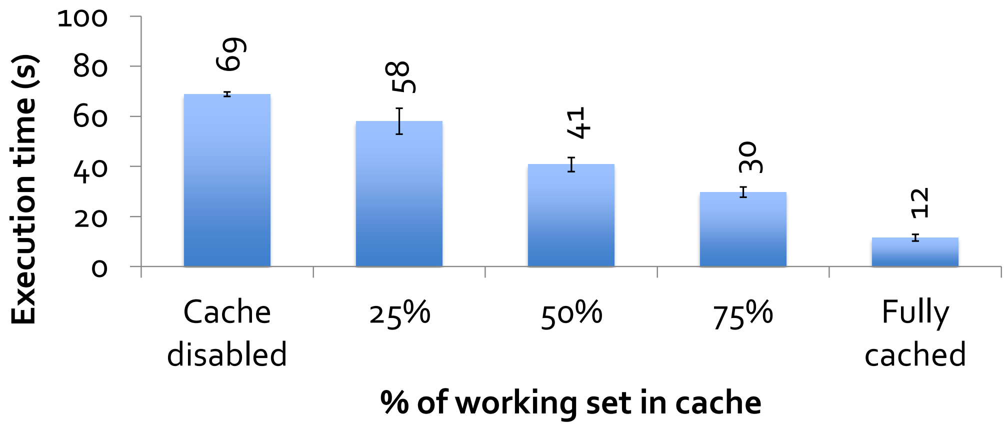
Operations

- Transformations (e.g. map, filter, groupBy)
- Actions (e.g. count, collect, save)



Lazy evaluation: Spark doesn't really do anything until it reaches an action! This helps Spark to optimize the execution and load only the data tat is really needed for evaluation.

Impact of Caching on Performance



Fault Recovery

RDDs track *lineage* information that can be used to efficiently recompute lost data

```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))
               .map(lambda s: s.split("\t")[2])
```

Spark

Programming with RDD's

SparkContext

- Main entry point to Spark functionality
- Available in shell as variable `sc`
- In standalone programs, you'd make your own

Creating RDDs

```
# Turn a Python collection into an RDD
> sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
> sc.textFile("file.txt")
> sc.textFile("directory/*.txt")
> sc.textFile("hdfs://namenode:9000/path/file")
```

Basic Transformations

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

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

# Keep elements passing a predicate
> even = squares.filter(lambda x: x % 2 == 0) // {4}

# Map each element to zero or more others
> nums.flatMap(lambda x: => range(x))
    > # => {0, 0, 1, 0, 1, 2}
```

Range object (sequence of numbers 0, 1, ..., x-1)

Basic Actions

```
> 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 operate 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
```

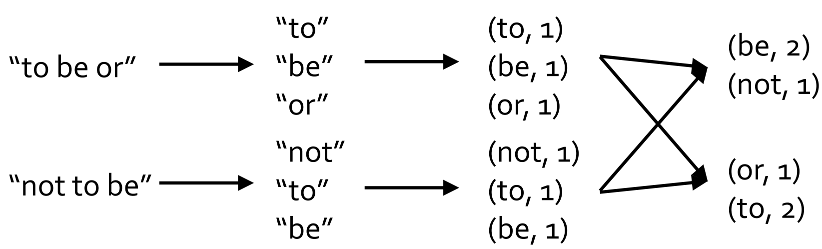
Java:
```
Tuple2 pair = new Tuple2(a, b);
pair._1 // => a
pair._2 // => b
```

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

Word Count (Python)

```
> lines = sc.textFile(“hamlet.txt”)
> counts = lines.flatMap(lambda line: line.split(“ ”))
                .map(lambda word => (word, 1))
                .reduceByKey(lambda x, y: x + y)
                .saveAsTextFile(“results”)
```



Word Count (Scala)

```
val textFile = sc.textFile("hamlet.txt")

textFile
  .flatMap(line => tokenize(line))
  .map(word => (word, 1))
  .reduceByKey((x, y) => x + y)
  .saveAsTextFile("results")
```

Spark

Word Count (Java)

```
val textFile = sc.textFile(“hamlet.txt”)

textFile
  .map(object mapper {
  def map(key: Long, value: Text) =
    tokenize(value).foreach(word => write(word, 1))
  })
 .reduce(object reducer {
  def reduce(key: Text, values: Iterable[Int]) = {
    var sum = 0
    for (value <- values) sum += value
    write(key, sum)
  })
 .saveAsTextFile(“results)
```

Other Key-Value Operations

