DEEP DIVE INTO RDDs

Life Cycle of a Spark Program

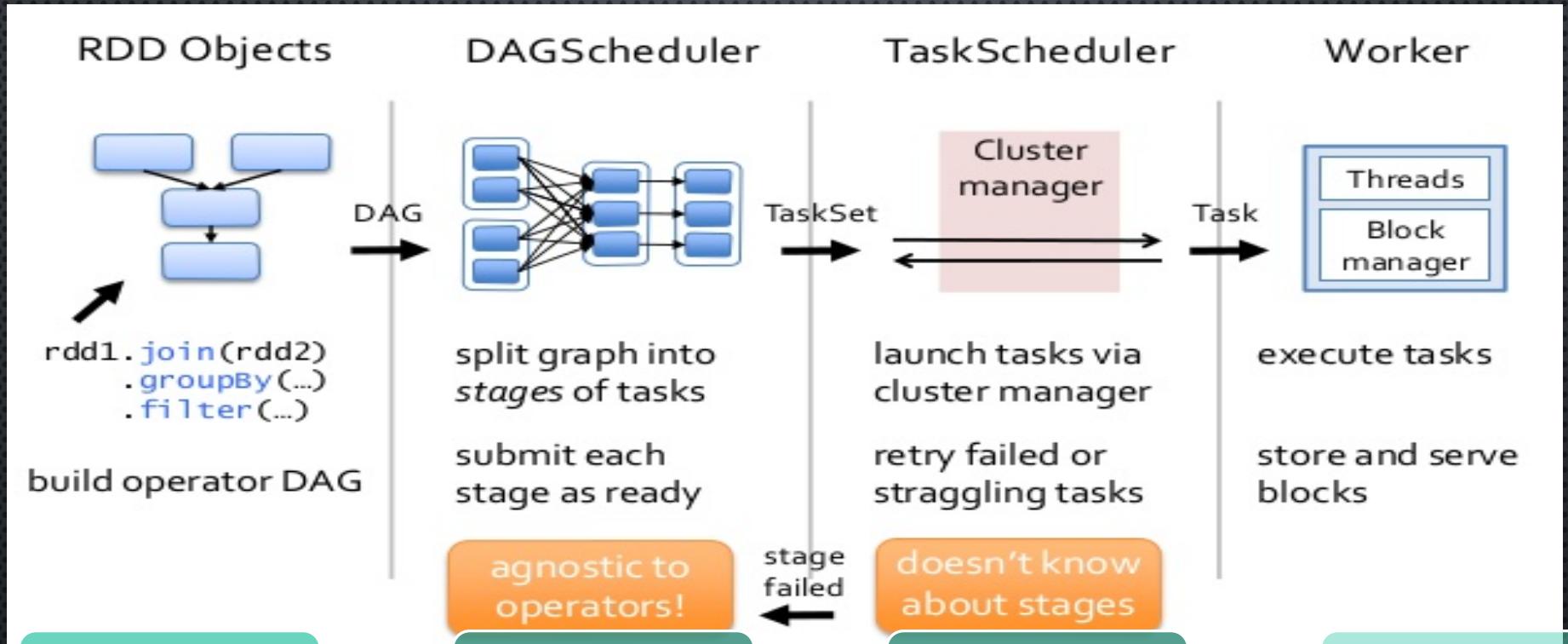
1) Create some input RDDs from external data or parallelize a collection in your driver program.

2) Lazily transform them to define new RDDs using transformations like `filter()` or `map()`

3) Ask Spark to `cache()` any intermediate RDDs that will need to be reused.

4) Launch actions such as `count()` and `collect()` to kick off a parallel computation, which is then optimized and executed by Spark.

Spark execution model



Create DAG of RDDs → Create logical execution plan → Schedule tasks → Execute tasks

Spark execution model

1. The user submits an application using spark-submit.
2. Spark-submit launches the driver program and invokes the main method specified by the user.
3. The driver program contacts the cluster manager to ask for resources to launch executors.
4. The cluster manager launches executors on behalf of the driver program.
5. The driver process runs through the user application. Based on the RDD actions and transformations in the program, the driver sends work to executors in the form of tasks.
6. Tasks are run on executor processes to compute and save results.
7. If the driver's main method exits or it calls SparkContext.stop(), it will terminate the executors and release resources from the cluster manager.

Spark execution model - Terminology

Term	Meaning
Application	User program built on Spark. Consists of a driver program and executors on the cluster.
Driver program	The process running the main() function of the application and creating the SparkContext
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
Worker node	Any node that can run application code in the cluster
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Task	A unit of work that will be sent to one executor
Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save,collect). within each Spark application, multiple “jobs” (Spark actions) may be running concurrently.
Stage	Each job gets divided into smaller sets of tasks called stages that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

=

Job #1

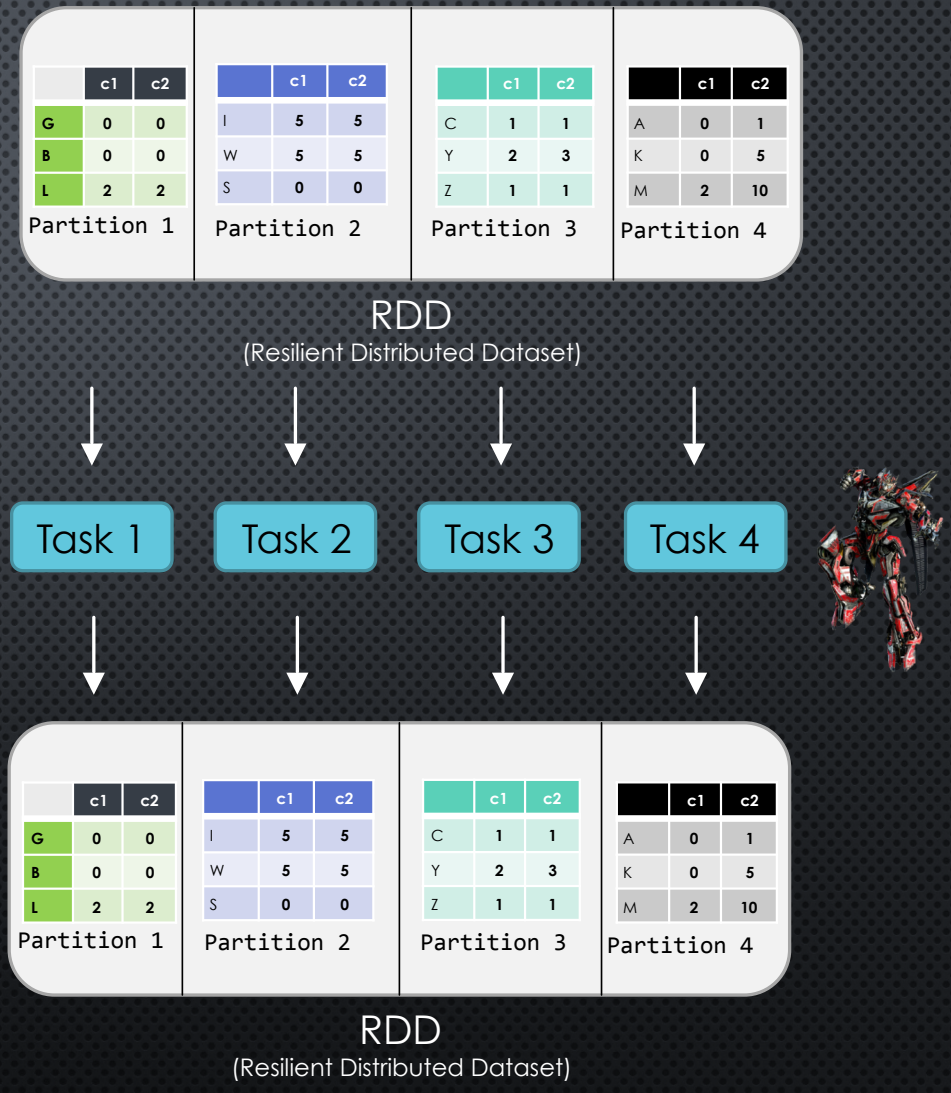
A physical stage will launch tasks that each do the same thing, but on different partitions of data.

Each task internally performs the same steps:

1) Fetch its input: either from data storage (for input RDDs) or an existing RDD, or shuffle outputs

2) Perform the operation or transformation to compute the RDD which it represents

3) Writing output to a shuffle, to external storage or back to the driver

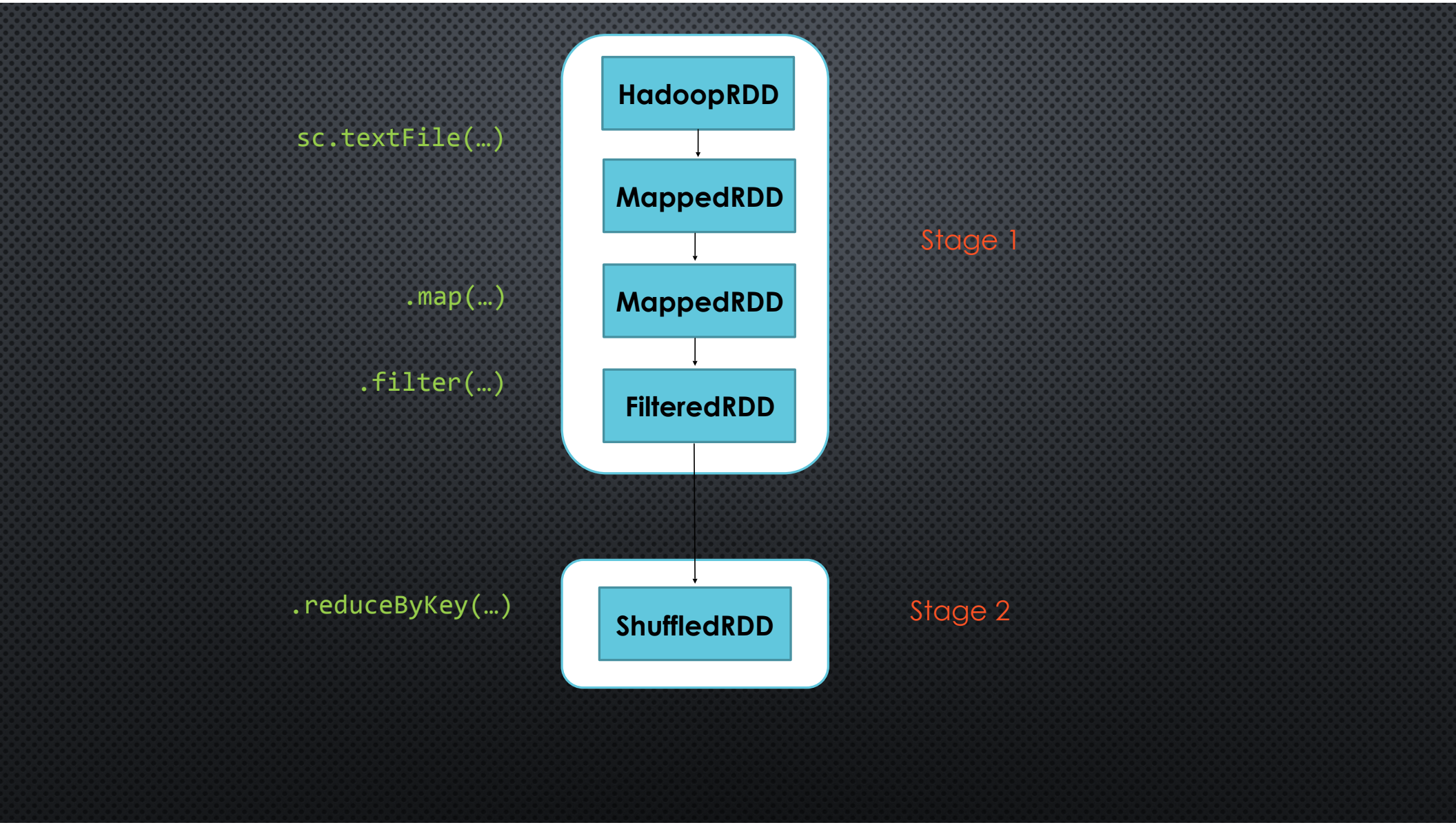


Spark execution summary:

1) User code defines a DAG (Direct Acyclic Graph) of RDDs
2) Actions force translation of the DAG to an execution plan.
3) Tasks are scheduled and executed on a cluster.

