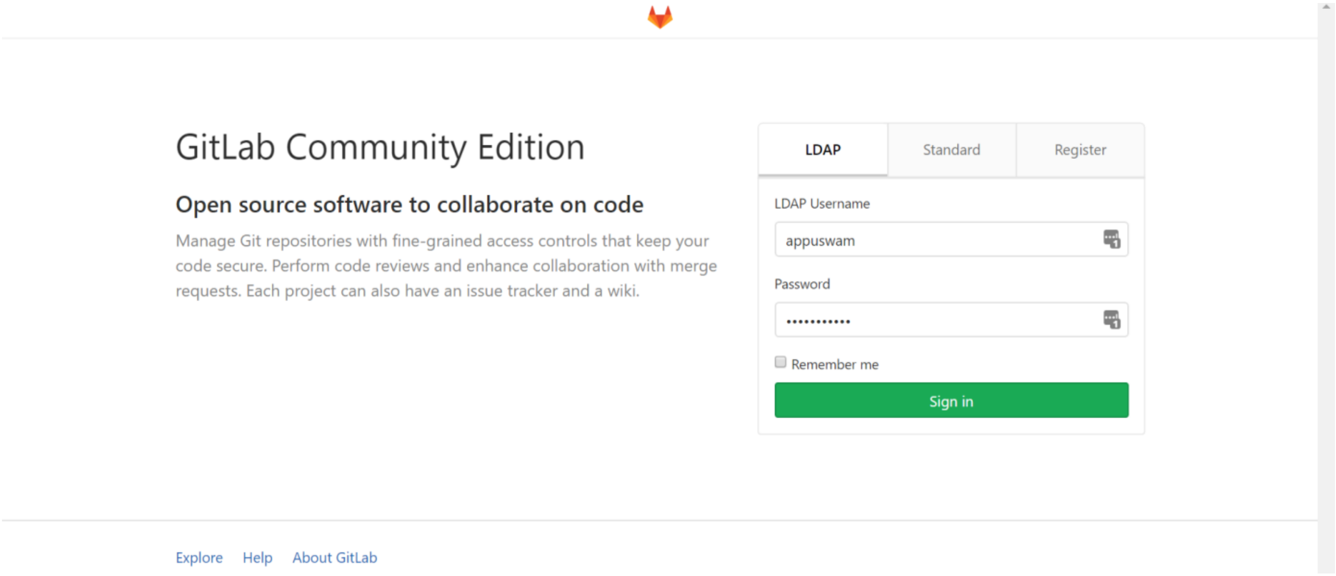
Programming models & runtime: Memory Hierarchy & Spark

Lecture 4

Lab preparation

- Use your credentials to login on gitlab.eurecom.fr (LDAP)
  - ***LAB REQUIRES GITLAB LOGIN***
  - ***DO THIS AS SOON AS POSSIBLE & EMAIL ME IF YOU CANNOT LOGIN***



- Brush up your python skills

Recap

- MapReduce introduced by Google
    - Simple programming model for building distributed applications that process vast amounts of data
    - Runtime for executing jobs on large clusters in a reliable, fault-tolerant manner

- Hadoop makes MapReduce broadly available
    - HDFS becomes central data repository
    - Becomes Defacto standard for batch processing

New applications, new workloads

- Iterative computations
  - Ex: More and more people aiming to get insights from data with machine learning
- Interactive computations, e.g., ad-hoc analytics
  - SQL engines like Hive and Pig drove this trend
- Despite Big Data, many working sets are not big

Memory (GB)	Facebook (% jobs)	Microsoft (% jobs)	Yahoo! (% jobs)
8	69	38	66
16	74	51	81
32	96	82	97.5
64	97	98	99.5
128	98.8	99.4	99.8
192	99.5	100	100
256	99.6	100	100

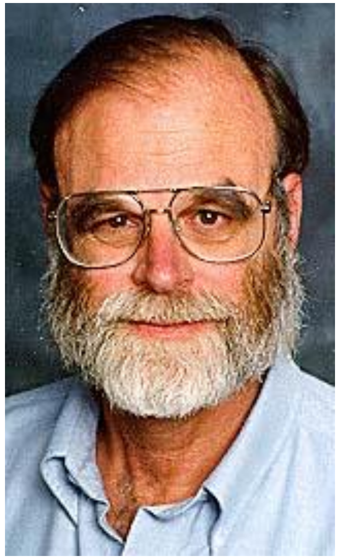
G Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica, ”Disk-Locality in Datacenter Computing Considered Irrelevant”, HotOS 2011

Mapreduce and new workloads

- MapReduce is built for batch processing
  - Entirely disk based: Input and output sit on HDFS
- Let us look at **k-means algorithm, 1 iteration**
  - HDFS Read
  - **Map**(Assign sample to closest centroid)
  - NETWORK Shuffle
  - **Reduce**(Compute new centroids)
  - HDFS Write
- Each iteration reads and writes data from disk-based HDFS
  - To understand why this is bad, let us look at the memory hierarchy

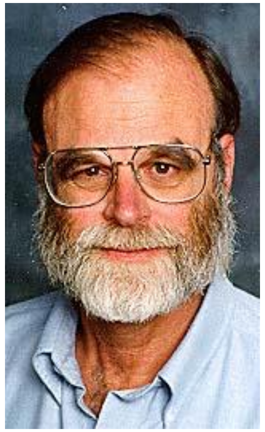
Understanding Memory Hierarchy: How Far Away is the Data?

10^9	Tape	Andromeda	2,000 Years
10^6	Disk	Pluto	2 Years
100	Memory	St. Tropez	1.5 hr
10	On Board Cache	This Building	10 min
1	Registers	This Room	1 min

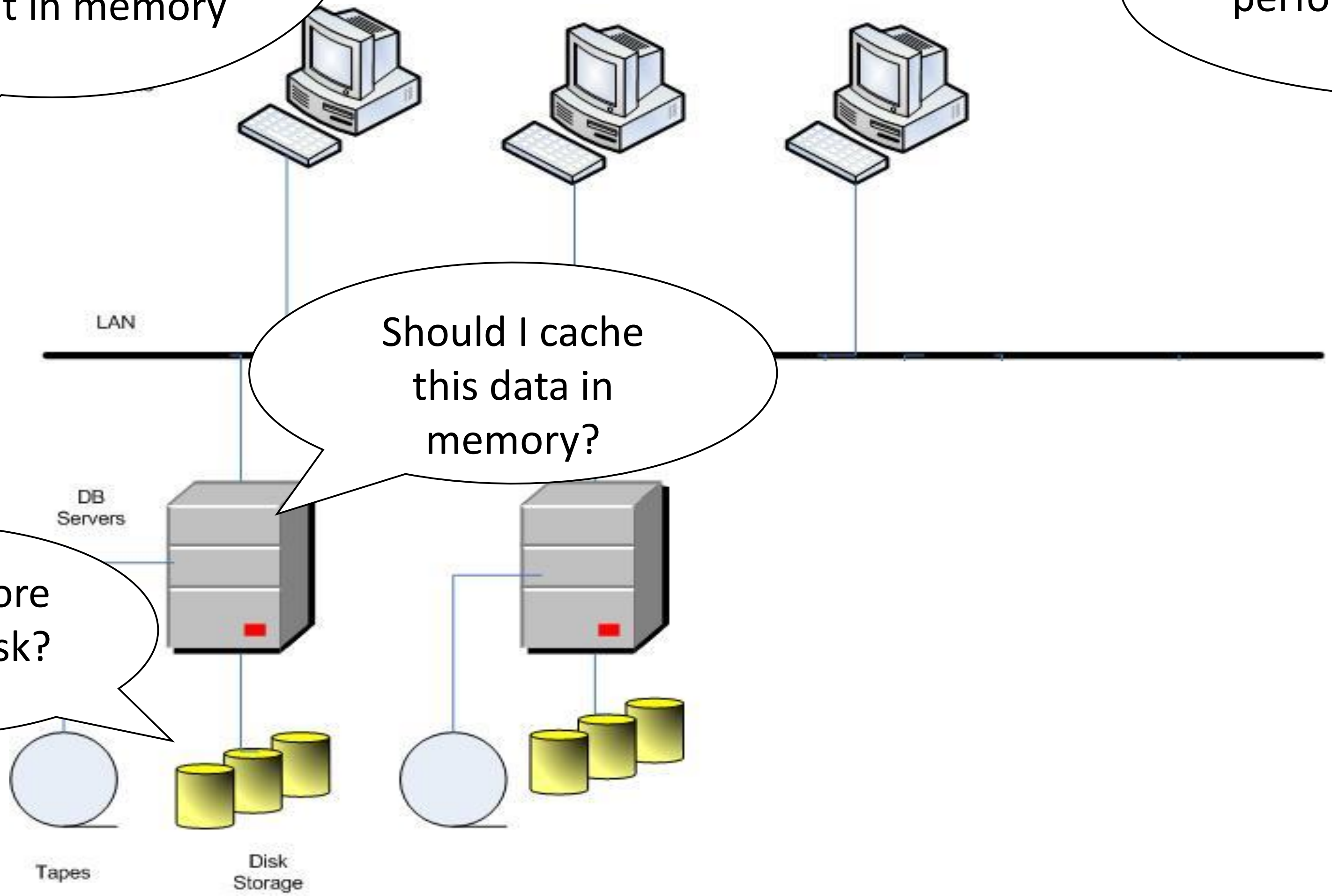


Jim Gray

1980s: Database Administrators Dilemma



If a data is accessed more than once within 5 minutes, cache it in memory



Should I cache this data in memory?

Should I store data on disk?

How do I improve the performance of my DB server?

Tandem Computers: Price/performance

- Tandem disk: **$1k/access**
  - cost: $15k for 180MB
  - performance: 15 accesses / second
- Tandem CPU + supporting hardware: **$1k/access**
  - Cost: $15,000
- Cost of accessing data from disk: **$2k/access**

- Memory cost: $5k for 1MB => **$5/KB**

Five-minute rule

- Cost of accessing data from disk: **$2k/access,** Memory cost: **$5/KB**
- If we keep 1KB in memory, assuming we have 1 access/sec
  - We save $2k of disk i/o by paying $5 for memory
- If we have 1 access every 10 secs => 0.1 access/sec
  - We save $200 of disk i/o by paying $5 for memory

.

.

.