```
> visits = sc.parallelize([ (“index.html”, “1.2.3.4”),
                            (“about.html”, “3.4.5.6”),
                            (“index.html”, “1.3.3.1”) ])

> pageNames = sc.parallelize([ (“index.html”, “Home”),
                               (“about.html”, “About”) ])

> visits.join(pageNames)
  # (“index.html”, (“1.2.3.4”, “Home”))
  # (“index.html”, (“1.3.3.1”, “Home”))
  # (“about.html”, (“3.4.5.6”, “About”))

> visits.cogroup(pageNames)
  # (“index.html”, ([“1.2.3.4”, “1.3.3.1”], [“Home”]))
  # (“about.html”, ([“3.4.5.6”], [“About”]))
```

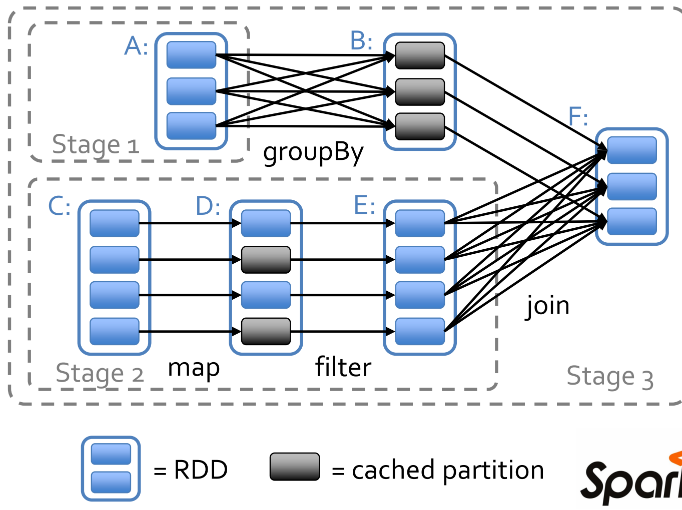
Setting the Level of Parallelism

All the pair RDD operations take an optional second parameter for number of tasks

