Spark Data Model 1

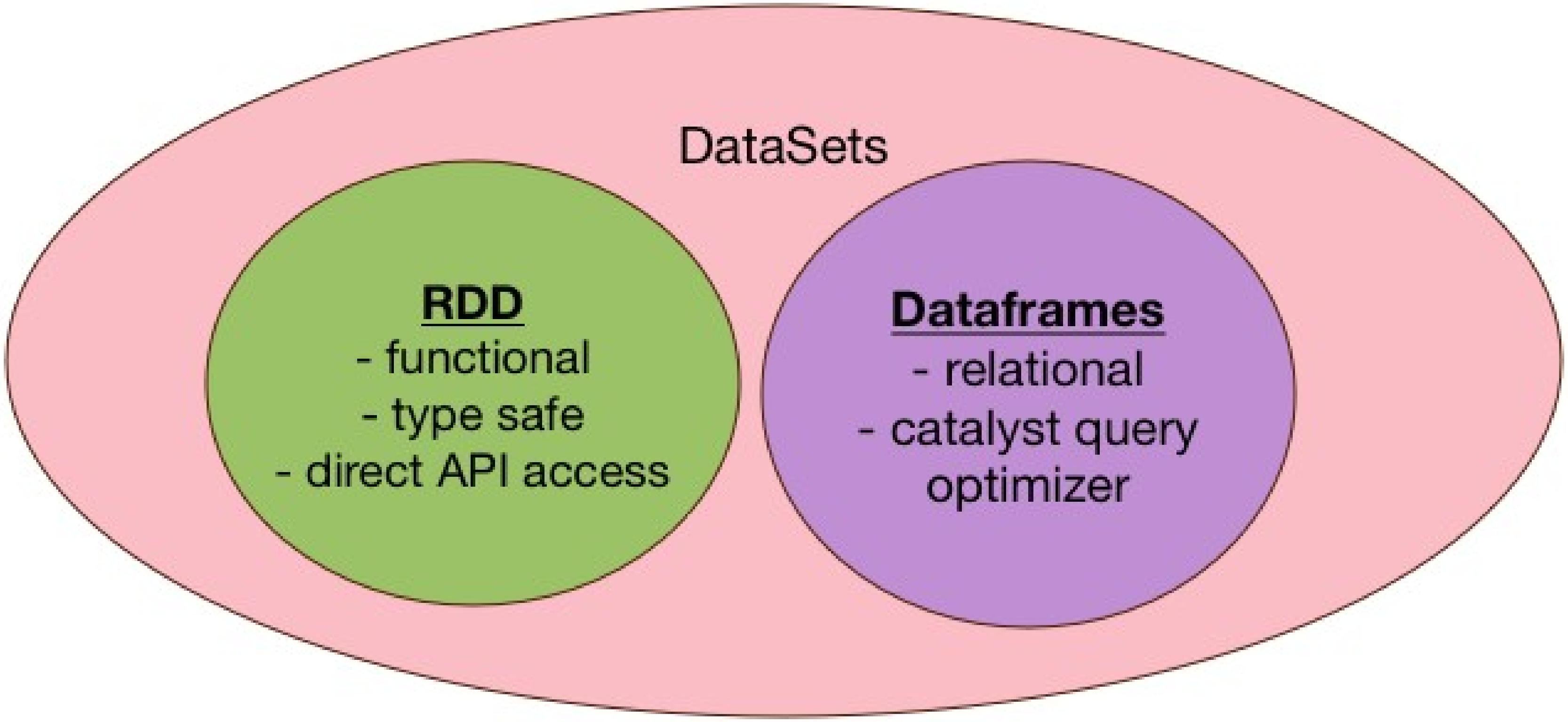
Data Model Overview
RDD Concepts
Spark Workflow
Working with RDDs
Caching
Key-Value Pairs

Lesson Objectives

- Understand various Spark data models
- Understand RDD, DataFrame, Dataset
- Be familiar with Spark's architecture and how it works

Data Model Overview

Spark Data Model Evolution



Spark Data Models

- There are three data models

RDD	DataFrame	Dataset
• Since v1 • Original data API • Gives complete control to developer • Can be a bit low level	• Since Spark 1.3 • Created to give high-level data access • Data has schema, organized as columns • Supports SQL • DataFrame = RDD + Schema • Catalyst query optimizer	• Since Spark v2 • Effort to unify RDD and DataFrame
Java, Python, Scala	Java, Python, Scala	Java, Scala, Python (partial support)

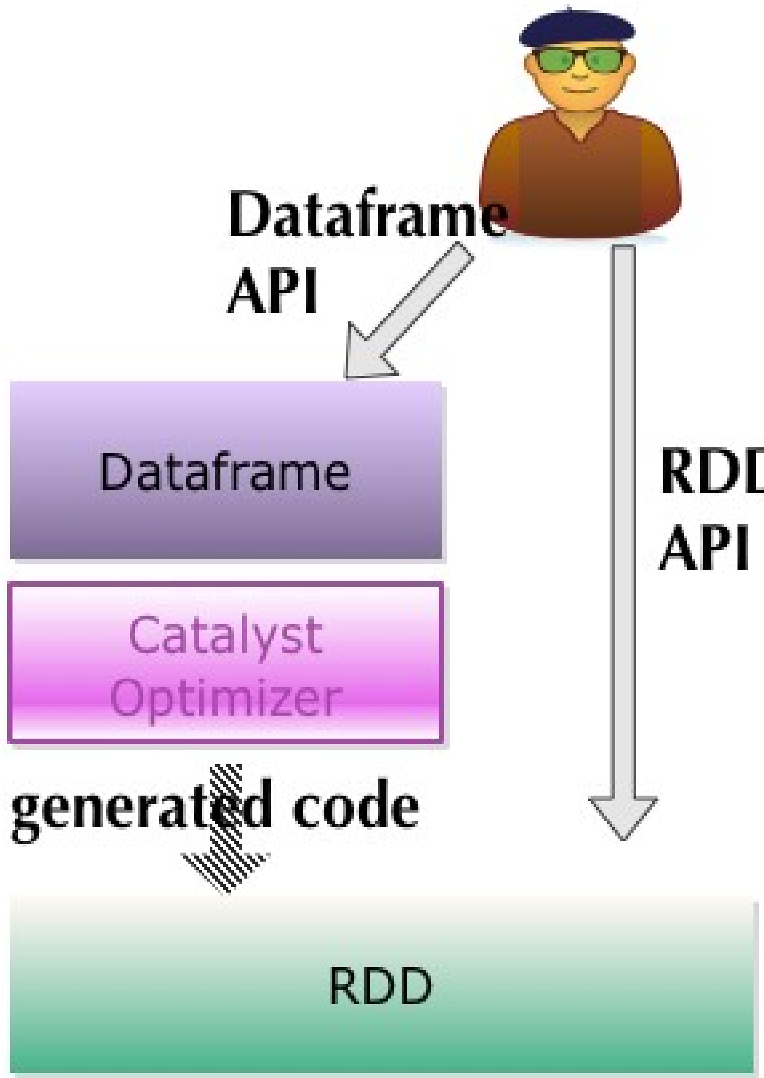
Spark Data Models: RDD

- RDDs are building block of Spark
- API in multiple languages: Java, Scala, Python, R
- Provide complete control of data manipulation
- Suited for unstructured data
  - Text, images, video
- Uses JVM heap for memory
  - Can be expensive for caching large data sets
  - Has Java Garbage Collection (GC) overhead
- Has been supported from the beginning

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
 2019-03-12

Spark Data Models: DataFrame

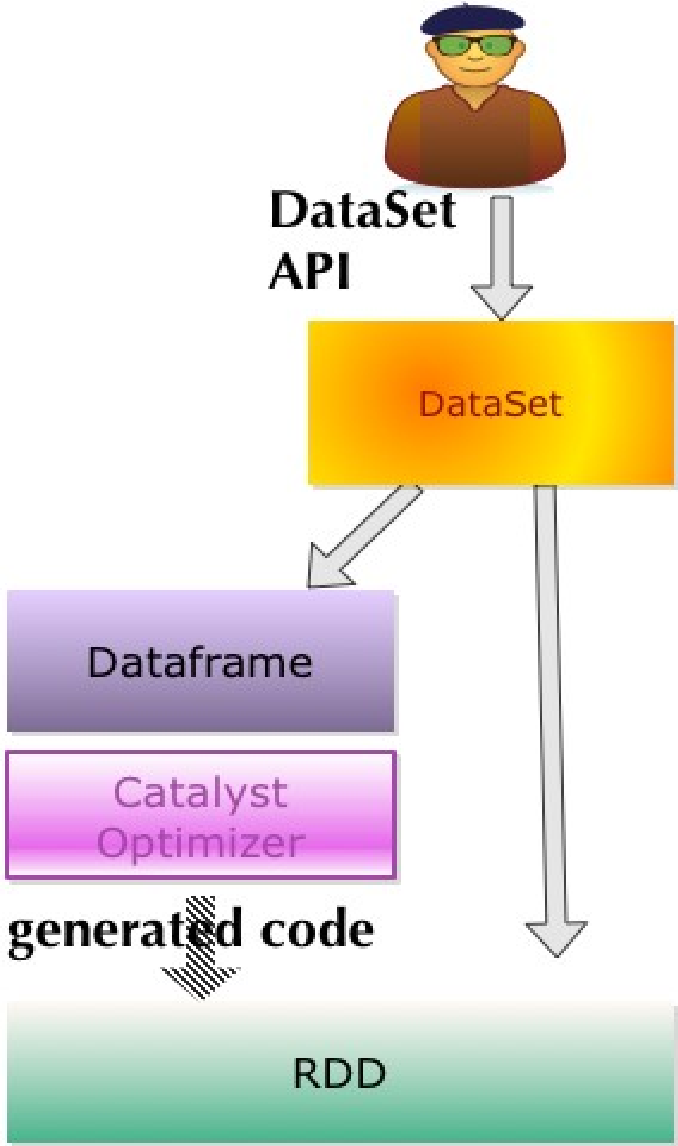
- DataFrames are built on top of RDDs
- Has schema information (column types …etc) (not found in RDDs)
- Schema helps with
  - Efficient storage (Tungsten)
  - Optimizer (Catalyst)
- Uses off-heap memory for caching, better performance than JVM heap
- Generally better performance than RDDs because of:
  - Better memory management
  - Optimized code
- Supported since V1.3
- ** In-depth coverage in next section



Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

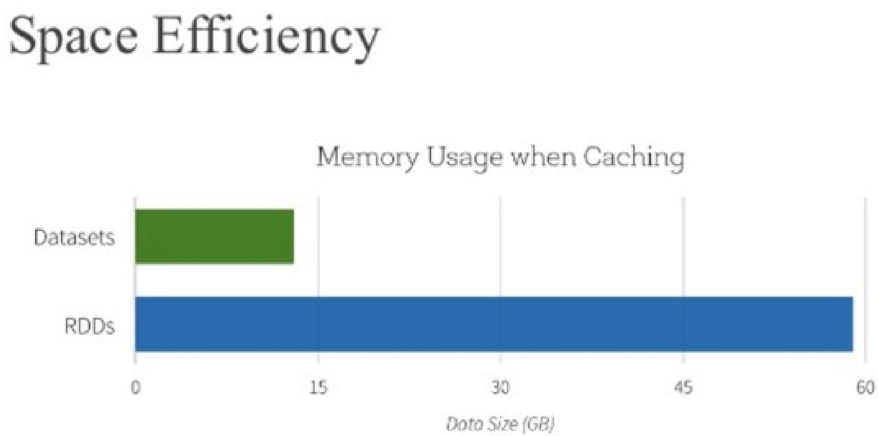
Spark Data Models: Dataset

- Dataset is introduced in Spark 2.0 (preview in 1.6)
- Offers a unified view of DataFrame and RDD
- Best of both worlds!
- Will be ***the*** API going forward



Dataset Features

- High performance:
  - Built on **Tungsten** execution engine and **Catalyst** optimizer
  - Tungsten provides efficient memory footprint
  - Catalyst optimizes the query
- Dataset provides type-safe, object-oriented API (Scala/Java)
- Provides schema
- Dataset API available in Java/Scala
  - Not in Python
  - Since Python is dynamic, most features of generic Dataset APIs are available



Tungsten

- Project Tungsten's goal is to improve Spark memory/CPU efficiency
  - Network & disk IO performance is considered fast enough
  - Modern hardware has more memory, GPUs (Graphics Processing Units)
- Fast memory encoding using "encoders"
- Efficient memory usage
  - Uses "off heap" memory allocation; manages memory explicitly (bypass JVM to avoid garbage collection)
  - Serialize objects in Tungsten binary format
- Cache Locality
  - Compute-friendly cache layout ("Cache aware data structures")
- "Whole Stage Code Generation"
  - Compiling queries into efficient Java functions
  - Uses Janino compiler (super small/super fast)

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Spark Data Models Comparison

Feature	RDD	DataFrame	Dataset
Good for	- Familiar API - Complete control - Good for semi-structured data - No need for schema - Strongly typed	- High-level API - With schema - Take advantage of Catalyst optimizer - Supports SQL - Generic typed - Optimized cache	- Unified view of RDD and DataFrame - Both strongly typed and generic type
Downside	- Cache not optimized - Code not optimized	- Newer API - Low level operations are not possible/hard	- Not fully supported in Python

Type Safety Comparison

	SQL	DataFrames	Datasets
Syntax error	Runtime	Compile Time	Compile Time
Analytical Error	Runtime	Runtime	Compile Time

- Catching type errors at compile time saves developer time and effort
- Sometimes data type is not known at compile type
  - Want to be flexible

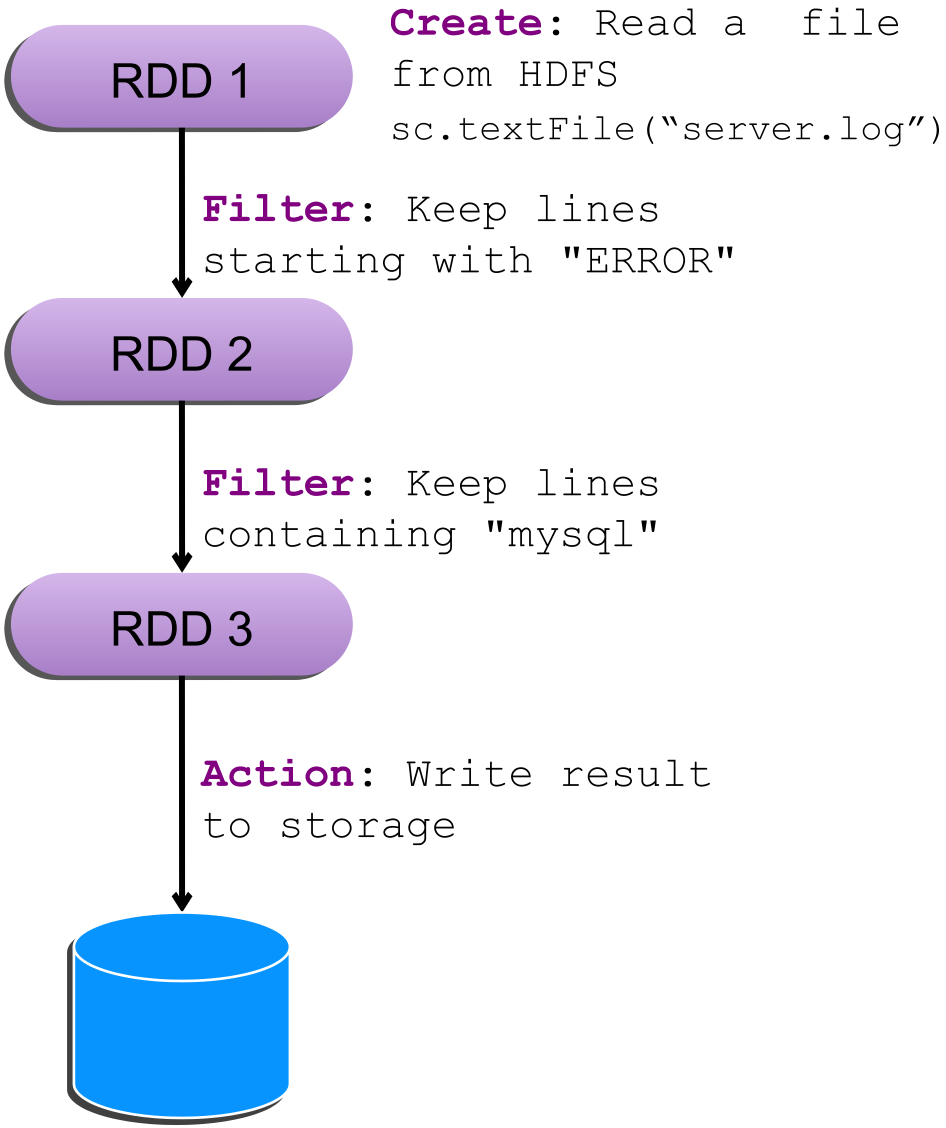
Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
 2019-03-12

RDD

What is an RDD?

- **Resilient Distributed Dataset**: **Core Spark data abstraction**
  - **Distributed collection** of elements
    - **Partitioned**–usually across **different nodes**
    - **Read-only** (immutable)
    - Operations execute **in parallel** on the partitions
  - **Fault tolerant**: Can recover from loss of a partition
    - The "Resiliency"
    - **Efficiently**–Re-computed, not stored
- Two operation types
  - **Transformation**: Lazy operation creating new RDD
    - E.g., `map()`, `filter()`
  - **Action**: Return a result or save it
    - E.g., `take()`, `save()`

RDD Lifecycle



- RDD is **created** by either:
  - Loading an external dataset
  - Distributing a collection
- RDD is **transformed**
  - E.g., filter out elements
  - Result: a **new RDD**
  - Often have a sequence of transformations
- Data is eventually **extracted**
  - By an **action** on an RDD
  - E.g., save the data
- At left, we read/transform a log file, then save the result.