PIPELINING

- In some cases, the physical set of stages will not be an exact 1:1 correspondence to the logical RDD graph
- This can happen when pipelining (collapsing of multiple RDDs into a single stage)
- Pipelining occurs when RDDs can be computed from its parents without data movement.
- For example: when a user calls both map and filter sequentially, those can be collapsed into a single transformation which first maps, then filters each element


sc.textFile(…)
.map(…)
.filter(…)
.reduceByKey(…)
HadoopRDD
MappedRDD
MappedRDD
FilteredRDD
ShuffledRDD
Stage 1
Stage 2

Simple use case to understand spark execution model and RDDs

Example

Goal: Find number of distinct names per “first letter”

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

Example

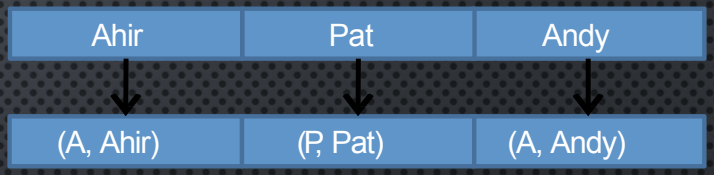
Goal: Find number of distinct names per “first letter”

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

Example

Goal: Find number of distinct names per “first letter”

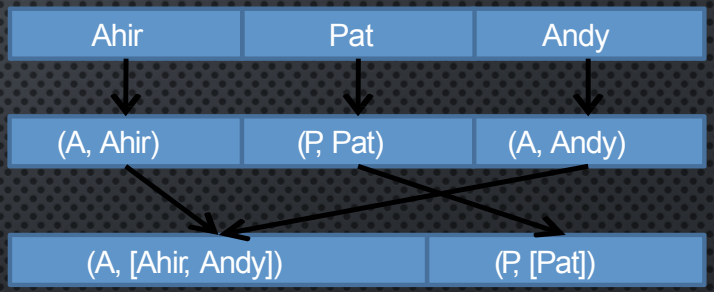
```
sc.textFile(“hdfs:/names”)
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues(names => names.toSet.size)
  .collect()
```



Example

Goal: Find number of distinct names per “first letter”

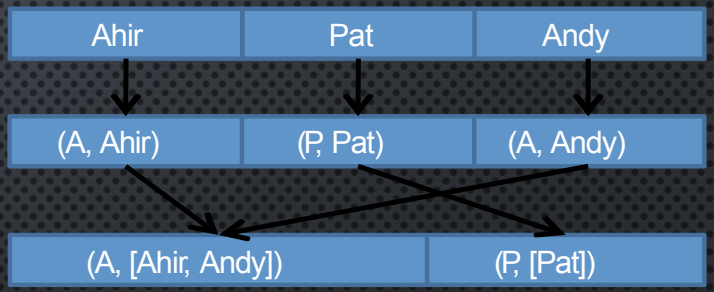
```
sc.textFile(“hdfs:/names”)
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues(names => names.toSet.size)
  .collect()
```



Example

Goal: Find number of distinct names per “first letter”

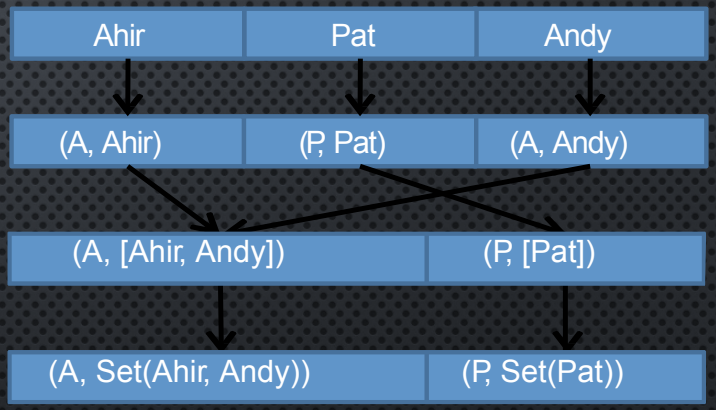
```
sc.textFile(“hdfs:/names”)
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues(names => names.toSet.size)
  .collect()
```



Example

Goal: Find number of distinct names per “first letter”

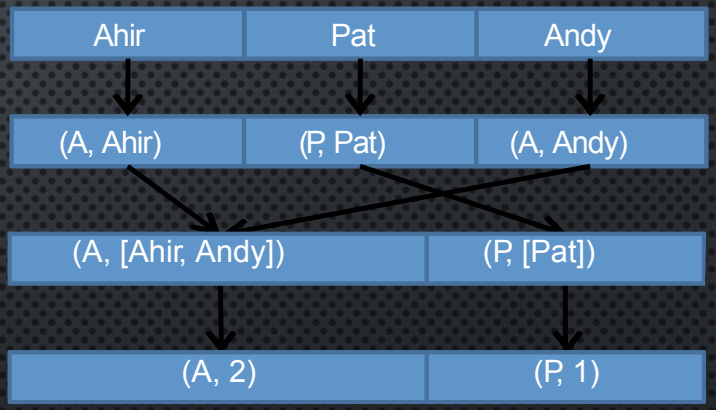
```
sc.textFile(“hdfs:/names”)
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues(names => names.toSet.size)
  .collect()
```



Example

Goal: Find number of distinct names per “first letter”

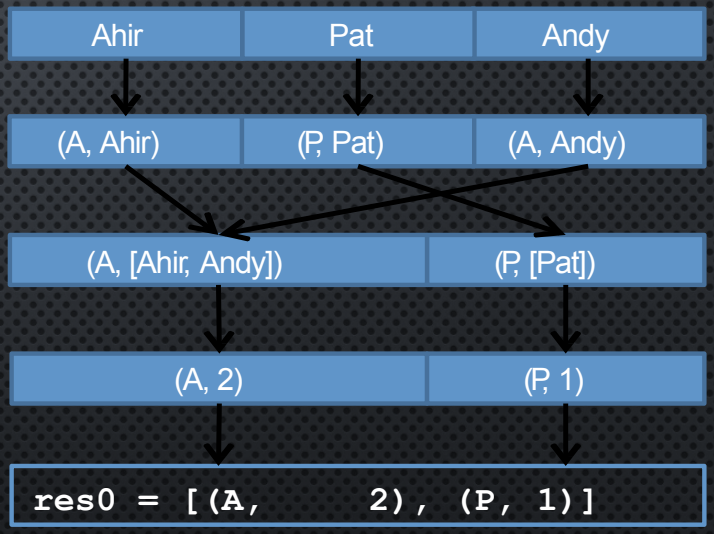
```
sc.textFile(“hdfs:/names”)
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues(names => names.toSet.size)
  .collect()
```



Example

Goal: Find number of distinct names per “first letter”

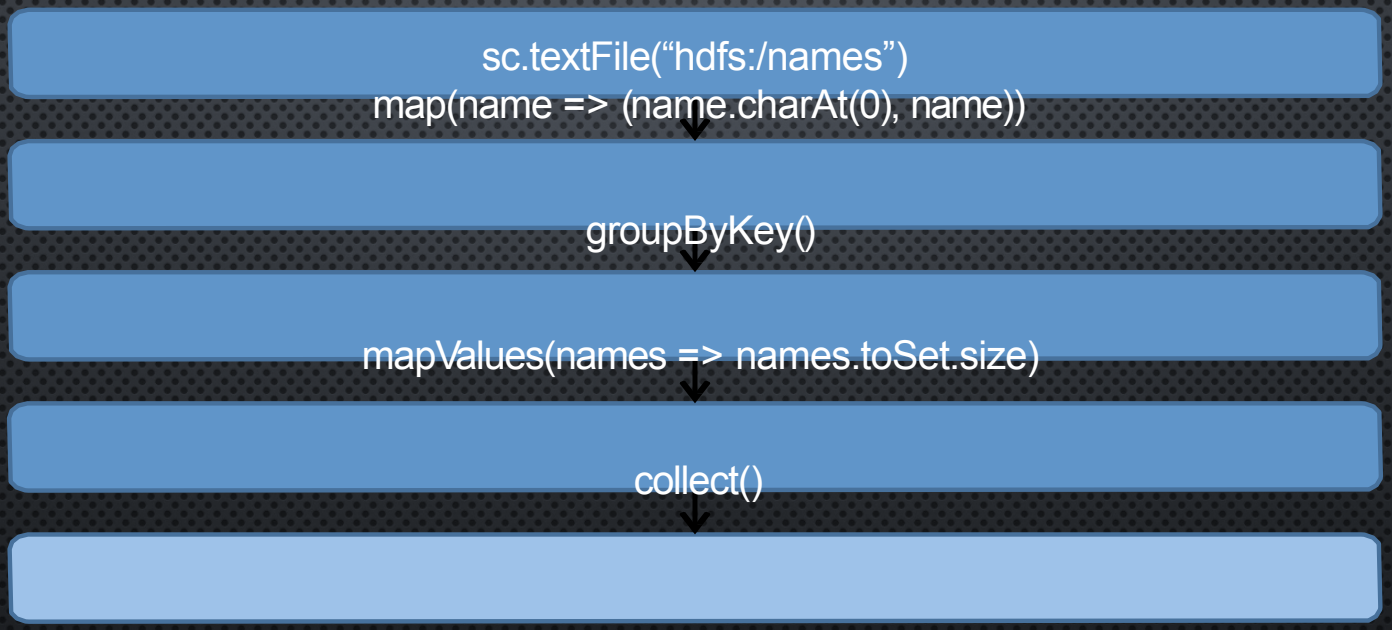
```
sc.textFile(“hdfs:/names”)

  .map(name => (name.charAt(0),          name))

  .groupByKey()

  .mapValues(names => names.toSet.size)

  .collect()
```



Spark Execution model for this example

1. Create DAG of RDDs to represent computation
2. Create logical execution plan for DAG
3. Schedule and execute individual tasks

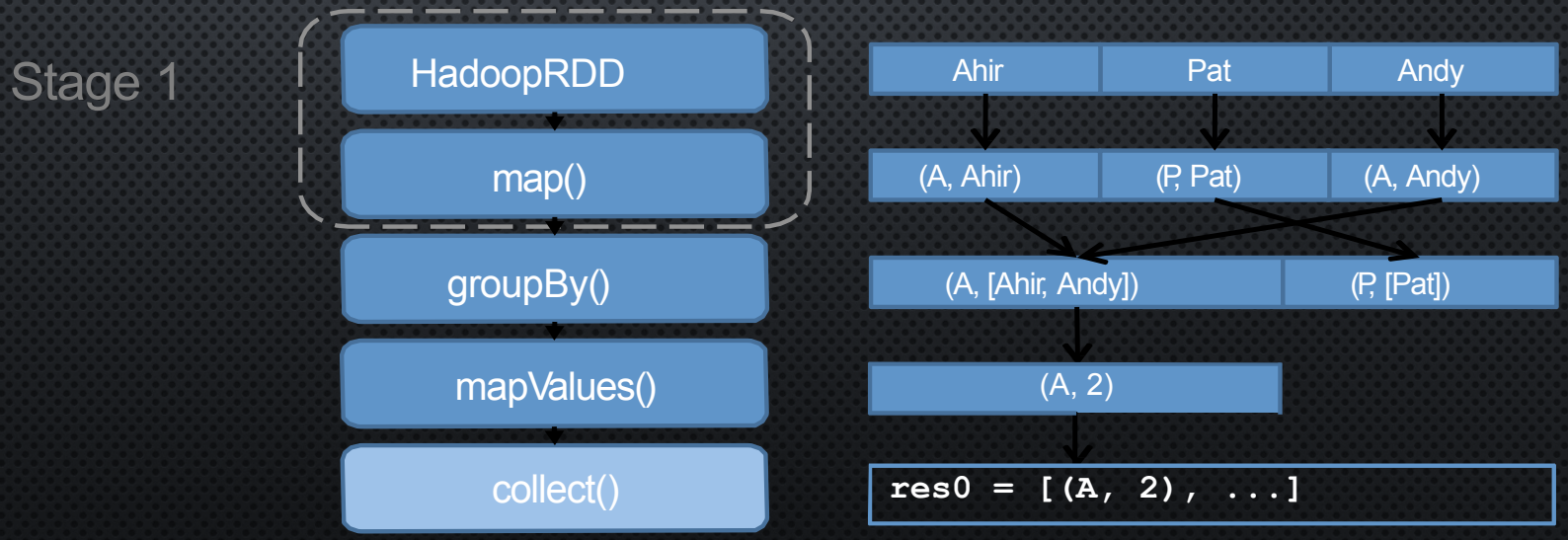
Step1: Create RDDs



Step1: Create RDDs

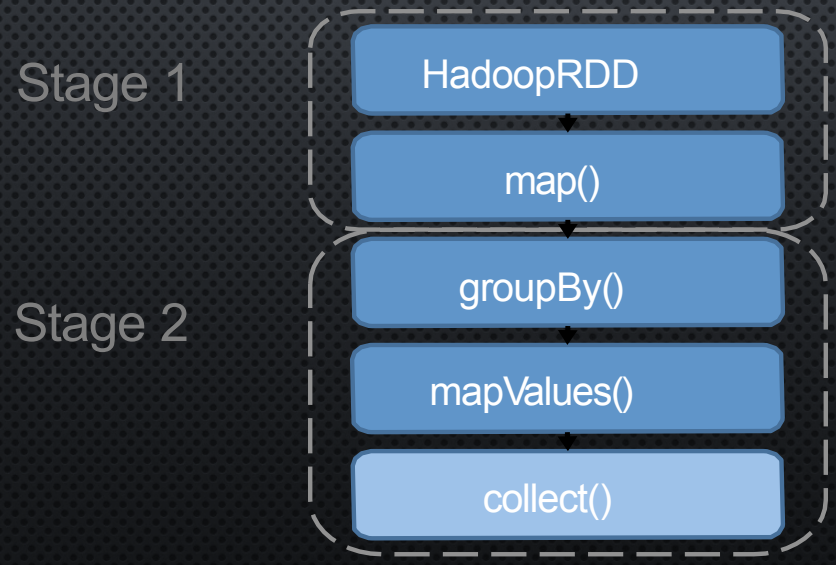
Step2: Create execution plan

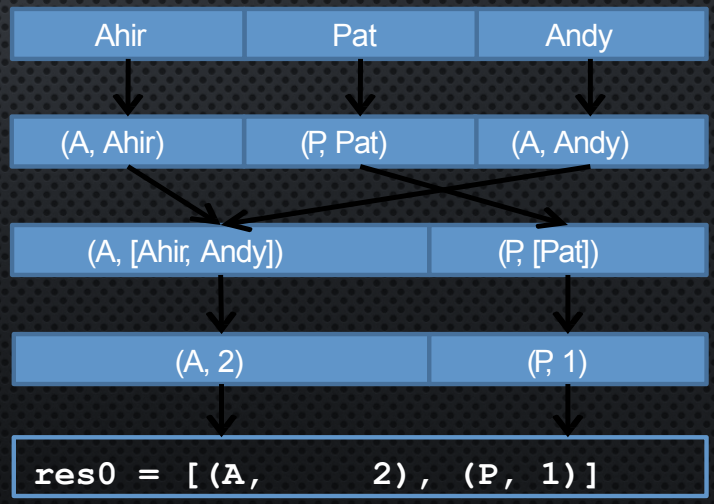
- Pipeline as much as possible
- Split into "stages" based on need to reorganize data



Step2: Create execution plan

- Pipeline as much as possible
- Split into "stages" based on need to reorganize data