```
> words.reduceByKey(lambda x, y: x + y, 5)
> words.groupByKey(5)
> visits.join(pageViews, 5)
```

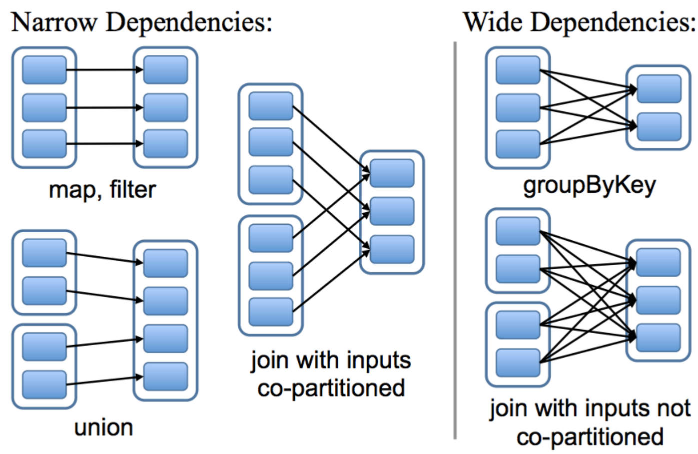
Under The Hood: DAG Scheduler

- General task graphs
- Automatically pipelines functions
- Data locality aware
- Partitioning aware to avoid shuffles



Directed Acyclic Graph (DAG)
A job is broken down to multiple stages that form a DAG.

Physical Operators



Narrow dependency is much faster than wide dependency because it does not require shuffling data between working nodes.

More RDD Operators

- `map`
- `filter`
- `groupBy`
- `sort`
- `union`
- `join`
- `leftOuterJoin`
- `rightOuterJoin`

- `reduce`
- `count`
- `fold`
- `reduceByKey`
- `groupByKey`
- `cogroup`
- `cross`
- `zip`

`sample`

`take`

`first`

`partitionBy`

`mapWith`

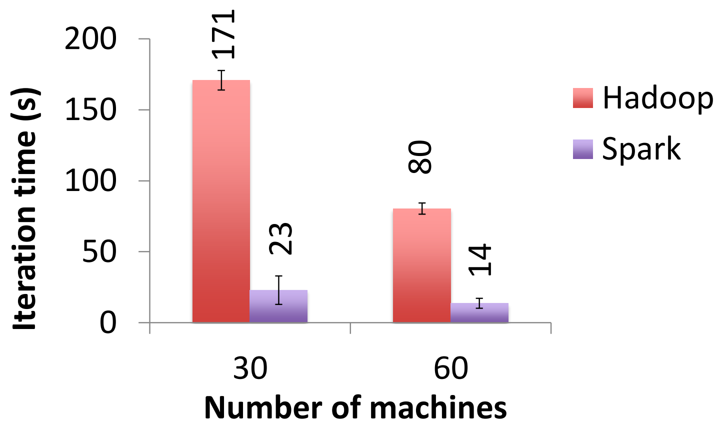
`pipe`

`save` ...


Learn Apache Spark. You Must.

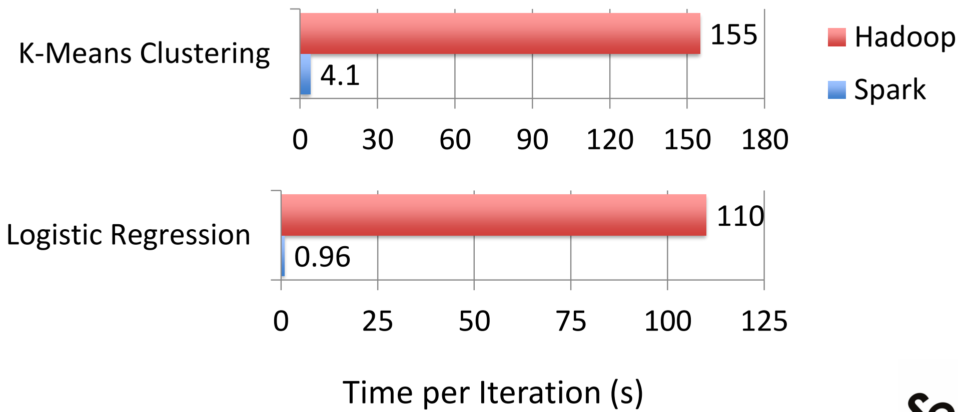
PERFORMANCE

PageRank Performance



Since spark avoids heavy disk i/o, it significantly improves the performance.

Other Iterative Algorithms