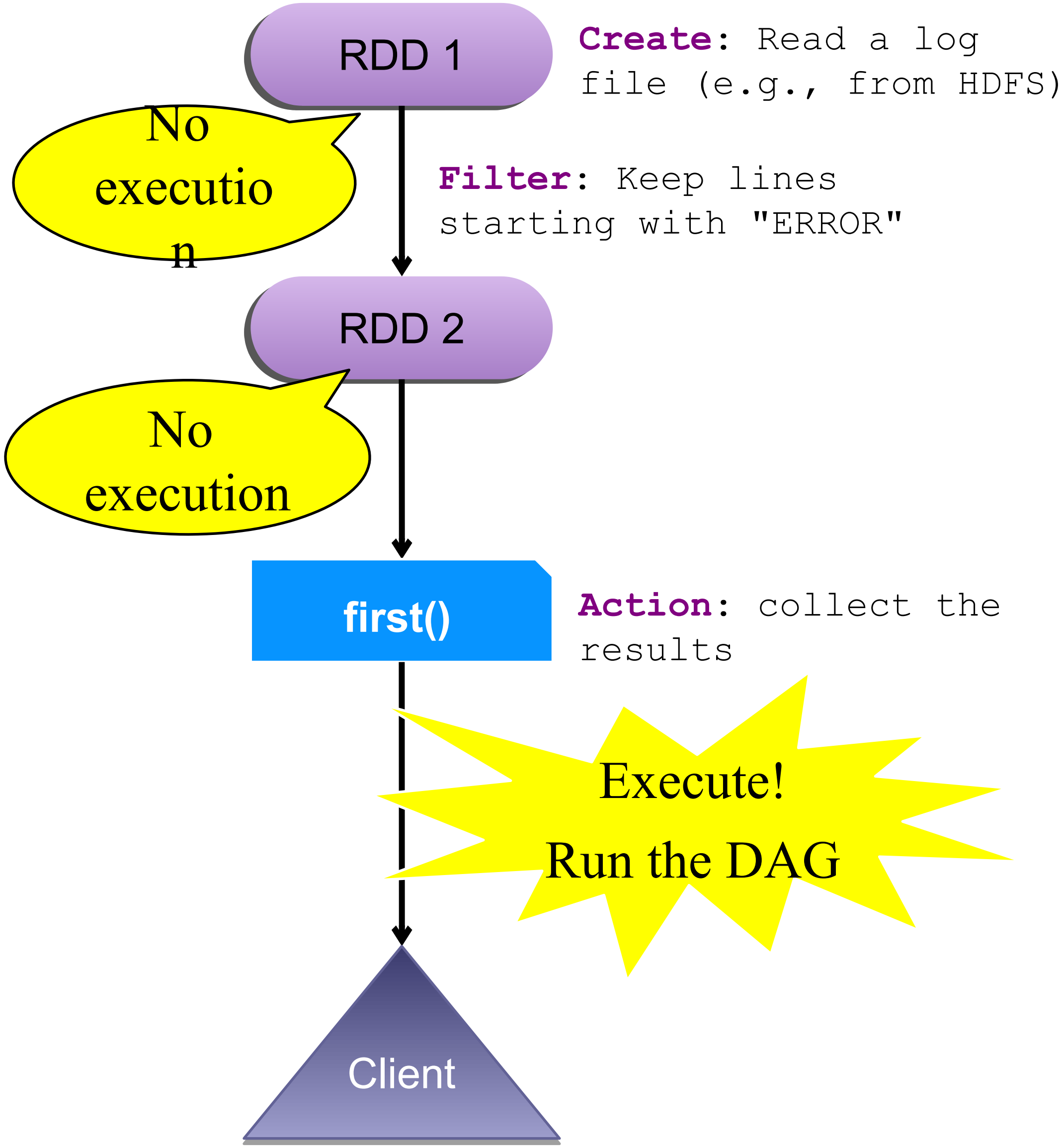
All Transformations are Lazy

- Spark engine **does not immediately** compute results
  - Transformations stored as a execution-graph (DAG)
  - They specify how to perform parallel computation

- The Execution-Graph is executed when an action occurs
  - When it needs to provide data

- Allows Spark to:
  - **Optimize** required calculations (we'll view this soon)
  - **Efficiently recover** RDDs on node failure (more on this later)

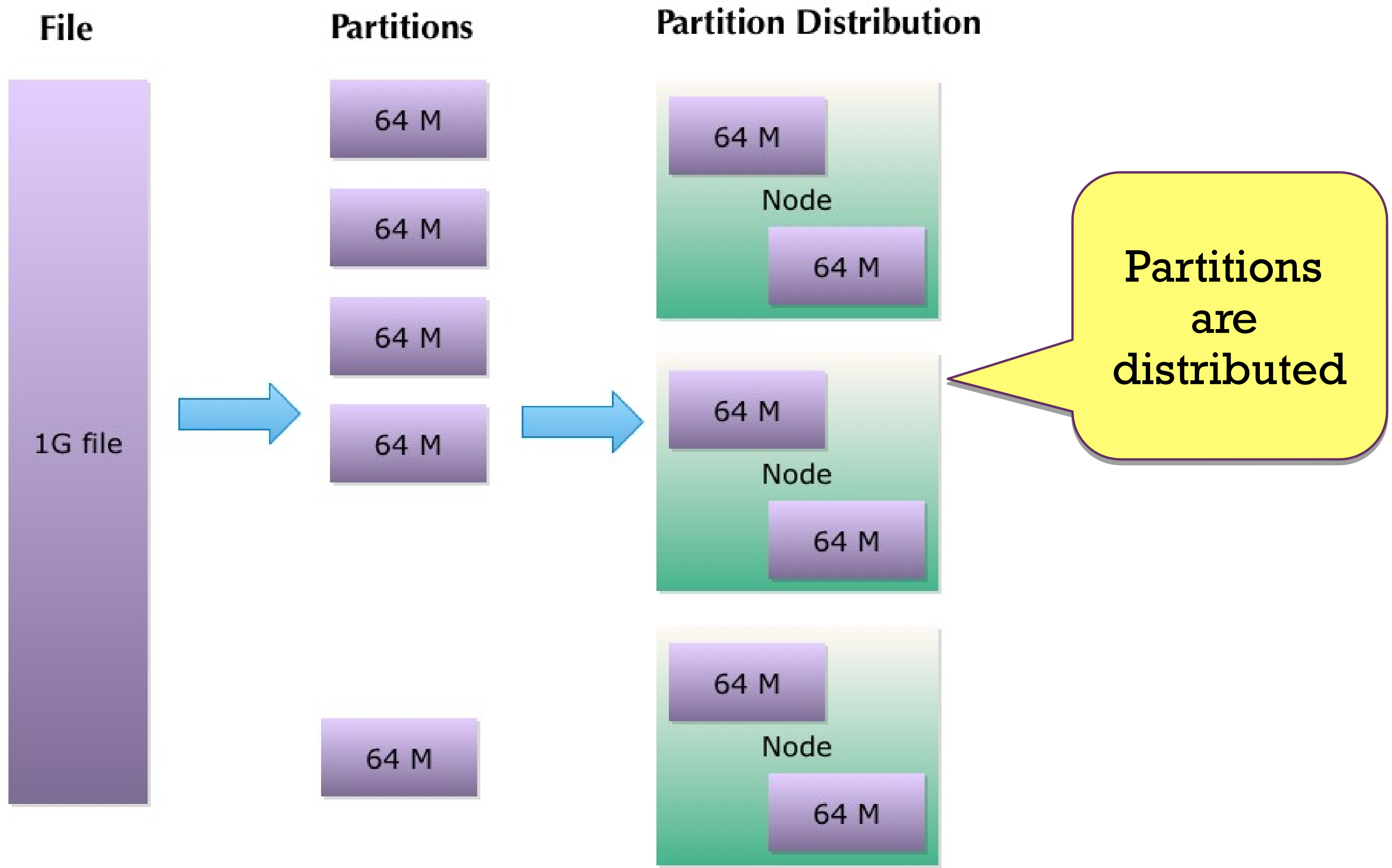
Lazy Evaluation



- This example reads a log file
  - It filters out all but error lines
  - Thus far, **no actual work was done**
- Client requests the first line
  - Triggers evaluation of the RDD DAG
  - **Now** the work is done
  - Result is sent to client
- Many possible optimizations
  - Stop filtering after the first ERROR line encountered
  - Doesn't even need to read all of the log file

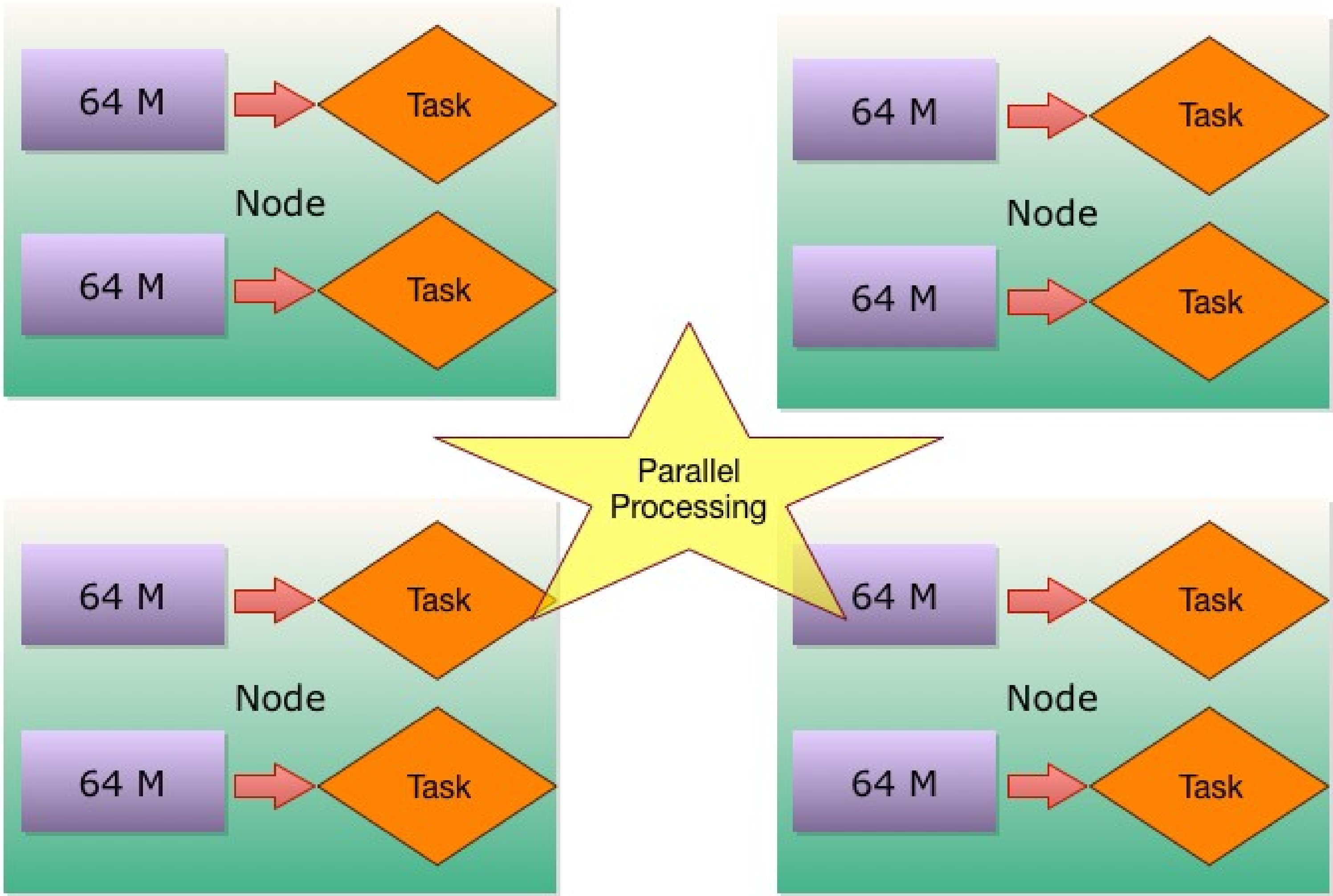
Partitioning

- Data is partitioned around the cluster
  - E.g., with HDFS, Spark creates partitions from HDFS blocks



Partitions and Parallel Processing

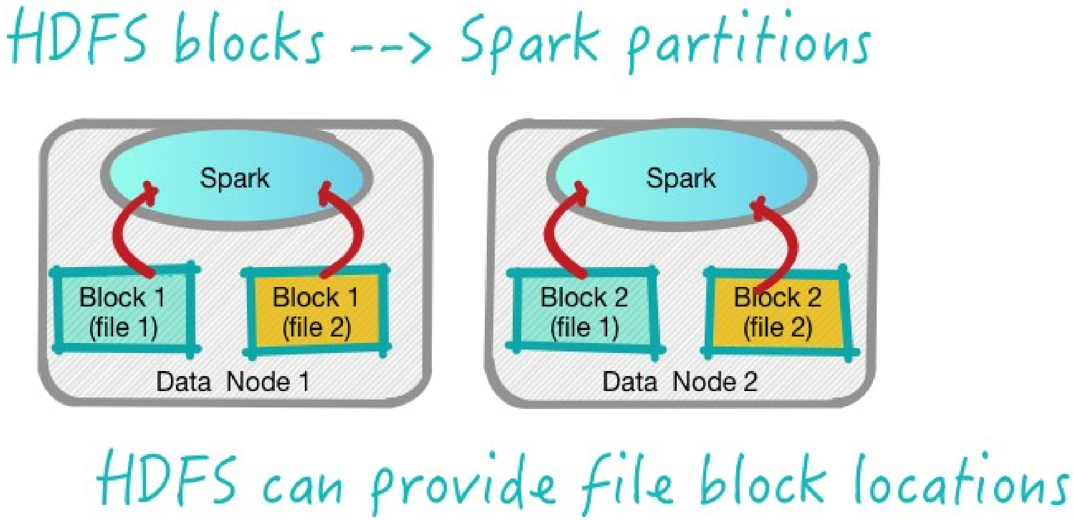
- Nodes execute tasks in parallel on the partitions
  - Spark will co-locate tasks with their data (HDFS block if HDFS)



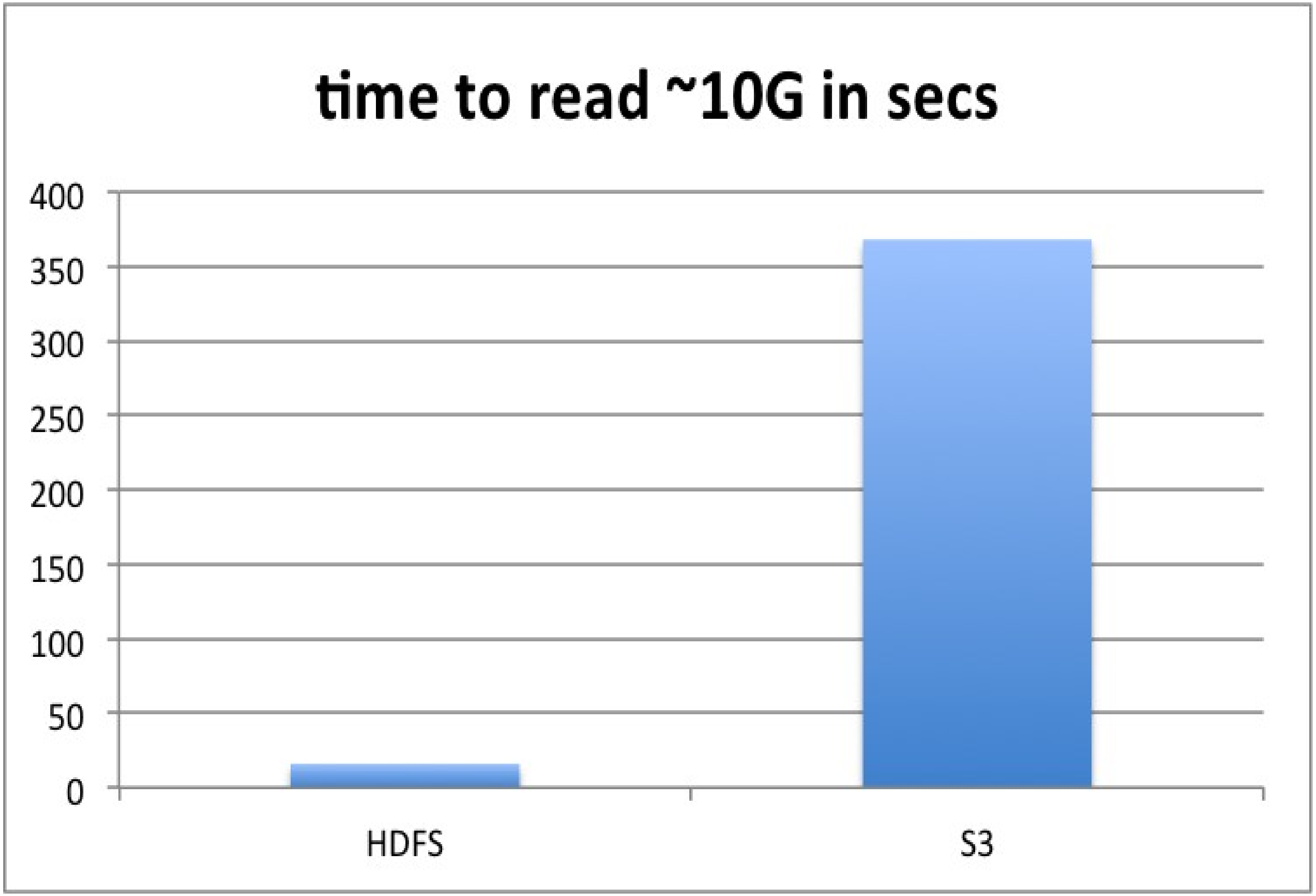
Spark & HDFS

- Spark can natively read / write data to HDFS
- HDFS can also provide 'location hints' for data, so Spark can do 'data local' processing.
  - → faster processing (no network IO)
- HDFS is a high-throughput distributed file system

```
val logs = sc.textFile("hdfs://namenode:9000/data/*.log")
logs.count
```