- Break even point : 1 access every 400 secs

400 seconds ~ 5 minutes

Formalizing the five-minute rule

$$\textit{BreakEvenReferenceInterval (seconds)} = (400 \text{ secs})$$

$$\frac{\textit{PagesPerMBofRAM}\ (1024)}{\textit{PricePerMBofDRAM}\ (\$5k)} \times \frac{\textit{PricePerDiskDrive}\ (\$30k)}{\textit{AccessPerSecondPerDisk}\ (15)}$$

- Customer: “What server do I buy for my 500MB, 600 a/s database?”
  - 80/20 rule of data accesses => 64% of accessess to 4% of database
- All-in-memory: 500 MB RAM = **$2.5M**
- Hybrid ram-disk suggested by 5 min rule = **$520k**
  - 4% data in main memory => <u>$100k</u> for 20 MB of RAM
  - Remaining 216 disk accesses (36%) with 14 disks (15 a/sec/disk)
  - Cost per disk = 15 * 2k/a/sec = 30k
  - Overall disk cost = 30k * 14 = <u>$420k</u>

Five-minute rule: then and now

Page size (4KB)	1987	Now
RAM-HDD	5 mins	5 hours

- RAM-HDD break-even 60x higher due to drop in DRAM price
  - Take away: Never ever go to disk!
- See “Five minute rule” CACM paper for more details
  - https://cacm.acm.org/magazines/2019/11/240388-the-five-minute-rule-30-years-later-and-its-impact-on-the-storage-hierarchy/fulltext

MapReduce/Hadoop and memory hierarchy

- Hadoop misaligned with five-minute rule
  - All data is stored in disk
  - Does not cache data in memory even if workload can fit
- Hadoop unfit for new classes of workloads
  - Interactive and iterative applications are bottlenecked by disk
- MapReduce was also too simple a computational model
  - Algorithm design with just map and reduce functions is non trivial

Hadoop ecosystem grows rapidly

General Batching	Specialized systems			
	Streaming	Iterative	Ad-hoc / SQL	Graph
MapReduce	Storm	Mahout	Pig	Giraph
	S4		Hive	
	Samza		Drill	
			Impala	

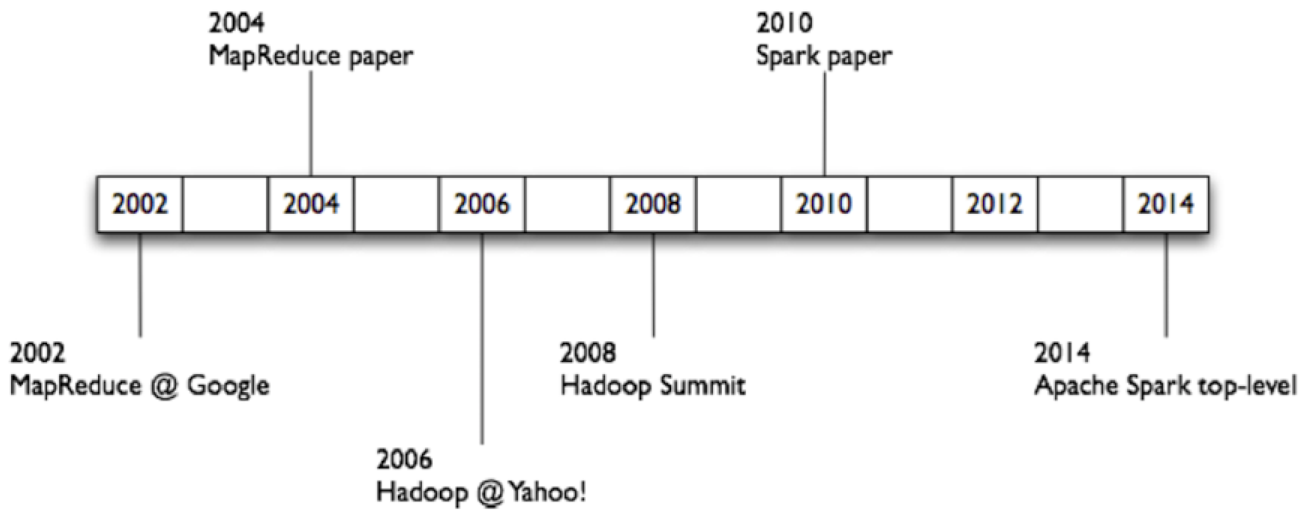
- Specialized systems emerged with no unified vision
  - Diverse APIs, sparse modules, high operational costs

Lighting a Spark

Flexible, in-memory data processing framework written in Scala

Central Ideas

- Exploit memory by caching data to enable fast data sharing
- Generalize the two-stage computational model of mapreduce to a Directed Acyclic Graph-based one that can support a richer API



Spark Fundamentals

Example of an application:

```
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
```

- **Spark Context**
- Resilient Distributed Datasets
- Transformations
- Actions

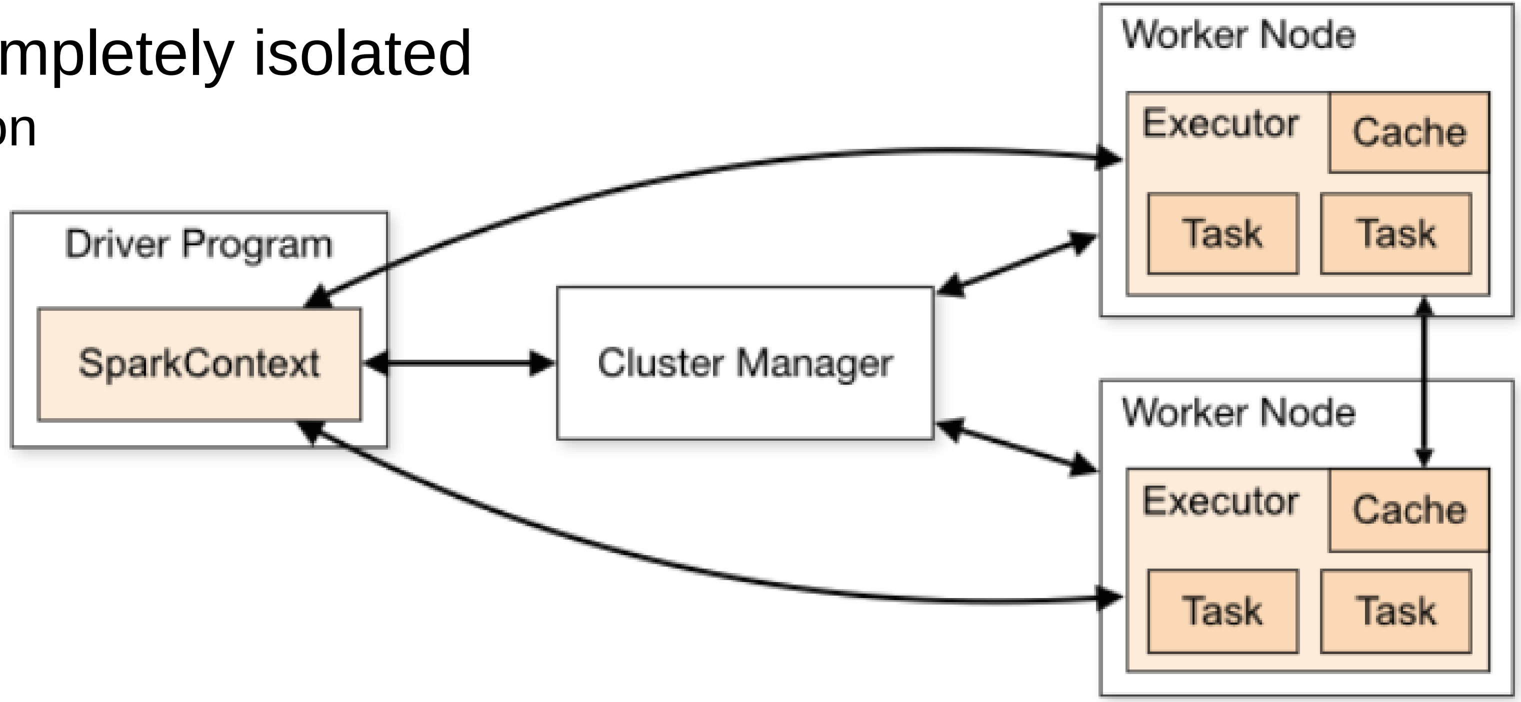
Spark Context

- Every Spark application requires a *spark context*: the main entry point to the Spark API
- Spark Context holds configuration information and represents connection to a Spark cluster
  - Could be local (single threaded or multithreaded)
  - Apache Mesos
  - Hadoop YARN

```
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
```

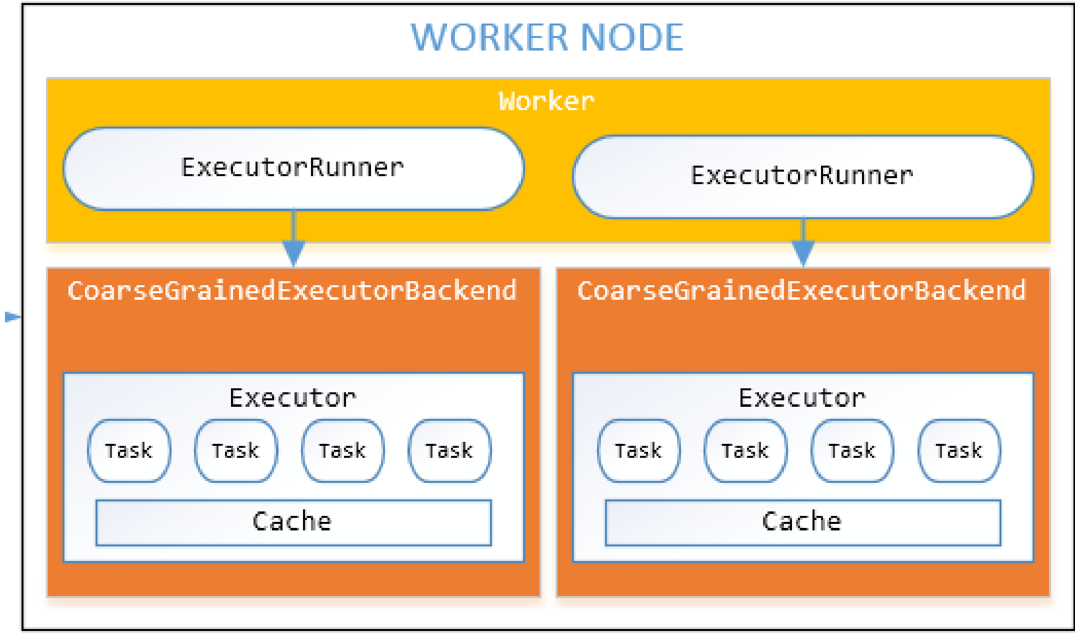
General Workflow

- Spark Application creates SparkContext which inits DriverProgram
- Connects to a cluster manager (manages and allocate resources)
- Acquires executors – worker processes to run computations
- Sends app code and *tasks* for the executors to run
- Benefit: Applications are completely isolated
  - Task scheduling per application



Worker Nodes and Executors

- Worker nodes are machines that run executors
  - Host one or multiple Workers, one JVM (= 1 UNIX process) per Worker
  - Each Worker can spawn one or more Executors
- Executors run tasks
  - Run in child JVM (= 1 UNIX process)
  - Execute one or more tasks using threads in a ThreadPool
- Benefit: Low-overhead
  - Task setup = thread spawning
  - 10-100x faster than running one task per JVM (Hadoop)



Spark Fundamentals

Example of an application:

```
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
```

- Spark Context
- **Resilient Distributed Datasets**
- Transformations
- Actions

Need for a new abstraction

- Need an efficient way to share data stored in memory
- Traditional way: Distributed shared memory abstraction
  - General purpose, extends single-node shared memory to a cluster
  - Applications can make fine-grained updates to any data in memory
  - Can be used to build very efficient applications
- Problem: Fault tolerance
  - Need to replicate data across nodes or log updates which is 10-100x slower than memory write
  - Too expensive for data-intensive apps
- Goal: In-memory abstraction that provides fault-tolerance and efficiency

Resilient Distributed Dataset

RDD (Resilient Distributed Dataset): Restricted form of DSM

- An immutable, partitioned collection of objects
- Can only be built through coarse-grained deterministic transformations

- **RDD are data structures that:**
  - Either point to a direct data source (e.g. HDFS)
  - Apply some transformations to its parent RDD(s) to generate new data elements

```
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
```

Spark Fundamentals

Example of an application:

```
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
```

- Spark Context
- Resilient Distributed Datasets
- **Transformations**
- **Actions**

RDD Operations

Two types of operations

Transformations: Define a new RDD based on current RDD(s)