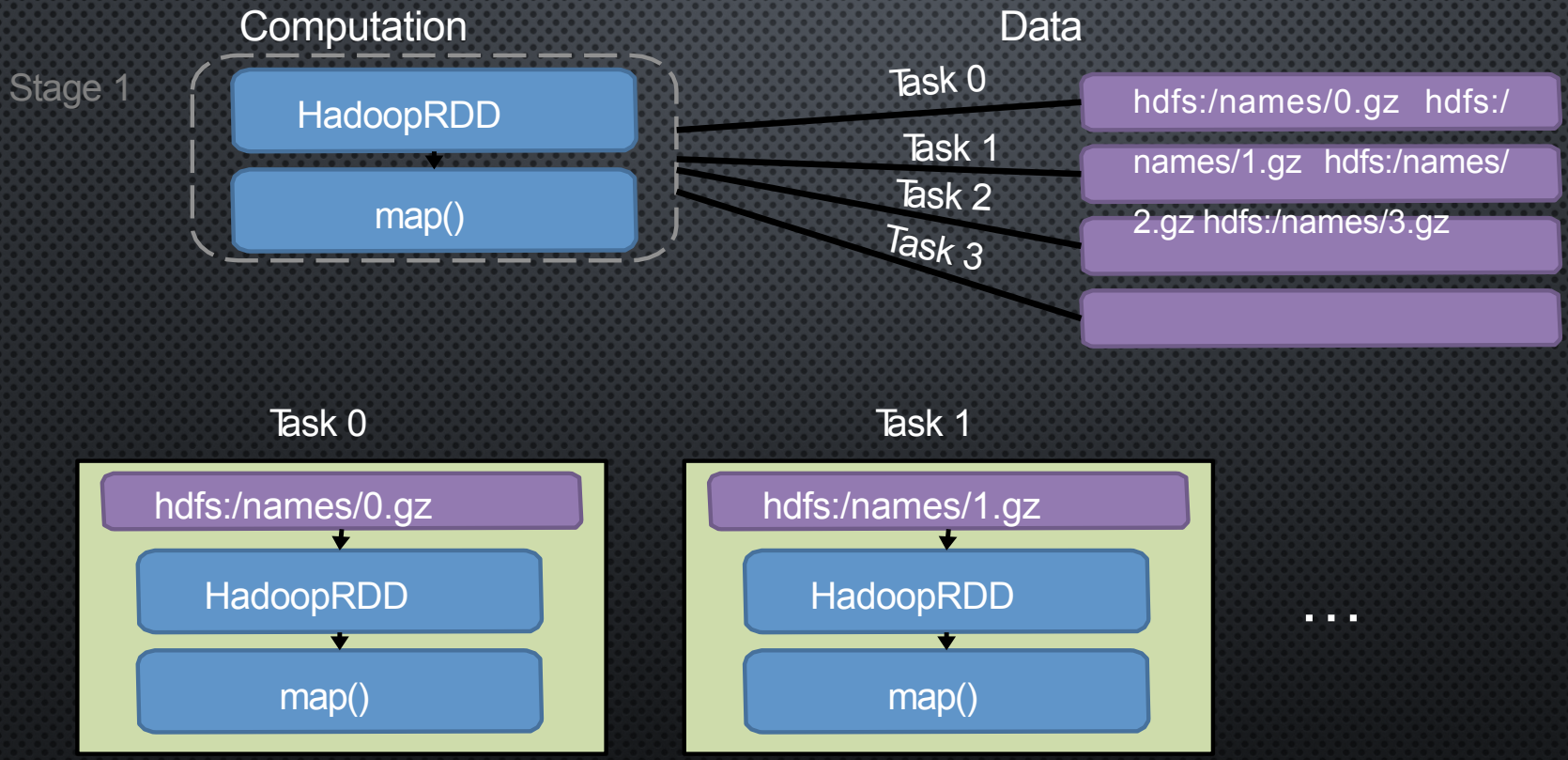




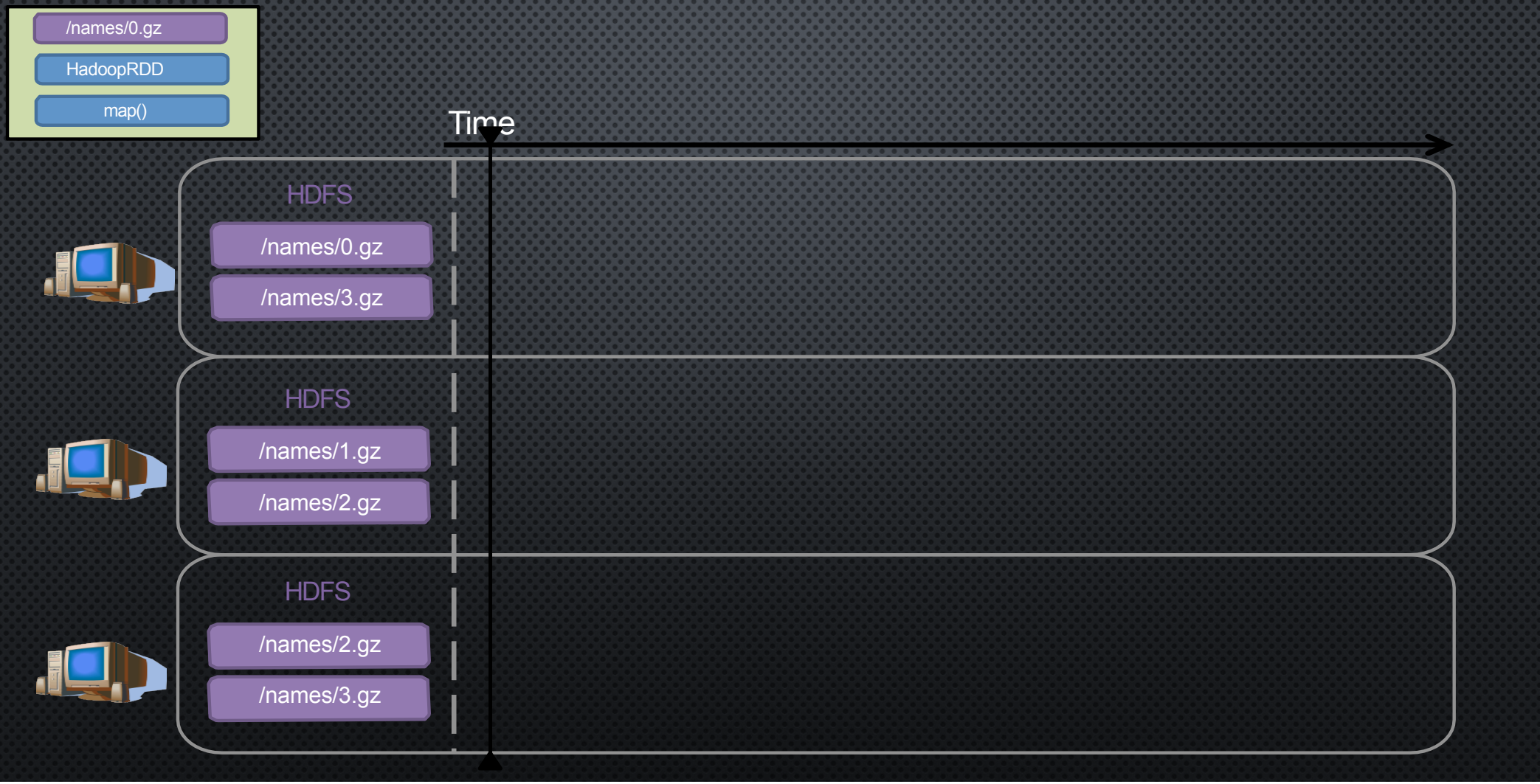
Step3: Schedule tasks

- Split each stage into tasks
- A task is data + computation
- Execute all tasks within a stage before moving on