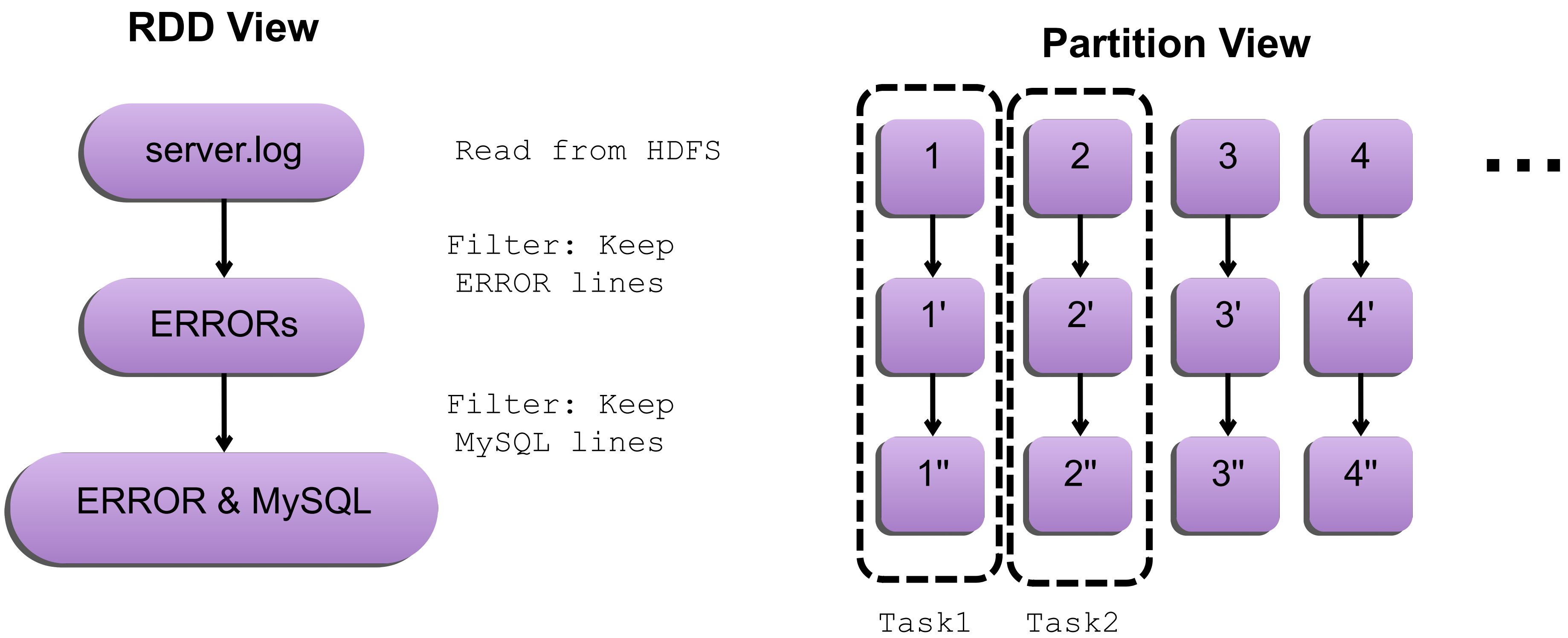


HDFS Vs. S3 (lower is better)



Transformations Generate New Partitions

- A transformation on a partition creates a partition of the new RDD/Dataset
  - Succeeding transformations on it may be pipelined on a task
  - Often, it can all be done with in-memory data (fast)
  - Some transformations require data shuffling (covered later)



Partitions Example

- We are reading a log file
  - Split into 3 partitions
  - Each line has : SEVERITY, COMPONENT, MSG

Partition 1	Partition 2	Partition 3
1. INFO, msg	6. INFO, msg	11. ERROR, MYSQL
2. ERROR, msg	7. ERROR, SPARK, msg	12. WARN, msg
3. ERROR, MYSQL, msg	8. ERROR, msg	13. INFO, msg
4. INFO, msg	9. WARN, msg	14. WARN, msg
5. ERROR, SPARK, msg	10. WARN, msg	15. INFO, msg

Transformations Example

Partition 1	Partition 2	Partition 3
INFO, msg	INFO, msg	ERROR, MYSQL
ERROR, msg	ERROR, SPARK, msg	WARN, msg
ERROR, MYSQL, msg	ERROR, msg	INFO, msg
INFO, msg	WARN, msg	WARN, msg
ERROR, SPARK, msg	WARN, msg	INFO, msg

Dataset1

Filter for lines containing ERROR

Partition 1a	Partition 2a	Partition 3a
		ERROR, MYSQL
ERROR, msg	ERROR, SPARK, msg	
ERROR, MYSQL, msg	ERROR, msg	
ERROR, SPARK, msg		

Dataset2

Uneven partitions

Transformations Example

Partition 1	Partition 2	Partition 3
INFO, msg	INFO, msg	ERROR, MYSQL
….	…..	……

Dataset1

Filter for lines containing ERROR

Partition 1a	Partition 2a	Partition 3a
		ERROR, MYSQL
ERROR, msg	ERROR, SPARK, msg	
ERROR, MYSQL, msg	ERROR, msg	
ERROR, SPARK, msg		

Dataset2

Filter for lines containing MYSQL

Partition 1b	Partition 2b	Partition 3b
		ERROR, MYSQL
ERROR, MYSQL, msg		

Dataset3

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @

 2019-03-12

Transformation Example

Dataset1

Partition 1	Partition 2	Partition 3
INFO, msg	INFO, msg	ERROR, MYSQL
….	…..	……

Filter for lines containing ERROR

Dataset2

Partition 1	Partition 2	Partition 3
		ERROR, MYSQL
ERROR, msg	ERROR, SPARK, msg	
ERROR, MYSQL, msg	ERROR, msg	
ERROR, SPARK, msg		

Coalesce (rebalances partitions)

Dataset3

Partition 1	Partition 2	Partition 3
ERROR, msg	ERROR, SPARK, msg	ERROR, MYSQL
ERROR, MYSQL, msg	ERROR, msg	ERROR, SPARK, msg

Even partitions

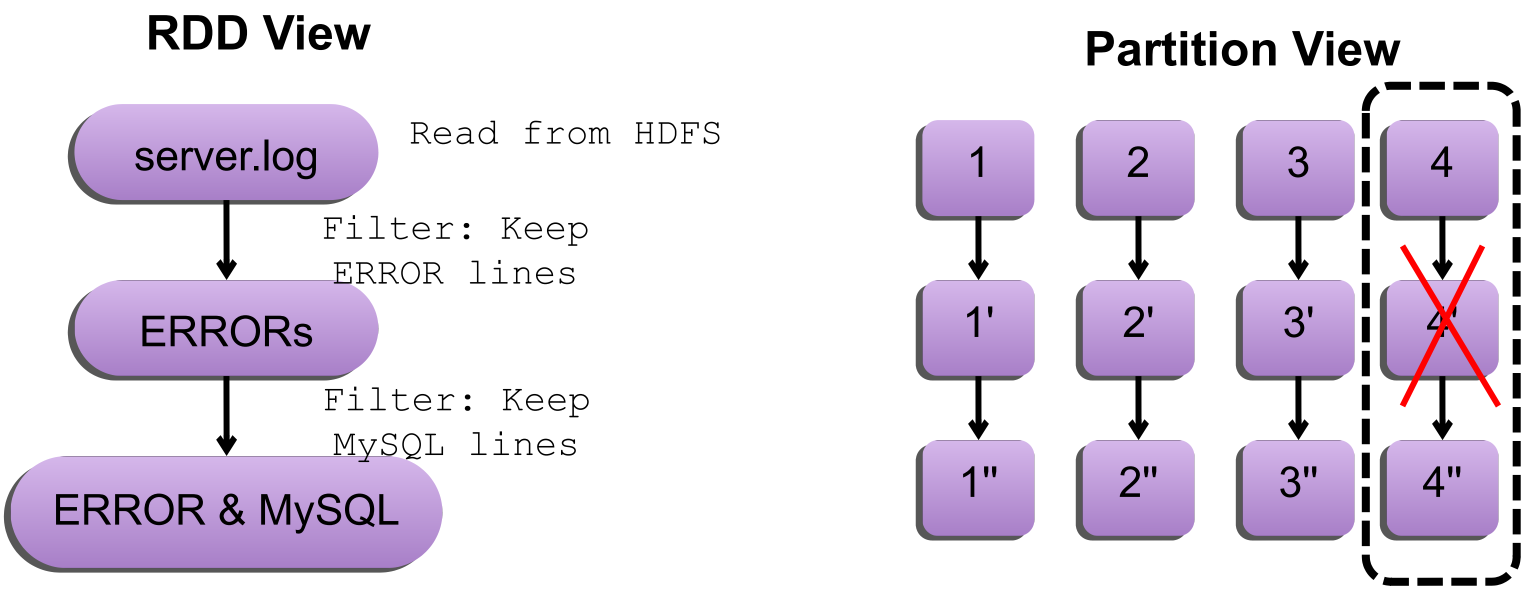
Coalesce vs. Repartition

- Repartition–either decreases or increases the number of partitions
- Coalesce–only decreases, and is more efficient

What	Why
Coalesce (numPartitions)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
Repartition (numPartitions)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

Fault Tolerance

- Spark tracks transformations that create an RDD/Dataset
  - **Lineage**: The series of transformations producing an RDD/Dataset
- A lost partition can be rebuilt from its lineage
  - E.g., if partition 4 is lost, Spark can read the HDFS block again, apply the transformations, and recover the partition
  - Efficient, and adds little overhead to normal operation

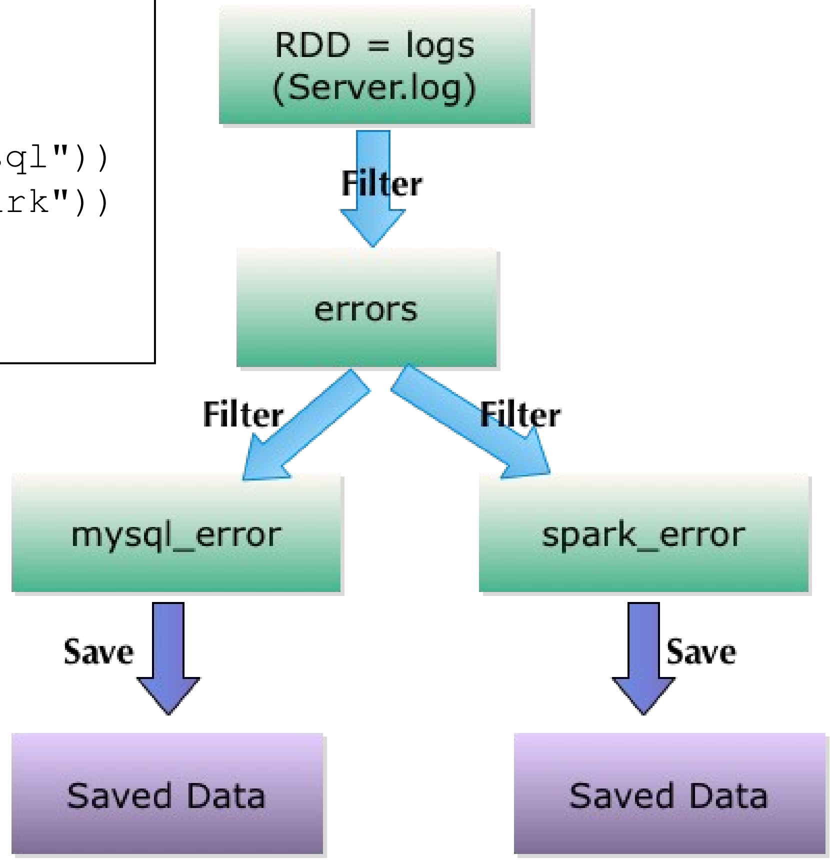


Anatomy of a Spark Job

Data Model Overview
RDD Concepts
➔ **Spark Workflow**
Working with RDDs
Caching
Key-Value Pairs

Spark Execution Workflow (DAG)

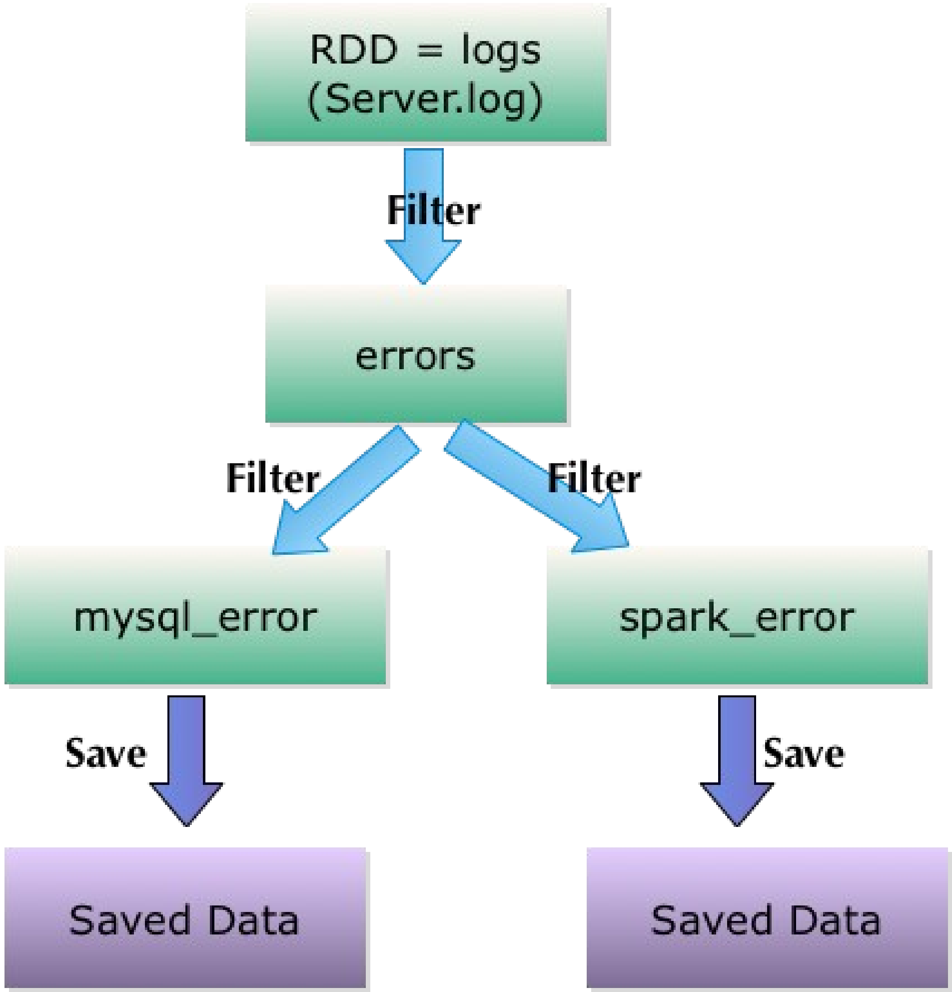
```
// sample job in scala

val logs = sc.textFile("server.log")

val errors = logs.filter(_.contains("Error"))
val mysqlError = errors.filter(_.contains("mysql"))
val sparkError = errors.filter(_.contains("spark"))

mysqlError.saveAsTextFile("mysql-error")
sparkError.saveAsTextFile("spark-error")
```



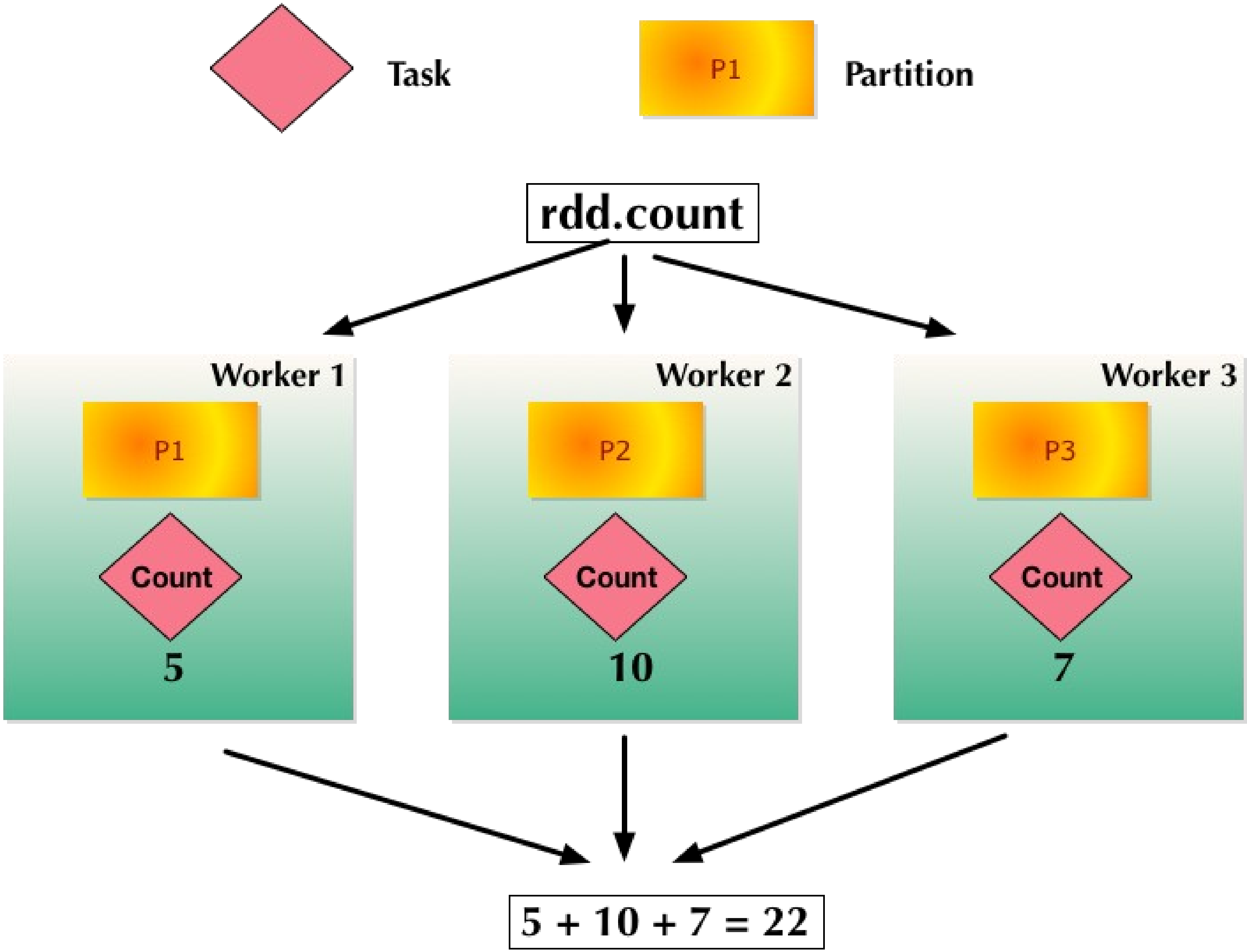
- Spark executes the workflow as a **DAG** (Direct Acyclic Graph)
  - Directed (data flows in a certain direction
  - Acyclic (no cycles/loops)

RDD/Datasets are Transient by Default

- Once an action completes, its RDDs/Datasets disappear (by design)
  - If you need one again, it's recomputed
- Here **'errors'** RDD/Dataset is transient
- You can tell Spark to persist an RDD to keep it in memory
  - Useful if you reuse an RDD and it is expensive to create

