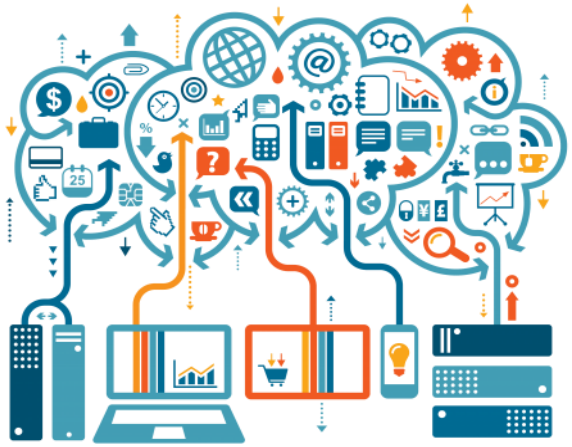
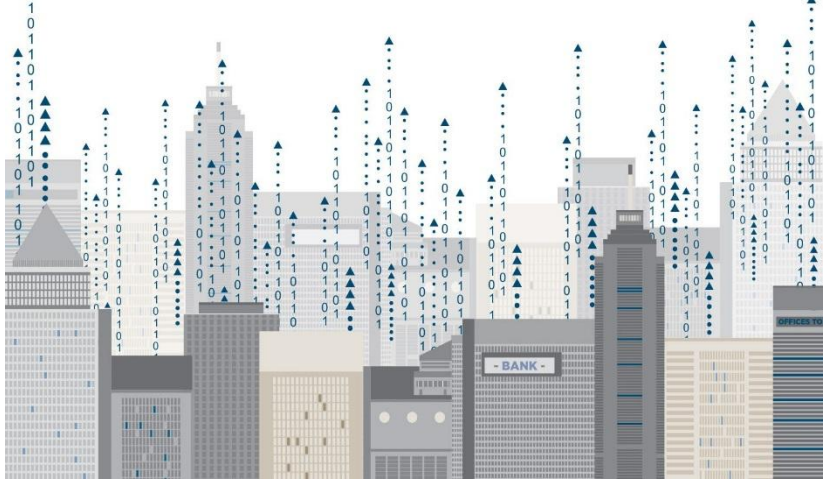




Big Data Analytics

Fall 2025

Lecture 12

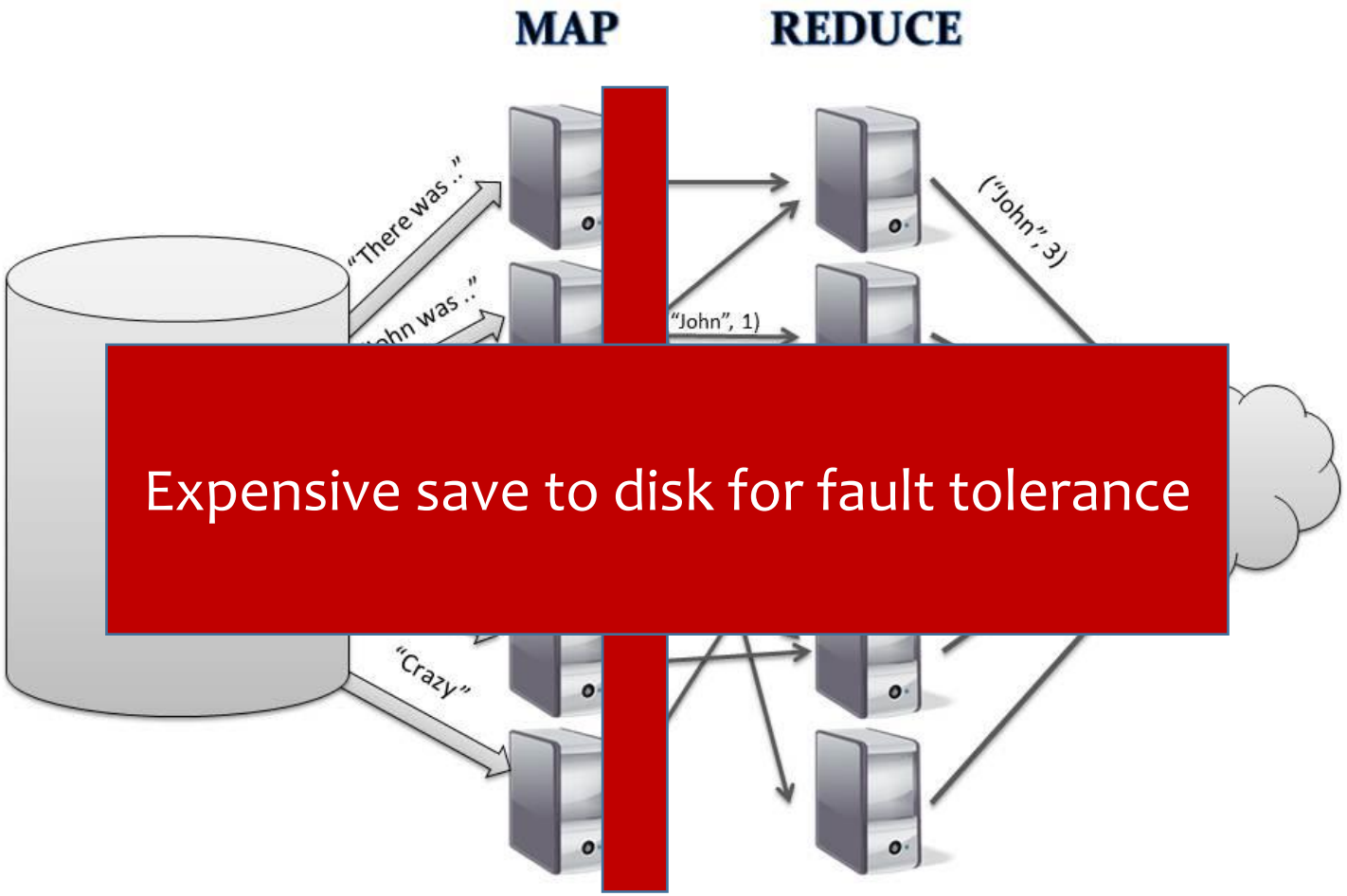
Why Spark?

- Another system for BDA

- Isn’t MapReduce good enough?

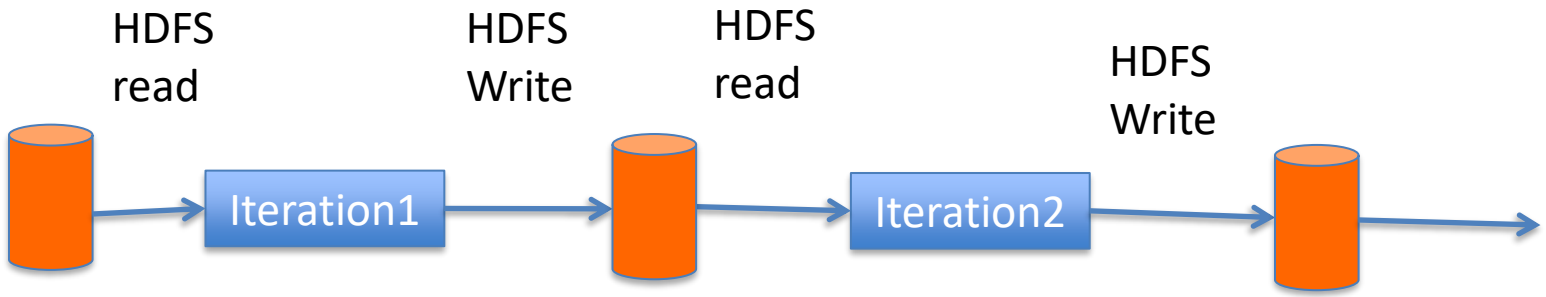
Why Spark?