Step3: Schedule tasks



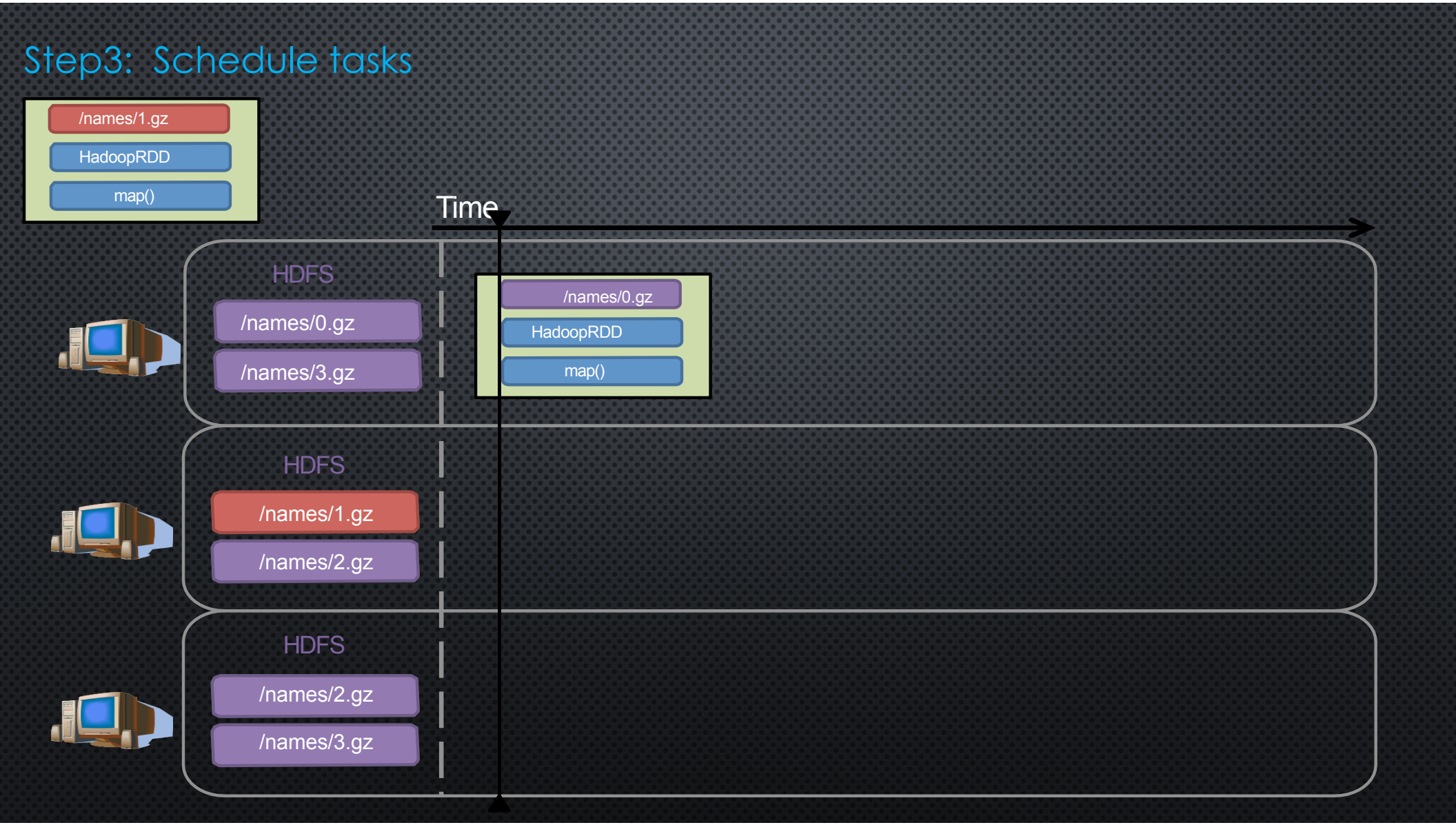
Step3: Schedule tasks



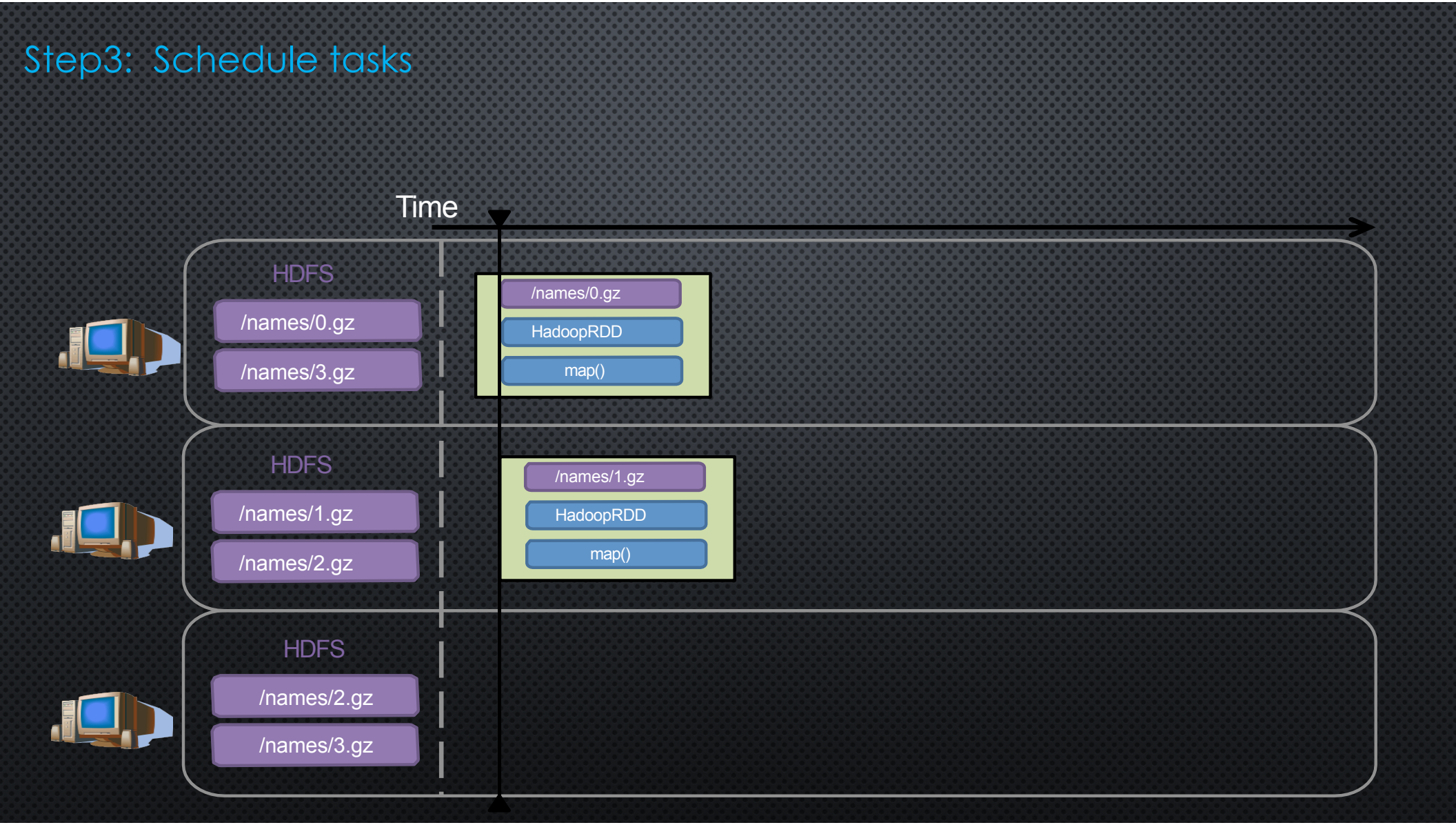
Step3: Schedule tasks

Step3: Schedule tasks

Step3: Schedule tasks

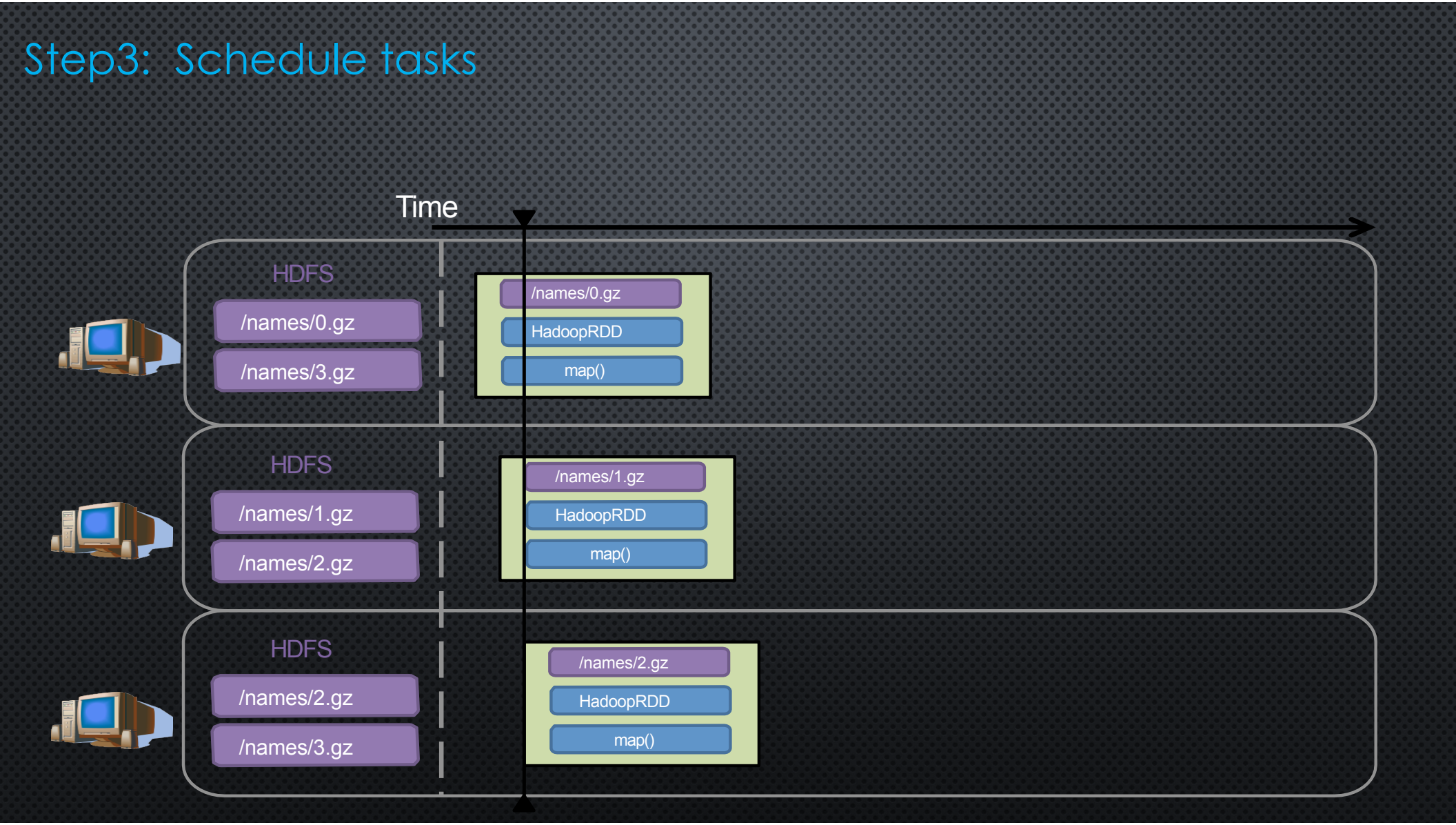


Step3: Schedule tasks

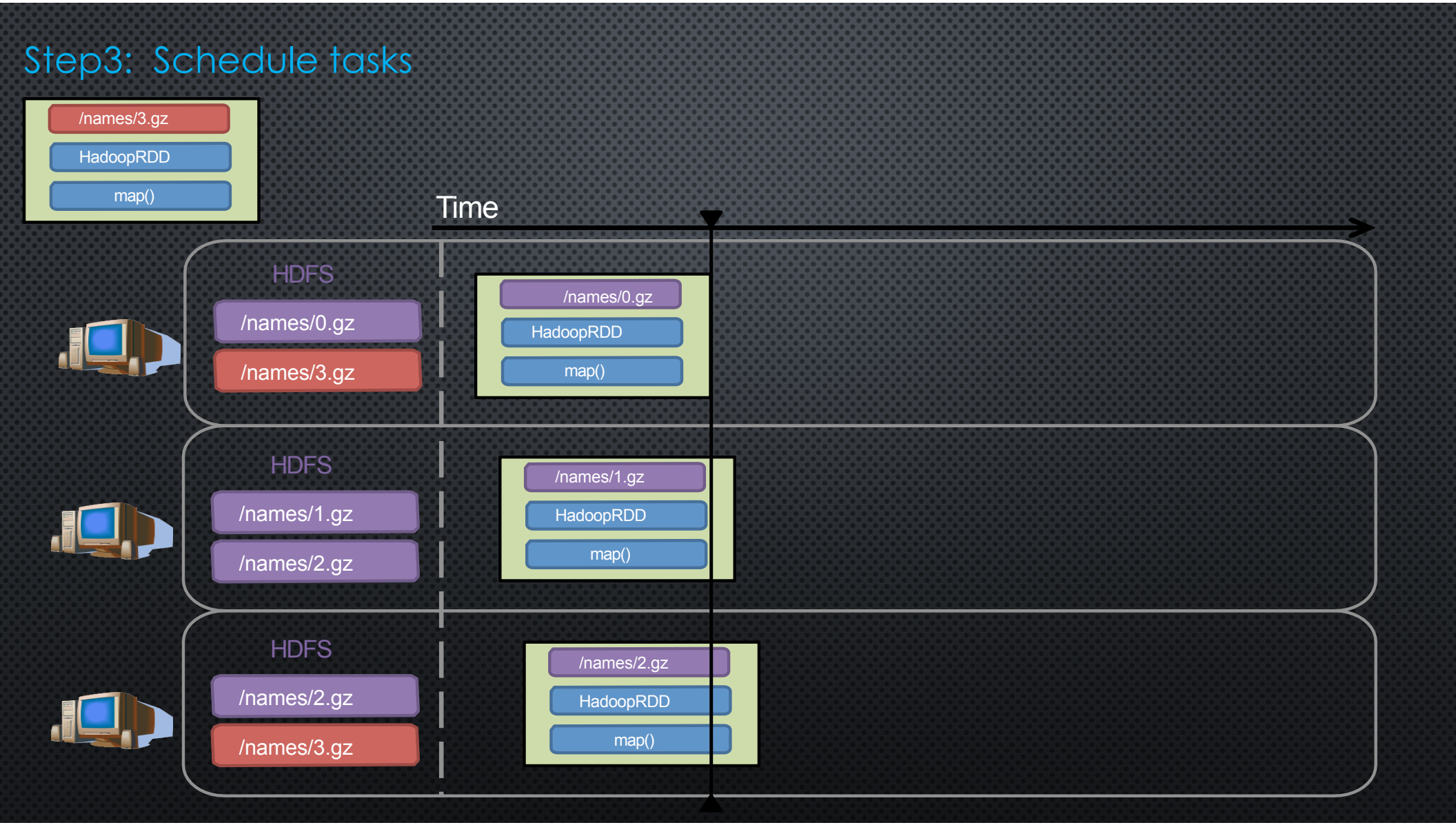


Step3: Schedule tasks

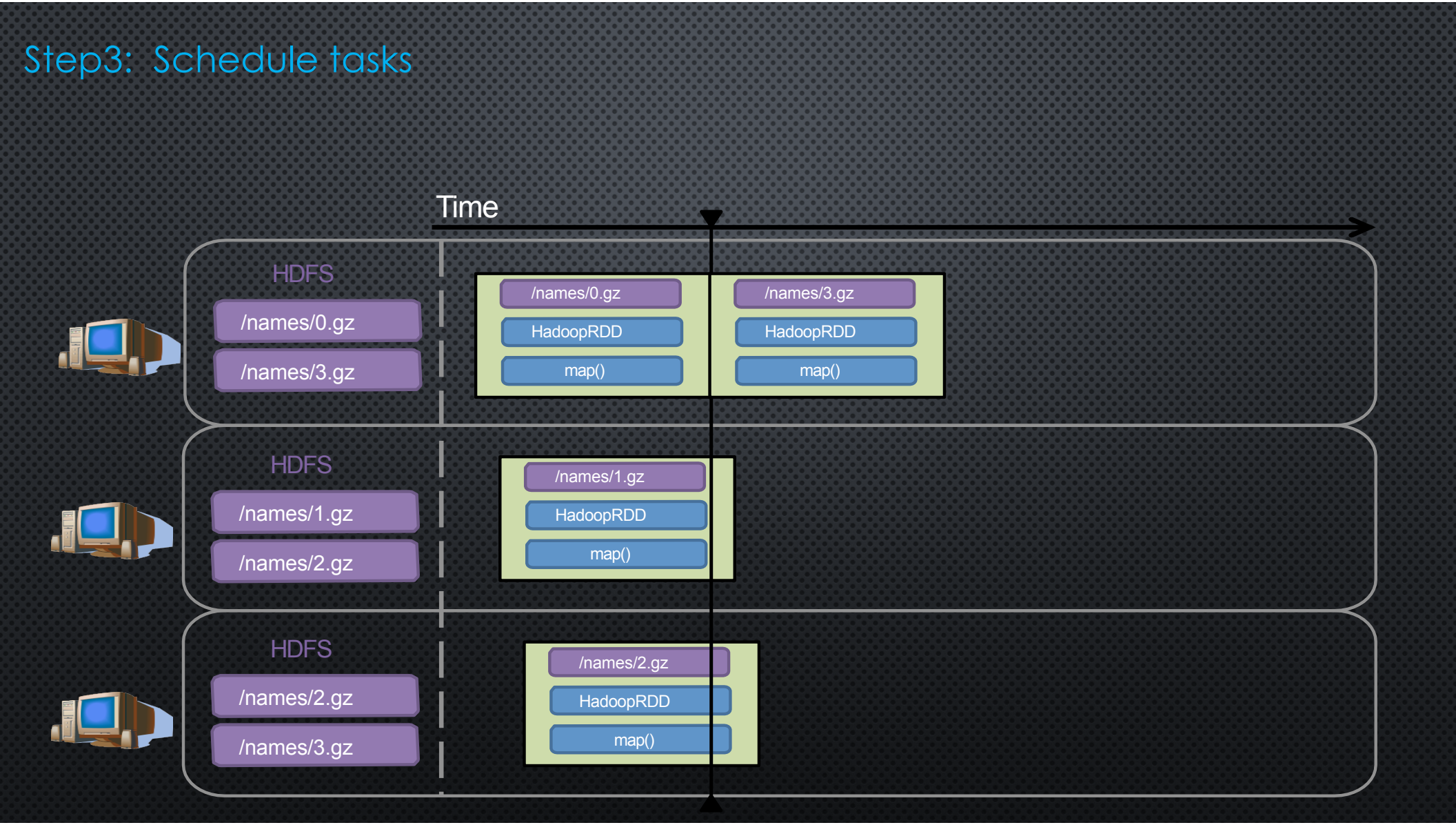
Step3: Schedule tasks



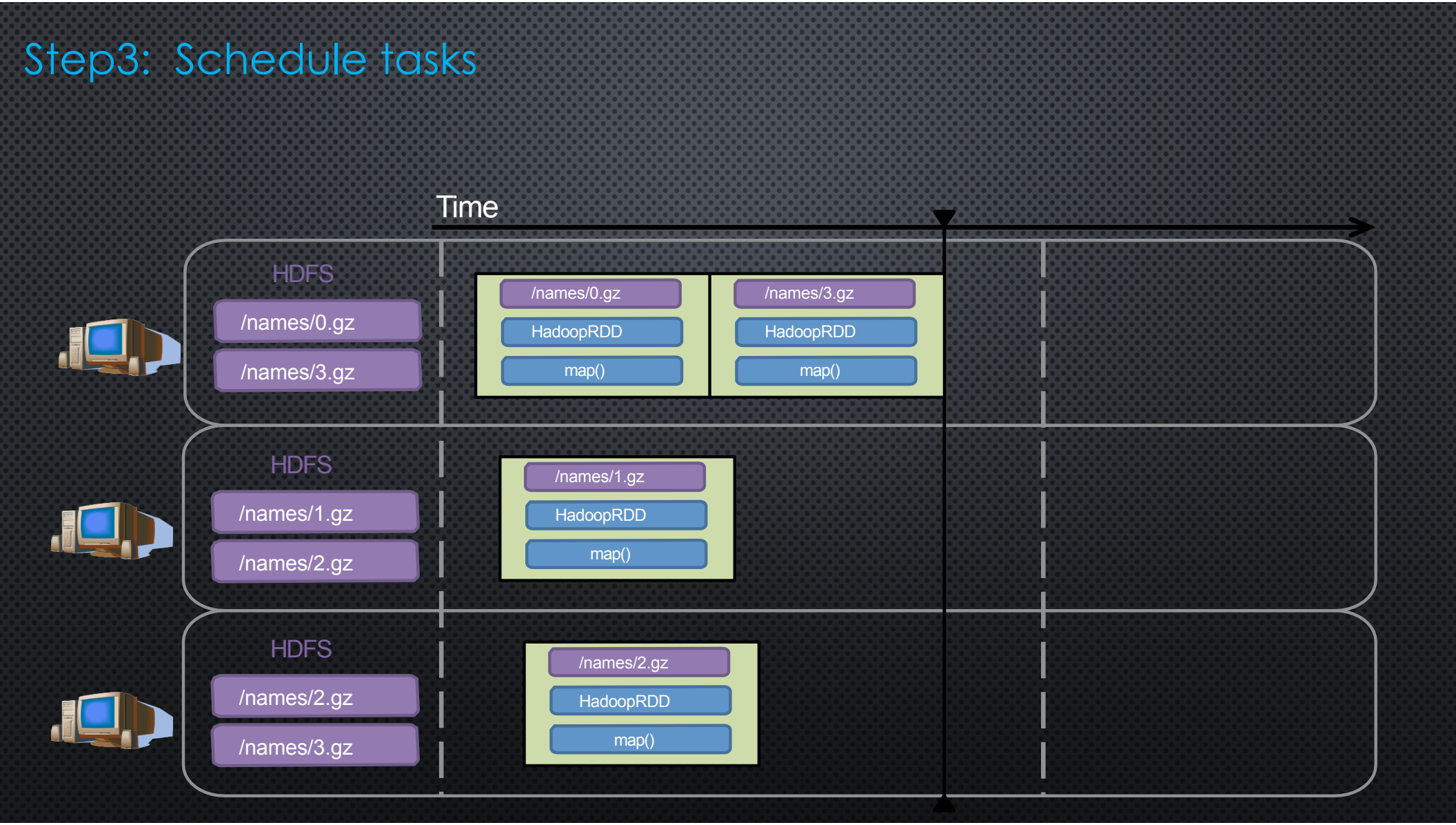
Step3: Schedule tasks



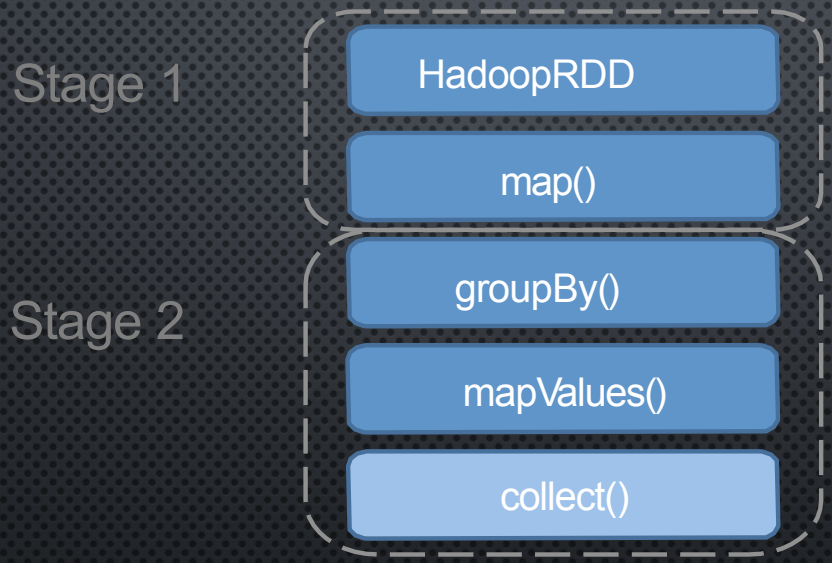
Step3: Schedule tasks



Step3: Schedule tasks

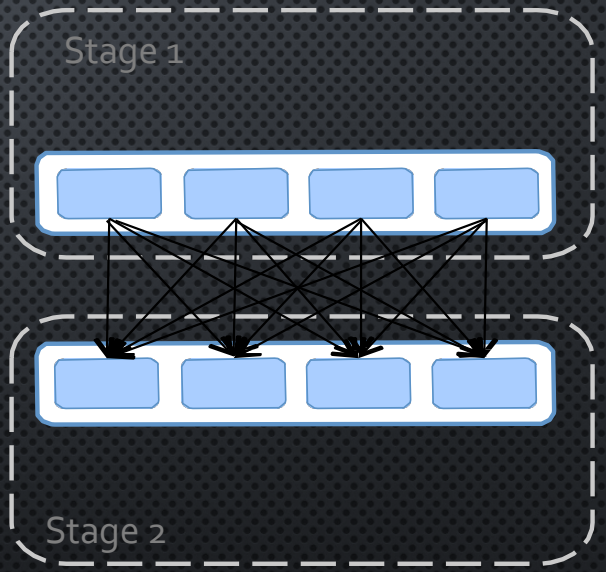


The Shuffle

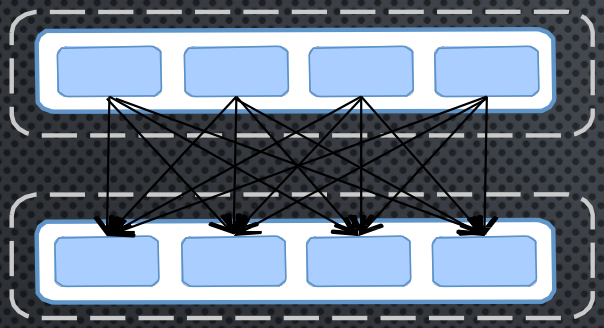


The Shuffle

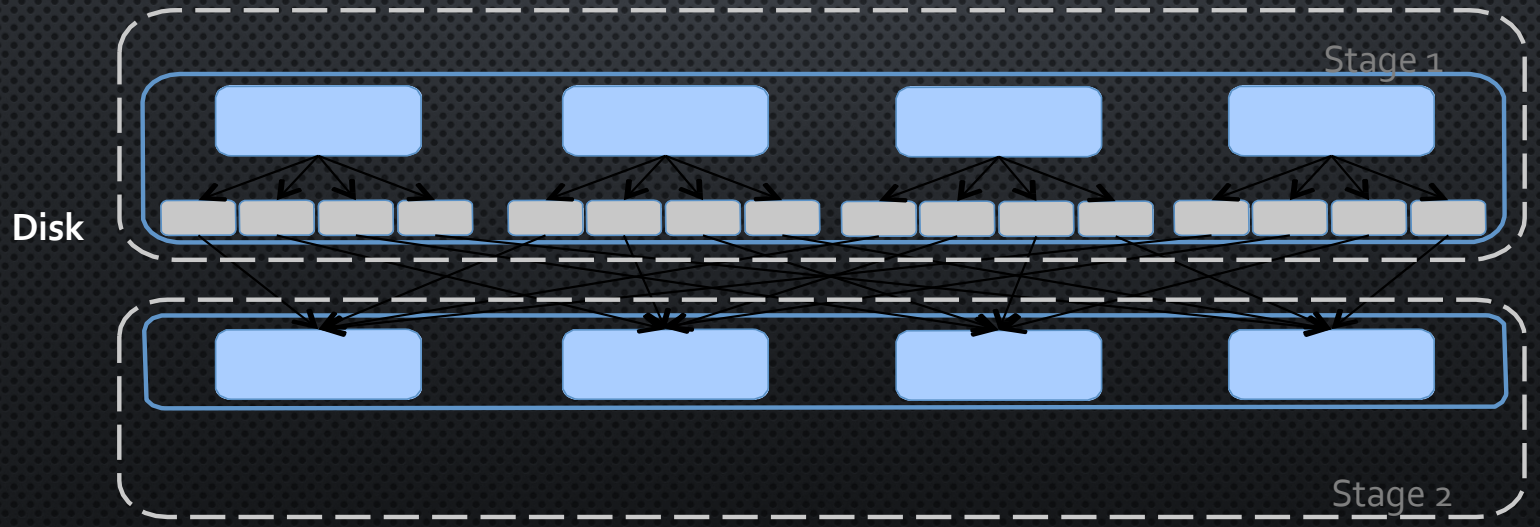
- Redistributes data among partitions
- Hash keys into buckets
- Optimizations:
  - Avoided when possible, if data is already properly partitioned
  - Partial aggregation reduces data movement



The Shuffle



- Pull-based, not push-based
- Write intermediate files to disk



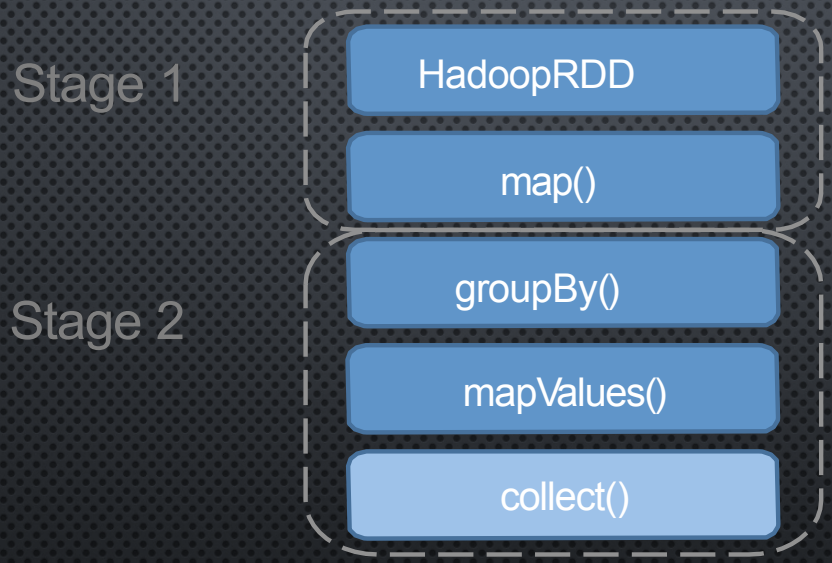
Execution of a groupBy

- Build hash map within each partition

  A => [Arsalan, Aaron, Andrew, Andrew, Andy, Ahir, Ali, …], E => [Erin, Earl, Ed, …]
  …

- Note: Can spill across keys, but a single key-value pair must fit in memory

Done!



What went wrong?

- Too few partitions to get good concurrency
- Large per-key groupBy()
- Shipped all data across the cluster

Common issue checklist

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of sorting and large keys in groupBys)
3. Minimize amount of data shuffled
4. Know the standard library

1 & 2 are about tuning number of partitions!

Importance of partition tuning

- Main issue: too few partitions
  - Less concurrency
  - More susceptible to data skew
  - Increased memory pressure for groupBy, reduceByKey, sortByKey, etc.
- Secondary issue: too many partitions
- Need "reasonable number" of partitions
  - Commonly between 100 and 10,000 partitions
  - Lower bound: At least ~2x number of cores in cluster
  - Upper bound: Ensure tasks take at least 100ms

Memory problems

- Symptoms:
  - Inexplicably bad performance
  - Inexplicable executor/machine failures (can indicate too many shuffle files too)
- Diagnosis:
  - Set spark.executor.extraJavaOptions to include
    - -XX:+PrintGCDetails