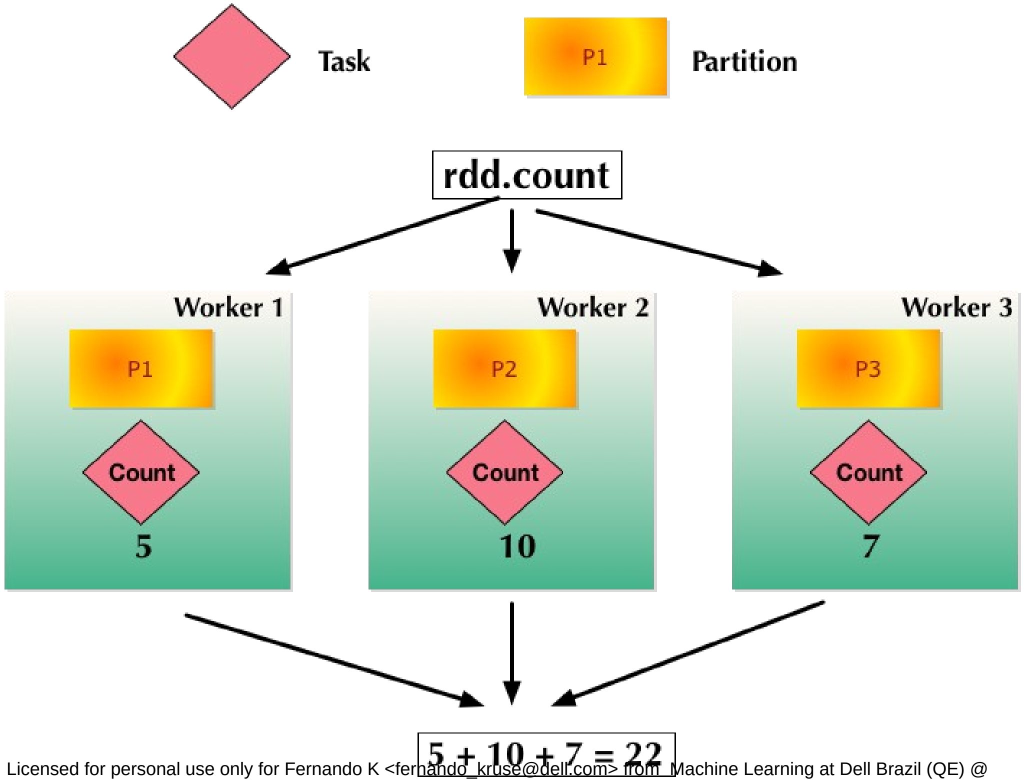


Distributed Execution



Distributed Execution Explained

- Here 'count' operation is 'parallelized' across workers
- Each worker runs a task
- Each task is operating on a partition
- And each task's count is then totaled together for the final count



Class Exercise

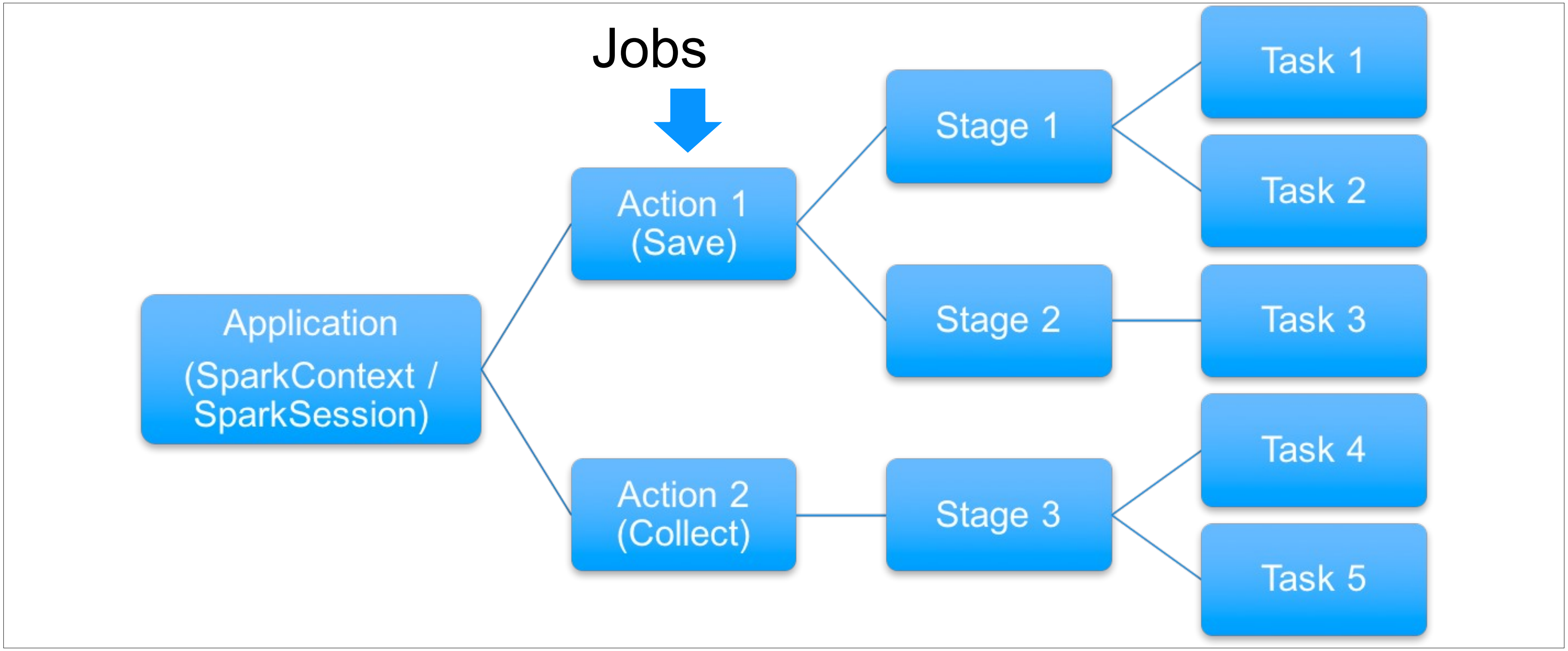
- Q1 : how can we find the MAX / MIN in a distributed fashion?

- Q2 : How can we find AVERAGE value in a distributed fashion?

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Anatomy of Spark Job

- Application can have many **actions → jobs**
- A **Job** may be executed in one or many stages (depending on the complexity)
- A **Stage** may have one or more **tasks**



Anatomy of a Spark Job : Stage

- Stage is
  - Collection of tasks that can be executed in ONE Executor
  - Without talking to another Executor
- If network communication is required then another Stage begins
  - E.g. shuffle operation
- Operations that cause a shuffle operation
  - Sort, groupByKey, Join
- Stages for a Job are usually executed in sequence
  - One Stage's output is fed as input another Stage

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Working with RDDs

Data Model Overview
RDD Concepts
Spark Workflow
➔ **Working with RDDs**
Caching
Key-Value Pairs

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Creating an RDD (Scala)

- Two ways to create
  - **Load data file(s)**: From local or distributed file system
  - **Parallelize a collection**: For small data or testing
    - It will all have to fit in memory on your driver node

```
// Turn a Scala collection into an RDD
val numbers = sc.parallelize (List(1,2,3,3))

// Create from local file
val oneFile = sc.textFile("README.md")

// Create from multiple files
val multiFile = sc.textFile("data/mllib/*.txt")

// Create from a file in HDFS
val hdfsFile =
     sc.textFile("hdfs://namehost:9000/student/myfile.txt")
```

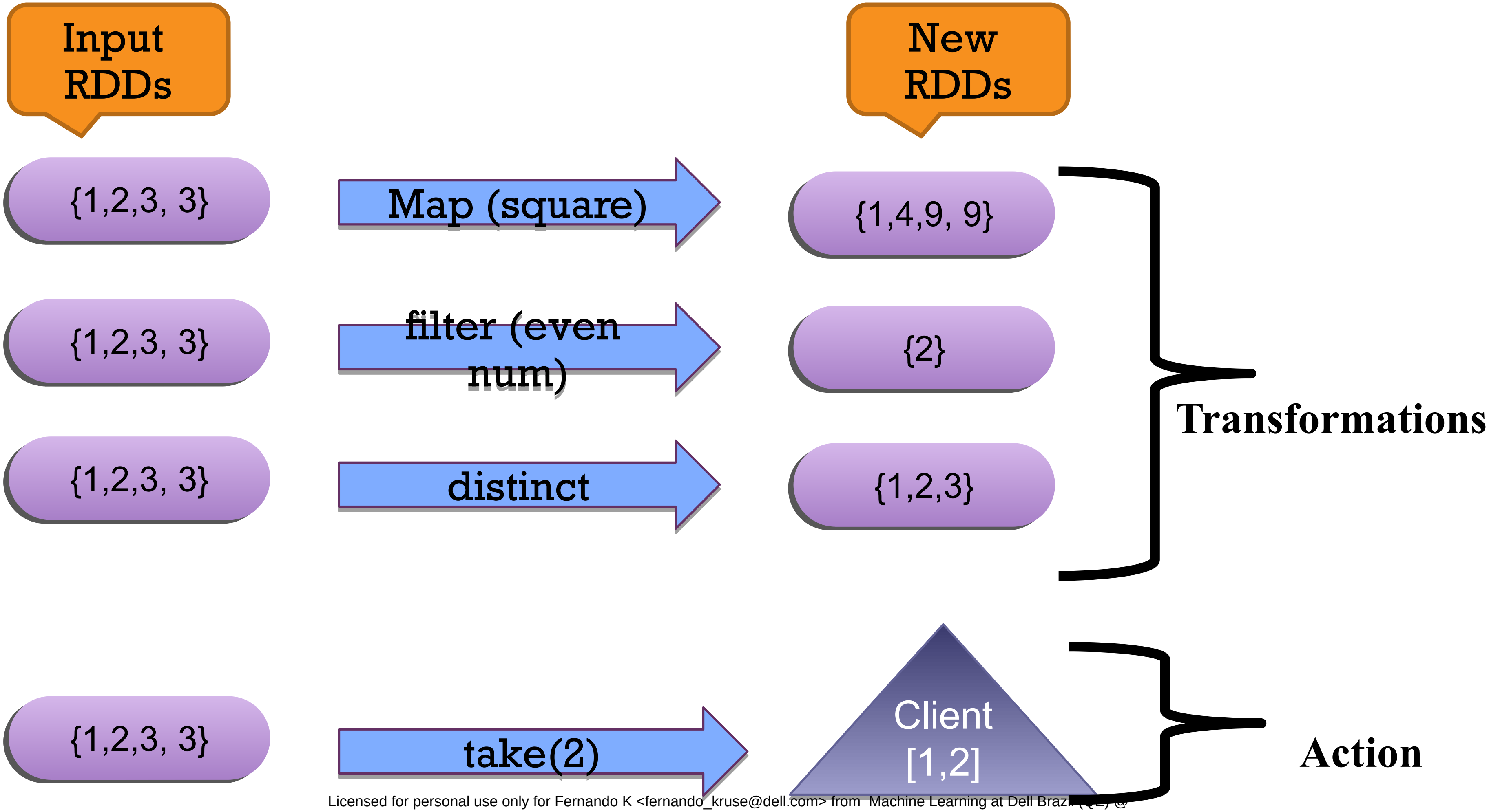
Creating an RDD (Python)

- Two ways to create
  - **Load data file(s)**: From local or distributed file system
  - **Parallelize a collection**: For small data or testing
    - It will all have to fit in memory on your driver node

```
// Turn a python collection into an RDD
numbers = sc.parallelize ( (1,2,3,3))

// Create from local file
oneFile = sc.textFile("README.md")

// Create from multiple files
multiFile = sc.textFile("data/mllib/*.txt")

// Create from a file in HDFS
hdfsFile =
     sc.textFile("hdfs://namehost:9000/student/myfile.txt")
```

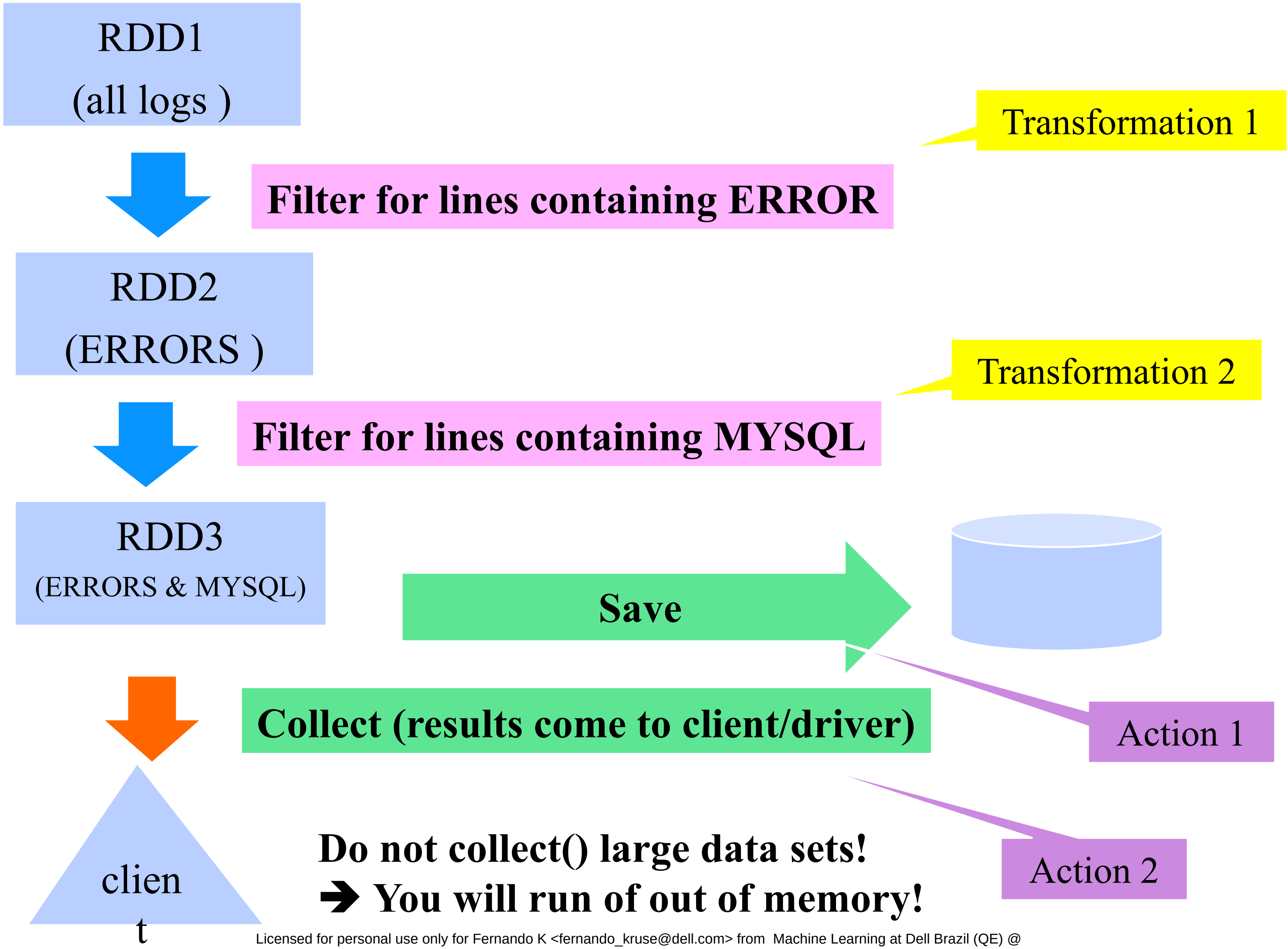
Transformation/Action Examples

- Below, we illustrate three transformations and an action
  - Let's look at how to code some of these



Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

RDD Workflow Example



Do not collect() large data sets!
➔ You will run of out of memory!

map() (Scala)

- `map(f: (T)=>U)` : Applies function f to all elements
  - `f()` takes one argument, returns a mapped value
  - **Return**: New RDD of mapped elements (may be a different type)
  - See next slide for illustration
- Below is a simple map example
  - Creates a new RDD, containing squares of the input RDD
  - Argument to `map()` is an anonymous function
  - See notes for a brief discussion

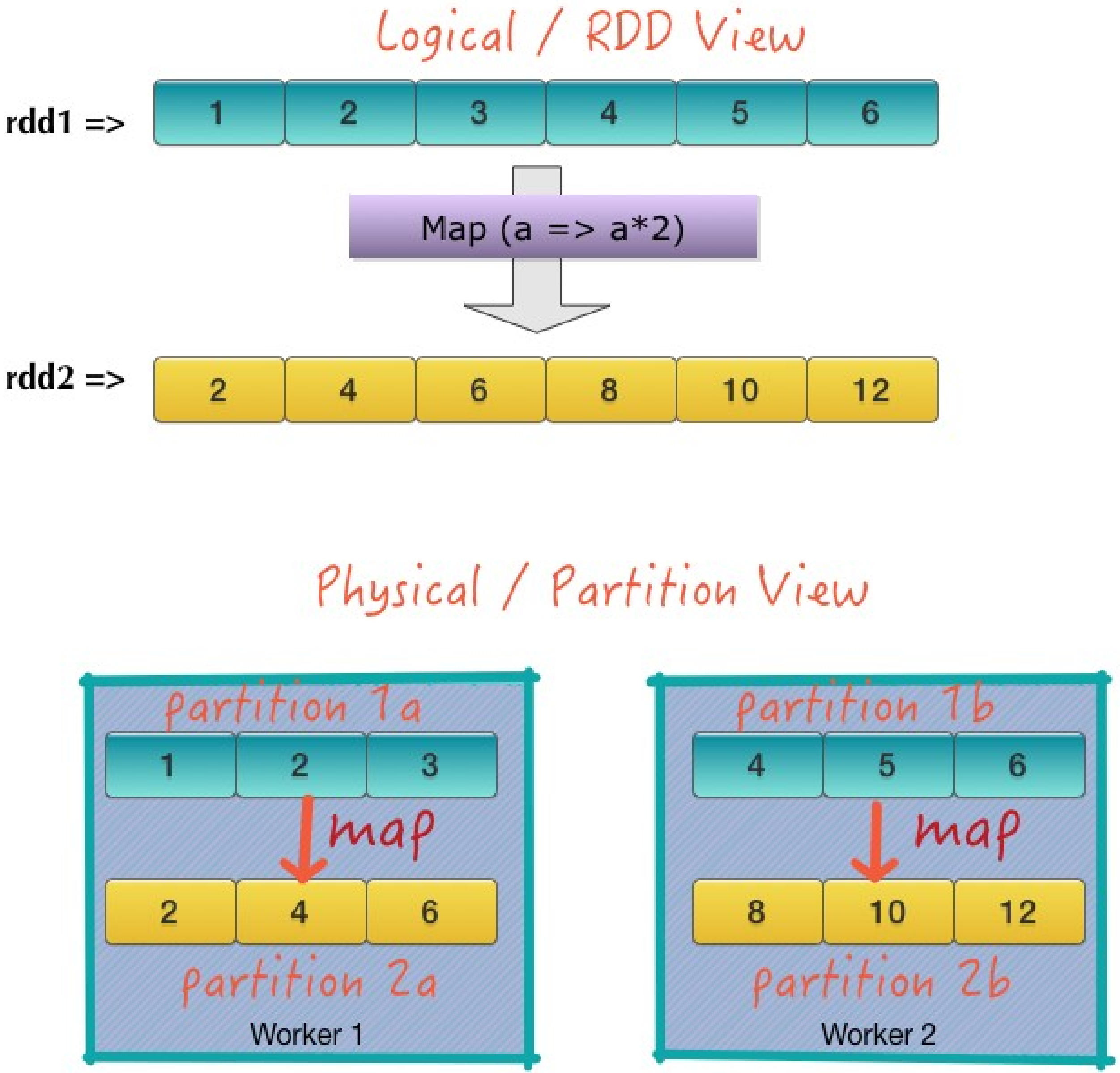

```
> val numbers = sc.parallelize (List(1,2,3,3))

// Map numbers RDD by squaring each element
> val squares=numbers.map(x=> x*x)

// Collect all data in the RDD
> squares.collect()
res0: Array[Int] = Array(1, 4, 9, 9)
```

map() (Python)