- MapReduce can be expensive for some applications e.g., Iterative
- Lacks efficient data sharing

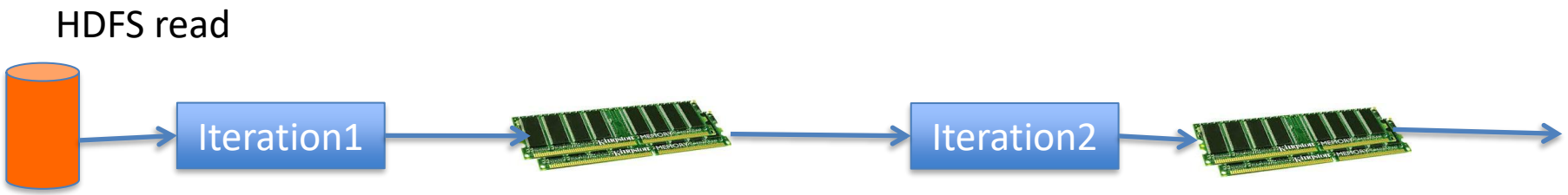

MAP
REDUCE
"There was .."
"John was .."
"Crazy"
"John", 1)
("John", 3)
Expensive save to disk for fault tolerance

Spark Uses Memory instead of Disk

Hadoop: Use Disk for Data Sharing



Spark: In-Memory Data Sharing



Sort competition

	Hadoop MR Record (2013)	Spark Record (2014)
Data Size	102.5 TB	100 TB
Elapsed Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400 physical	6592 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**

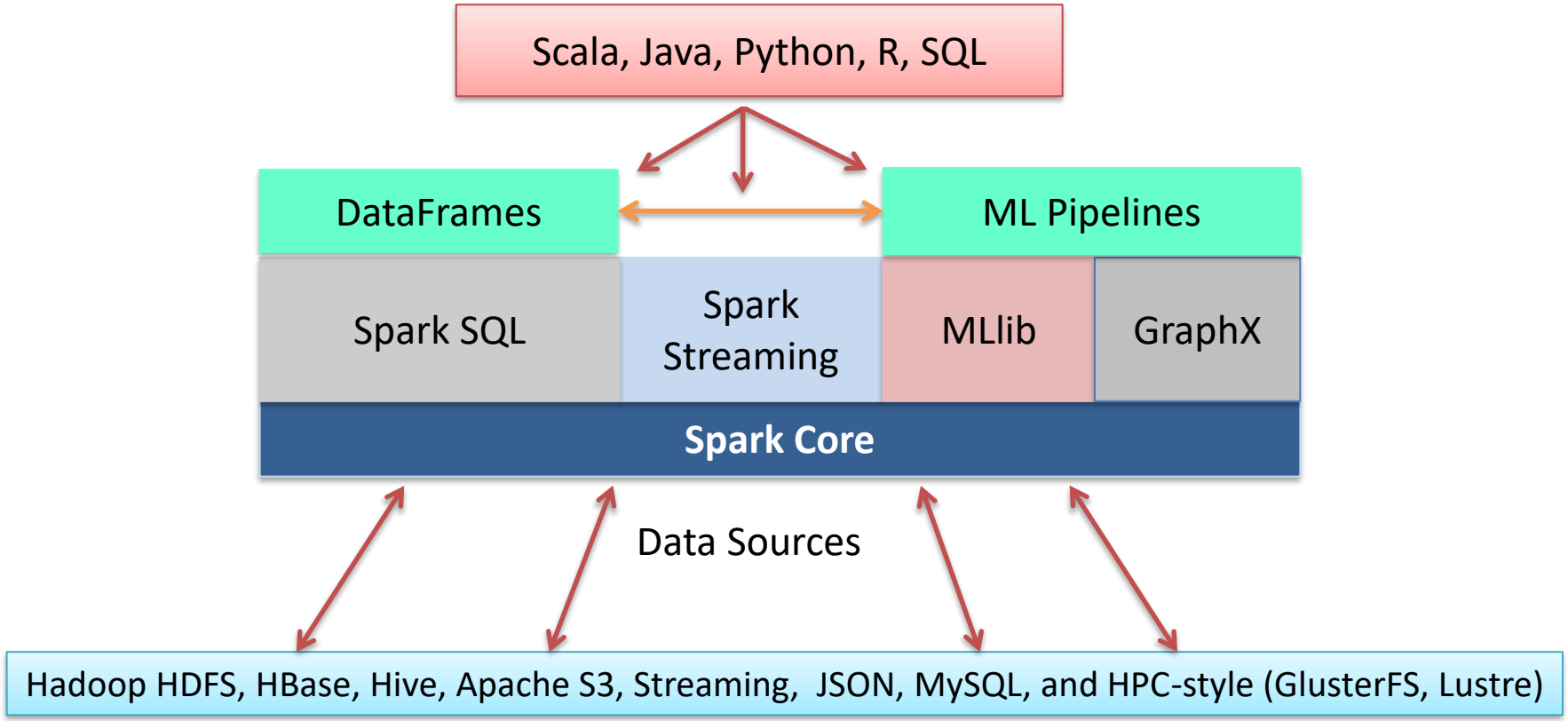
Spark, 3x faster with 1/10 the nodes

Sort benchmark, Daytona Gray: sort of 100 TB of data (1 trillion records)
http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

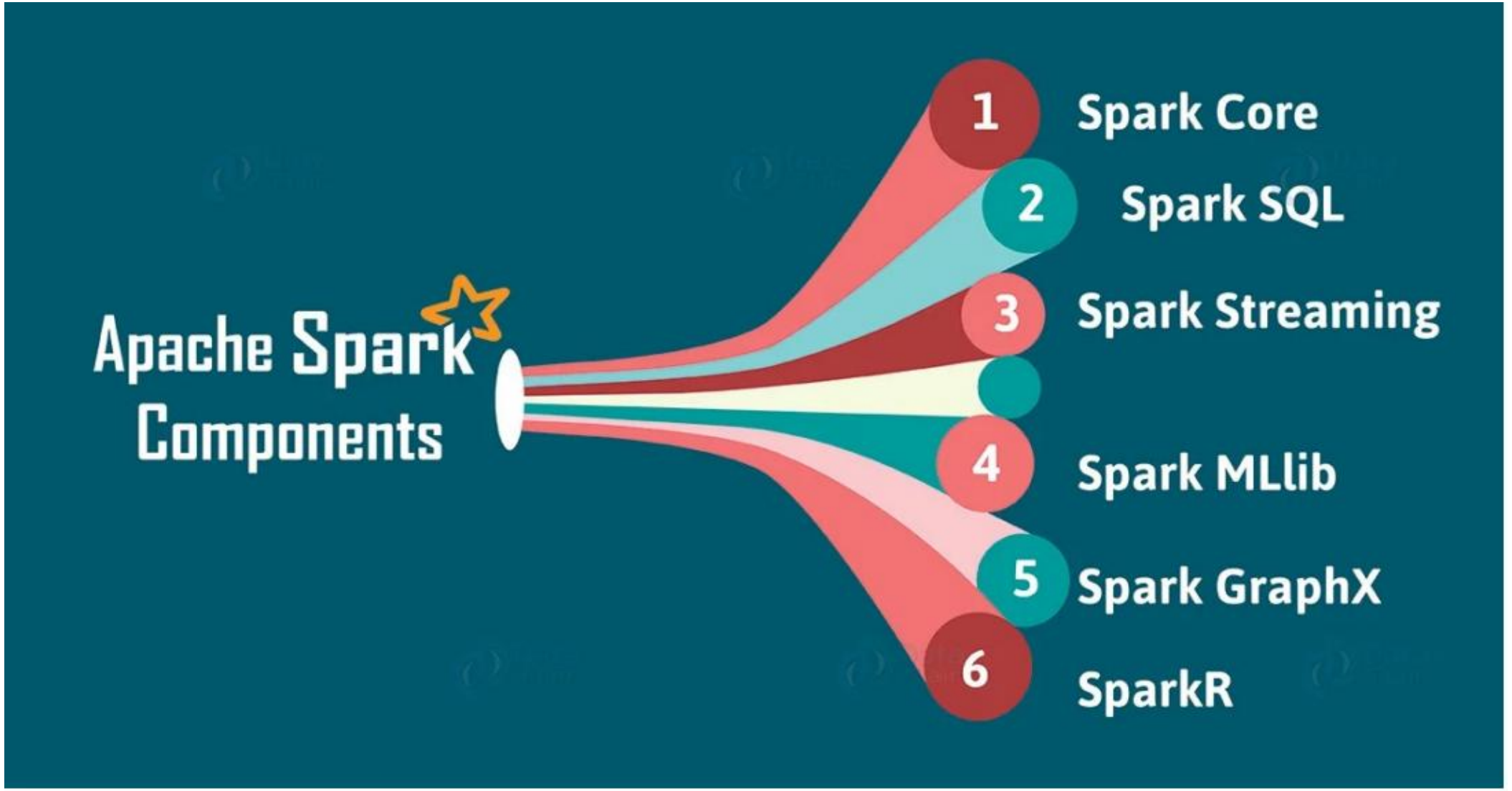
- Apache Spark is a lightning-fast cluster computing technology, designed for fast computation.
- It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing.
- The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application.