    - -XX:+HeapDumpOnOutOfMemoryError
  - Check dmesg for oom-killer logs
- Resolution:
  - Increase spark.executor.memory
  - Increase number of partitions
  - Re-evaluate program structure (!)

Fixing our mistakes

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

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of large groupBys and sorting)
3. Minimize data shuffle
4. Know the standard library

Fixing our mistakes

```
sc.textFile("hdfs:/names")
  .repartition(6)
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues { names => names.toSet.size }
  .collect()
```

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of large groupBys and sorting)
3. Minimize data shuffle
4. Know the standard library

Fixing our mistakes

```
sc.textFile("hdfs:/names")
  .repartition(6)
  .distinct()
  .map(name =>  (name.charAt(0),      name))
  .groupByKey()
  .mapValues   {   names =>  names.toSet.size       }
  .collect()
```

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of large groupBys and sorting)
1. Minimize data shuffle
2. Know the standard library

Fixing our mistakes

```
sc.textFile("hdfs:/names")
  .repartition(6)
  .distinct()
  .map(name =>  (name.charAt(0),      name))
  .groupByKey()
  .mapValues   {   names =>  names.size    }
  .collect()
```

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of large groupBys and sorting)
3. Minimize data shuffle
4. Know the standard library

Fixing our mistakes

```
sc.textFile("hdfs:/names")
  .distinct(numPartitions = 6)
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues { names => names.size }
  .collect()
```

1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of large groupBys and sorting)
3. Minimize data shuffle
4. Know the standard library

Fixing our mistakes

```
sc.textFile("hdfs:/names")
  .distinct(numPartitions = 6)
  .map(name => (name.charAt(0), 1))
  .reduceByKey(_ + _)
  .collect()
```

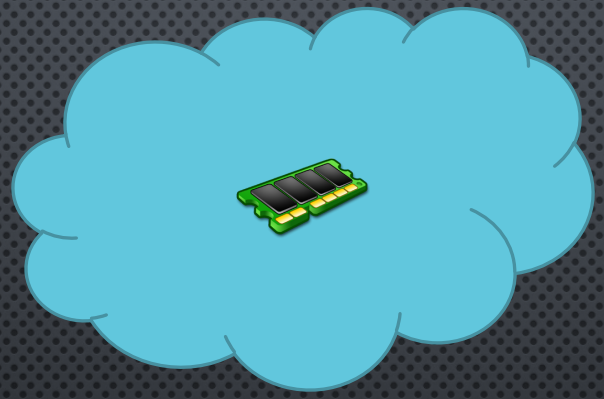
1. Ensure enough partitions for concurrency
2. Minimize memory consumption (esp. of large groupBys and sorting)
3. Minimize data shuffle
4. Know the standard library

Fixing our mistakes

```
sc.textFile("hdfs:/names")
  .distinct(numPartitions = 6)
  .map(name => (name.charAt(0), 1))
  .reduceByKey(_ + _)
  .collect()
```