- `map(f: (T)=>U)` : Applies function f to all elements
  - **Return**: New RDD of mapped elements (may be a different type)
  - See next slide for illustration
- Below is a simple map example
  - Creates a new RDD, containing squares of the input RDD
  - Uses Python Lambda expression
  - See notes for a brief discussion


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

// Map numbers RDD by squaring each element
> squares=numbers.map(lambda x: x*x)

// Collect all data in the RDD
> squares.collect()
[1, 4, 9, 9]
```

Map in Spark Illustrated



Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

filter() Details (Scala)

- **`filter( f(T=>Bool) )`**: Filters elements on predicate `f()`
  - `f()` takes one argument, returns a boolean
  - **Return**: New RDD with elements where `f(element)==true`
    - It has the same element type as the parent

- Below is a simple filter example
  - Creates a new RDD, containing odd numbers from the input RDD

```
> val numbers = sc.parallelize (List(1,2,3,3))

// Filter numbers RDD by ODDs out of elements
> val odds=numbers.filter(x=> x%2 == 1)

> odds.collect()
Array[Int] = Array(1, 3, 3)
```

filter() Details (Python)

- **`filter( lambda: expression`**
  - Lambda expression evaluates to boolean (true / false)
  - **Return**: New RDD with elements where `f(element)==true`
    - It has the same element type as the parent

- Below is a simple filter example
  - Creates a new RDD, containing odd numbers from the input RDD

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

odds = numbers.filter(lambda x: x % 2 == 1)

odds.collect()
[1, 3, 3]
```

Actions Overview (Scala)

- Below, we illustrate some common actions and results
- Actions materialize data

```
> val numbers = sc.parallelize (List(1,2,3,3))
> val odds=numbers.filter(x=> x%2 == 1)

// Get count of RDD
> numbers.count()
Long = 4

> odds.count()
Long = 3

// Get first two elements
> odds.take(2)
Array[Int] = Array(1, 3)
```

Actions Overview (Python)

- Below, we illustrate some common actions and results
- Actions materialize data

```
numbers = sc.parallelize ([1,2,3,3])
odds = numbers.filter(lambda x: x % 2 == 1)

// Get count of RDD
numbers.count()
4

odds.count()
3

// Get first two elements
odds.take(2)
[1,2]
```

reduce() Details (Scala)

- **`reduce(f: (T, T) ⇒ T):`** reduces elements using f
  - `f()` operates on two elements of the RDD, returns one element
    - For example, adding two numbers together
  - **Return**: Single element of the same type
    - f is applied repeatedly until only one element left (the result)

- Below, we show two examples of reduce

```
> val numbers = sc.parallelize (List(1,2,3,4))

// Reduce by adding all numbers together
> numbers.reduce ( (a,b)=> a+b)
Int = 10

// Reduce by multiplying (factorial)
> numbers.reduce ( (a,b)=> a*b)
Int = 24
```

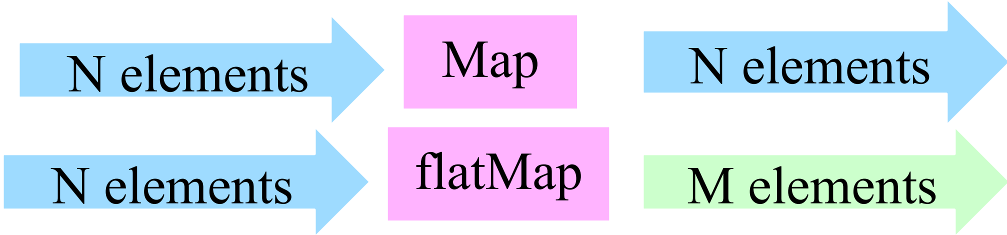
reduce() Details (Python)

- `reduce(f: (T, T) ⇒ T):` reduces elements using f
  - `f()` operates on two elements of the RDD, returns one element
    - For example, adding two numbers together
  - **Return**: Single element of the same type
    - f is applied repeatedly until only one element left (the result)
- Below, we show two examples of reduce

```
>>> numbers = sc.parallelize ([1,2,3,4])

// Reduce by adding all numbers together
>>> sum = numbers.reduce(lambda accum, n:accum + n)
>>> print(sum)
10

// Reduce by multiplying (factorial)
>>> factorial = numbers.reduce(lambda accum, n:accum * n)
>>> print(factorial)
24
```

flatMap() Details (Scala)

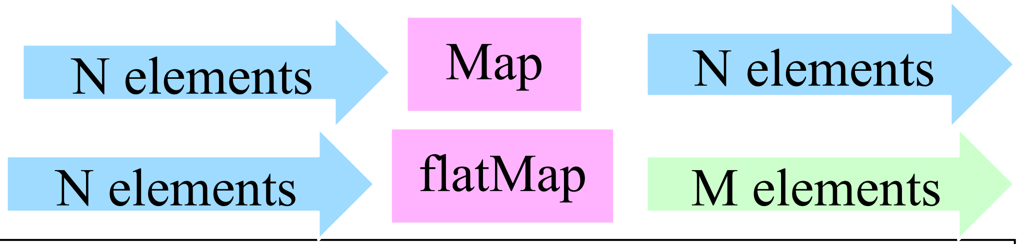
- **flatMap(f: (T) ⇒ TraversableOnce[U])**: Applies function f to all elements, then flattens the results
  - f returns an object that can be iterated over (e.g., a collection)
  - The elements in each iterator are then combined to create the RDD ("flattened")




```
> val numbers = sc.parallelize (List(1,2,3))

// Map each number by creating list of 3 numbers
> val mapped = numbers.flatMap(a=> List(a-1, a, a+1))
mapped: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[4]
at flatMap at <console>:23

> mapped.collect  // 9 element - 3 from each original
Array[Int] = Array(0, 1, 2, 1, 2, 3, 2, 3, 4)
```

flatMap() Details (Python)

- **flatMap(f: (T) ⇒ TraversableOnce[U])**: Applies function f to all elements, then flattens the results
  - f returns an object that can be iterated over (e.g., a collection)
  - The elements in each iterator are then combined to create the RDD ("flattened")




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

// Map each number by creating list of 3 numbers
mapped = numbers.flatMap(lambda x: [x-1, x, x+1])

mapped.collect()  // 9 element - 3 from each original
[0, 1, 2, 1, 2, 3, 2, 3, 4]
```

union() Details (Scala)

- **union(other: RDD[T]): RDD[T]**: Returns union of this RDD with another RDD
- Operates on two RDDs
- Duplicates are included
  - Use distinct to remove them

```
> val odds = sc.parallelize (List(1,3,5,7))
> val evens = sc.parallelize (List(2,4,6,8))

// Create union
> val all = odds.union(evens)
all: org.apache.spark.rdd.RDD[Int] = UnionRDD[2] at
union at <console>:25

> all.collect.sorted
Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8)
```

union() Details (Python)

- **union(other: RDD[T]): RDD[T]**: Returns union of this RDD with another RDD
- Operates on two RDDs
- Duplicates are included
  - Use distinct to remove them

```
>>> odds = sc.parallelize ([1,3,5,7])
>>> evens = sc.parallelize ([2,4,6,8])

// Create union
>>> a = odds.union(evens)