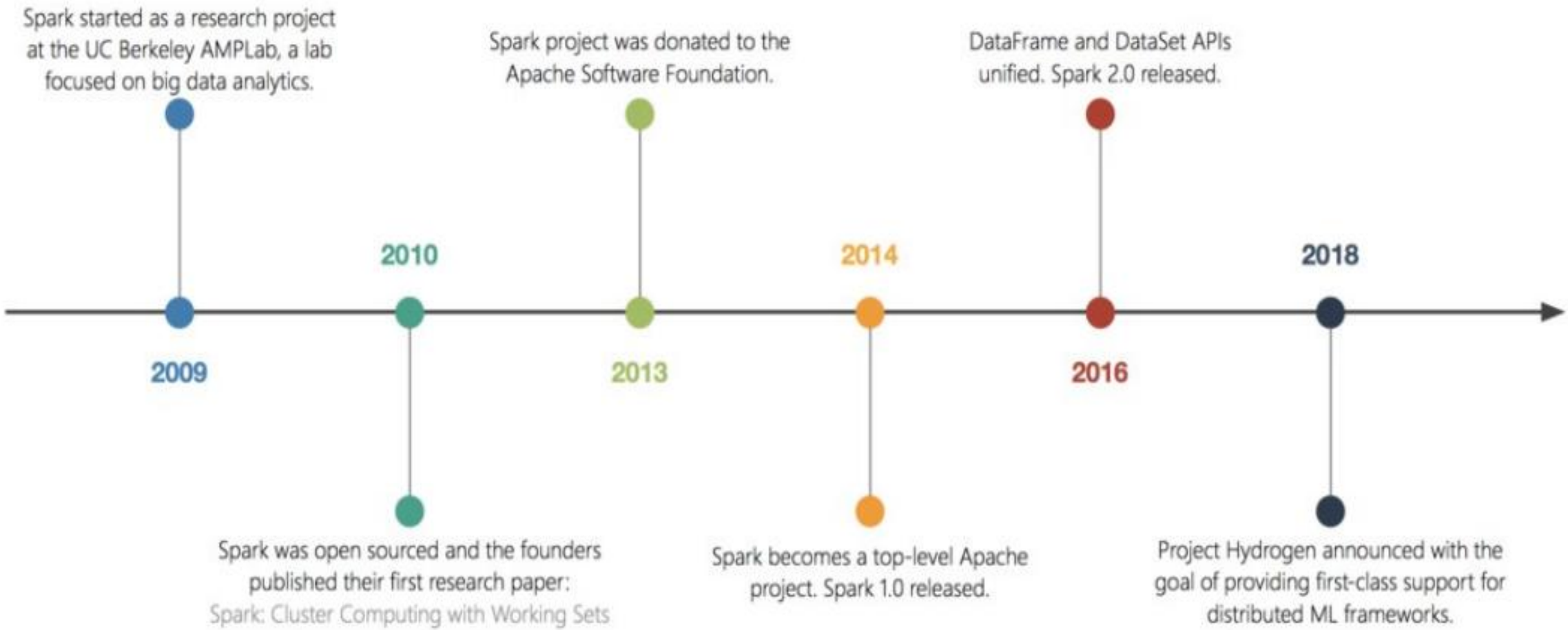
Apache Spark

Apache Spark supports data analysis, machine learning, graphs, streaming data, etc. It can read/write from a range of data types and allows development in multiple languages.