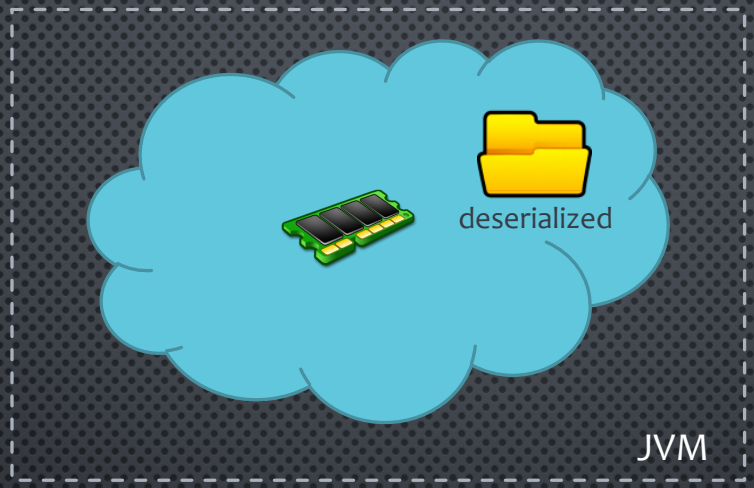
Original:

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



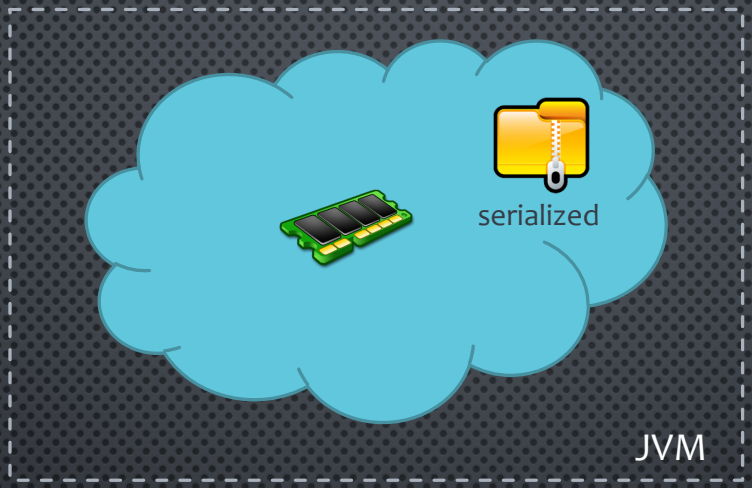
MEMORY AND PERSISTENCE


Vs.

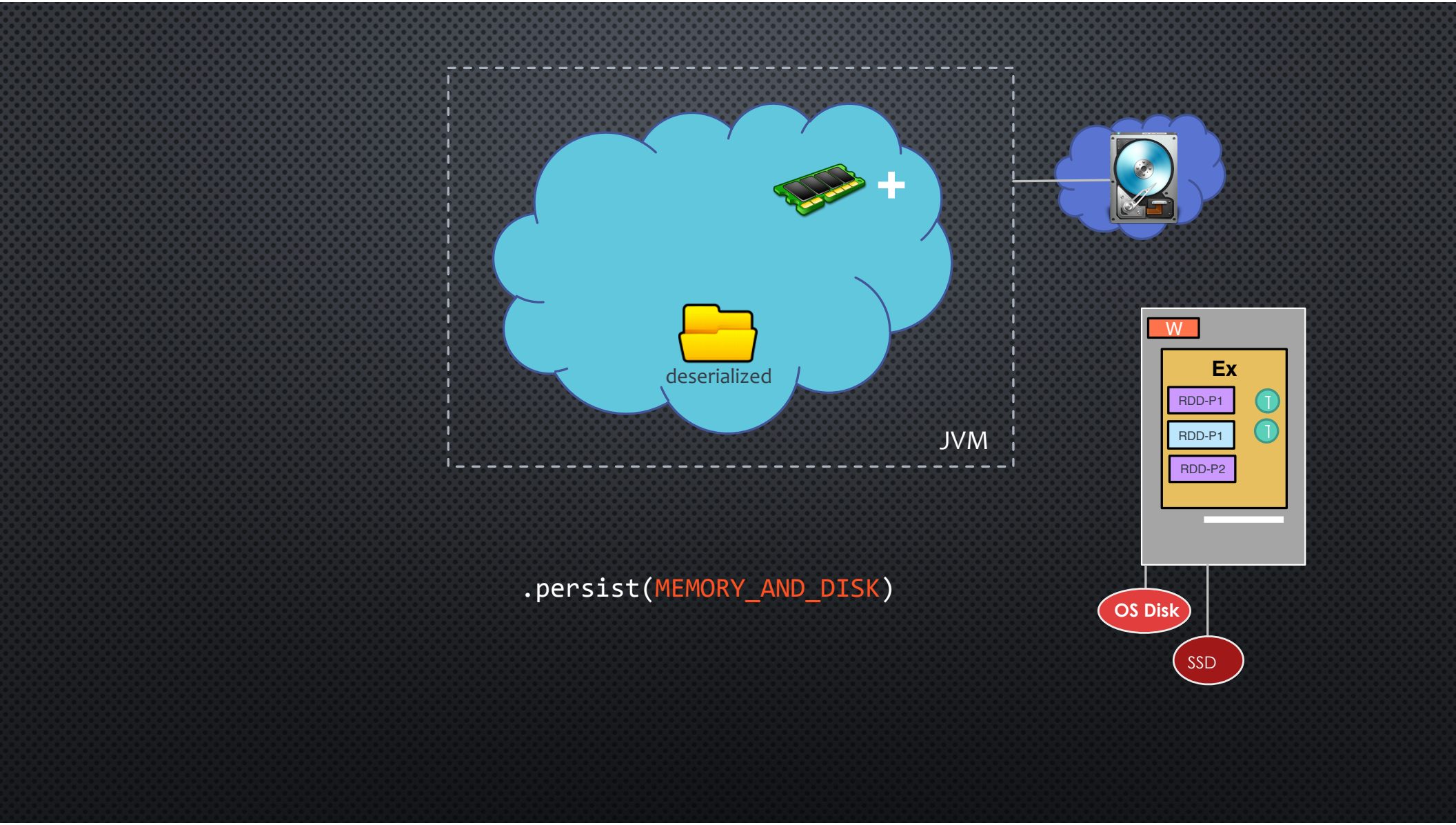



```
RDD.cache()  ==  RDD.persist(MEMORY_ONLY)
```

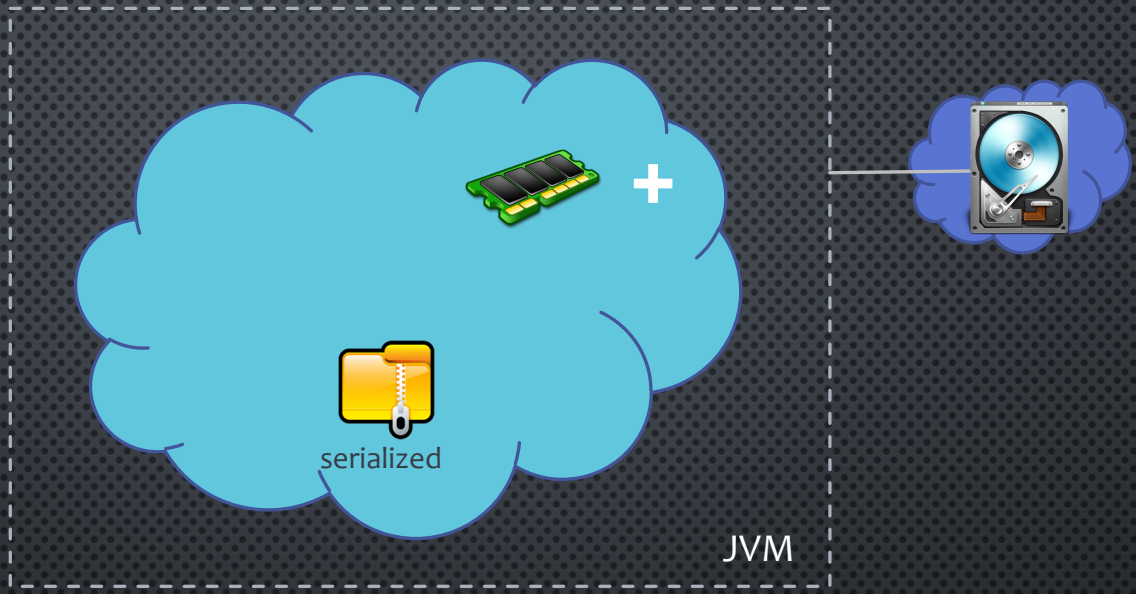
most CPU-efficient option



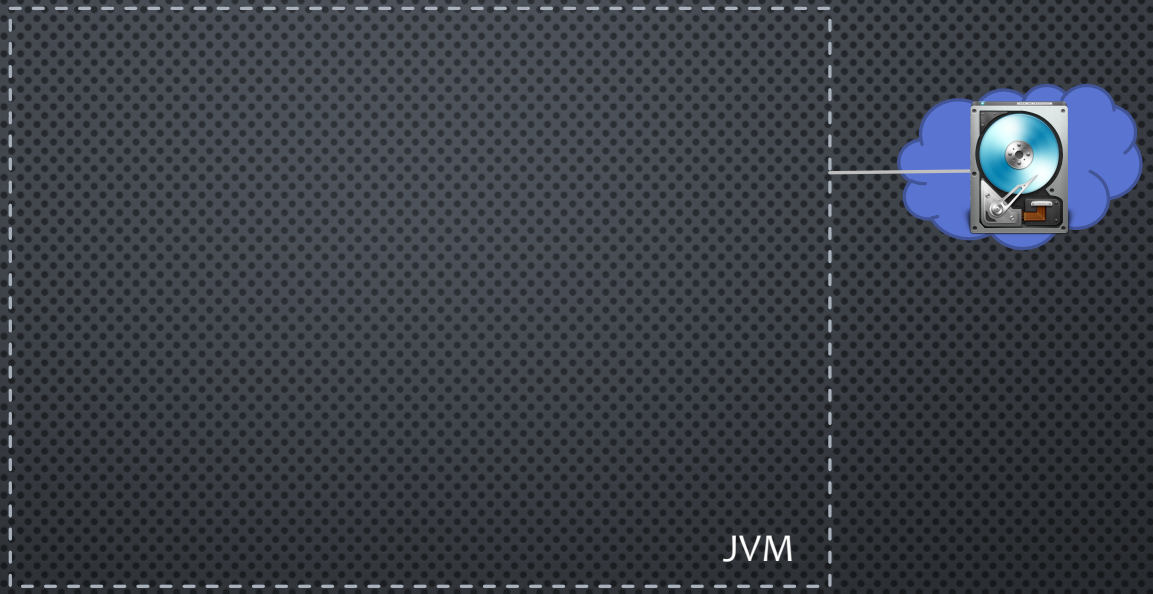

```
RDD.cache()  ==  RDD.persist(MEMORY_ONLY_SER)
```