Actions: return values

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

val file = sc.textFile("hdfs://...")        Transformation

val errors = file.filter(_.contains("ERROR"))        Transformation
    an RDD

errors.cache()

errors.count()        Action
```

RDD Transformations

- Set of operations that define how to transform an RDD
  - Examples: map(), filter(), groupByKey(), sortByKey(), etc.
- As in relational algebra, the application of a transformation to an RDD yields a new RDD
  - RDD are immutable
- Transformations are lazily evaluated
  - Computation that performs the transformation is not performed immediately

RDD Actions

- Actions trigger computation of the chain of transformations
- Some actions only store data to an external data source (e.g. HDFS)
  - Ex: saveAsTextFile(*file*) – save to text file(s)
- Others fetch data from the RDD (and its transformation chain) upon which the action is applied, and convey it to the driver
  - count() – return the number of elements
  - take(*n*) – return an array of the first *n* elements
  - collect()– return an array of all elements
  - …

Lazy Execution of RDDs (1)

Data in RDDs is not processed until an action is performed

File: purplecow.txt

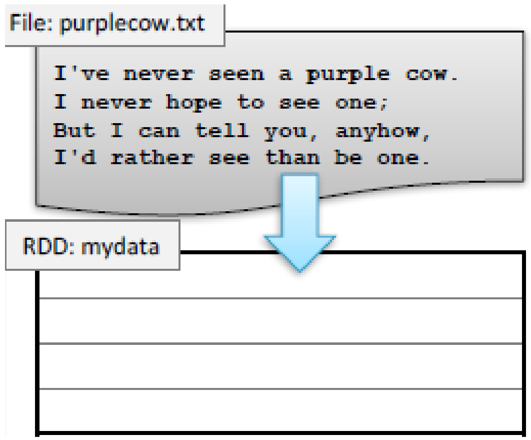
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

```
>
```

Lazy Execution of RDDs (2)

Data in RDDs is not processed until an action is performed

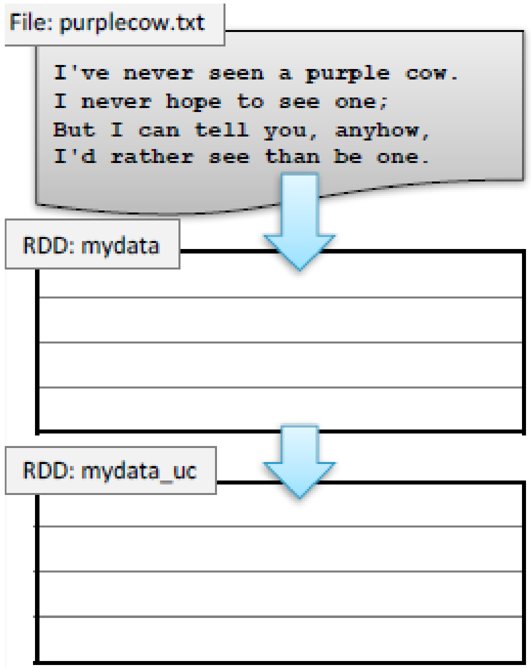
```
> val mydata = sc.textFile("purplecow.txt")
```



Lazy Execution of RDDs (3)

Data in RDDs is not processed until an action is performed

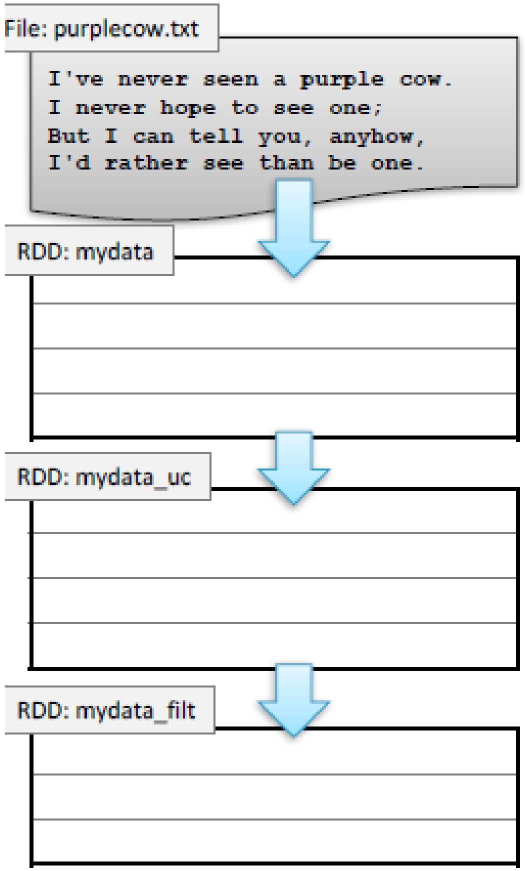
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```



Lazy Execution of RDDs (4)

Data in RDDs is not processed until an action is performed

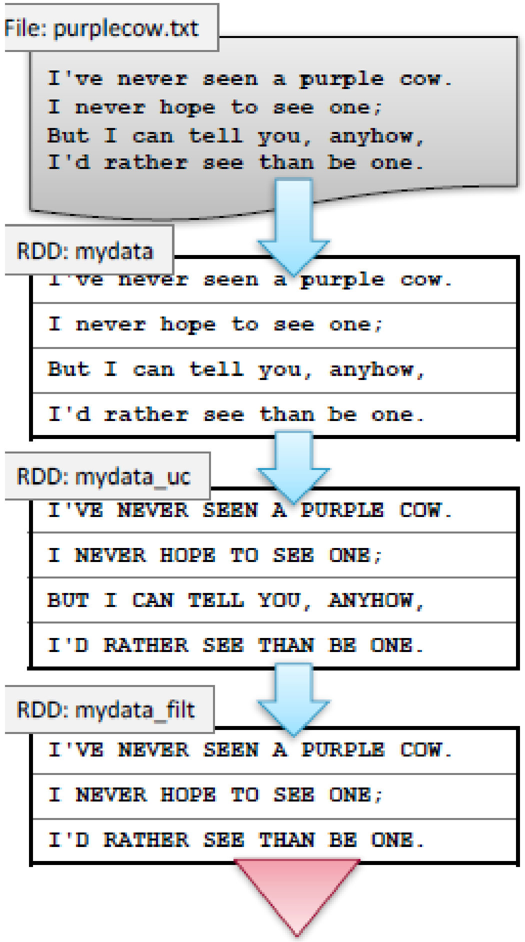
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```



Lazy Execution of RDDs (5)

Data in RDDs is not processed until an action is performed

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



Output Action "triggers" computation

RDD & Spark

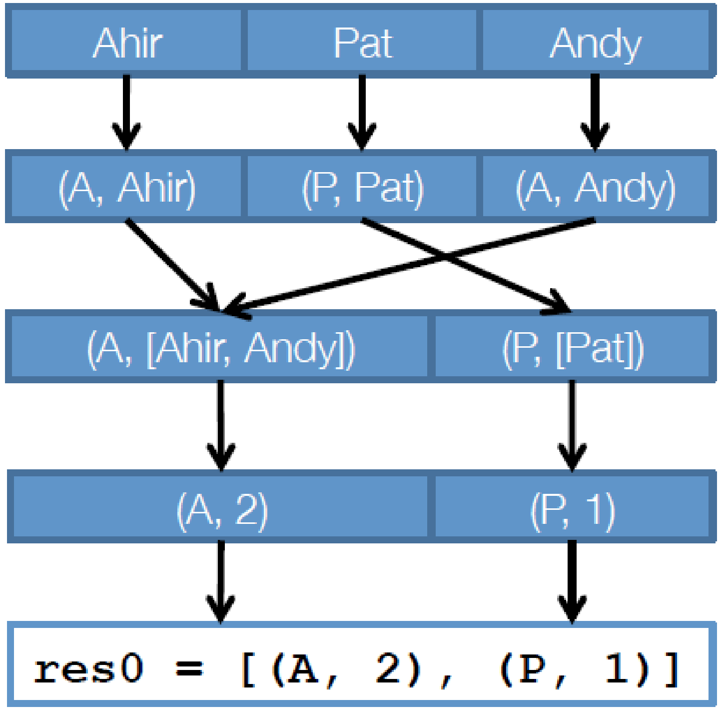
Key Idea: Write applications in terms of transformations on distributed datasets. One RDD per transformation.

- RDD can be organized into a DAG showing how data flows.
- RDD can be saved and reused with controllable persistence (e.g. caching in RAM)
- RDD is automatically rebuilt on failure

Spark DAG execution: An Example

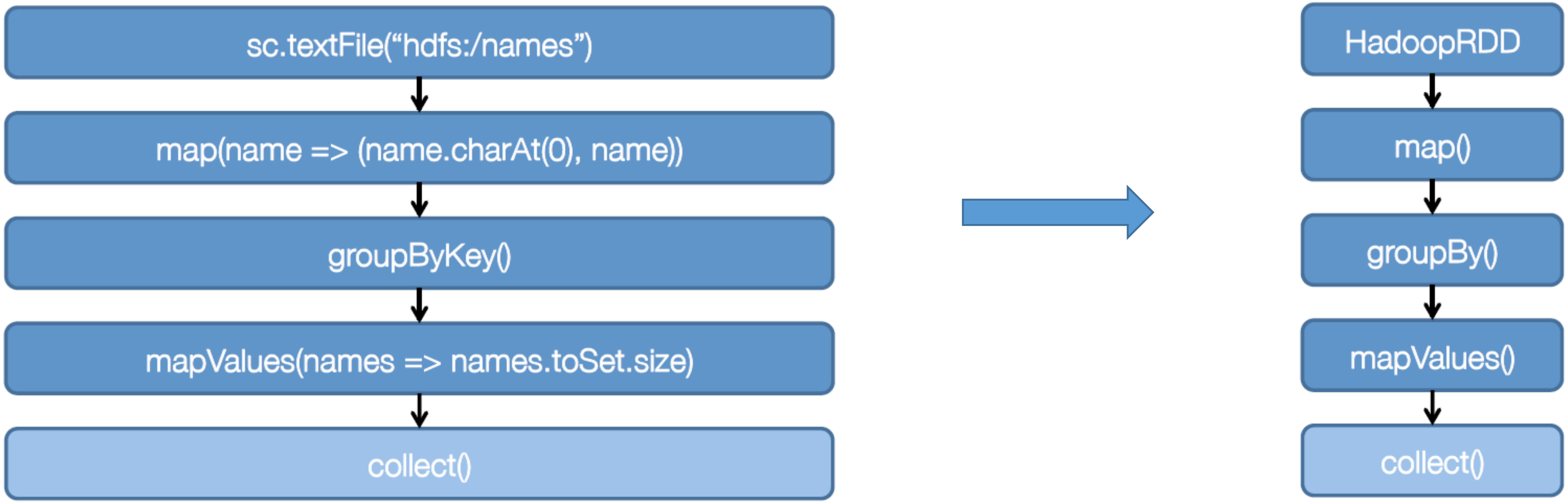
- Goal: Find the number of distinct names per first letter