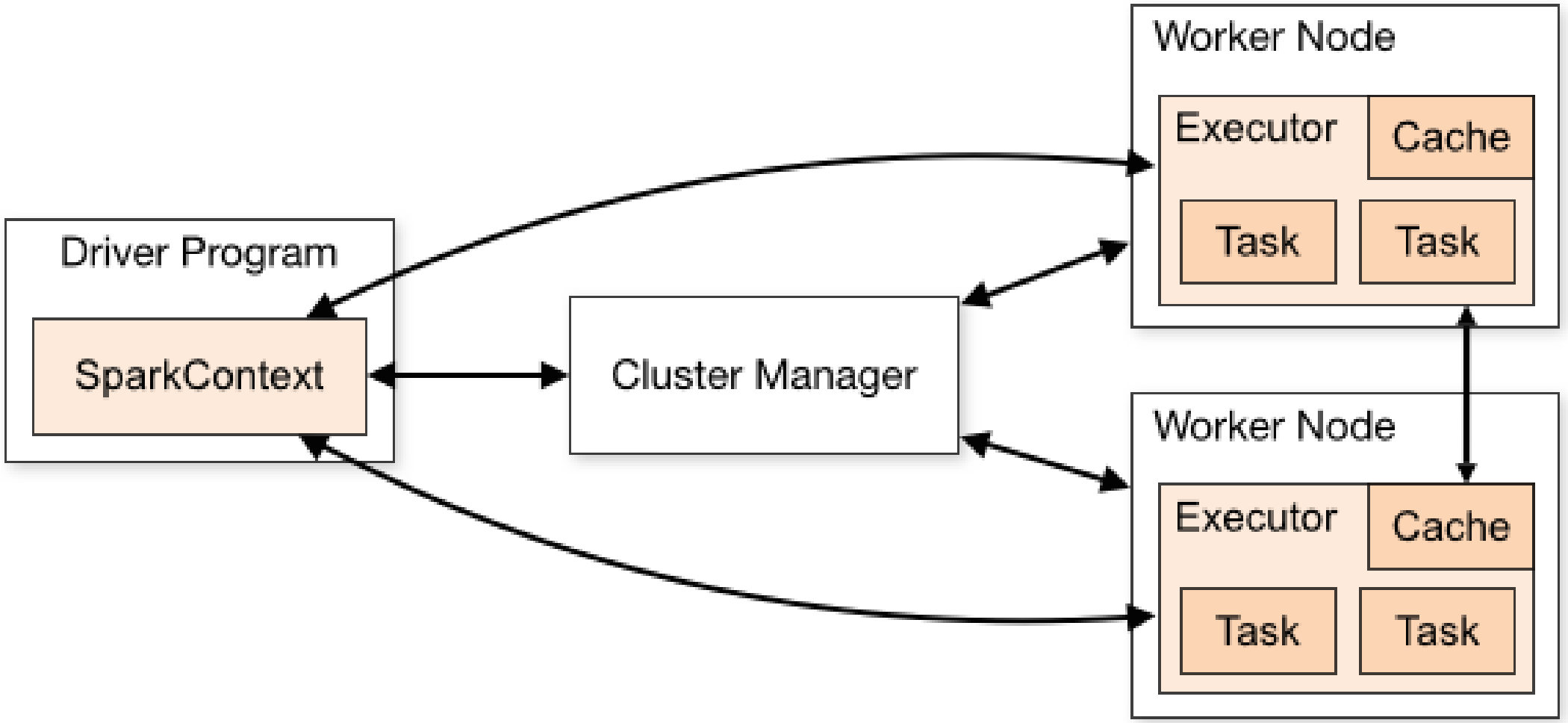


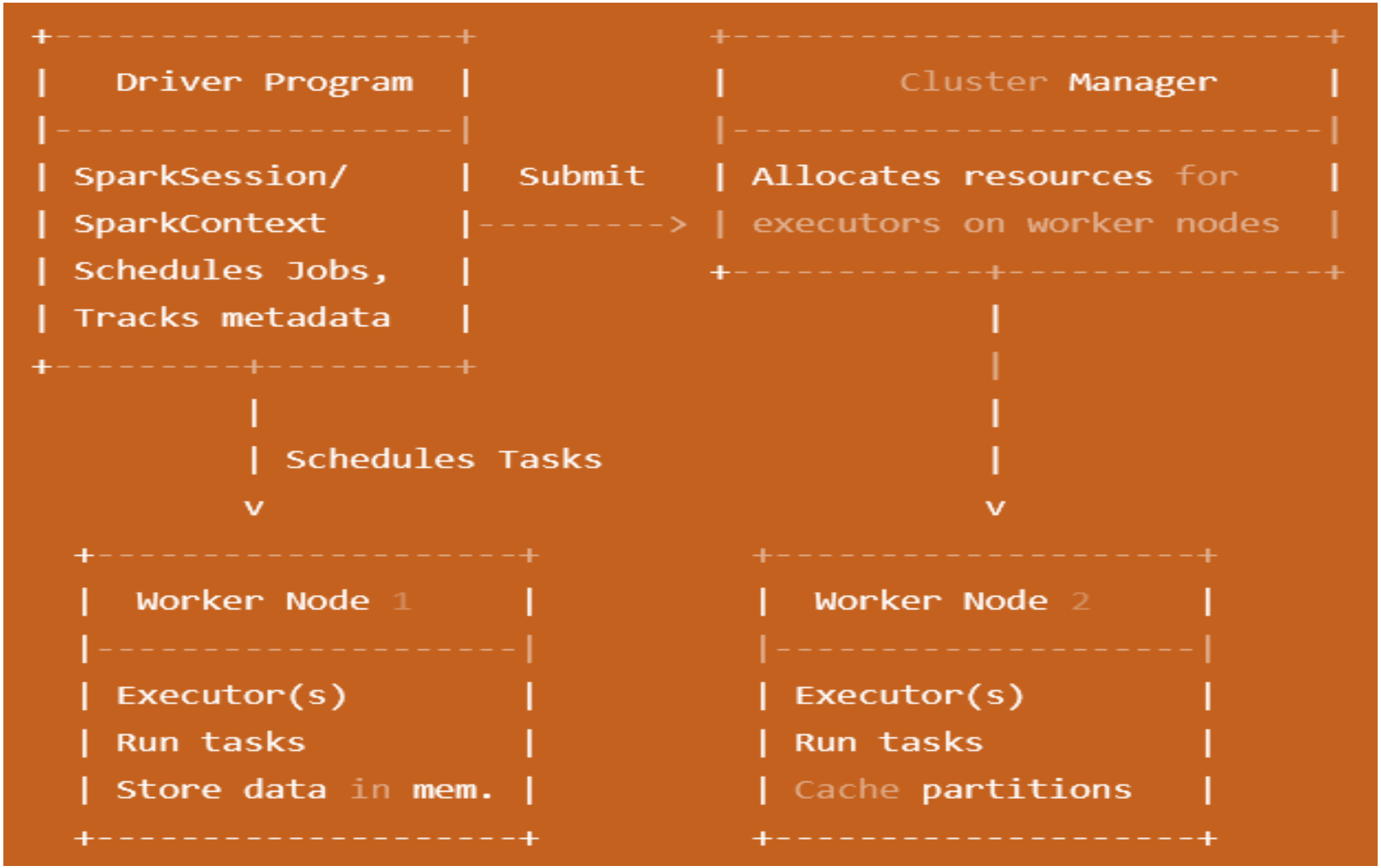
- Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming.
- Apart from supporting all these workload in a respective system, it reduces the management burden of maintaining separate tools.

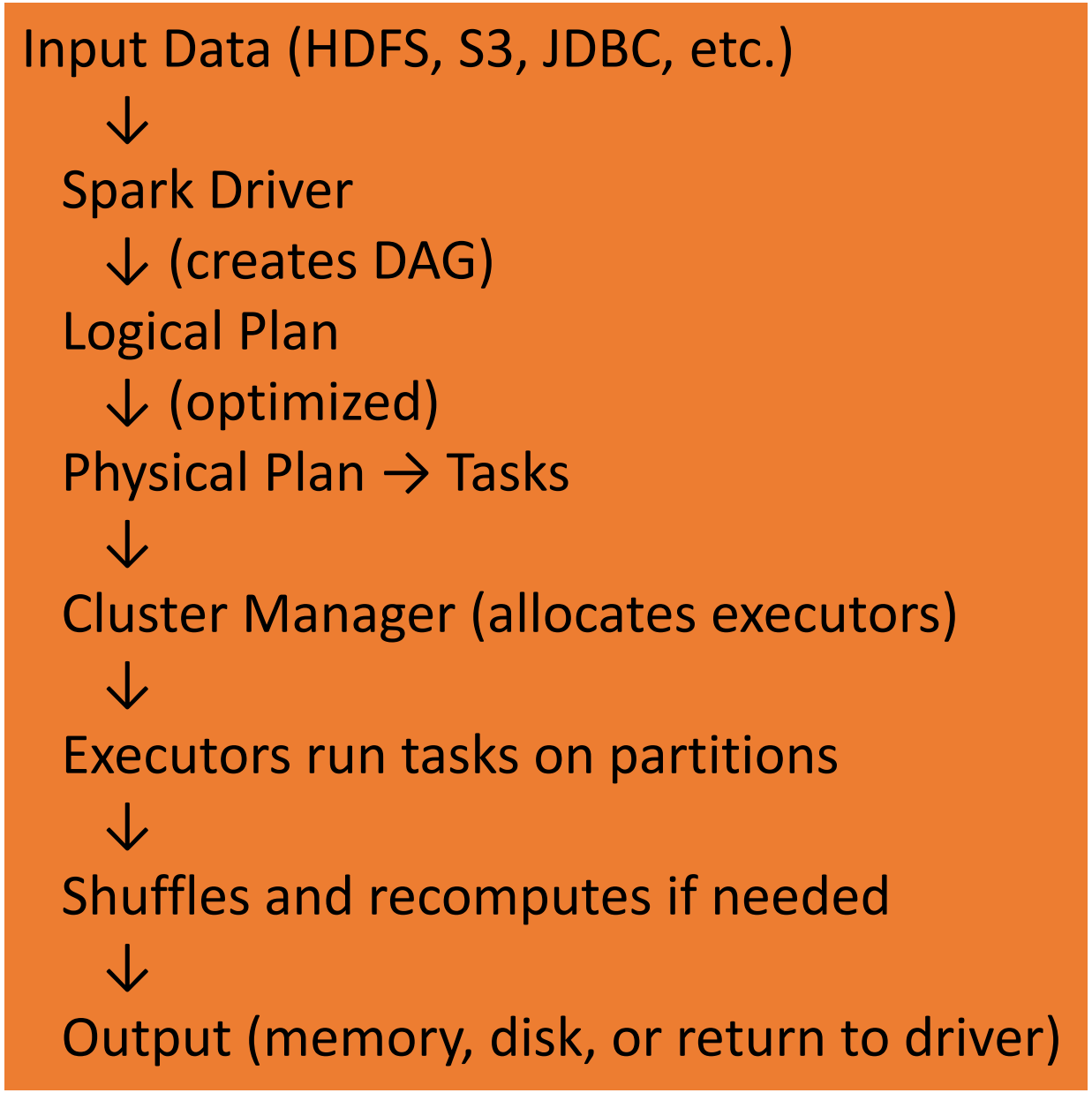

Apache Spark
Components
1
Spark Core
2
Spark SQL
3
Spark Streaming
4
Spark MLlib
5
Spark GraphX
6
SparkR