>>> a.collect()
[1, 3, 5, 7, 2, 4, 6, 8]
```

RDD Transformations Summary (1 of 2)

RDD r = {1,2,3,3}

Transformation	Description	Example	Result
map(func)	apply func to each element in RDD	r.map(x => x*2)	{2,4,6,6}
filter(func)	Filters through each element when func is true (aka grep)	r.filter(x=> x % 2 == 1)	{1,3,3}
distinct	Removes dupes	r.distinct()	{1,2,3}
flatMap	Like map, but can output more than one result per element		
mapPartitions	Like map, but runs on the whole partition not on each element		

RDD Transformations Summary (2 of 2)

RDD r1 = {1,2,3,3} RDD r2 = {2,4}

Transformation	Description	Example	Result
union(RDD)	Merges two RDDs (duplicates are kept)	r1.union(r2)	{1,2,3,3,2,4}
Intersection(RDD)	Returns common elements in two RDDs	r1.intersection(r2)	{2}
subtract(RDD)	Takes away elements from one	r1.subtract(r2)	{1,3,3}
sample	Take a small sample from RDD		

RDD Actions Summary

- Note that actions return values, not an RDD
  - E.g., `count()` returns a long, and `take()` returns an Array

RDD r = {1,2,3,3}

Action	Description	Example	Result
count()	Counts all records in an rdd	r.count()	4
first()	Extract the first record	r.first ()	1
take(n)	Take first N lines	r.take(3)	[1,2,3]
collect()	Gathers all records for RDD. All data has to fit in memory of ONE machine (don't use for big data sets)	r.collect()	[1,2,3,3]
saveAsTextFile ()	Saves to storage		
	…… many more ….		

Review the RDD Documentation

- We'll take a few minutes to briefly review the RDD API docs
  - They're hosted on the Spark Apache website

Mini-Lab

- Browse to **http://spark.apache.org/docs/latest/**
  - On the top menu bar, go to **API Docs | Scala**
  - In the left hand pane, find the **`org.apache.spark.rdd`** package
  - Within that package, click on the RDD entry
  - This brings you to the RDD API documentation
- Review the RDD API briefly
  - In particular, look at `map()`, `filter()`, `count()`, and `take()`

Lab 3.1 & 3.1b : RDD & Dataset Basics

Lab

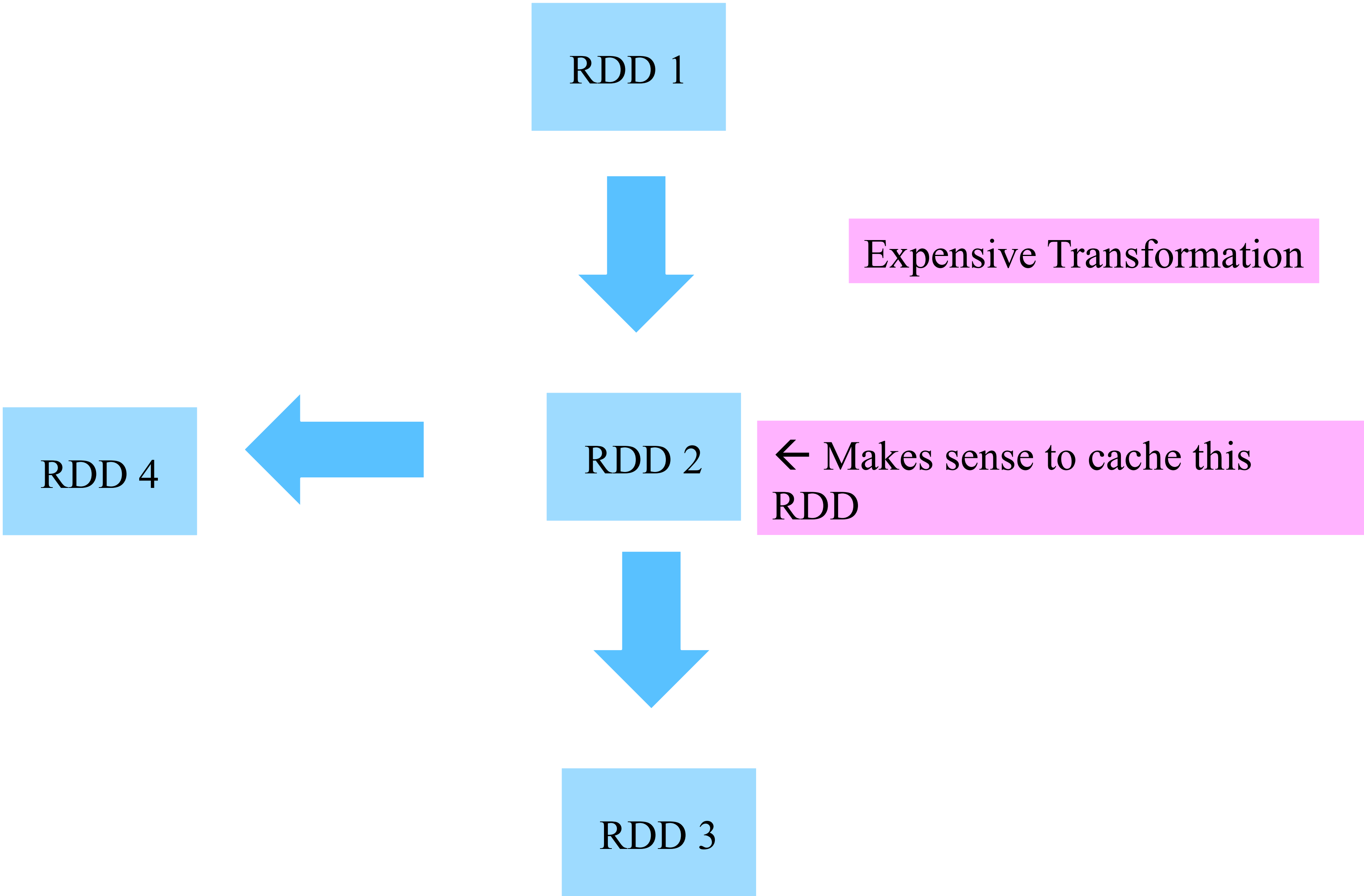
- **Overview**: In this lab, we will create and work with RDDs and Datasets
- **Builds on previous labs**: Lab 2.1 for general setup
- **Approximate time**: 30-40 minutes
- **Instructions:**
  - **Standalone:**
    - **3.1-rdd-basics**
    - **3.1b-dataset-basics**
  - **Hadoop: spark/2-RDD.md**
- **Solution (Instructor to update):**
  - **/spark/solutions/3-rdd/3.1-RDD-basics-solutions.md**
  - **/spark/labs-solutions/3-rdd/3.1b-dataset-basics-solutions.md**

Caching

Motivation for Caching

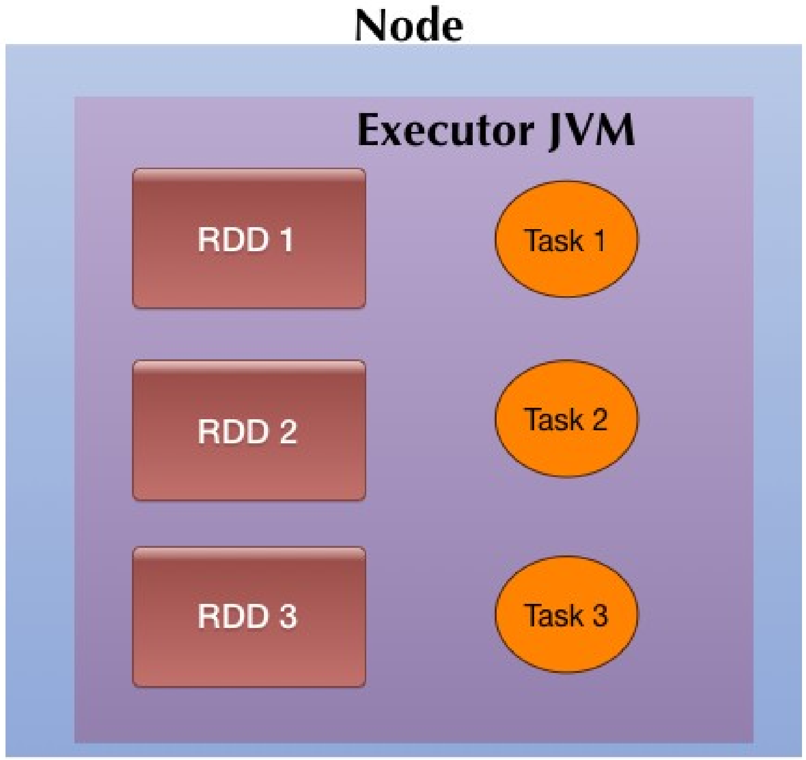
- Standard Spark job sequence:
  - Build a graph of transformations
  - Upon an action, run the transformations, get the result
  - Don't save any intermediate RDDs
- This is intentional
  - You may have a LOT of (big) data being processed
  - Sometimes, though, you do want to cache an RDD
- Spark can persist an RDD across operations
  - You must explicitly ask for this
- Caching use cases
  - Saving a result of an extensive computation
  - Iterative workloads (machine learning)

Case For Caching



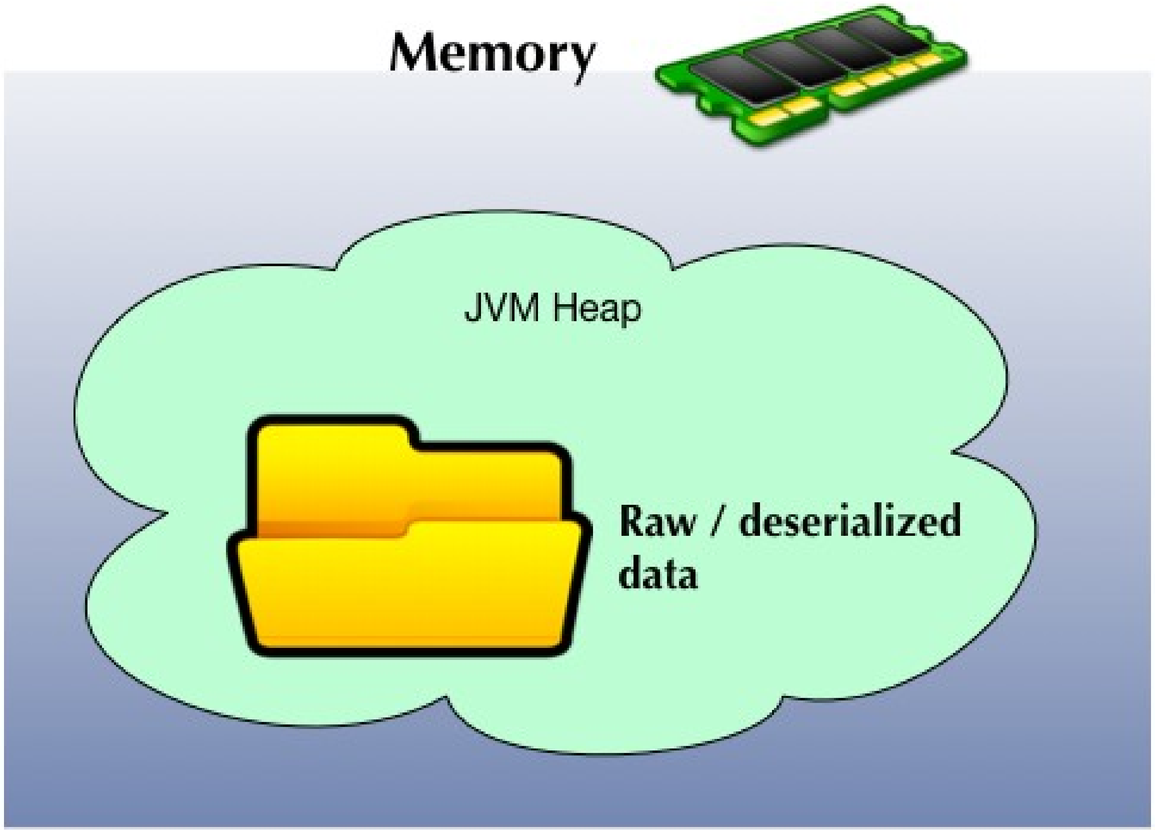
Caching Mechanics

- Cache into
  - Memory
  - Disk
  - Or combination
- Memory caching is done by executors on worker nodes
- Beware of JVM memory limits
  - Min JVM memory: 4-8G
  - Max JVM memory: 40G (larger will take longer to GC)
  - New GC in Java 8 ('G1') might be able to help



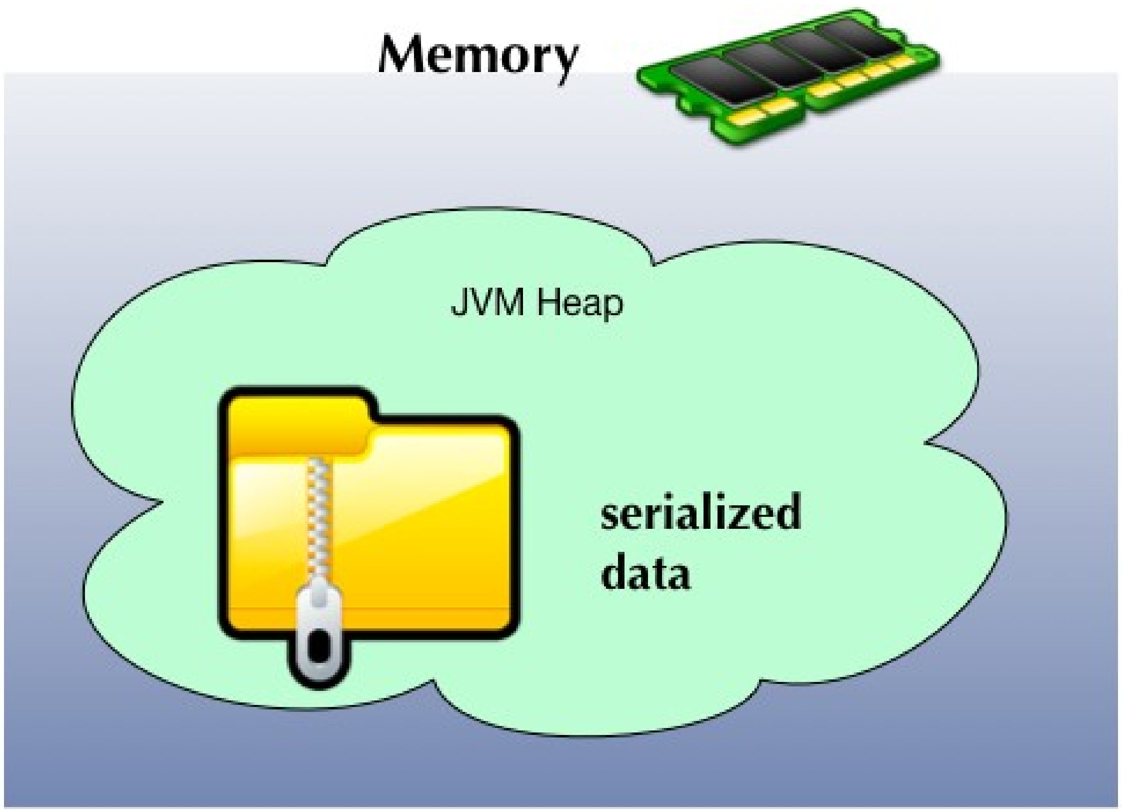
RDD Memory Caching - Raw

- Rdd.cache() == rdd.persist(MEMORY_ONLY)
- Most CPU efficient
- Data stored as "raw"/deserialized
- Takes up memory (3x–5x)
- 1G raw data might use 3G–5G memory



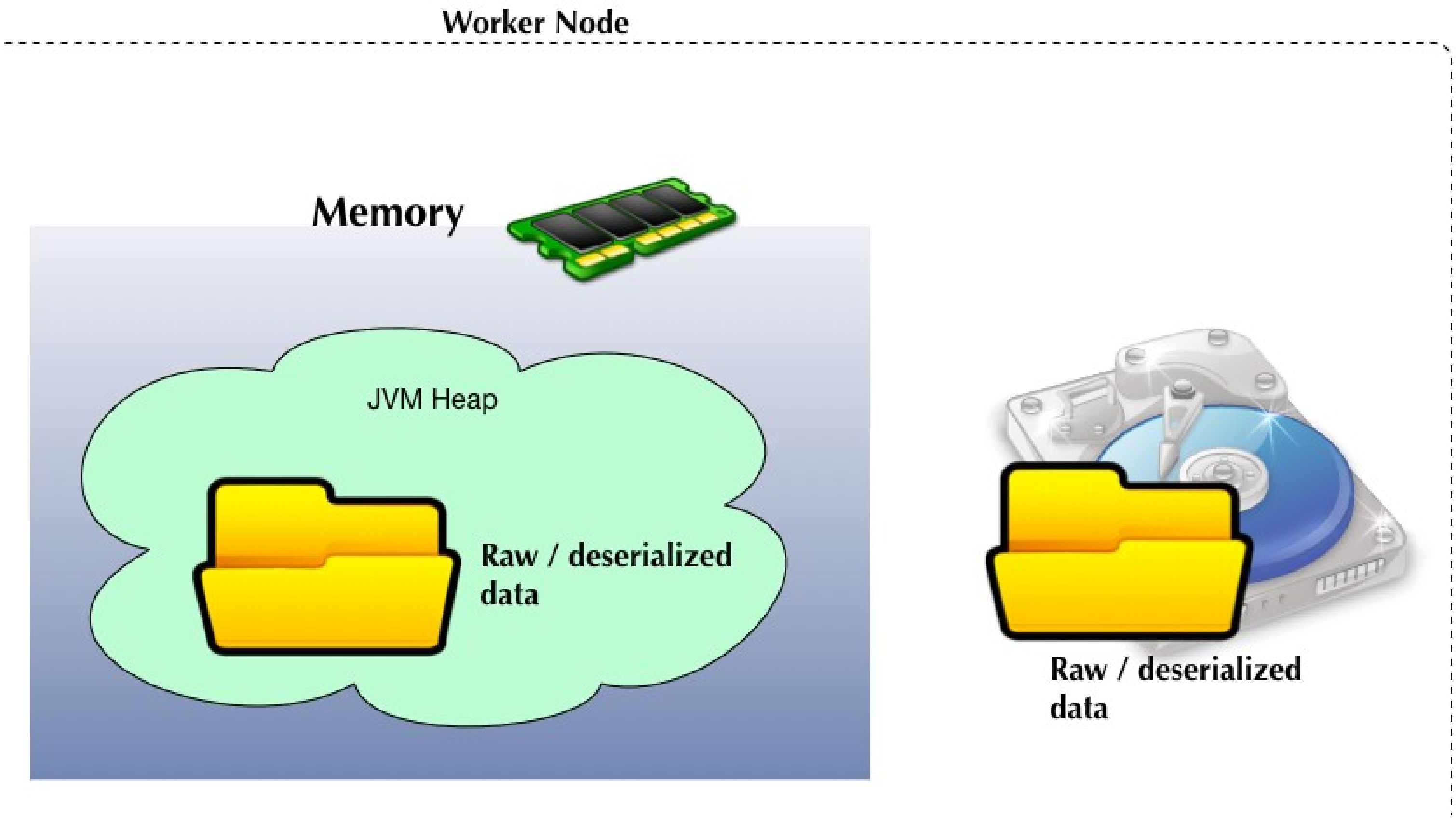
RDD Memory Caching 2–Serialized

- Rdd.persist(MEMORY_ONLY_SER)
- Most memory efficient option
- Little overhead (1G data might take about the same memory)
- CPU intensive (need to serialize/ de-serialize)
- Default Java serializer is OK
- Use 'kryo' serializer (version 2) for high, fast performance
  - Kryo is also more compact than Java.



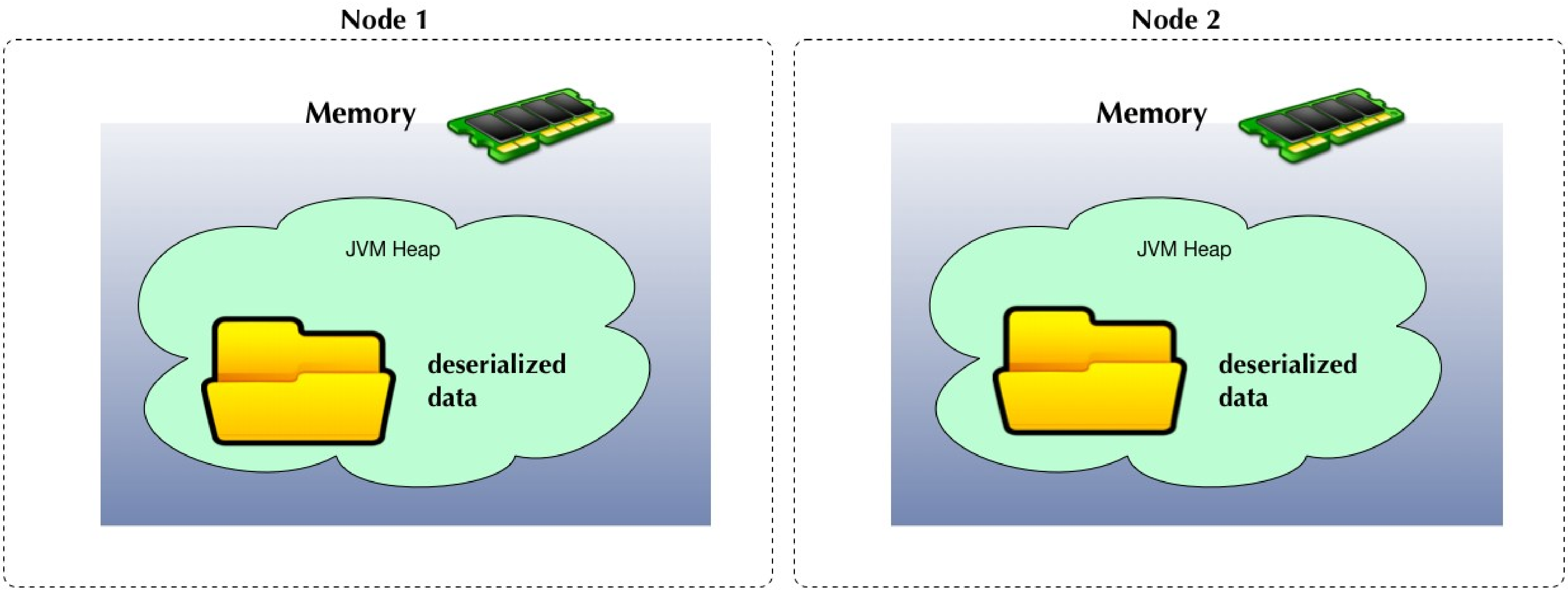
Memory and Disk Caching

- Rdd.persist(MEMORY_AND_DISK)
- Both in memory & disk
- Can survive memory eviction



Caching on Multiple Nodes

- RDD.persist(MEMORY_ONLY_2)
- Survives a node failure



RDD Persistence Levels

Storage Level	Behavior
MEMORY_ONLY (default level)	Store as deserialized Java objects in JVM. If RDD doesn't fit in memory, some partitions not cached, and recomputed.
MEMORY_AND_DISK	Store as deserialized Java objects in JVM. If RDD doesn't fit in memory, store those partitions on disk, and read as needed.
MEMORY_ONLY_SER	Store as serialized Java objects. Generally more space-efficient than deserialized, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2,MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (Tachyon)	RDD in serialized format in Tachyon. Reduces garbage collection overhead as well as other benefits.

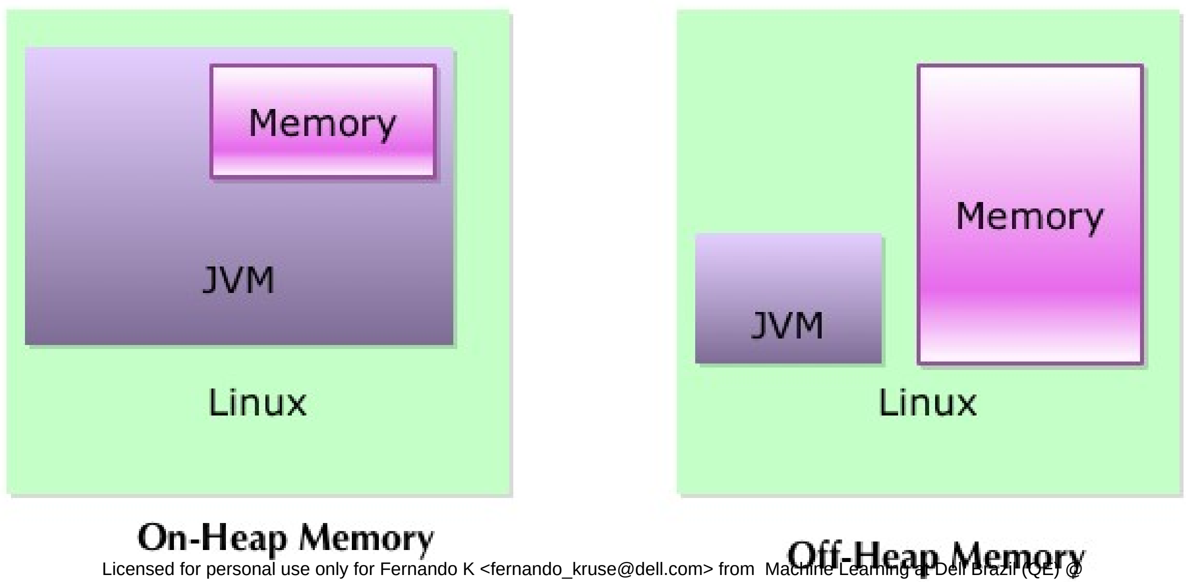
Off-Heap Caching

- Caching large amounts of data in JVM heap is problematic
  - Garbage collectors will "pause" all operations in JVM when they are reclaiming a large amount of memory (100G+)
  - This will make executor process look "dead" → causes all kinds of failures in the cluster
- JVM memory limitations have become a blocker in Big Data
- Solutions were created to "bypass" JVM
  - Cassandra was the first system to experiment with "off-heap" with very good results
  - Many others followed the example

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
 2019-03-12

Off-Heap Caching with Tachyon

- Tachyon by-passes JVM and allocates memory directly from Linux (like "malloc" ☺)
- Manages memory explicitly (no JVM and no Garbage Collector involved)
- Uses "custom encoders" to store Java objects in a very compact form
- Datasets by default use Tungsten engine for caching

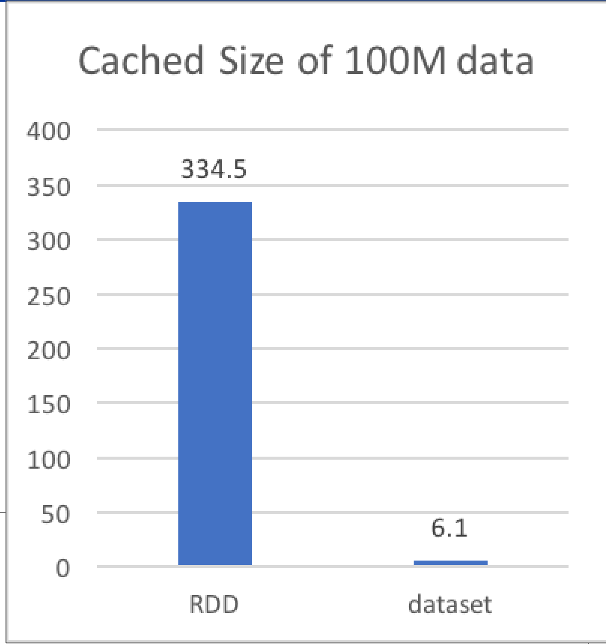


Comparing Caching Performance: RDD vs. DataSet

```
// rdd
val rdd = sc.textFile("100M.data")
rdd.count // 0.9 secs
rdd.cache
rdd.count // 1.7 sec  (first time after cache!)
rdd.count // 0.046 sec