```
sc.textFile("hdfs:/names")
    .map(name => (name.charAt(0), name))
    .groupByKey()
    .mapValues(names => names.toSet.size)
    .collect()
```



Spark Execution (1)

1. Create a DAG of RDDs to represent computation



Spark Execution (2)

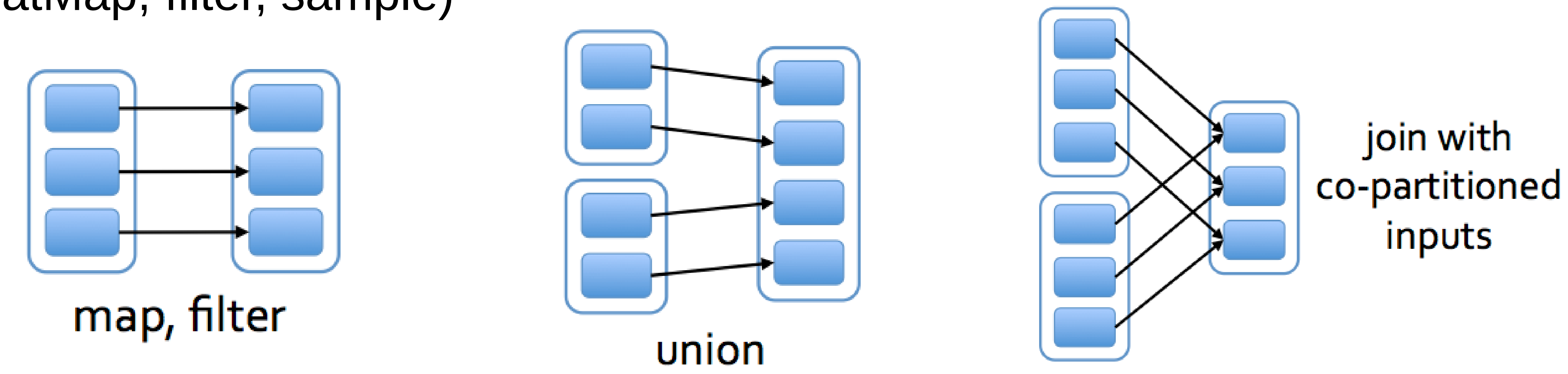
1. Create a DAG of RDDs to represent computation
2. Create logical execution plan for the DAG
    - Split DAG into “stages” based on dependencies
    - Pipeline as much as possible

RDD: Data Set vs Partition Views

Much like in Hadoop MapReduce, each RDD is stored physically in multiple nodes as input partitions

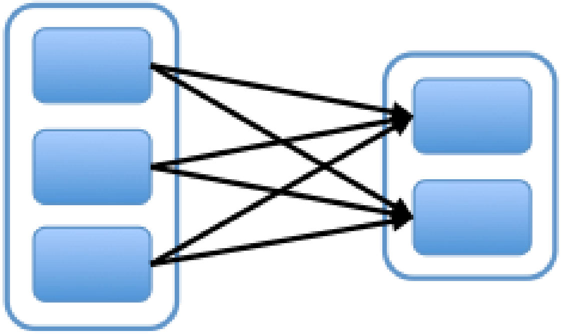
A word about dependencies (1)

- Dependencies determine the need to shuffle data
  - Two types: Narrow and wide
- Narrow dependencies
  - Each partition of the parent RDD is used by at most one partition of the child RDD
  - Task can be executed locally and we don’t have to shuffle. (E.g. map, flatMap, filter, sample)

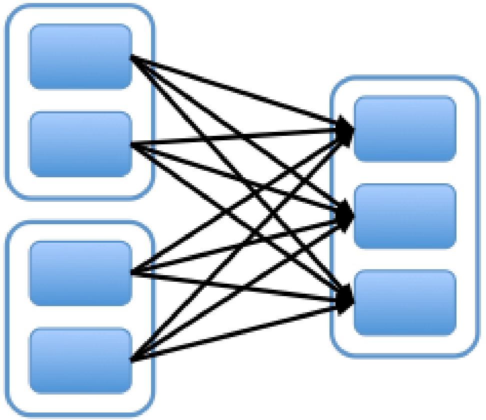


A word about dependencies (2)

- Wide dependencies
  - Multiple child partitions may depend on one partition of the parent RDD
  - We have to shuffle data (E.g. sortByKey, reduceByKey, groupByKey, cogroupByKey, join, cartesian)



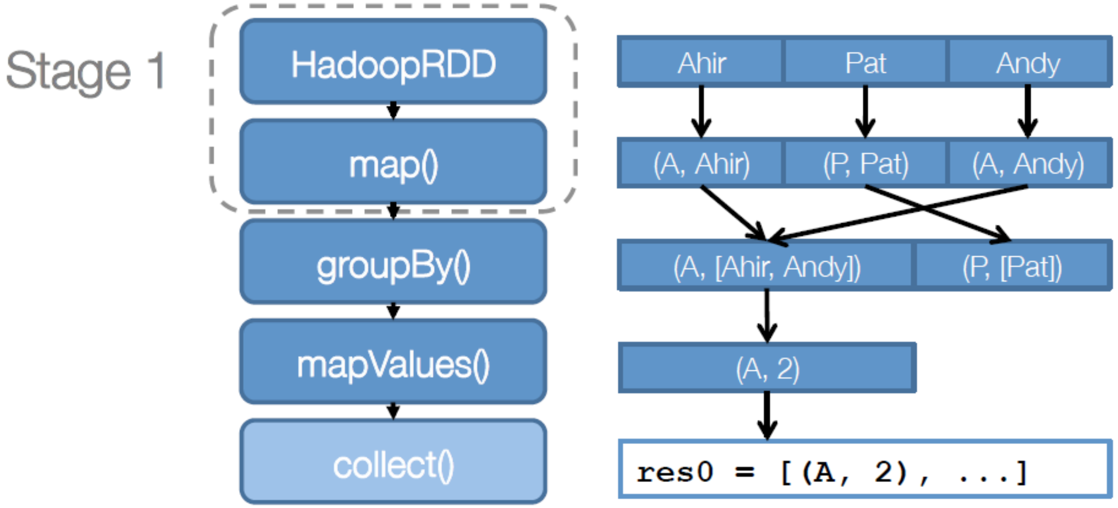
groupByKey

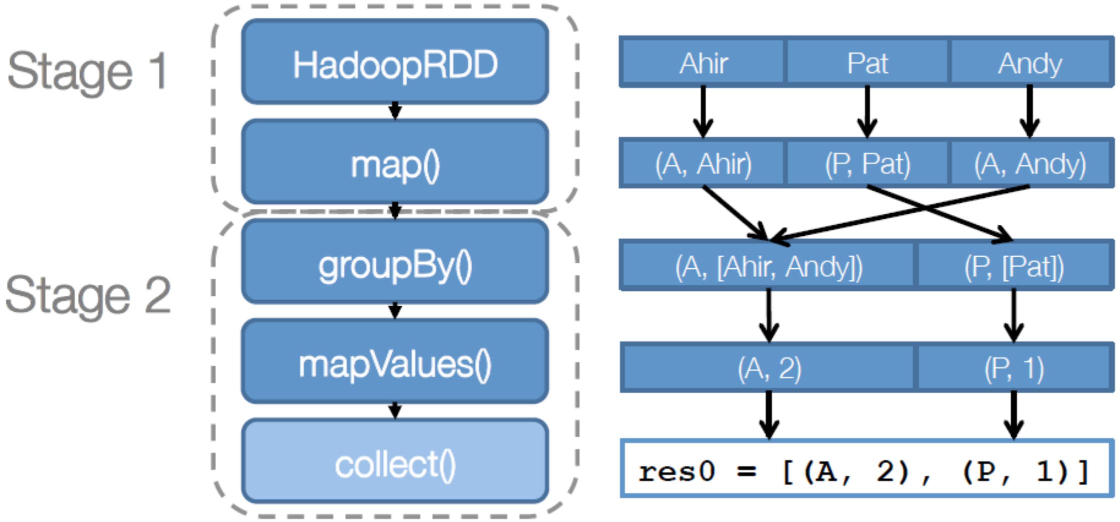


join with inputs not co-partitioned

How does Spark execute this job?

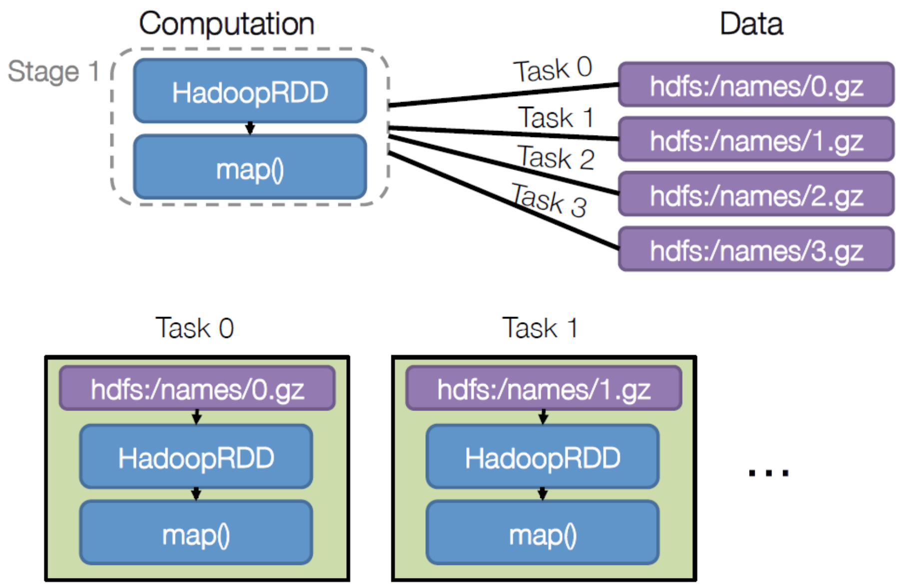
1. Create a DAG of RDDs to represent computation
2. Create logical execution plan for the DAG
   - Pipeline as much as possible
   - Split DAG into “stages” based on need to shuffle data





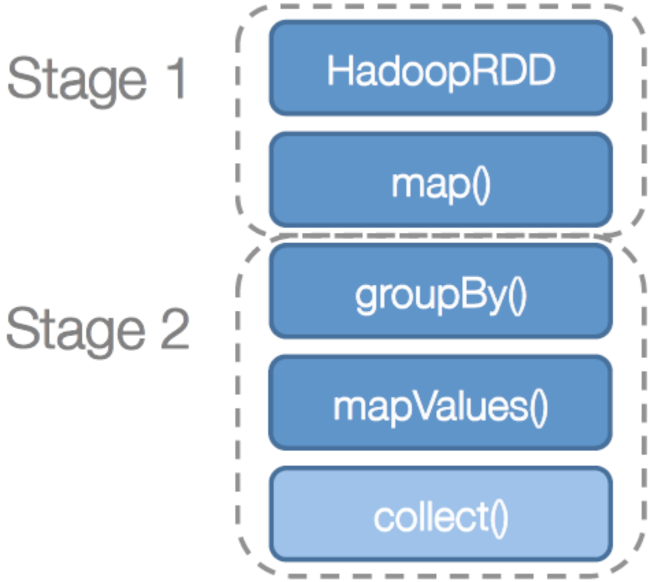
Spark Execution (3)

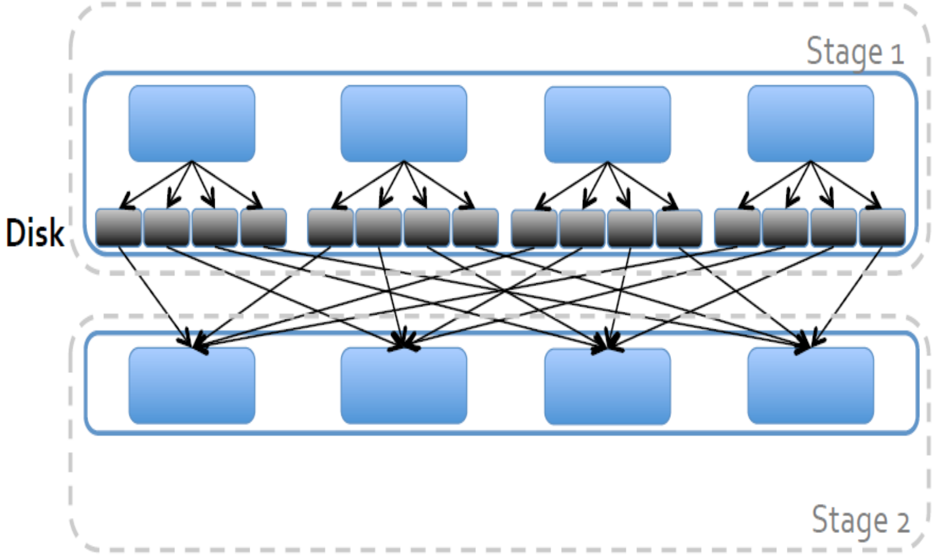
1. Create a DAG of RDDs to represent computation
2. Create logical execution plan for the DAG
3. Split each stage into tasks and execute tasks stage by stage
   - Task = Data + Computation
   - In this example, all tasks from stage 1 would be executed together first



Spark Execution (3)

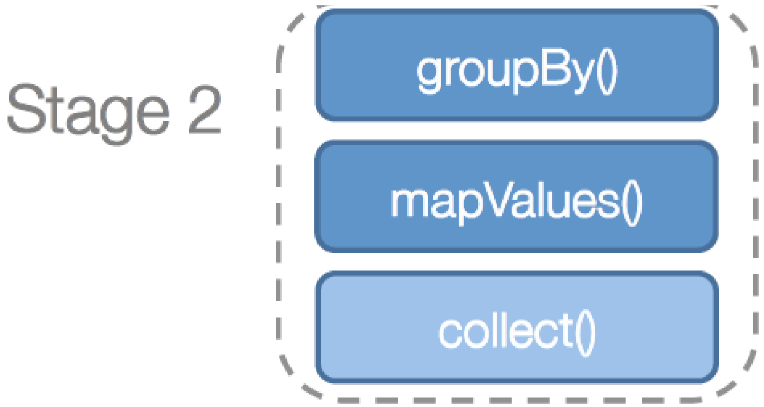
1. Create a DAG of RDDs to represent computation
2. Create logical execution plan for the DAG
3. Split each stage into tasks and execute tasks stage by stage
   - In this example, all tasks from stage 1 would be executed together first
   - After stage 1, pull-based shuffle occurs (intermediates written to files and pulled)





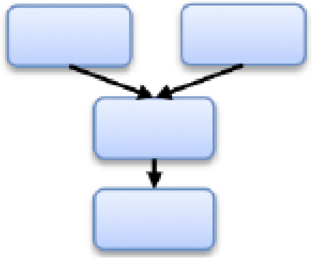
Spark Execution (3)

1. Create a DAG of RDDs to represent computation
2. Create logical execution plan for the DAG
3. Split each stage into tasks and execute tasks stage by stage
   - In this example, all tasks from stage 1 would be executed together first
   - After stage 1, pull-based shuffle occurs (intermediates written to files and pulled)
   - Now, tasks from stage 2 are executed (operators pipelined in each task)



Putting it all together

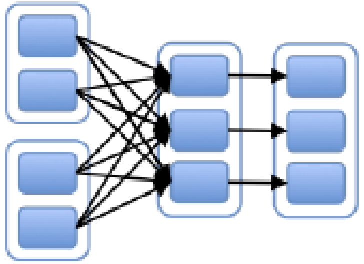
RDD Objects