Spark started as a research project at the UC Berkeley AMPLab, a lab focused on big data analytics.
2009
2010
Spark was open sourced and the founders published their first research paper:
Spark: Cluster Computing with Working Sets
Spark project was donated to the Apache Software Foundation.
2013
2014
Spark becomes a top-level Apache project. Spark 1.0 released.
DataFrame and DataSet APIs unified. Spark 2.0 released.
2016
2018
Project Hydrogen announced with the goal of providing first-class support for distributed ML frameworks.

Basic Spark Cluster Architecture


Worker Node
Executor
Cache
Task
Task
Driver Program
SparkContext
Cluster Manager
Worker Node
Executor
Cache
Task
Task


Driver Program
SparkSession/
SparkContext
Schedules Jobs,
Tracks metadata
Submit
Cluster Manager
Allocates resources for
executors on worker nodes
Schedules Tasks
Worker Node 1
Executor(s)
Run tasks
Store data in mem.
Worker Node 2
Executor(s)
Run tasks
Cache partitions


Input Data (HDFS, S3, JDBC, etc.)
↓
Spark Driver
↓ (creates DAG)
Logical Plan
↓ (optimized)
Physical Plan → Tasks
↓
Cluster Manager (allocates executors)
↓
Executors run tasks on partitions
↓
Shuffles and recomputes if needed
↓
Output (memory, disk, or return to driver)

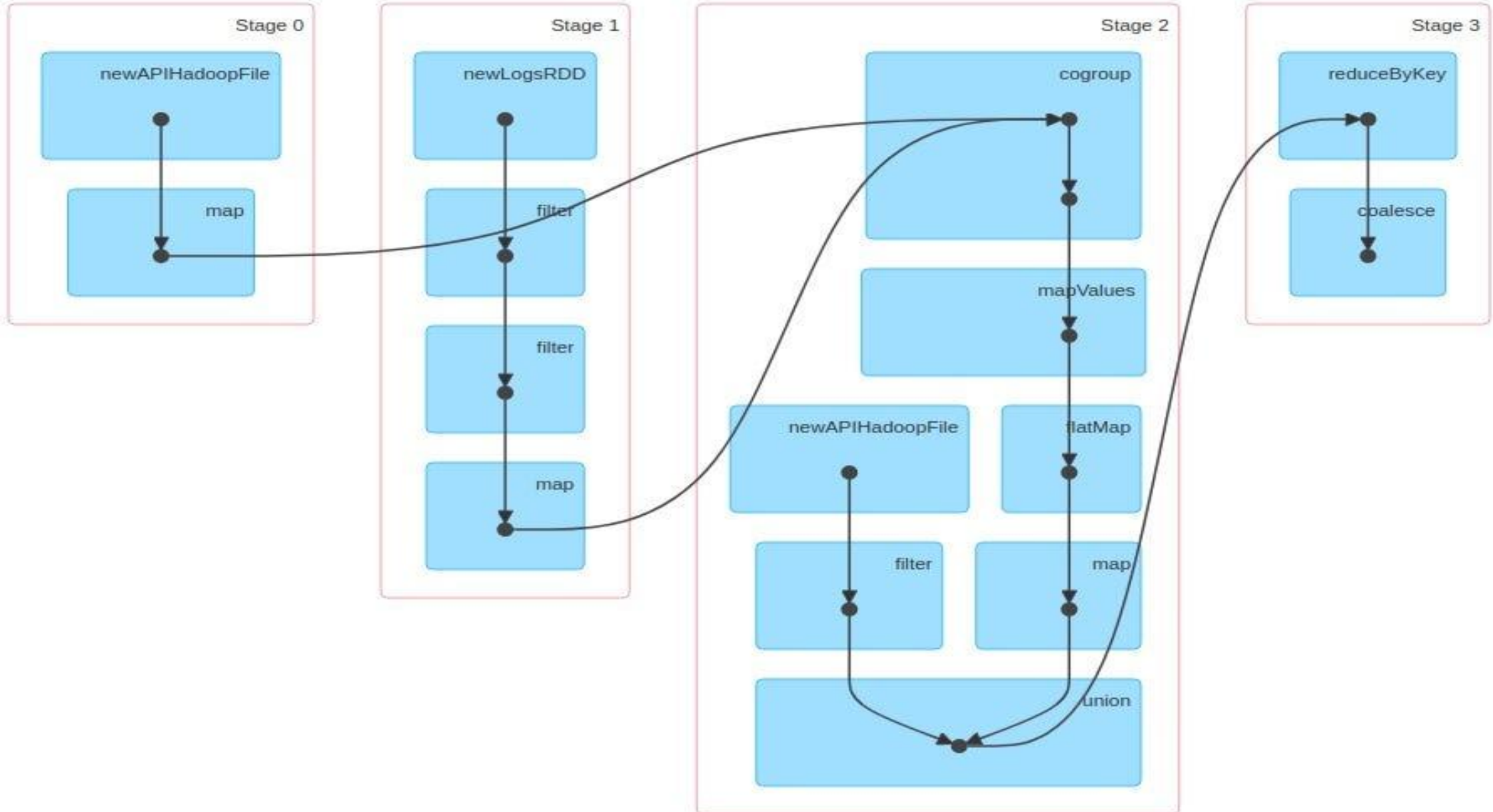
Spark Clusters!

- A group of nodes (master-slave) that work together to execute Spark applications in parallel.
- Components:
  - Driver — coordinates the job
  - Cluster Manager — allocates resources
  - Executors — run the actual computation on worker nodes.