// dataset
val dataset = spark.read.textFile("100M.data")
dataset.count //  1.2 sec
dataset.cache
dataset.count  // 1.7 secs (first time after cache!)
dataset.count // 0.033 secs
```

Comparing Caching Performance: RDD vs. DataSet

- Tungsten caching is really effective!



Storage

RDDs

100M data

dataset = spark.read.textFile()

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
*FileScan text [value#0] Batched: false, Format: Text, Location: InMemoryFileIndex[file:/home/ubuntu/data/twinkle/100M.data], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<value:string>	Memory Deserialized 1x Replicated	2	100%	6.1 MB	0.0 B
data/twinkle/100M.data	Memory Deserialized 1x Replicated	4	100%	334.5 MB	0.0 B

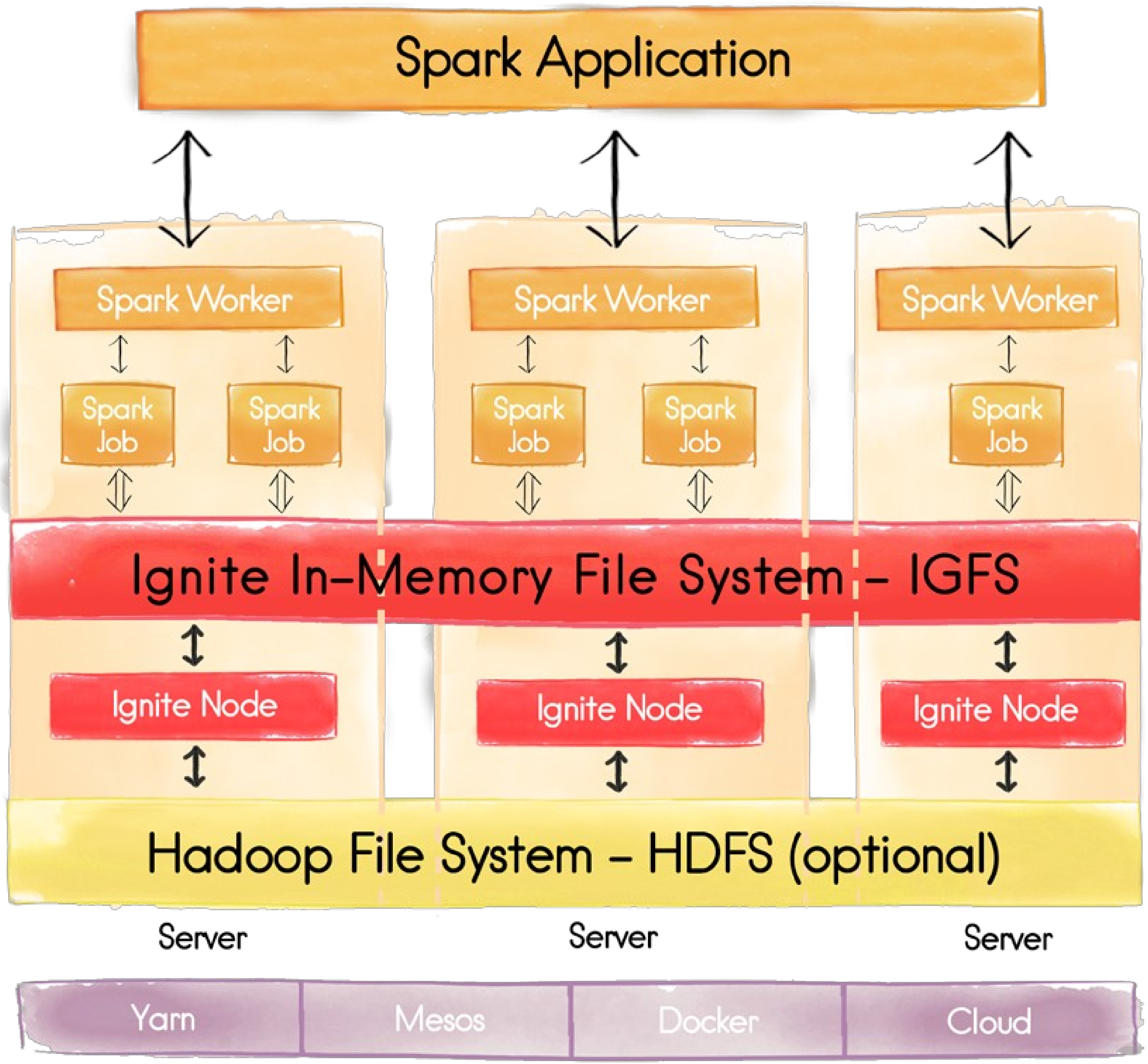
rdd = sc.textFile()

Distributed In-Memory File Systems

- **"Memory is the new disk"**
- Memory prices have been falling
  Year 2000 = $1000/GB
  Year 2016 = $3/GB
- Typical Hadoop/Spark node has 100–300 G memory
  - 10 node cluster @ 256 GB each = 2 TB of distributed memory!
- In-memory processing is very attractive for iterative workloads like machine learning
- Baidu uses 100 node spark cluster with 2 PB of memory

In-Memory File Systems

- Tachyon (Now "Alluxio")
  - Came out of Berkeley AMP lab (same incubator as Spark)
- Ignite
  - From GridGain

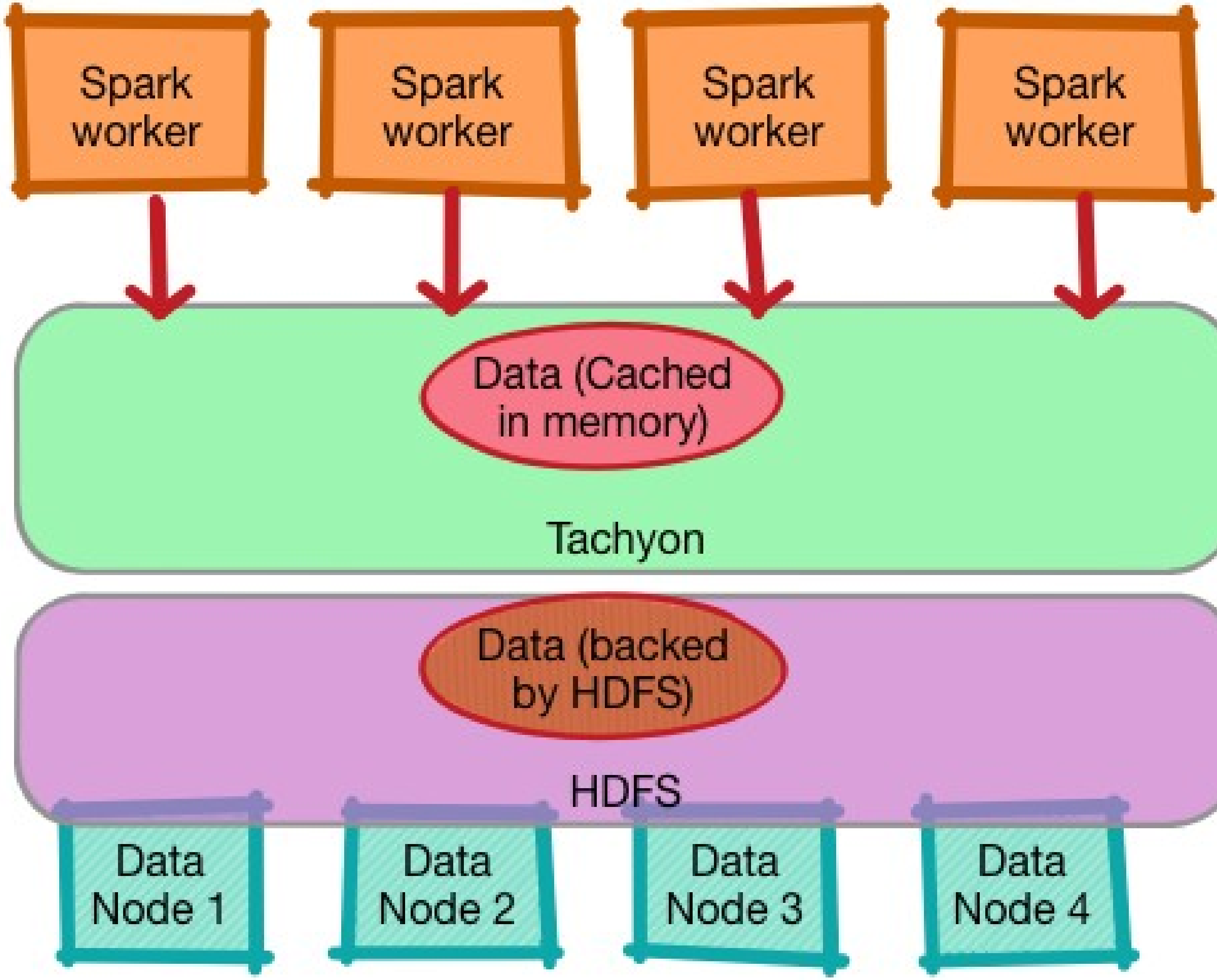


Source: The Apache Software Foundation

Tachyon File System (Alluxio)

- TachyonFS is a **distributed, in-memory file system**
- Data is backed by HDFS for safety
- (Diagram next slide)
- Doesn't use JVM for storage
  - No need to worry about Garbage Collection (GC)
  - Can accommodate very large data sets (PB)
- Features:
  - Automatic data promotion from HDFS --> memory (based on usage)
  - Configurable cache policies (Latest used/most used, etc.)
  - Pin data in memory (high usage data)

Tachyon File System (Alluxio)



Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Using RDD Persistence

- RDDs use the following methods for persistence:
  - **`persist(newLevel: StorageLevel)`**: Set this RDDs storage level to `newLevel` (only valid if storage level never set)
  - **`persist()`**: Same as `persist(StorageLevel.MEMORY_ONLY)`
  - **`cache()`**: Same as `persist(StorageLevel.MEMORY_ONLY)`
  - `MEMORY_ONLY` is the default for Cache. Use Persist for other options.
- Below, we give a usage example:
  - Note that the call to cache doesn't have any immediate affect.
  - It's added to the DAG, and when it's eventually created, Spark knows to cache the RDD.

```
val visits = sc.textFile("visits.txt").map(...)
val pageNames = sc.textFile("pages.txt").map(...)
val joined = visits.join(pageNames) // can be expensive!
joined.cache() // cached, so no need to re-compute
```

Lab: Caching

Lab

- **Overview**:
  In this lab, we will persist some of our RDDs
  - We'll examine how that affects the performance of a job

- **Builds on previous labs**:
  Lab for general setup

- **Approximate time**:
  15-20 minutes

- **Instructions**:
  3.6-caching

Guidelines for Using Caching

- **Use Datasets and DataFrames with Tungsten!**

- If you have to use RDD…
  - If your RDDs fit in memory, use the default (`MEMORY_ONLY`)
  - Most CPU efficient
  - If not, try using `MEMORY_ONLY_SER` and select a fast serialization library
  - Don't spill to disk unless RDD computation is expensive or filters a lot of data
  - Otherwise, recomputing may be as fast as reading from disk

Key-Value Pairs [For Reference. Not Covered in Class]

Spark's Pair RDDs

- Many operations work on key-value pairs
  - Generally doing aggregation (Grouping, counting, etc.)
  - Pair RDDs support this directly
- **Pair RDD**: RDD containing key/value pairs
  - Elements of form **`(key, value)`** - e.g. `(apple, 1)`
  - Support special operations (e.g. `groupByKey()`, `join()`)
  - Some operations operate on keys in parallel - fast
- Pair RDD Example: Similar to word count RDDs
  - **key**: The word, **value**: The count

```
{ (apple, 1), (pear, 1), (apple, 1), (grape, 1),
(pear, 1), (apple, 1) }
```

Operations for KV RDDs

- Key-Value RDDs provide few more operations
  - Usually centered around 'key'
- **PairRDDFunctions** provide specialized operations on key-value RDDs
  - countByKey()
  - groupByKey()
  - reduceByKey()
  - sortByKey()
- org.apache.spark.rdd.PairRDDFunctions<K,V>

countByKey() Overview

- **countByKey()** : Count all values per key
  - Return: A Map of key -> counts (no longer an RDD)
  - Frequency count
  - Below, we illustrate conceptually and in code

```
{ (apple, 1), (pear, 1), (apple, 1), (grape, 1),
(pear, 1), (apple, 1) }
==>
{(apple, 3), (pear, 2), (grape, 1)}
```

```
scala> val wordPairsRDD = sc.parallelize( Array(
    ("apple",1), ("pear",1), ("apple",1),
    ("grape",1), ("pear",1), ("apple",1)) )

scala> wordPairsRDD.countByKey()
Res6: Map[String,Long] = Map(apple -> 3, grape -> 1,
pear -> 2)
```

groupByKey() Overview

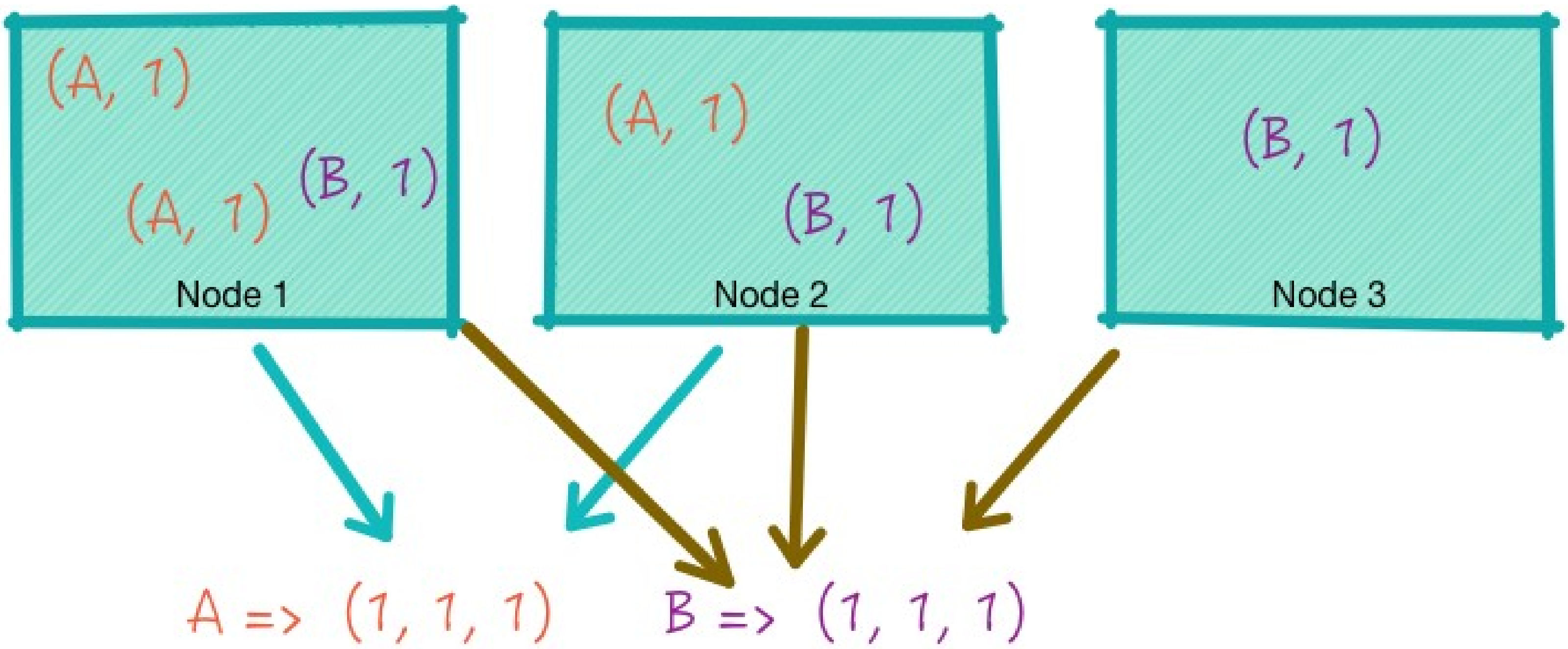
- **groupByKey()** : Group all values with the same key
  - Return: New RDD of pairs formed of (key, (all values))
  - There are performance implications (covered later)
  - Below, we illustrate conceptually and in code

```
{ (apple, 1), (pear, 1), (apple, 1), (grape, 1),
(pear, 1), (apple, 1) }
==>
{ (pear,(1, 1)), (apple,(1, 1, 1)), (grape,(1)) }
```

```
scala> val wordPairsRDD = sc.parallelize( Array(
    ("apple",1), ("pear",1), ("apple",1),
    ("grape",1), ("pear",1), ("apple",1)) )

scala> wordPairsRDD.groupByKey().collect()
res6: Array[(String, Iterable[Int])] =
Array((pear,CompactBuffer(1, 1)),
(apple,CompactBuffer(1, 1, 1)),
(grape,CompactBuffer(1)))
```

GroupByKey and Shuffle

- GroupBy usually involves a shuffle phase
- Data can be in multiple nodes, and they have to be brought together.
- During shuffle, nodes exchange data over the network
- This can be expensive!



Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

join() on Two Pair RDDs

- Spark supports joins on Pair RDDs
  - Via `join` (an inner join) `leftOuterJoin`, and `rightOuterJoin`
  - Many use cases that use joins
  - Joins happen on ‘keys’
- Assume you had the two RDDs below
  - We illustrate joining them at bottom
  - The code is simple - we'll see performance considerations later

RDD r1 = { (apple, 1), (pear, 1), (apple, 1), (grape, 1), (pear, 1), (apple, 1) }

RDD r2 = { (apple, ripe), (pear, unripe) }

```
scala> r1.join(r2).collect
res8: Array[(String, (Int, String))] = Array((pear,
(1,unripe)), (pear,(1,unripe)), (apple,(1,ripe)),
(apple,(1,ripe)), (apple,(1,ripe)))
```

PairRDD Example


```
val data = sc.textFile("data/people.csv")
data: org.apache.spark.rdd.RDD[String] = data/people.csv MapPartitionsRDD[141]..

data.foreach(println)
John,M,35
Jane,F,40
Mike,M,18
Sue,F,19

val people = data.map(line => {
                          val tokens = line.split(",") // split the line
                          val name = tokens(0)
                          val gender = tokens(1)
                          val age = tokens(2).toInt
                          (name, gender, age) // create a tuple
                       })
people: org.apache.spark.rdd.RDD[(String, String, Int)] = MapPartitionsRDD[142]..


people.foreach(println)
(John,M,35)
(Mike,M,18)
(Jane,F,40)
(Sue,F,19)

val males = people.filter {case (name, gender, age) => gender == "M"}
males: org.apache.spark.rdd.RDD[(String, String, Int)] = MapPartitionsRDD[143]..

males.foreach(println)
(John,M,35)
(Mike,M,18)
```

```
John,M,35
Jane,F,40
Mike,M,20
Sue,F,19
```

Code Walkthrough (1 of 3)

- `data` is simple RDD of String type
- Each line is an element

```
val data = sc.textFile("data/people.csv")
data: org.apache.spark.rdd.RDD[String] =
data/people.csv MapPartitionsRDD[141]..

data.foreach(println)
John,M,35
Jane,F,40
Mike,M,18
Sue,F,19
```

Code Walkthrough (2 of 3)

- We are converting a flat RDD (data) into an RDD of Tuples (people)
- Use simple text parsing to create a Tuple
- data: RDD[String] → people: RDD[(String, String, Int)]

```
val people = data.map(line => {
                          val tokens = line.split(",") // split the line
                          val name = tokens(0)
                          val gender = tokens(1)
                          val age = tokens(2).toInt
                          (name, gender, age) // create a tuple
                       })
people: org.apache.spark.rdd.RDD[(String, String, Int)] =
MapPartitionsRDD[142]..


people.foreach(println)
(John,M,35)
(Mike,M,18)
(Jane,F,40)
(Sue,F,19)
```

Code Walkthrough (3 of 3)

- Using Scala case comparison operator (match expression) to process Tuples
- **`{case (x,y,z) => (x == 1) && (y > 3) }`**

```
val males = people.filter {case (name, gender, age) =>
gender == "M"}
males: org.apache.spark.rdd.RDD[(String, String, Int)] =
MapPartitionsRDD[143]..