```
rdd1.join(rdd2)
.groupBy(...)
.filter(...)
```

Build the operator DAG

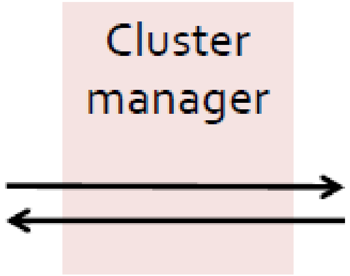
DAG Scheduler



Split the DAG into *stages* of *tasks*

Submit each stage and its tasks as ready

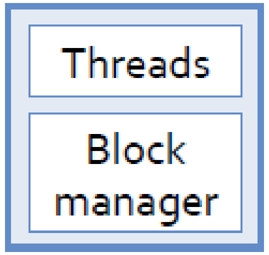
Task Scheduler



Launch tasks via Master

Retry failed and strag-gler tasks

Worker



Execute tasks

Store and serve blocks

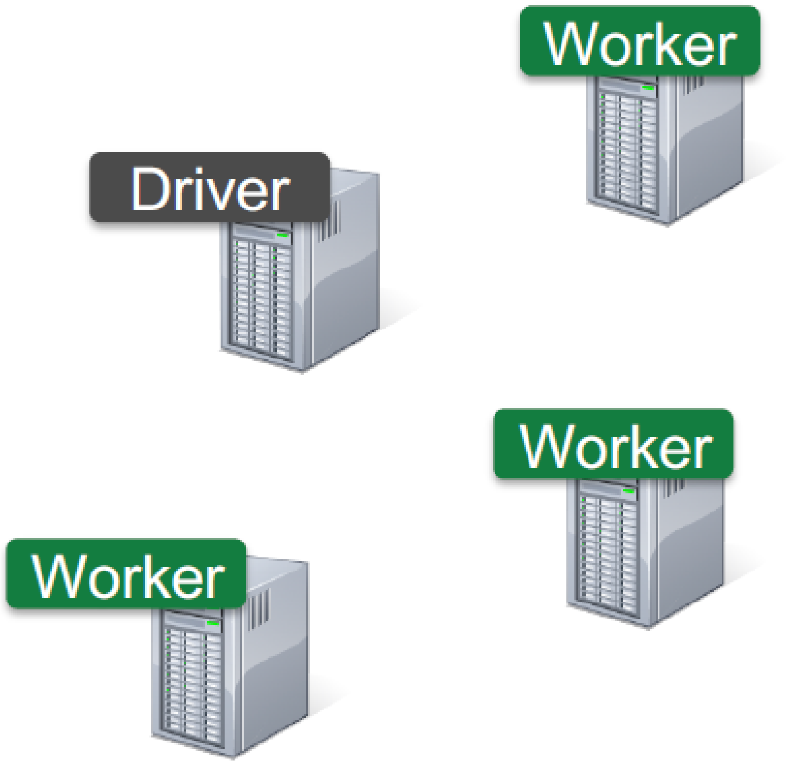
RDD & Spark

Key Idea: Write applications in terms of transformations on distributed datasets. One RDD per transformation.

- RDD can be organized into a DAG showing how data flows.
- RDD can be saved and reused with controllable persistence (e.g. caching in RAM)
- RDD is automatically rebuilt on failure
- RDD provides extensibility

Example: Log Mining

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

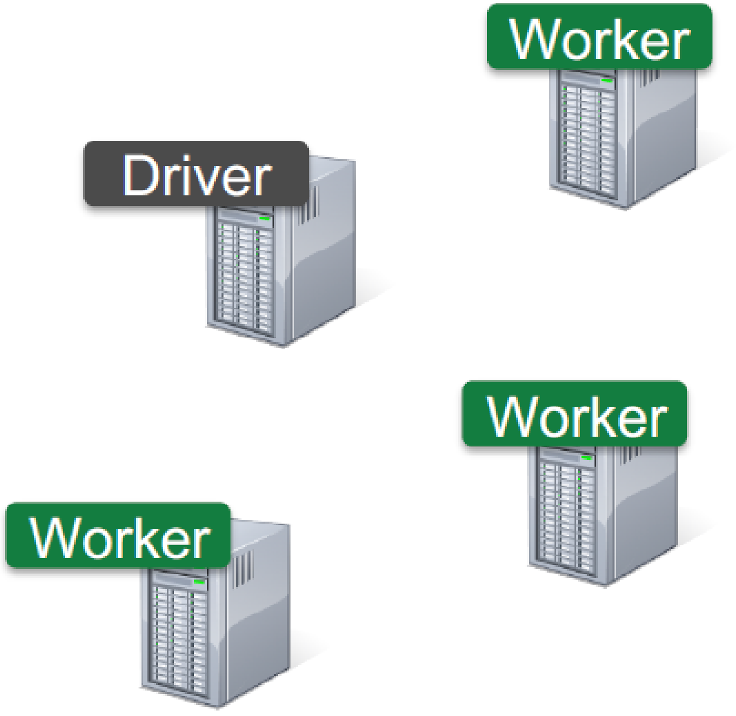


Example: Log Mining

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

Base RDD