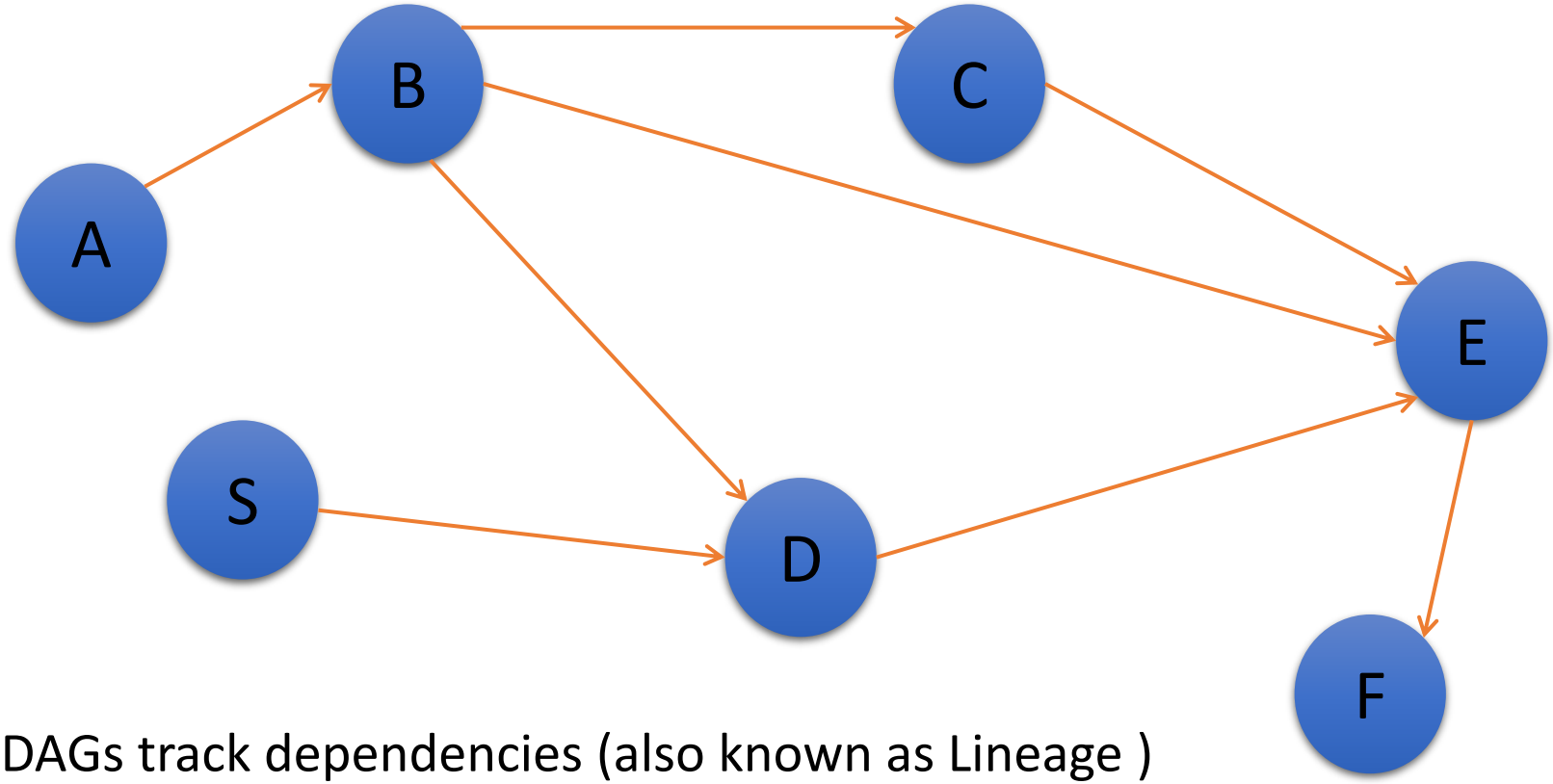
- Tasks: The smallest execution units that run inside executors.
- Job → Stage → Task: Execution Hierarchy - Spark breaks an application into multiple stages and tasks based on transformations.

Spark Clusters!

- Spark builds a logical execution plan (DAG)
- It tracks the sequence of transformations from input to output
- When an action (e.g., count(), collect()) is called, Spark submits the DAG to the DAG Scheduler
- The DAG is divided into stages based on shuffle boundaries
- Each stage is divided into multiple tasks, one per partition


Stage 0
newAPIHadoopFile
map
Stage 1
newLogsRDD
filter
filter
map
Stage 2
cogroup
mapValues
newAPIHadoopFile
flatMap
filter
map
union
Stage 3
reduceByKey
coalesce

Directed Acyclic Graphs (DAG)



DAGs track dependencies (also known as Lineage)

- nodes are RDDs
- arrows are Transformations

Spark Clusters!

- The Task Scheduler sends tasks to executors on worker nodes
- Executors execute tasks in parallel
- They may cache intermediate RDDs in memory for reuse (.cache() / .persist())
- Results of transformations (like map, reduce) are computed locally
- Once all stages finish, results are returned to the:
  - driver (for actions like collect()) or
  - written to storage (for actions like saveAsTextFile()).

A Core Idea – Understand it First

Core Idea: Partition

HDFS File Block	Size	Spark Partition
Block 1	128 MB	Partition 0
Block 2	128 MB	Partition 1
Block 3	128 MB	Partition 2
Block 4	128 MB	Partition 3
Block 5	128 MB	Partition 4

Core Idea: Partition

- Spark creates RDDs/DataFrames/DataSets of input data
- A partition is the smallest logical chunk of data that a RDD or DataFrame is divided into
- Each partition is processed independently and in parallel by one task on an executor
- A partition = a subset of your dataset that Spark processes as one unit of work.

Core Idea: Partition

- Spark is built for parallelism
- If you have a 1 TB dataset, Spark doesn't load it all into one machine
- It splits it into, say, 100 partitions → each partition is processed by a different executor thread or node
- Data is partitioned when you create an RDD or DataFrame:

```
rdd = spark.sparkContext.textFile("data.txt", minPartitions=4)
```

Core Idea: Partition

- Each Partition is processed by 1 Task
- The number of tasks per stage = number of partitions.
- 10 partitions → 10 tasks → processed in parallel.
- More partitions → more parallelism, but also more scheduling overhead.

Core Idea: Shuffle

- Redistribute data across partitions (change the data in partition) so that records with the same key end up in the same partition.
- It happens between stages — when Spark must group, join, or aggregate data that's spread across the cluster.
- Spark "shuffles" data — i.e., it mixes and redistributes it — so that related data lands together for the next computation.
- You can also redistribute partitions across nodes!

Core Idea: Shuffle

- Shuffling is triggered whenever an operation requires data movement between executors — i.e., when the next computation depends on data from multiple partitions.

Transformation	Why it causes shuffle
reduceByKey()	Needs all values for same key in one place
groupByKey()	Groups all values by key
join()	Needs to match keys from two datasets
distinct()	Needs to remove duplicates globally
repartition()	Redistributes data evenly across new partitions
sortByKey()	Needs to order data across partitions

Core Idea: Shuffle

val rdd = sc.parallelize(Seq(10, 20, 30, 40, 50, 60), 3)

Partition 0: 10, 20
Partition 1: 30, 40
Partition 2: 50, 60

val newRDD = rdd.repartition(2)

Partition 0: 20, 40, 60
Partition 1: 10, 30, 50

Node 1: (A,1) (B,2)
Node 2: (A,3) (C,4)

Node 1: (A,1), (A,3) → reduce to (A,4)
Node 2: (B,2), (C,4) → reduce to (B,2), (C,4)

Back to Cluster Architecture

Driver

- Runs the main SparkContext (your SparkSession)
- Defines transformations and actions on RDDs/DataFrames.
- Converts your code → DAG (Directed Acyclic Graph) → execution plan
- Communicates with the Cluster Manager to request resources (executors)
- Schedules tasks on those executors
- Driver = brain of Spark jobs

Cluster Manager

- Launch executor processes on available nodes
- Allocate CPU cores and memory to the Spark application.
- Monitor node health and resource usage.

Cluster Manager	Description
Standalone	Built-in lightweight manager (default for testing/small clusters)
YARN	Hadoop's resource manager — common in enterprise setups
Mesos	General-purpose cluster manager (less common now)
Kubernetes	Container-native deployment for Spark 3.x+

Worker Nodes and Executors

- Each worker node runs one or more executors.
- Executor = JVM process that runs tasks on data partitions.
- Executors:
  - Read data from HDFS/S3/etc.
  - Perform transformations.
  - Store intermediate results in memory.
  - Communicate results back to the driver.