males.foreach(println)
 (John,M,35)
 (Mike,M,18)
```

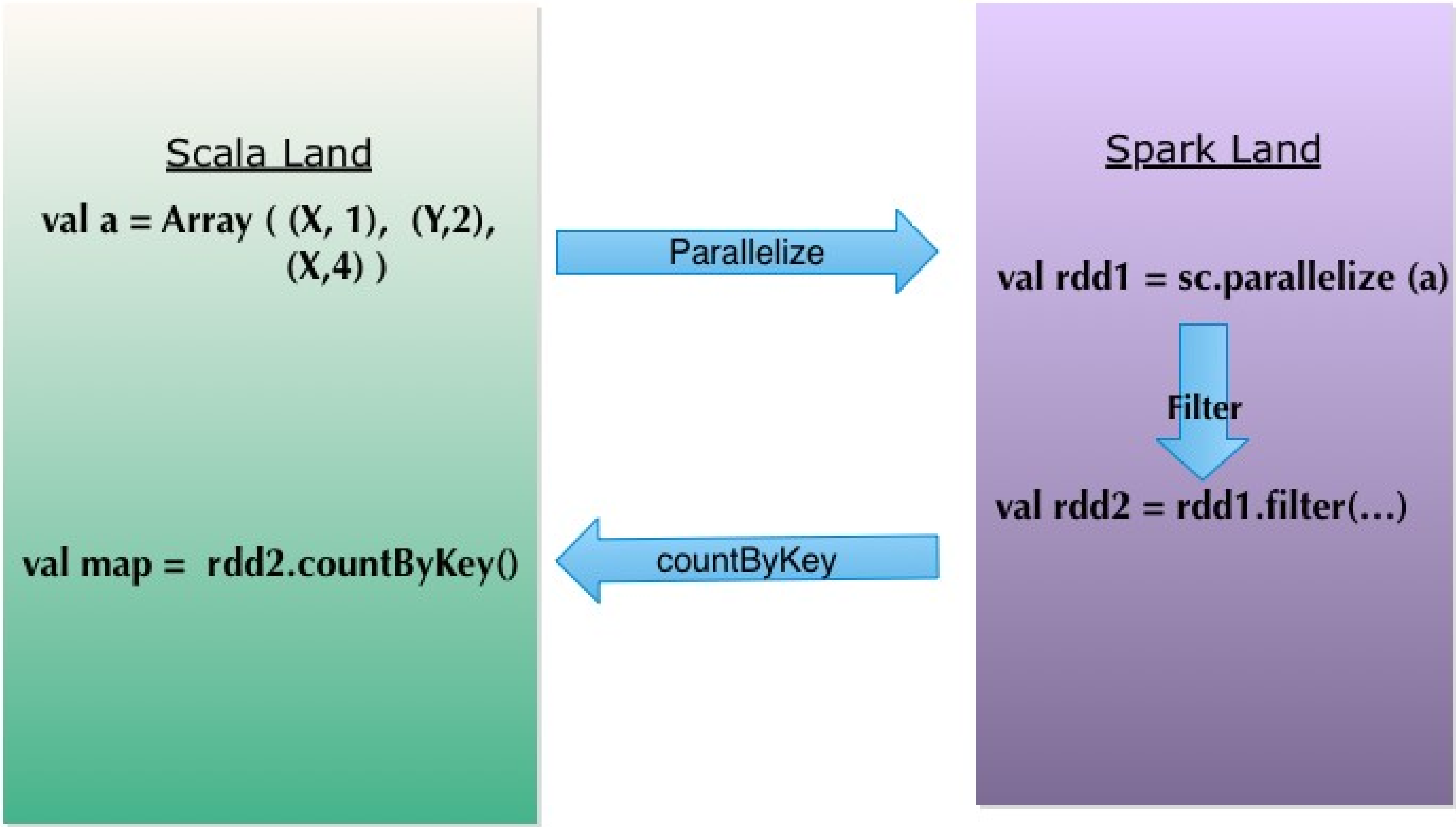
Spark ← → Scala

- Transformation operations will give another RDD
  - Filter, map, groupByKey
- Actions can give you Scala objects
  - Count, countyByKey
- If not sure of what you got, just type the variable in Scala shell to see the type

See diagram in next slide

```
scala> val wordPairsRDD = sc.parallelize( Array(
    ("apple",1), ("pear",1), ("apple",1),
    ("grape",1), ("pear",1), ("apple",1)) )
// result is an RDD

scala> wordPairsRDD.countByKey()
Res6: Map[String,Long] = Map(apple -> 3, grape -> 1, pear ->
2)
// result is a Scala map
```

Spark <-> Scala



Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
2019-03-12

Lab 3.3: Key/Value Pair RDDs

Lab

- **Overview**: In this lab, we will create and work with Pair RDDs
  - We'll work with various transformations on the RDDs
- **Builds on previous labs**: Lab 2.1 for general setup
- **Approximate Time**: 20-30 minutes
- **Follow: 3-rdd / 3.3-rdd-kv.md**

reduceByKey() Overview

- **reduceByKey(f(V,V)=>V)** : Combine values with same key using f
  - Return: New RDD of pairs formed of (key, (combined value))

- Below, we illustrate conceptually and in code
  - The anonymous function **(a,b) => a+b** adds the values together
  - Alternate syntax for this is **reduceByKey(_+_)**

```
{ (apple, 1), (pear, 1), (apple, 1), (grape, 1), (pear, 1),
(apple, 1) }
==>
{ (pear,2), (apple,3), (grape,1) }
```

```
// wordPairsRDD as per previous example
scala> wordPairsRDD.reduceByKey((a,b) => a+b).collect()
res7: Array[(String, Int)] = Array((pear,2), (apple,3),
(grape,1))
```

Word Count in Spark

- Simple word count is the prototypical Hadoop application
  - But it's non-trivial in Hadoop
- Let's look at it in Spark/Scala below
  - **`line`**: String representing some text (could read a file also)
  - **`line.split`**`("\\s+"))`: Split the input into words (1)
  - **`map(word => (word,1))`**: Map into Pair RDD of form (word,1)
    - Produces `{ (apple, 1), (pear, 1), (apple, 1), (grape, 1), (pear, 1), (apple, 1) }`
  - **`reduceByKey(_ + _)`**: Add up the counts for each key (word)
    - Produces the counts

    `{ (pear,2), (apple,3), (grape,1) }`

```
val line = "apple pear apple grape pear apple"
val wordPairsRDD = sc.parallelize(line.split("\\
s+")).map(word => (word,1))
val countsRDD = wordPairsRDD.reduceByKey(_ + _)
```

Alternate Word Count in Spark

- This version does it somewhat differently
  - We start with the Pair RDD of form (word,1)
  - **groupByKey()**: Group the elements by key - Entries in the result are of form `(word, Iterable[count])`
    - Produces `{ (pear,(1, 1)), (apple,(1, 1, 1)), (grape,(1)) }`
  - **map(t => (t._1, t._2.sum)**: Add up the counts for each key (word) - produces same result as previously

    `{ (pear,2), (apple,3), (grape,1) }`

```
val countsRDD = wordPairsRDD.
                             .groupByKey()
                             .map(t => (t._1,
t._2.sum))
```

Pair Transformation Overview

- Below, we list some of the transformations on Pair RDDs
  - Note for coding: These are defined in `PairRDDFunctions` (1)

RDD r = { (apple, 1), (pear, 1), (apple, 1), (grape, 1), (pear, 1), (apple, 1) }

Transformation	Description	Example	Result
groupByKey()	Group values with same key	r.groupByKey()	{ (pear,(1, 1)), (apple, (1, 1, 1)), (grape,(1)) }
reduceByKey(func)	Combine all values with the same key using func as combiner	r.reduceByKey((a,b) = > a + b) r.reduceByKey(_+_)	{ (pear,2), (apple,3), (grape,1) }
sortByKey()	Return RDD sorted by key	r.sortByKey()	{ (apple,1), (apple,1), (apple,1), (grape,1), (pear,1), (pear,1) }
keys	Return RDD of keys	r.keys	{apple, pear, apple, grape, pear, apple}
values	Return RDD of values	r.values	{1, 1, 1, 1, 1, 1}
mapValues(func)	Apply func to each value	r.mapValues(a => a + 1)	{ (apple, 2), (pear, 2), (apple, 2), (grape, 2), (pear, 2), (apple, 2) }

Optional/Backup Slides

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

[Bonus] Lab: MapReduce

Lab

- **Overview**:
  In this lab, we will code the Word Count example using MapReduce in Spark

- **Builds on previous labs**:
  Lab for general setup

- **Approximate time**:
  15-20 minutes

- **Follow**:
  3-rdd/3.4-mapreduce.md

[Bonus] Lab: Clickstream Analysis

Lab

- **Overview**:
  In this lab, we will use MapReduce to analyze clickstream data

- **Builds on previous labs**:
  Lab for general setup

- **Approximate time**:
  30-40 minutes

- **Follow**:
  3-rdd/3.5-clickstream.md

RDD Attributes

- Partitions
  - Distributed across cluster
- Dependencies
  - Parent RDD
  - Used to recompute
- F(x): Compute function
  - Function to compute this RDD from parents
- Preferred Locations [Optional]
  - For storages that support location hints
  - HDFS/Cassandra
- Example: **HadoopRDD**
  - Partitions: Corresponds to HDFS block
  - Dependencies: None
  - Compute Function: Read HDFS block

Types of RDD

RDD	Description	Dependencies	Partitions
HadoopRDD	HDFS files	None	HDFS blocks (1-1 mapping)
FilteredRDD	Created by running filter on parents	Parents	Parent's partition (1-1 mapping)
MappedRDD	Result of map function	Parents	Parent's partition (1-1 mapping)
PairRDD	(K,V) pair		
JoinedRDD	Joining 2 RDDs	Shuffle each parent	One per reduce task
ShuffledRDD		Shuffle sources	
UnionRDD		Parents unioned	

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @
 2019-03-12

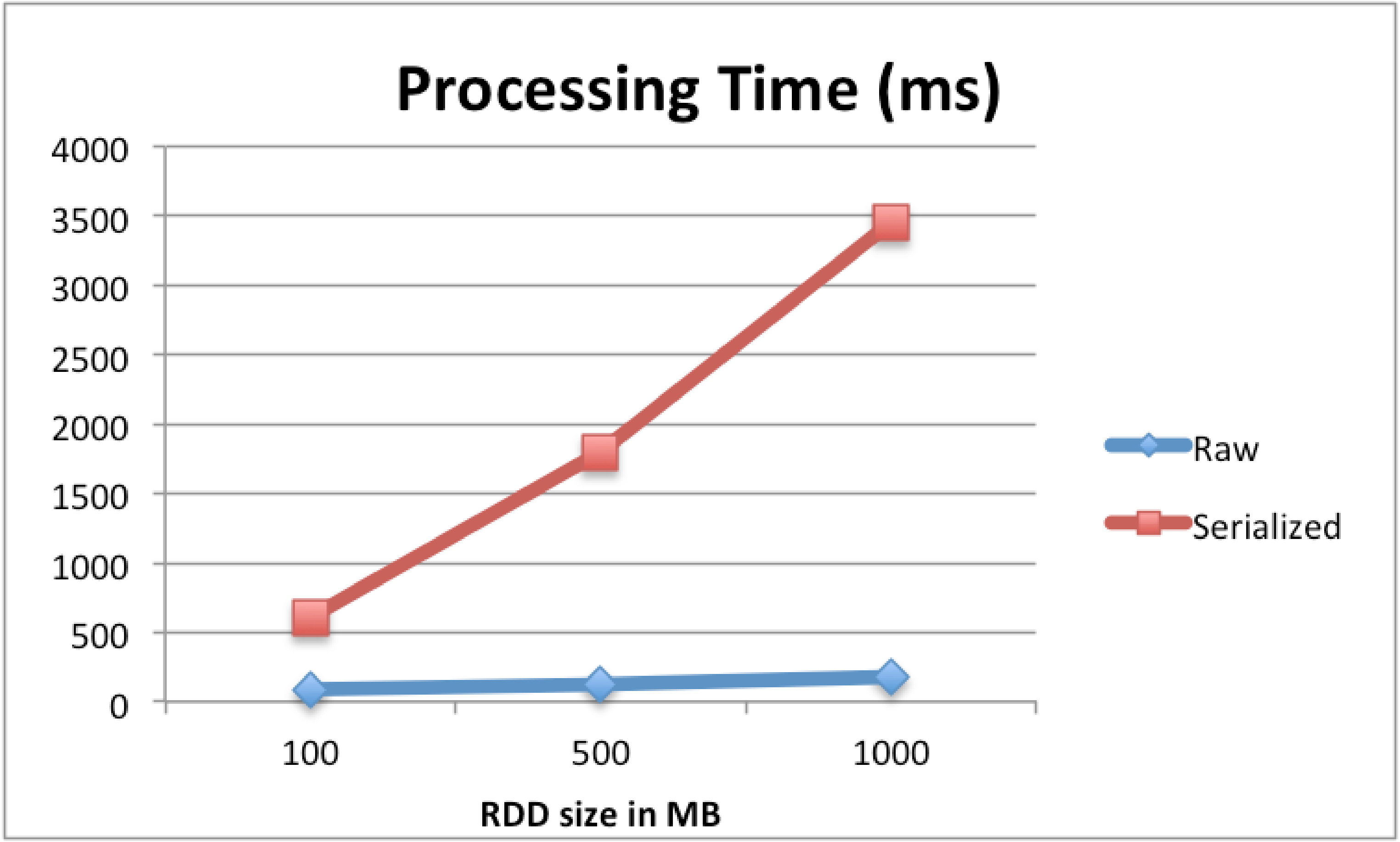
Lab: Operations on Multiple RDDs

Lab

- **Overview**:
  In this lab, we will work with some of the operations that use multiple RDDs
- **Builds on previous labs**:
  Lab for general setup
- **Approximate time**:
  5 minutes
- **Follow**:
  3-rdd/3.2-rdd-multi.md

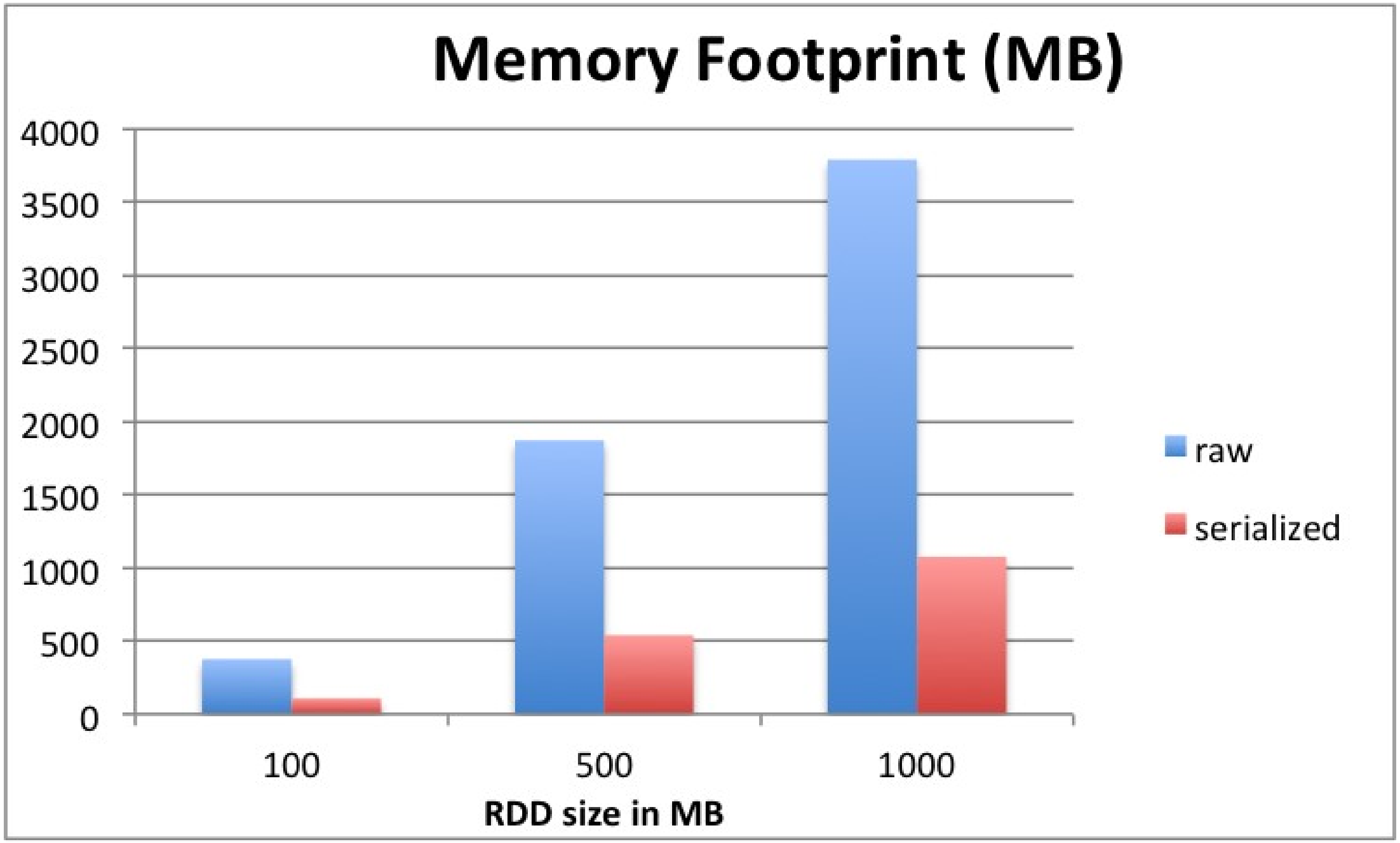
Understanding Memory Caching Implications

- Raw caching consumes more memory (2-5x)
- But is faster to process



Understanding Memory Caching Implications

- Serialized caching uses less memory
- Processing time is more



Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @

Guidelines to Using Persistence

- If your RDDs fit in memory, use the default (MEMORY_ONLY)
  - Most CPU efficient
- If not, try using MEMORY_ONLY_SER and select a fast serialization library
- Don't spill to disk unless RDD computation is expensive or filters a lot of data.
  - Otherwise, recomputing may be as fast as reading from disk

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Guidelines to Using Persistence

- Use replicated storage for fast fault recovery.
  - You have fault recovery anyway, but replicated has less down time
- For environments with high memory or multiple apps, OFF_HEAP has some advantages.
  - Lets multiple executors share memory pool in Tachyon
  - Reduces garbage collection
  - Cached data are not lost when an executor crashes

Review Questions

1. What is an RDD, DataFrame, DataSet?
2. Of the three above, which is the preferred interface?
3. What is a DAG in Spark execution?
4. What are Key/Value pairs?
5. When is caching used in Spark?

Anatomy of Spark Job

- Application can have many **actions → jobs**
- A **Job** may be executed in one or many stages (depending on the complexity)
- A **Stage** may have one or more **tasks**