```
lines = spark.textFile("hdfs://...")
```

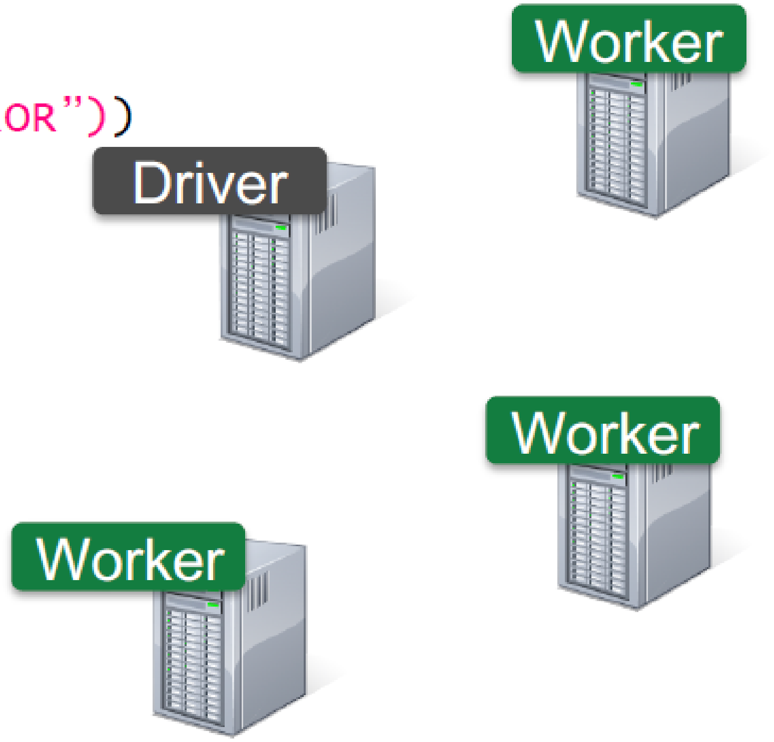


Example: Log Mining

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

Transformed RDD

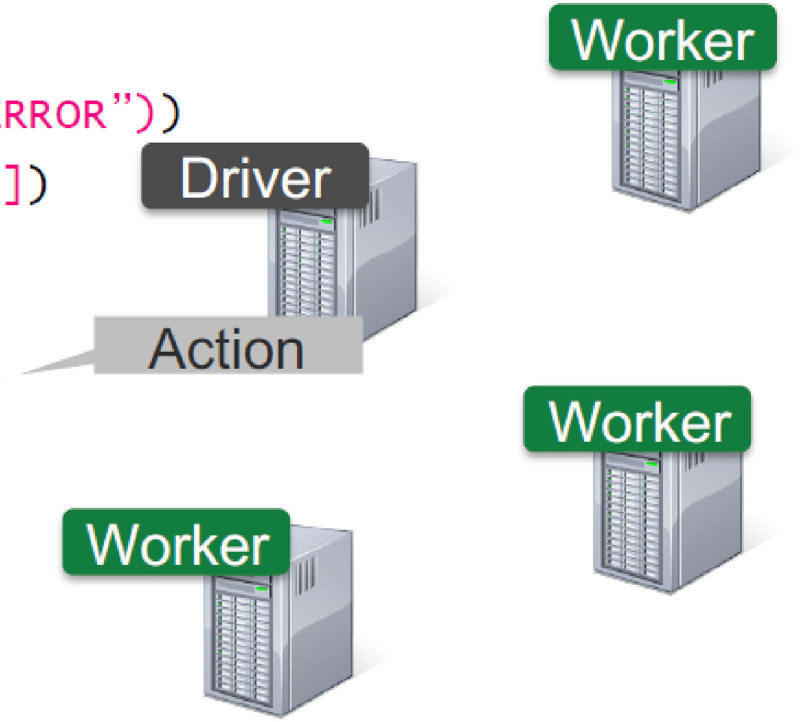
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```



Example: Log Mining

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

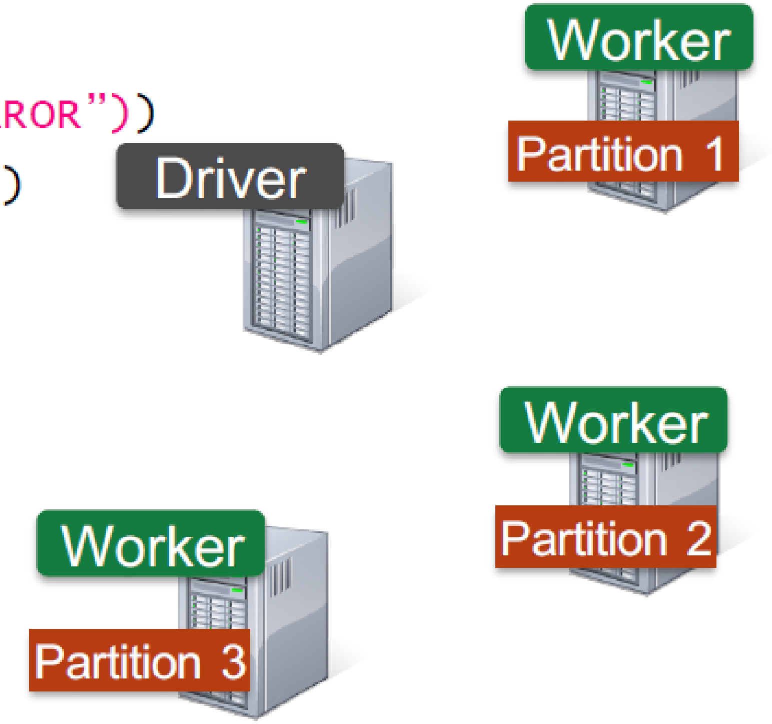
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```



Example: Log Mining

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

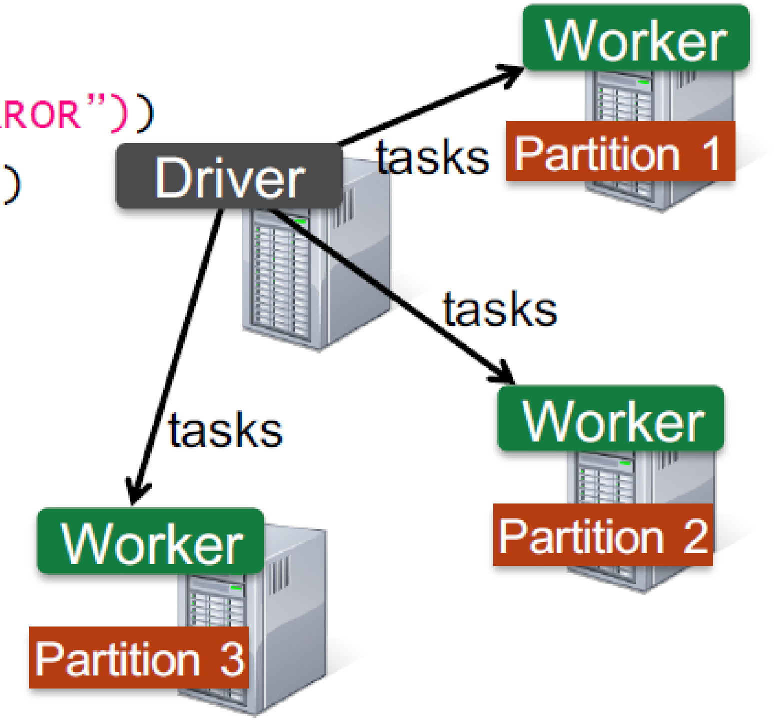
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```



Example: Log Mining

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

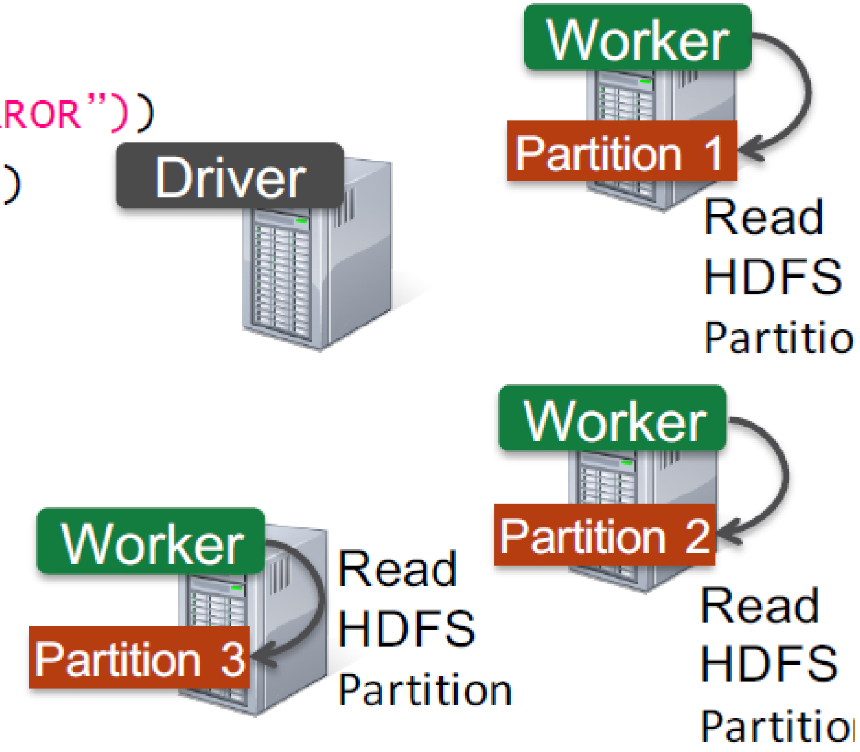
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```



Example: Log Mining

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

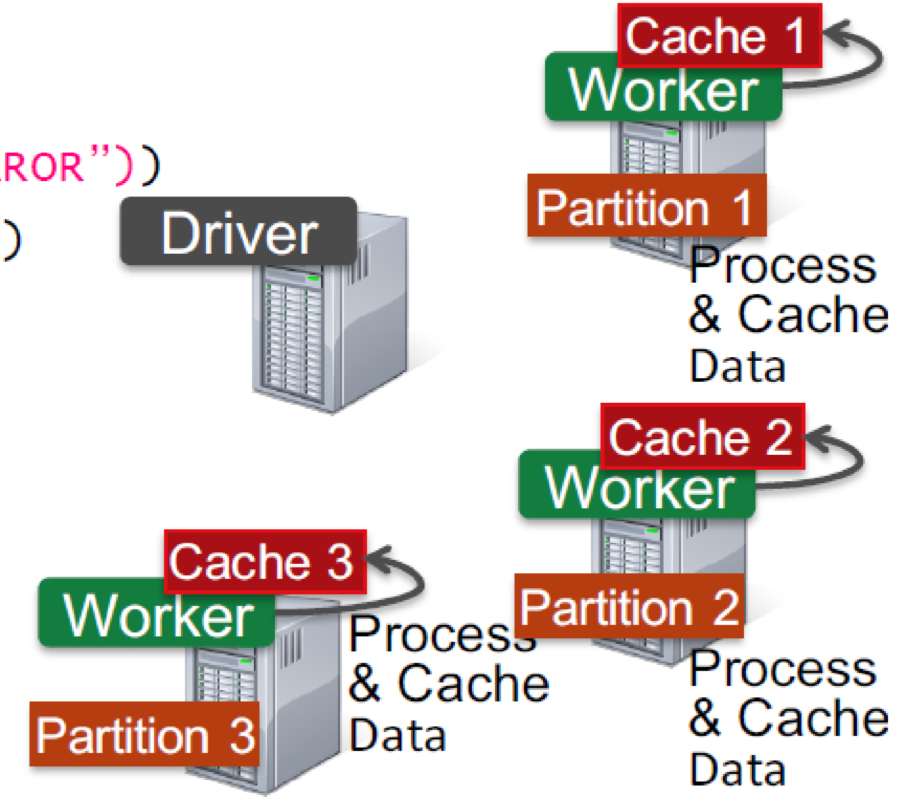
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```



Example: Log Mining

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

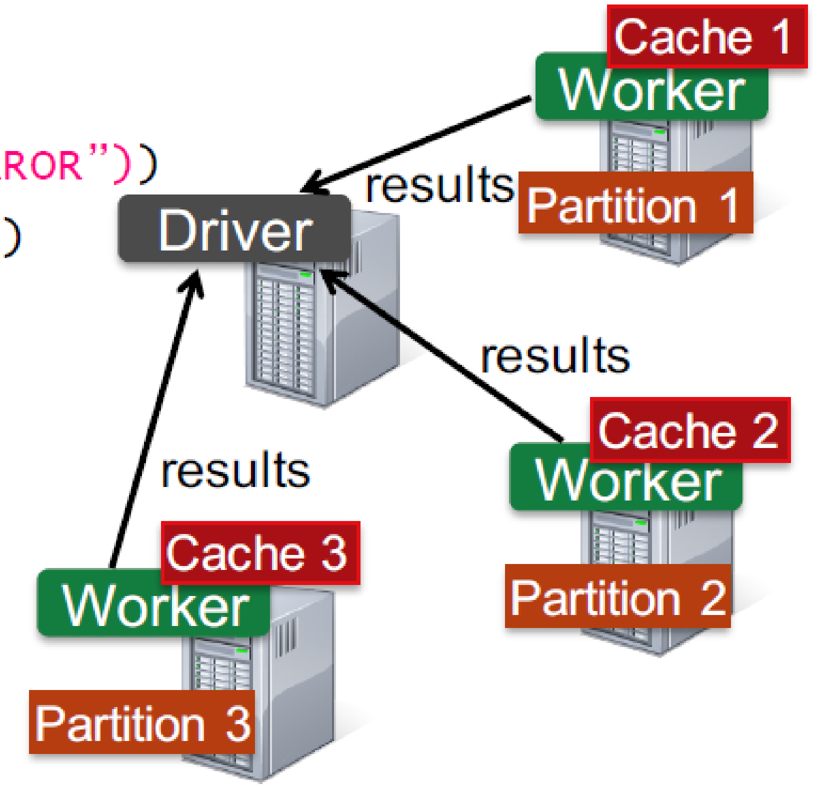
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```



Example: Log Mining

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

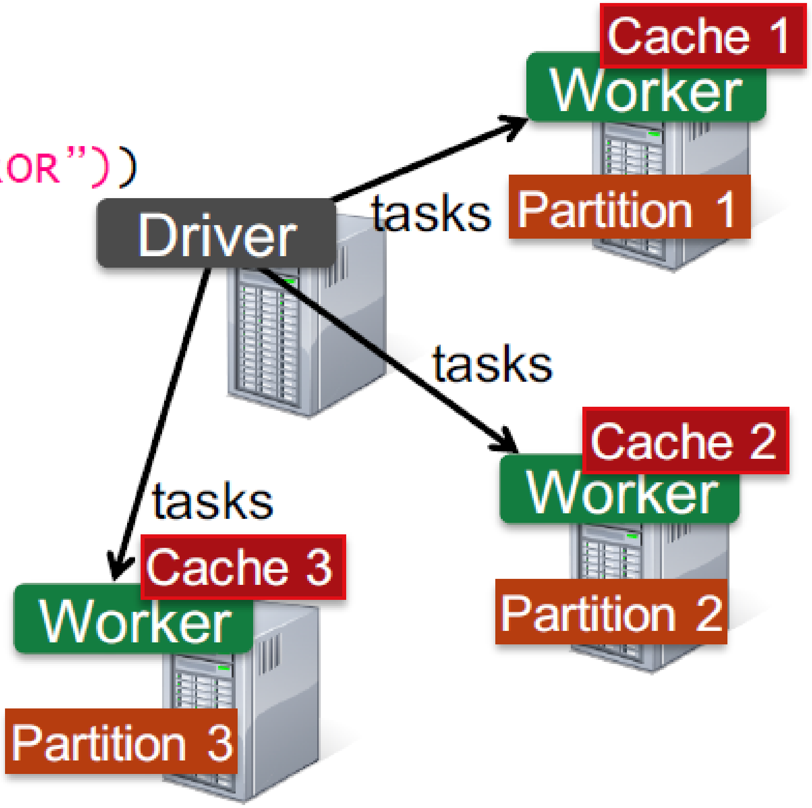
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```



Example: Log Mining

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

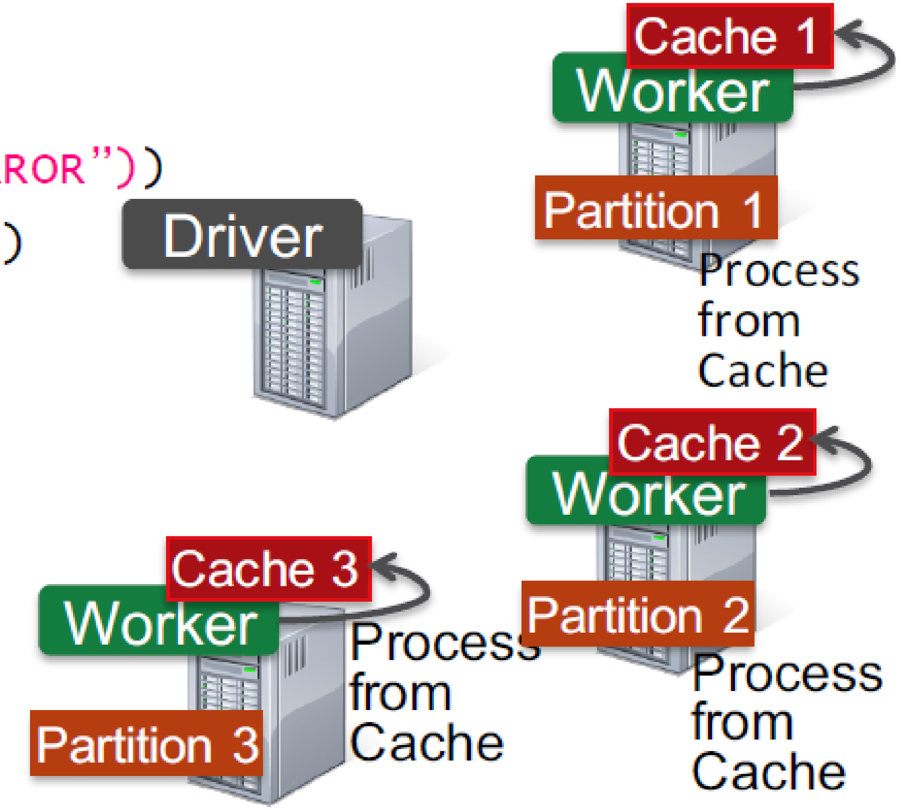
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



Example: Log Mining

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

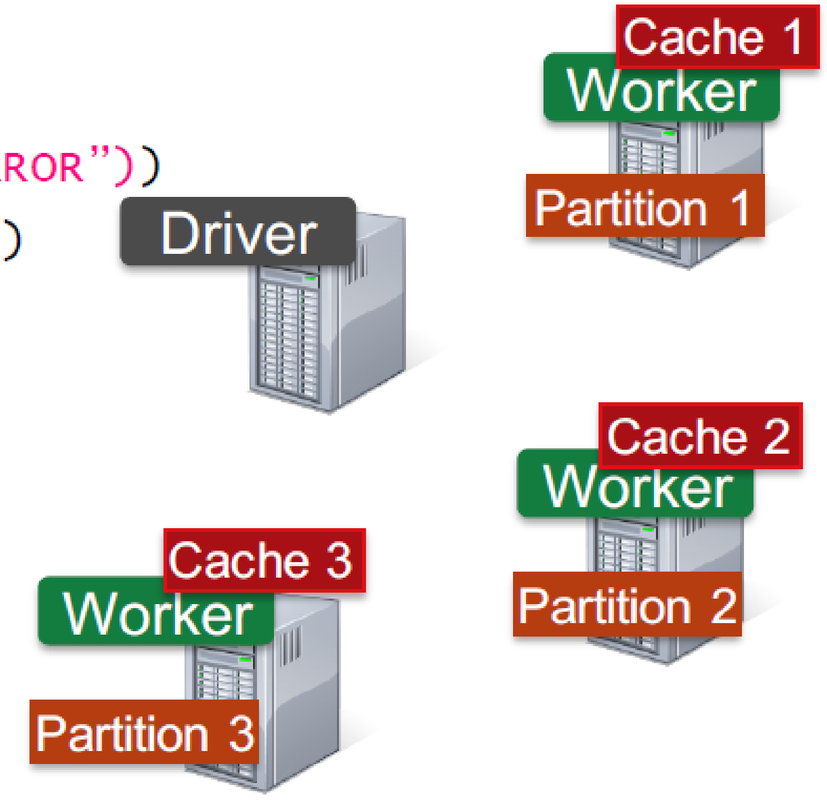
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



Cache your data → Faster Results

Full-text search of Wikipedia

- 60GB on 20 EC2 machines
- 0.5 sec from mem vs. 20s for on-disk

RDD & Spark

Key Idea: Write applications in terms of transformations on distributed datasets. One RDD per transformation.