Clustering Steps

- Step 1: Start Spark Application

spark-submit --master yarn --deploy-mode cluster my_app.py

- Step 2: Driver Connects to Cluster Manager
  - The driver connects to Cluster Manager
  - It requests resources (executors + memory)

- Step 3: Cluster Manager Allocates Executors
  - Cluster Manager launches executor JVMs on worker nodes.
  - Each executor registers back with the driver
  - Cluster topology is now formed

Driver ⇄ Cluster Manager ⇄ Executors (across nodes)

Clustering Steps

- Closure = function + variables it needs
- Step 4: Driver Distributes Tasks
  - The driver sends serialized tasks (closures) to executors.
  - Executors execute those tasks on their local data partitions.

- Step 5: Executors Return Results
  - Executors process tasks and return results (or write to disk/cloud).
  - Driver coordinates task retries and stage completion.

```
spark-submit --master local[*] app.py                    # local

spark-submit --master spark://host:7077 app.py           # standalone

spark-submit --master yarn app.py                        # Hadoop

spark-submit --master k8s://<api-server> app.py          # Kubernetes
```

Logical to Physical Flow

Logical Step	Physical Action
Spark creates a DAG	Driver builds execution plan
Stages identified	Cluster manager allocates executors
Tasks assigned	Executors run tasks on worker nodes
Results collected	Driver aggregates output

Fault Tolerance

- If a worker/executor fails → cluster manager reassigns tasks
- If a driver fails → whole job restarts (unless checkpointed)
- RDD/DataFrame lineage allows recomputation of lost partitions

RDD Lineage

- Lineage = the history of transformations used to create an RDD from a source
- Spark does not store intermediate data in memory or disk by default
- If an RDD is lost (executor fails), Spark can recompute it using its lineage
- Lineage forms a DAG of transformations.

```
from pyspark import SparkContext

sc = SparkContext("local[*]", "LineageExample")

# Step 1: Create an initial RDD from a list
rdd1 = sc.parallelize([1, 2, 3, 4, 5])

# Step 2: Apply a map transformation
rdd2 = rdd1.map(lambda x: x * 2)

# Step 3: Apply a filter transformation
rdd3 = rdd2.filter(lambda x: x > 5)
```

```
rdd1: [1,2,3,4,5]       (source)
   |
   v
rdd2 = rdd1.map(x*2)     -> [2,4,6,8,10]
   |
   v
rdd3 = rdd2.filter(x>5)  -> [6,8,10]
```

The Science of Spark – Resilient Distributed Datasets

RDDs

- It's the core data abstraction in Spark (introduced in Spark 1.0)
- Fault-tolerant collection of elements that can be operated on in parallel – can be cached for fast reuse (in-memory)
- Each RDD is:
  - Immutable: once created, it can't be changed (can be transformed)
  - Distributed: partitioned across nodes in a cluster
  - Lazy evaluated: transformations are not executed until an action is called
  - Resilient: automatically recovers from node failures using lineage

RDDs

- RDD divided into partitions
- Each partition is processed by one task
- Spark maintains DAG of all transformations

```
rdd1 = sc.textFile("hdfs://data/logs.txt")
rdd2 = rdd1.filter(lambda line: "ERROR" in line)
rdd3 = rdd2.map(lambda line: (line.split(" ")[0], 1))
rdd4 = rdd3.reduceByKey(lambda a, b: a + b)
```

- Each transformation (filter, map, reduceByKey) creates a new RDD

Creating RDDs

- rdd = sc.textFile("hdfs://namenode:9000/data.txt")
- rdd = sc.parallelize([1, 2, 3, 4, 5])
- rdd2 = rdd.map(lambda x: x * 2)

Type	Description	Examples	Executes Immediately?
Transformations	Create new RDDs from existing ones	map, filter, flatMap, reduceByKey, groupByKey, join	❌ Lazy
Actions	Return results or write to storage	count, collect, take, saveAsTextFile, reduce	✅ Yes

RDD Operations

Type	Description	Examples	Executes Immediately?
Transformations	Create new RDDs from existing ones	map, filter, flatMap, reduceByKey, groupByKey, join	❌ Lazy
Actions	Return results or write to storage	count, collect, take, saveAsTextFile, reduce	✅ Yes

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

rdd2 = rdd.map(lambda x: x * 2)          # Transformation (lazy)

result = rdd2.collect()                  # Action (triggers execution)
```

When `collect()` is called, Spark executes all transformations in a pipeline.

RDD Execution Model

- What happens when an action is called?

1. Spark builds the DAG
2. The DAG is split into stages based on shuffle boundaries
3. Each stage is composed of tasks, one per partition
4. Tasks are distributed to executors for execution
5. Results are collected back to the driver (for actions like collect)

RDD Persistence and Caching

- To speed up iterative computations (like ML), Spark allows caching