Spark outperforms Hadoop in iterative programs because it tries to keep the data that will be used again in the next iteration in memory. In contrast with Hadoop which always read and write from/to disk.

HADOOP ECOSYSTEM AND SPARK

YARN

Hadoop's (original) limitations:

Can only run MapReduce

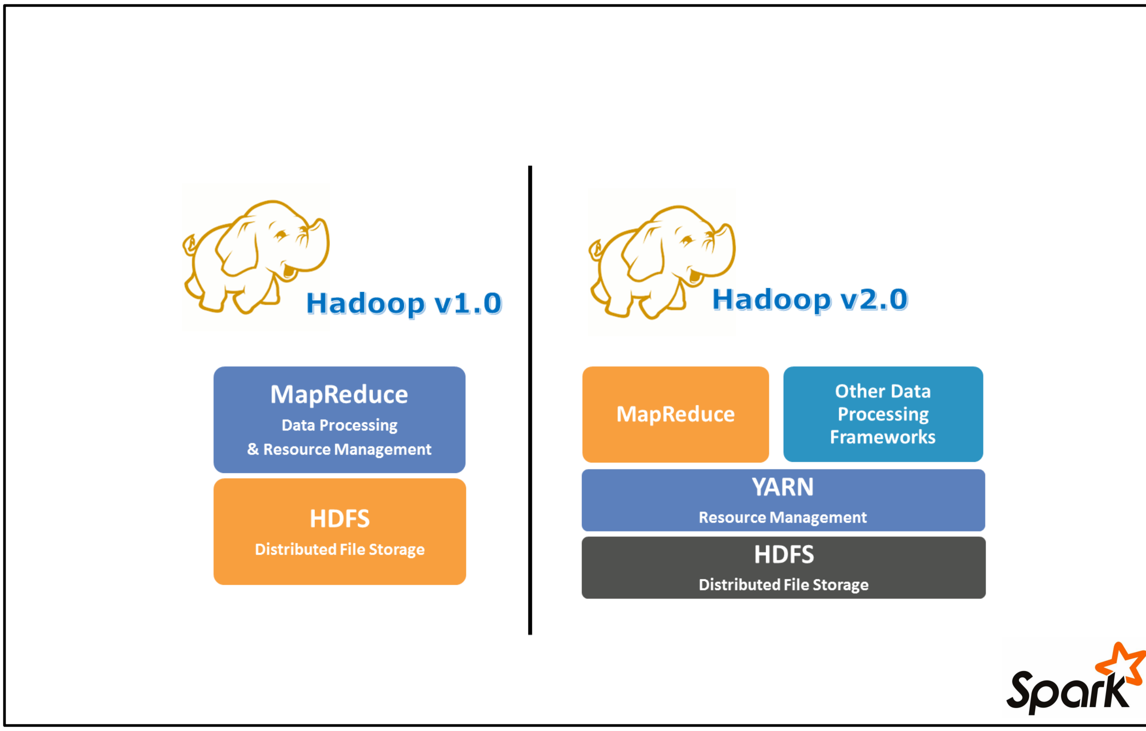
What if we want to run other distributed frameworks?

YARN = Yet-Another-Resource-Negotiator

Provides API to develop any generic distributed application

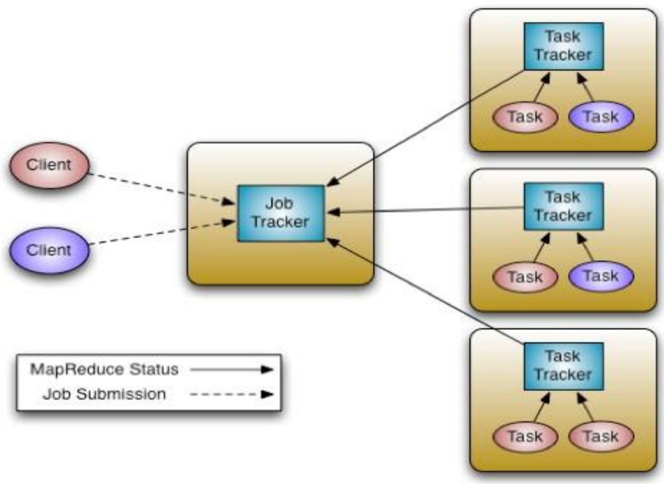
Handles scheduling and resource request

MapReduce (MR2) is one such application in YARN



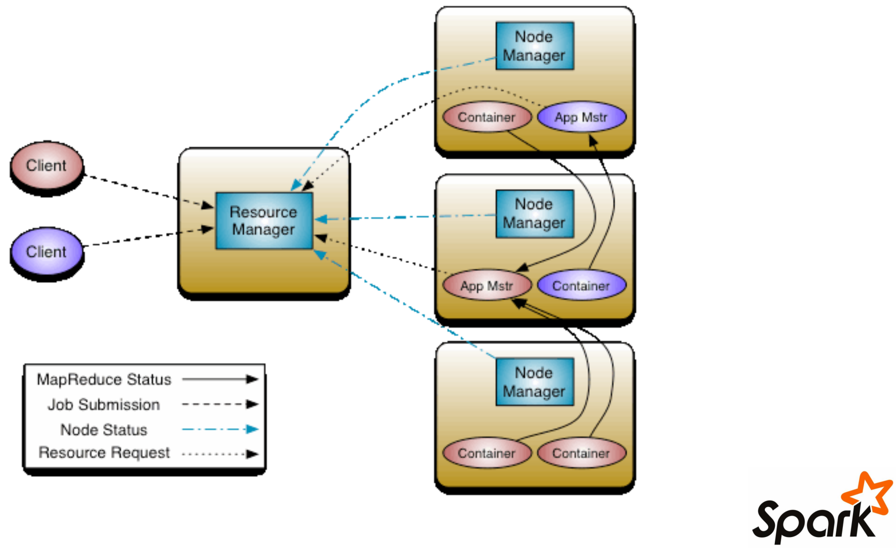
In Hadoop v1.0, the architecture was designed to support Hadoop MapReduce only. But later we realised that it is a good idea if other frameworks can also run on Hadoop cluster (rather than building a separate cluster for each framework). So in v2.0, YARN provides a general resource management system that can support different platforms on the same physical cluster.

Hadoop v1.0



The Job tracker in v1.0 was specific to Hadoop jobs.

Hadoop v2.0



But the resource manager in v2.0 can support different types of jobs (e.g., Hadoop, Spark,…).

Spark Architecture