- RDD can be organized into a DAG showing how data flows.
- RDD can be saved and reused with controllable persistence (e.g. caching in RAM)
- RDD can be automatically rebuilt on failure

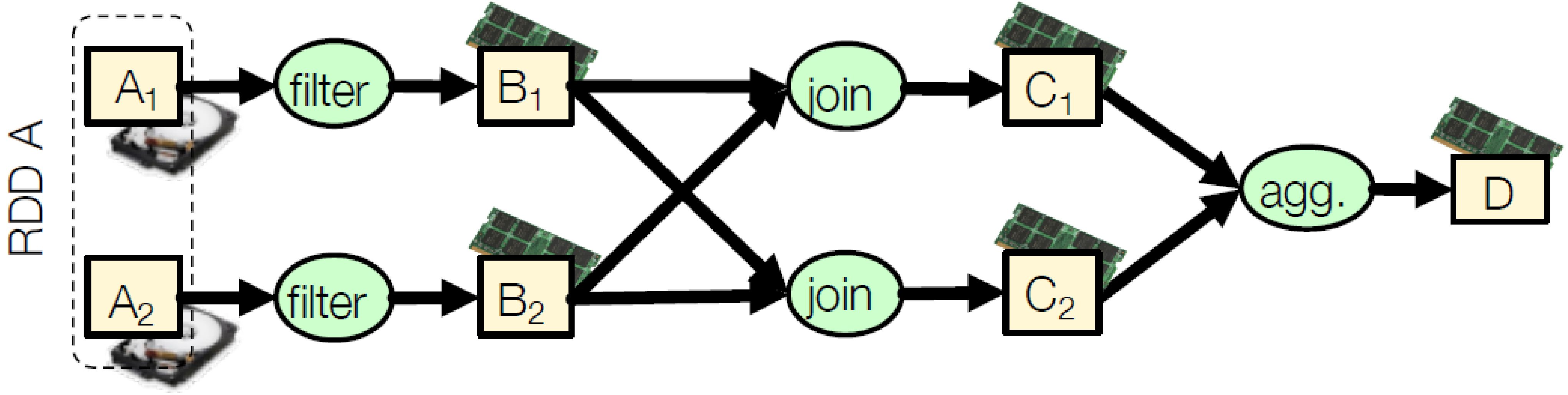
RDD: Immutability and lineage

- RDD are created once, then reuse without changes
- Avoids data inconsistency problems (no simultaneous updates) → Correctness
- Easily live in memory as on disk → Caching
- Safe to share across processes/tasks → Improves performance
- The DAG of RDD also encodes lineage information
  - Each RDD keeps track of its parent
  - Lineage is used in Spark to recreate RDD under failure

Fault Recovery Example

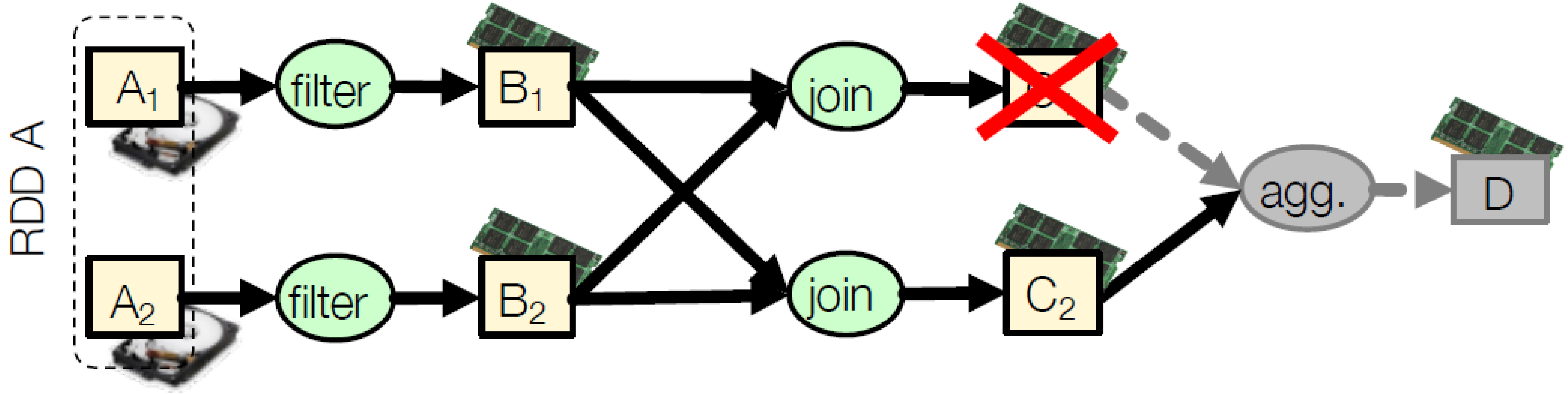
Two-partition RDD A={A_1, A_2} stored on disk

1) filter and cache → RDD B
2) join→ RDD C
3) aggregate → RDD D



Fault Recovery Example

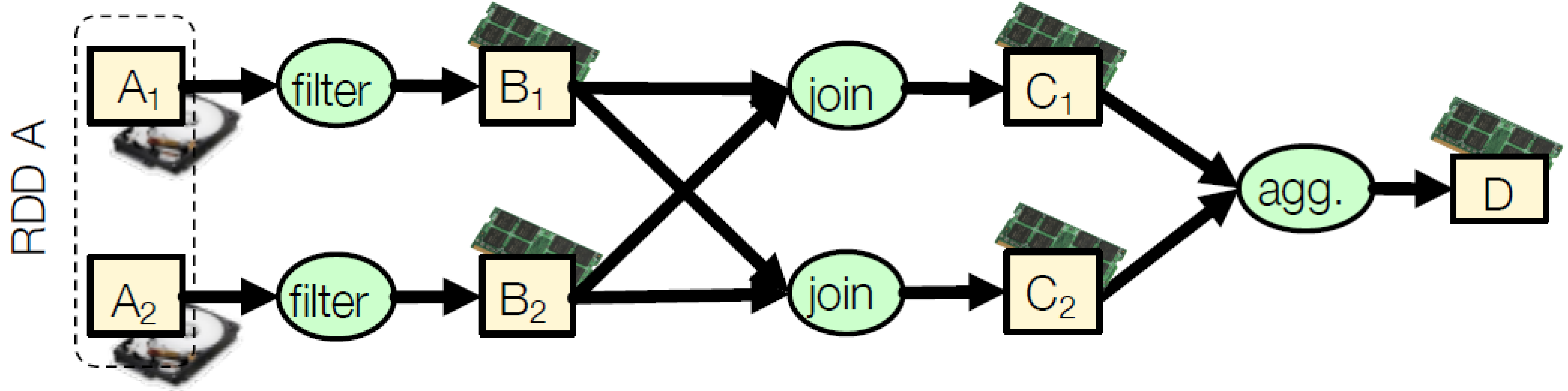
C_1 lost due to node failure before "aggregate" finishes



Fault Recovery Example

C_1 lost due to node failure before reduce finishes

Reconstruct C_1, eventually, on different node



Resilient Distributed Dataset: Summary

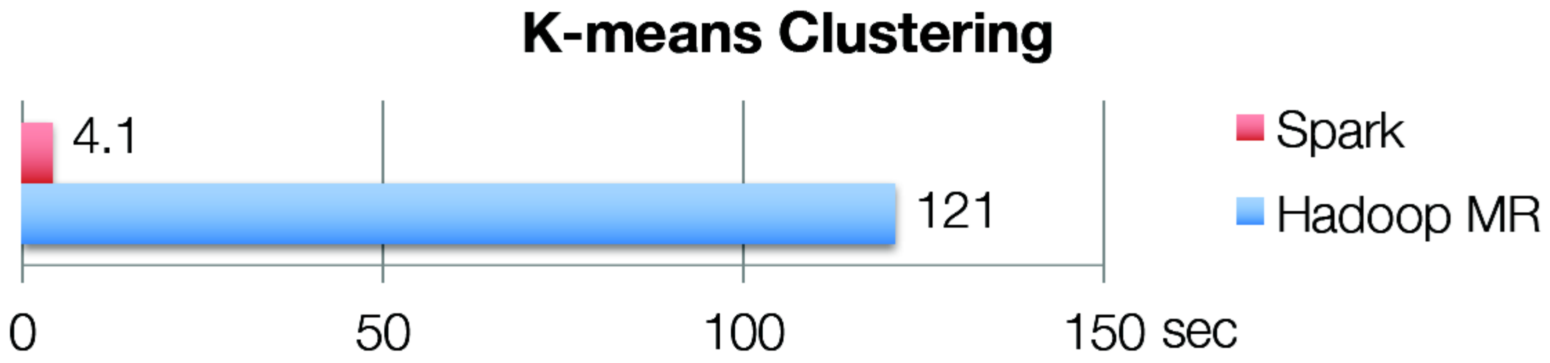
Resilient – Recover from node failures with low-overhead

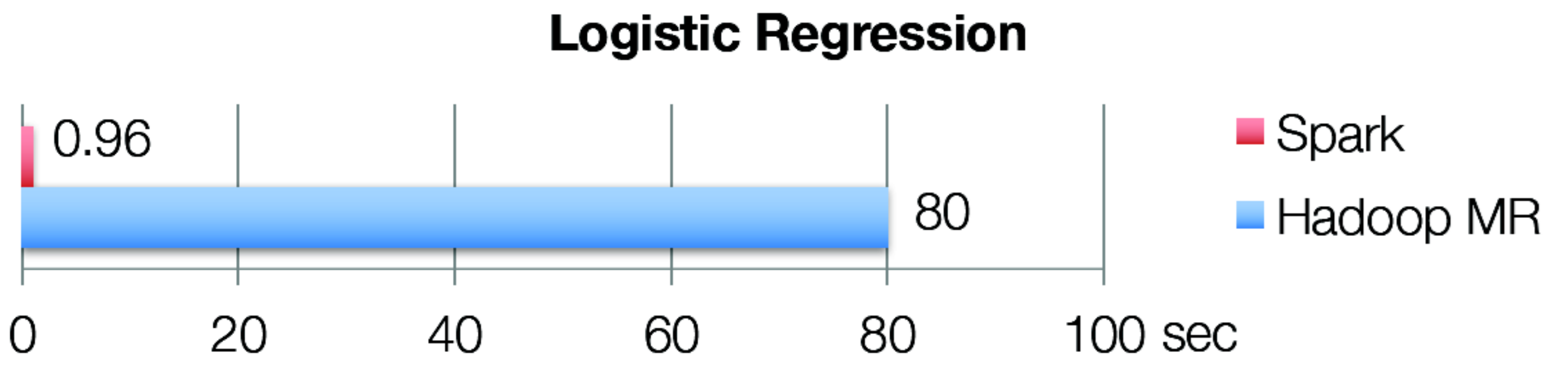
Distributed – partitioned parallelism across the cluster

Dataset – RDD created from a file or using coarse-grained transformations

- Benefits over DSM
  - Lineage tracking instead of checkpointing or logging
  - Can mitigate stragglers using backup tasks like MapReduce
  - Can exploit data locality automatically by placing work like MapReduce
  - Can exploit memory by caching RDDs
  - Can deal with out-of-memory situations gracefully instead of swapping

Iterative Algorithms: Spark vs MapReduce





Spark Programming: Transformations

Creating RDD

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

Basic Transformations