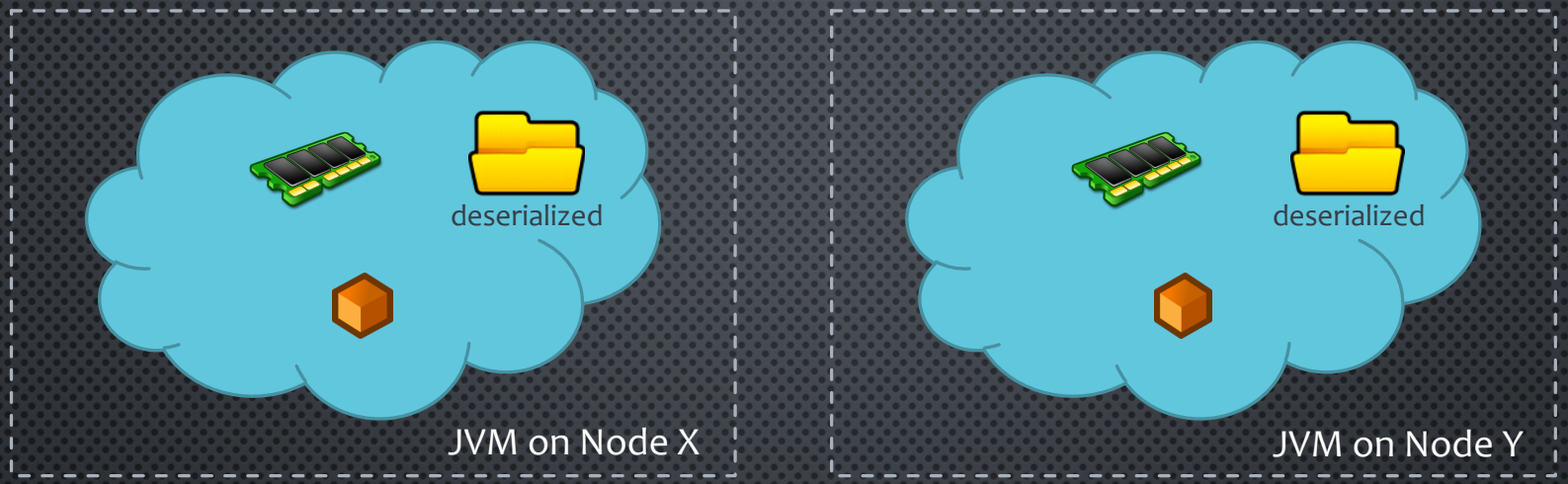

```
.persist(MEMORY_AND_DISK)
```



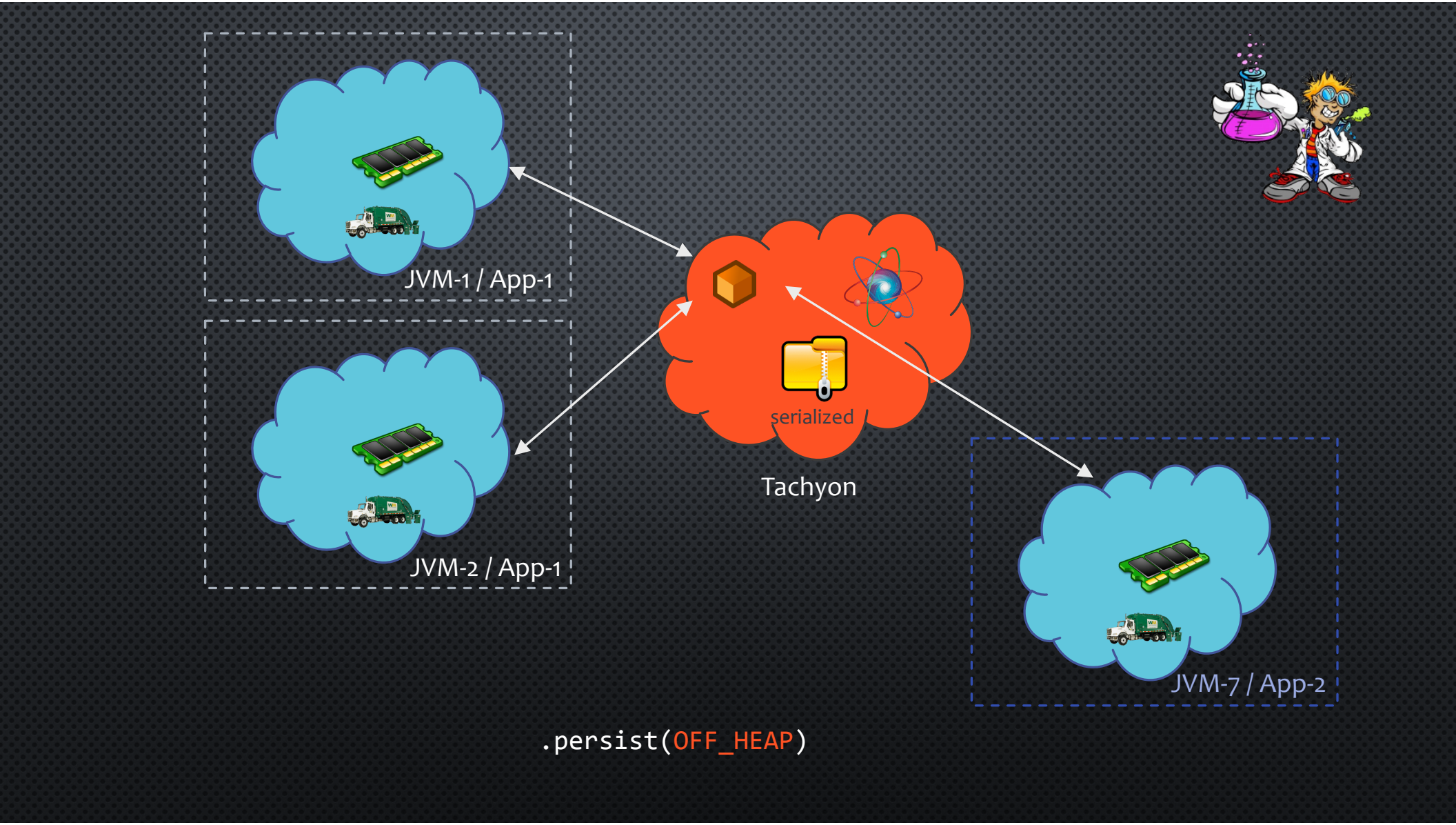

```
.persist(MEMORY_AND_DISK_SER)
```



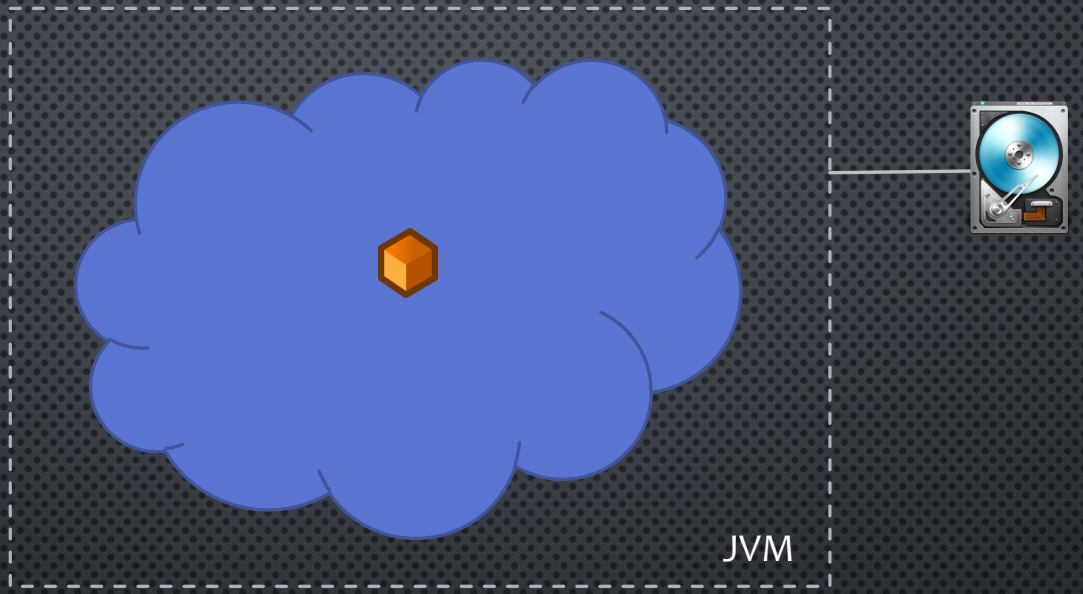

```
.persist(DISK_ONLY)
```



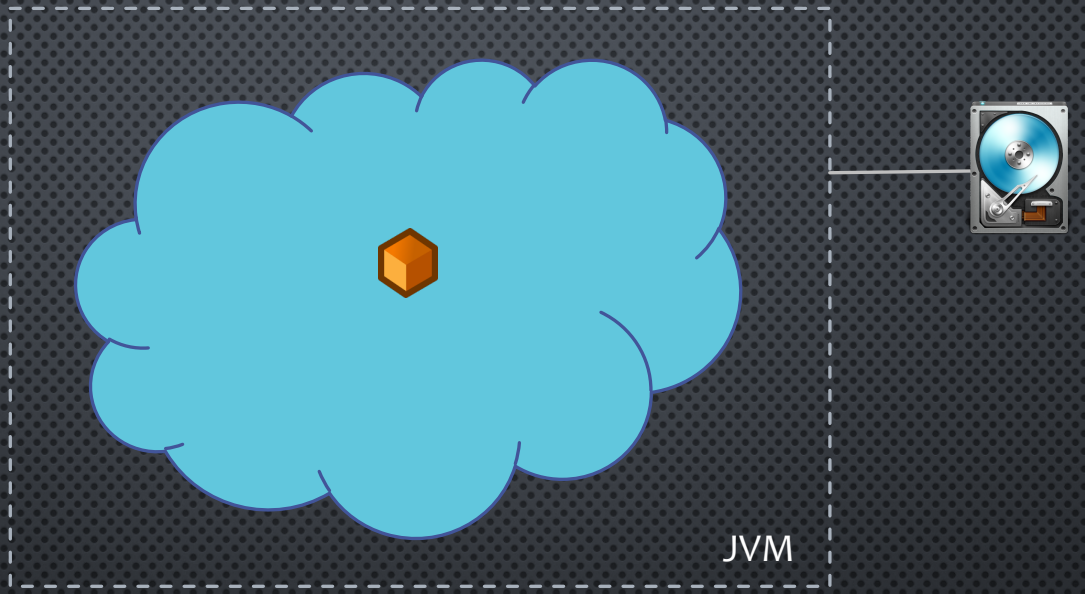

```
RDD.persist(MEMORY_ONLY_2)
```



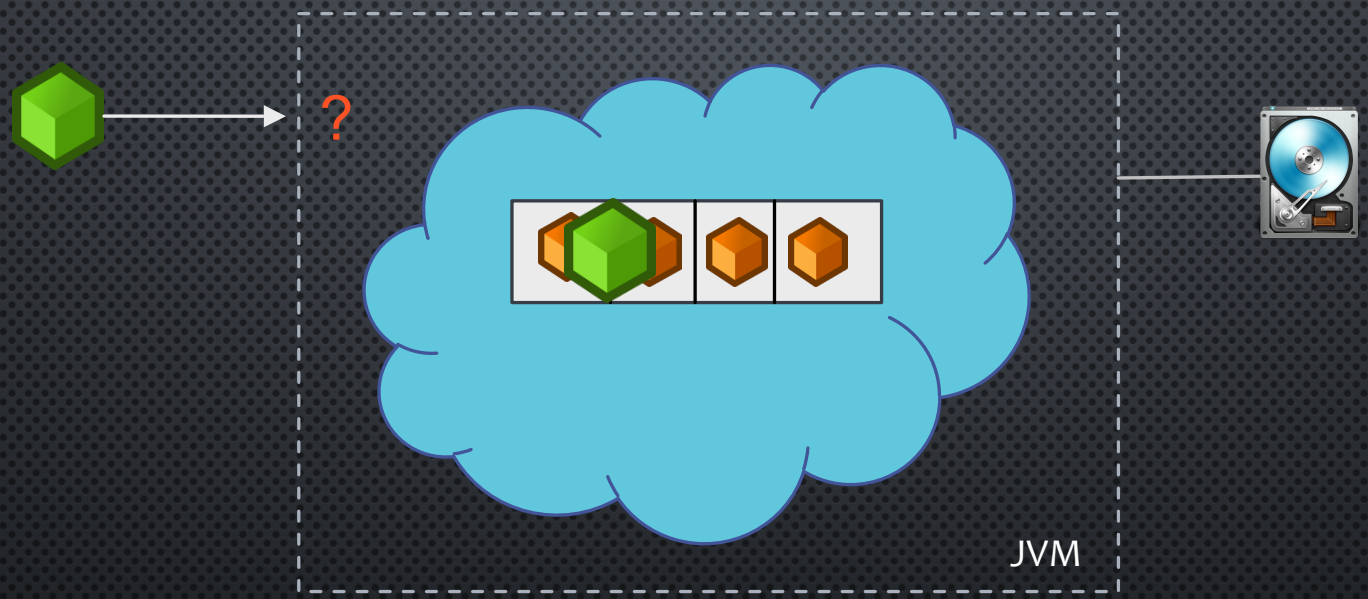

```
.persist(OFF_HEAP)
```



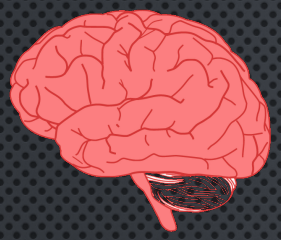

```
.unpersist()
```




```
.unpersist()
```


?
JVM

- If RDD fits in memory, choose MEMORY_ONLY
- If not, use MEMORY_ONLY_SER w/ fast serialization library
- Don't spill to disk unless functions that computed the datasets are very expensive or they filter a large amount of data. (recomputing may be as fast as reading from disk)
- Use replicated storage levels sparingly and only if you want fast fault recovery (maybe to serve requests from a web app)



Remember!

Intermediate data is automatically persisted during shuffle operations

PySpark: stored objects will always be serialized with Pickle library, so it does not matter whether you choose a serialized level.

DATA SERIALIZATION

"Often, this will be the first thing you should tune to optimize a Spark application."

- Spark docs

Java serialization

vs.

Kryo serialization

- Uses Java's **ObjectOutputStream** framework
- Works with any class you create that implements **java.io.Serializable**
- You can control the performance of serialization more closely by extending **java.io.Externalizable**
- Flexible, but quite slow
- Leads to large serialized formats for many classes

- Recommended serialization for production apps
- Use Kyro version 2 for speedy serialization (10x) and more compactness
- Does not support all **Serializable** types
- Requires you to *register* the classes you'll use in advance
- If set, will be used for serializing shuffle data between nodes and also serializing RDDs to disk

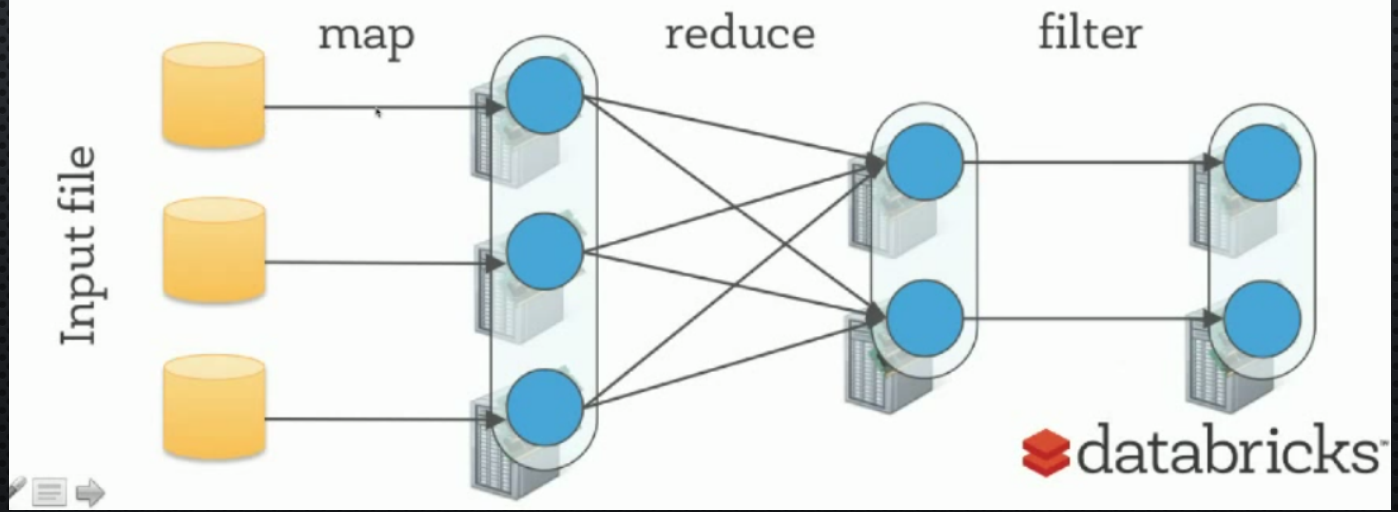
```
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
```