```
# 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}
```

Spark Programming: 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
```

Example: Word Count Driver

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
val sc = new SparkContext("spark://...", "MyJob", spark
home", "additional jars")
```

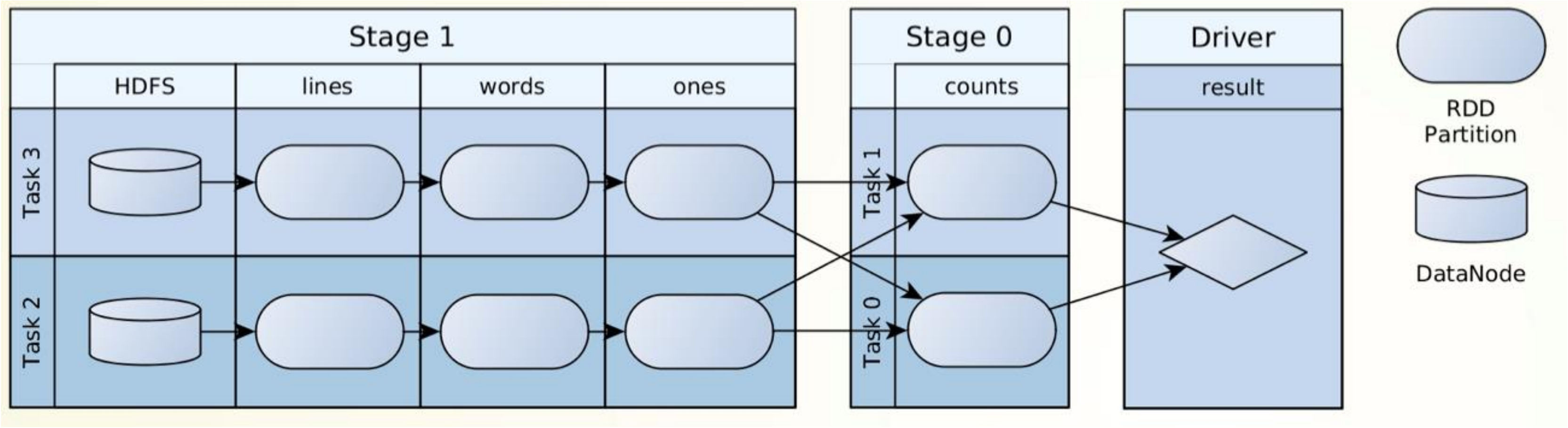
- **Driver and SparkContext**
  - A SparkContext initializes the application driver, the latter then registers the application to the cluster manager, and gets a list of executors
  - Then, the driver takes full control of the Spark job

Example: Word Count Code

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(w => (w, 1))
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

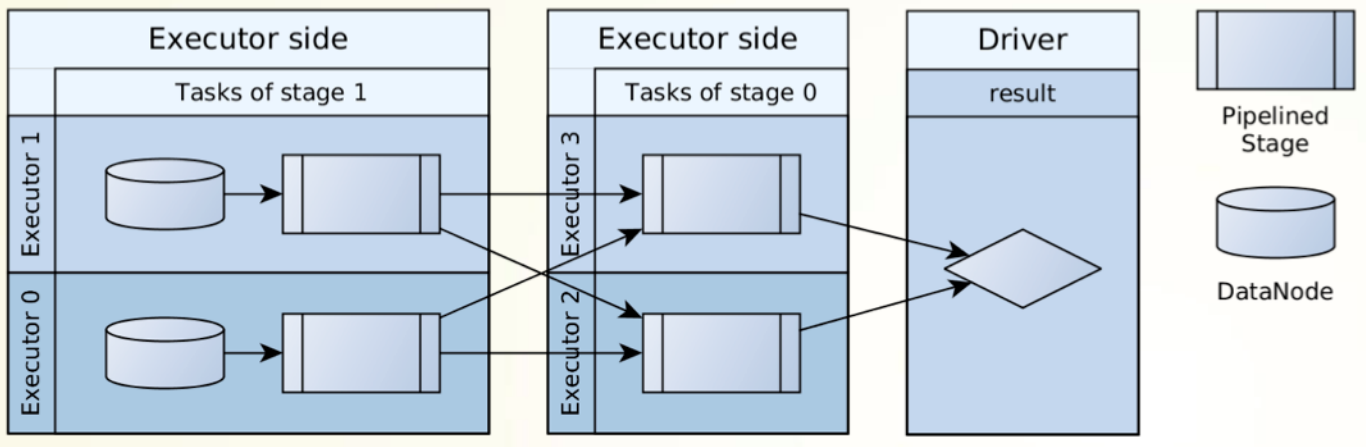
- **RDD lineage DAG is built on driver side with**
  - Data source RDD(s)
  - Transformation RDD(s), which are created by transformations
- **Job submission**
  - An action triggers the DAG scheduler to submit a job

Example: Word Count DAG



- **Directed Acyclic Graph**
  - Built from the RDD lineage
- **DAG scheduler**
  - Transforms the DAG into stages and turns each partition of a stage into a single task
  - Decides what to run

Example: Word Count Execution Plan



- **Spark Tasks**
  - Serialized RDD lineage DAG + closures of transformations
  - Run by Spark executors
- **Task scheduling**
  - The driver side task scheduler launches tasks on executors according to resource and locality constraints
  - The task scheduler decides where to run tasks

Example: Word Count Shuffle

```
val lines = sc.textFile("input")
val words = lines.flatMap(_.split(" "))
val ones = words.map(w => (w, 1))
val counts = ones.reduceByKey(_ + _)
val result = counts.collectAsMap()
```

- **reduceByKey transformation**
  - Induces the shuffle phase as we have a wide dependency
  - Like in Hadoop MapReduce, intermediate <key,value> pairs are stored on the local file system
- **Automatic combiners!**
  - The reduceByKey transformation implements map-side combiners to pre-aggregate data

Comparison with Hadoop MapReduce

```
public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
 }

 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }
```

Spark: Summary

- Simplicity (Easier to use)
  - Rich APIs for Scala, Java, and Python
- Generality: APIs for different types of workloads
  - Batch, Streaming, Machine Learning, Graph
- Low Latency (Performance)
  - In-memory processing and caching
- Fault-tolerance
  - Lineage and immutability of RDD can be used for recomputation

Spark Ecosystem: A Unified Pipeline

The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)