FAULT TOLERANCE & LINEAGE

Fault Tolerance

RDDs track *lineage* info to rebuild lost data

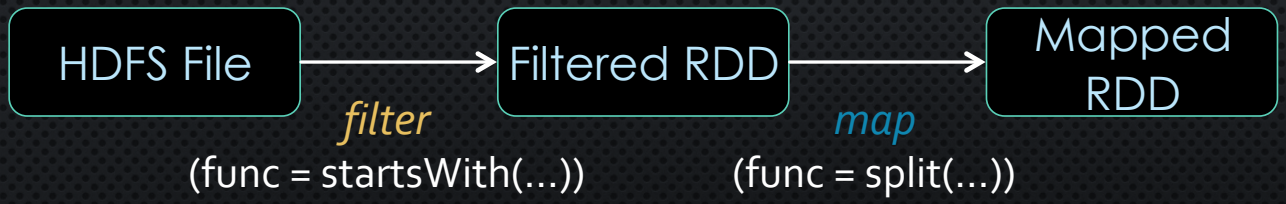
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



EASY 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])
```



LINEAGE

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica

University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, and can be implemented efficiently by extending a data flow system with dataflow operators. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that reuse intermediate results across multiple computations. Data reuse is common in many iterative machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is interactive data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (e.g., between two MapReduce jobs) is to write it to an external stable storage system, e.g., a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing the problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [21] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (e.g., looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, e.g., to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called resilient distributed datasets (RDDs) that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance efficiently. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (e.g., cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to this interface, RDDs provide an interface based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute just

[1] Checkpointing the data in some RDDs may be useful when the lineage graph becomes long and lossy recovery would take a lot of time.

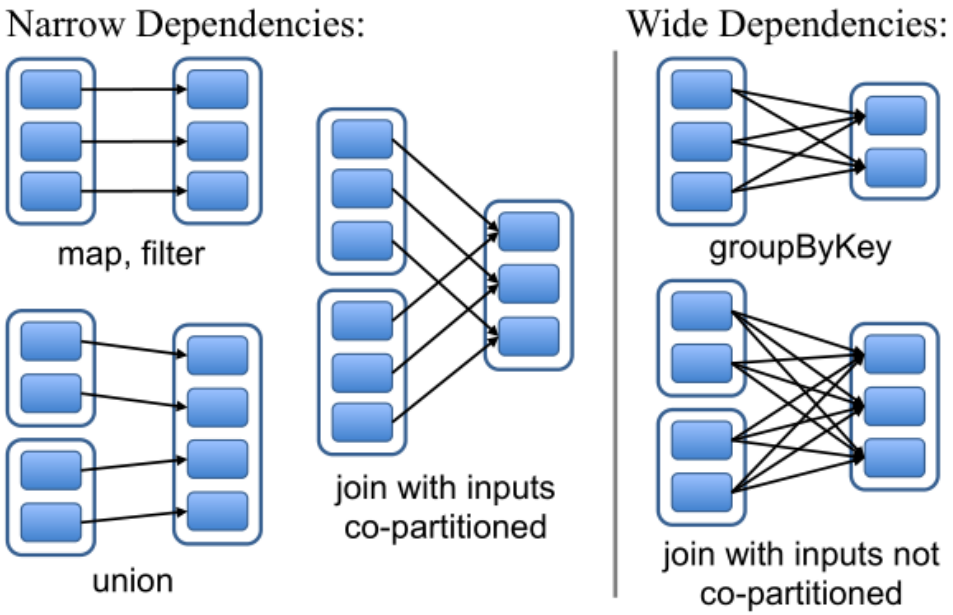
"One of the challenges in providing RDDs as an abstraction is choosing a representation for them that can track lineage across a wide range of transformations."

"The most interesting question in designing this interface is how to represent dependencies between RDDs."

"We found it both sufficient and useful to classify dependencies into two types:

- narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD
- wide dependencies, where multiple child partitions may depend on it."

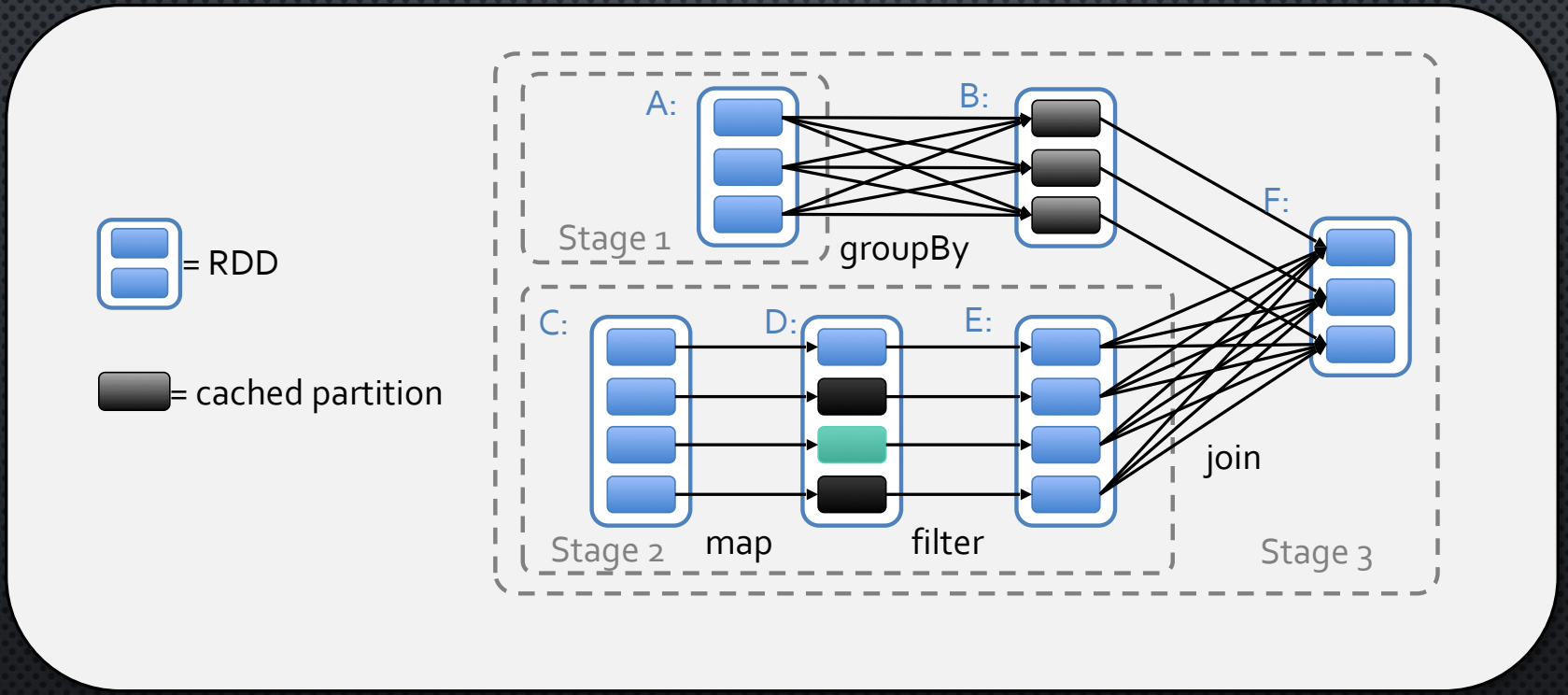
LINEAGE



Examples of narrow and wide dependencies.

Each box is an RDD, with partitions shown as shaded rectangles.

LINEAGE


= RDD
= cached partition
A:
B:
F:
Stage 1
groupBy
C:
D:
E:
join
Stage 2
map
filter
Stage 3

LINEAGE

Dependencies: Narrow vs Wide

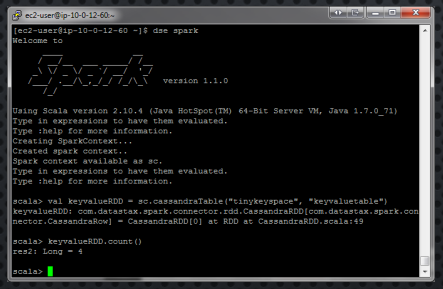
"This distinction is useful for two reasons:

1) Narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a map followed by a filter on an element-by-element basis.

In contrast, wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce-like operation.

2) Recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution."

To display the lineage of an RDD, Spark provides a **toDebugString** method:




```
scala> input.toDebugString

res85: String =
(2) data.text MappedRDD[292] at textFile at <console>:13
 | data.text HadoopRDD[291] at textFile at <console>:13

scala> counts.toDebugString
res84: String =
(2) ShuffledRDD[296] at reduceByKey at <console>:17
 +-(2) MappedRDD[295] at map at <console>:17
    | FilteredRDD[294] at filter at <console>:15
    | MappedRDD[293] at map at <console>:15
    | data.text MappedRDD[292] at textFile at <console>:13
    | data.text HadoopRDD[291] at textFile at <console>:13
```

DATA LOCALITY

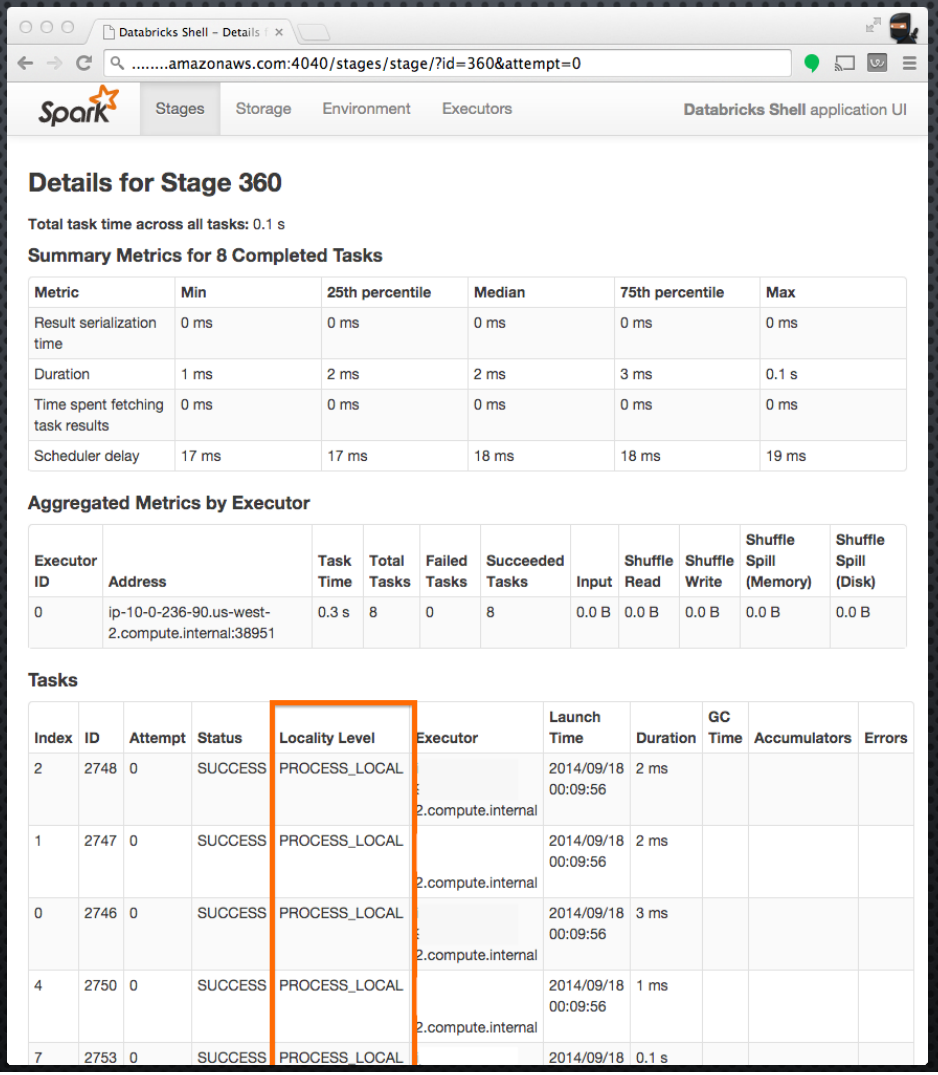
Data locality is how close data is to the code processing it. There are several levels of locality based on the data's current location. In order from closest to farthest:

PROCESS_LOCAL data is in the same JVM as the running code. This is the best locality possible
NODE_LOCAL data is on the same node. Examples might be in HDFS on the same node, or in another executor on the same node. This is a little slower than PROCESS_LOCAL because the data has to travel between processes
NO_PREF data is accessed equally quickly from anywhere and has no locality preference
RACK_LOCAL data is on the same rack of servers. Data is on a different server on the same rack so needs to be sent over the network, typically through a single switch
ANY data is elsewhere on the network and not in the same rack

it is faster to ship serialized code from place to place than a chunk of data because code size is much smaller than data. Spark builds its scheduling around this general principle of data locality.

The wait timeout for fallback between each level can be configured individually

Checking Data Locality



Details for Stage 360

Total task time across all tasks: 0.1 s

Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	0 ms	0 ms	0 ms
Duration	1 ms	2 ms	2 ms	3 ms	0.1 s
Time spent fetching task results	0 ms	0 ms	0 ms	0 ms	0 ms
Scheduler delay	17 ms	17 ms	18 ms	18 ms	19 ms

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input	Shuffle Read	Shuffle Write	Shuffle Spill (Memory)	Shuffle Spill (Disk)
0	ip-10-0-236-90.us-west-2.compute.internal:38951	0.3 s	8	0	8	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

Tasks

Index	ID	Attempt	Status	Locality Level	Executor	Launch Time	Duration	GC Time	Accumulators	Errors
2	2748	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	2 ms			
1	2747	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	2 ms			
0	2746	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	3 ms			
4	2750	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	1 ms			
7	2753	0	SUCCESS	PROCESS_LOCAL		2014/09/18	0.1 s			

Types of RDDs

- HadoopRDD
- FilteredRDD
- MappedRDD
- PairRDD
- ShuffledRDD
- UnionRDD
- PythonRDD
- DoubleRDD
- JdbcRDD
- JsonRDD
- SchemaRDD
- VertexRDD
- EdgeRDD
- CassandraRDD *(DataStax)*
- GeoRDD *(ESRI)*
- EsSpark *(ElasticSearch)*

Pair RDDs

- **Pair RDD are special form of RDDs**
  - Each element must be a key value pair (a two element tuple)
  - Keys and values can be any type

- **Why?**
  - Use with Map-Reduce algorithms
  - Many additional functions are available for common data processing algorithms such as
  - Sorting, Joining, Grouping, Counting etc

Python create pair RDD using the first word as the key

```
input.map(lambda x: (x.split(" ")[0], x))
```