```
rdd = sc.textFile("data.txt").map(process)
rdd.cache()  # or rdd.persist(StorageLevel.MEMORY_ONLY)
rdd.count()  # triggers computation
```

- The RDD's partitions are stored in memory (RAM) across executors.

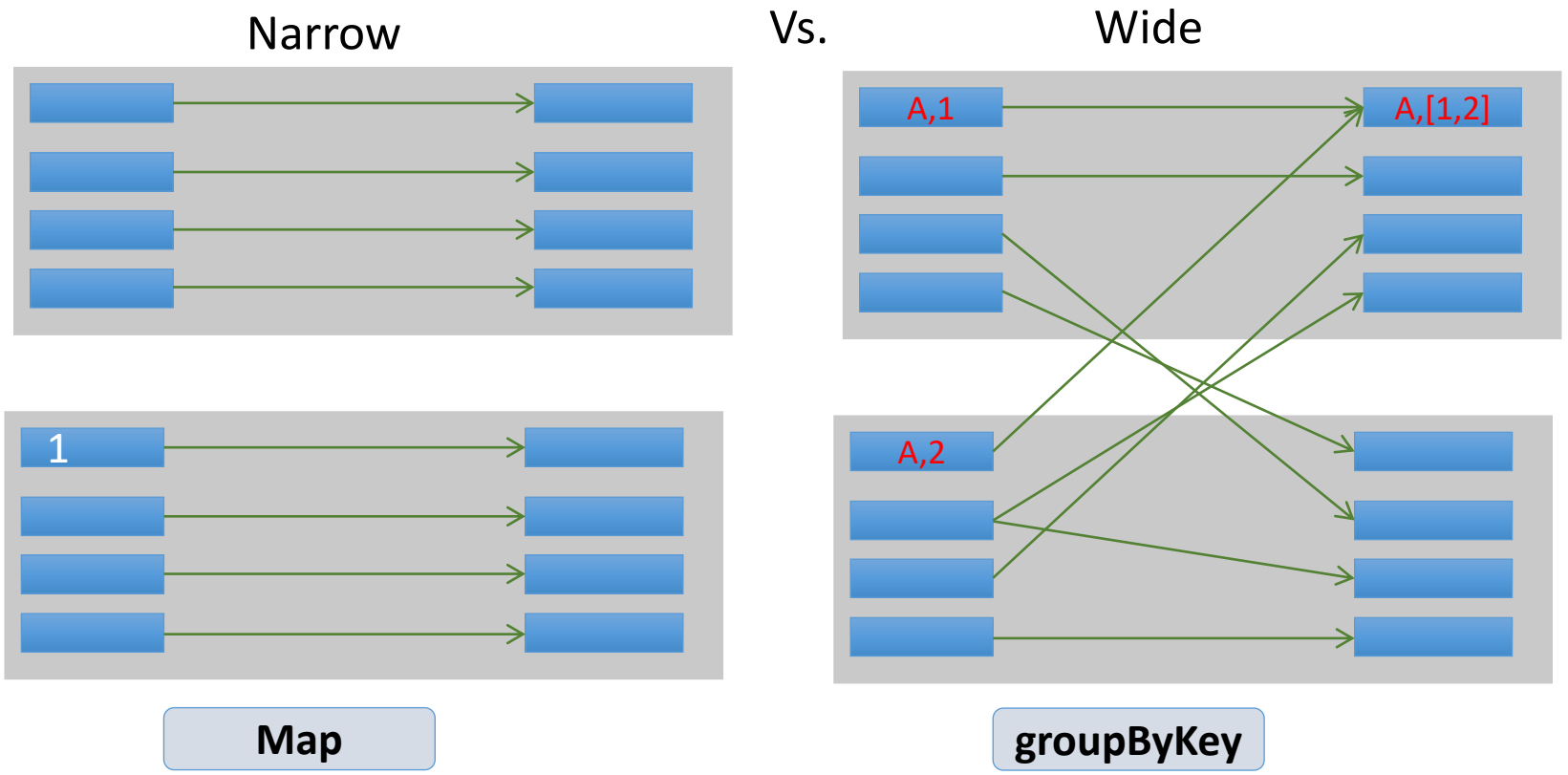
Storage Level	Description
MEMORY_ONLY	Keep data in memory; recompute if lost
MEMORY_AND_DISK	Spill to disk if memory full
DISK_ONLY	Store only on disk
MEMORY_ONLY_SER	Store serialized objects in memory (less space)

Narrow vs Wide Transformation

- A shuffle means data is redistributed between nodes — expensive operation.

Type	Example	Shuffling?	Parallelism Impact
Narrow	map, filter, flatMap	No	Can be done within partition
Wide	groupByKey, reduceByKey, join	Yes	Requires shuffle between executors

Narrow Vs. Wide transformation



Fault Tolerance: Lineage Graph

- If a node fails and a partition is lost, Spark recomputes it using the DAG.
- No need for data replication like Hadoop MapReduce.

Common Transformations

Operation	Description
map(f)	Apply function to each element
flatMap(f)	Flatten nested results
filter(f)	Select elements matching condition
distinct()	Remove duplicates
sample()	Random sampling
union()	Combine two RDDs
intersection()	Common elements
groupByKey()	Group values by key
reduceByKey()	Aggregate values per key
sortByKey()	Sort RDD by key
join()	Join two pair RDDs

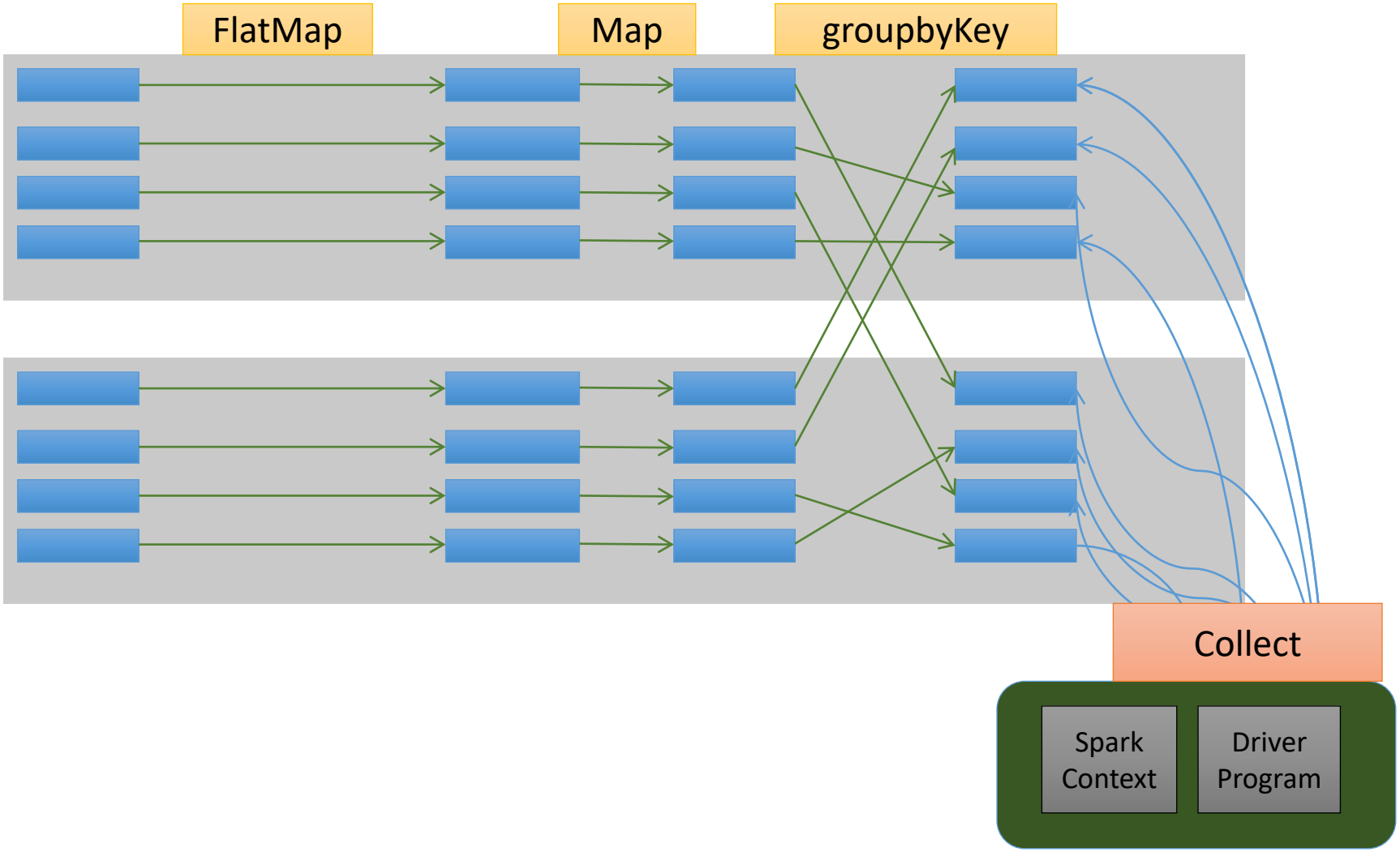
Common Actions

Action	Description
collect()	Bring all data to driver
count()	Count number of elements
first()	Return first element
take(n)	Return first n elements
reduce(f)	Aggregate elements using function
saveAsTextFile(path)	Write RDD to storage
foreach(f)	Run function on each element (for side effects)

Actions

- What is an action
  - The final stage of the workflow
  - Triggers the execution of the DAG
  - Returns the results to the driver
  - Or writes the data to HDFS or to a file

Spark Workflow



Optimizing RDD - 1

- Use persistence (cache() or persist()) for iterative algos - like ML training, graph traversal, PageRank) - same RDD used repeatedly
- Without caching, each iteration recomputes all the previous transformations and reloads data from disk (e.g., HDFS) every time
- Cache: RDD stays in memory: Spark stores RDD partitions in executor memory after the first computation - later actions reuse them directly.

```
rdd = sc.textFile("hdfs://data.txt").map(parse)

for i in range(10):
    rdd = rdd.map(transform)
    print(rdd.count())
```

```
rdd = sc.textFile("hdfs://data.txt").map(parse).cache()

for i in range(10):
    rdd = rdd.map(transform)
    print(rdd.count())
```

Optimizing RDD - 2

- Avoid groupByKey() → use reduceByKey() instead (less shuffle)
- Both groupByKey() and reduceByKey() aggregate data by key, but they behave very differently under the hood.
- For groupByKey(), Spark shuffles **all values** for each key across the network.
- Large data movement between executors
- High memory overhead (stores all values for a key before combining)

```
data = [("a", 1), ("a", 2), ("b", 3)]
rdd = sc.parallelize(data)
```

"a" → [1,2]
"b" → [3]

```
rdd.groupByKey().mapValues(sum).collect()
```

Optimizing RDD - 2

- Using reduceByKey(), Spark performs **local aggregation first** (on each partition) and then shuffles only partial results.

- Use reduceByKey(), aggregateByKey(), or combineByKey() for aggregations.

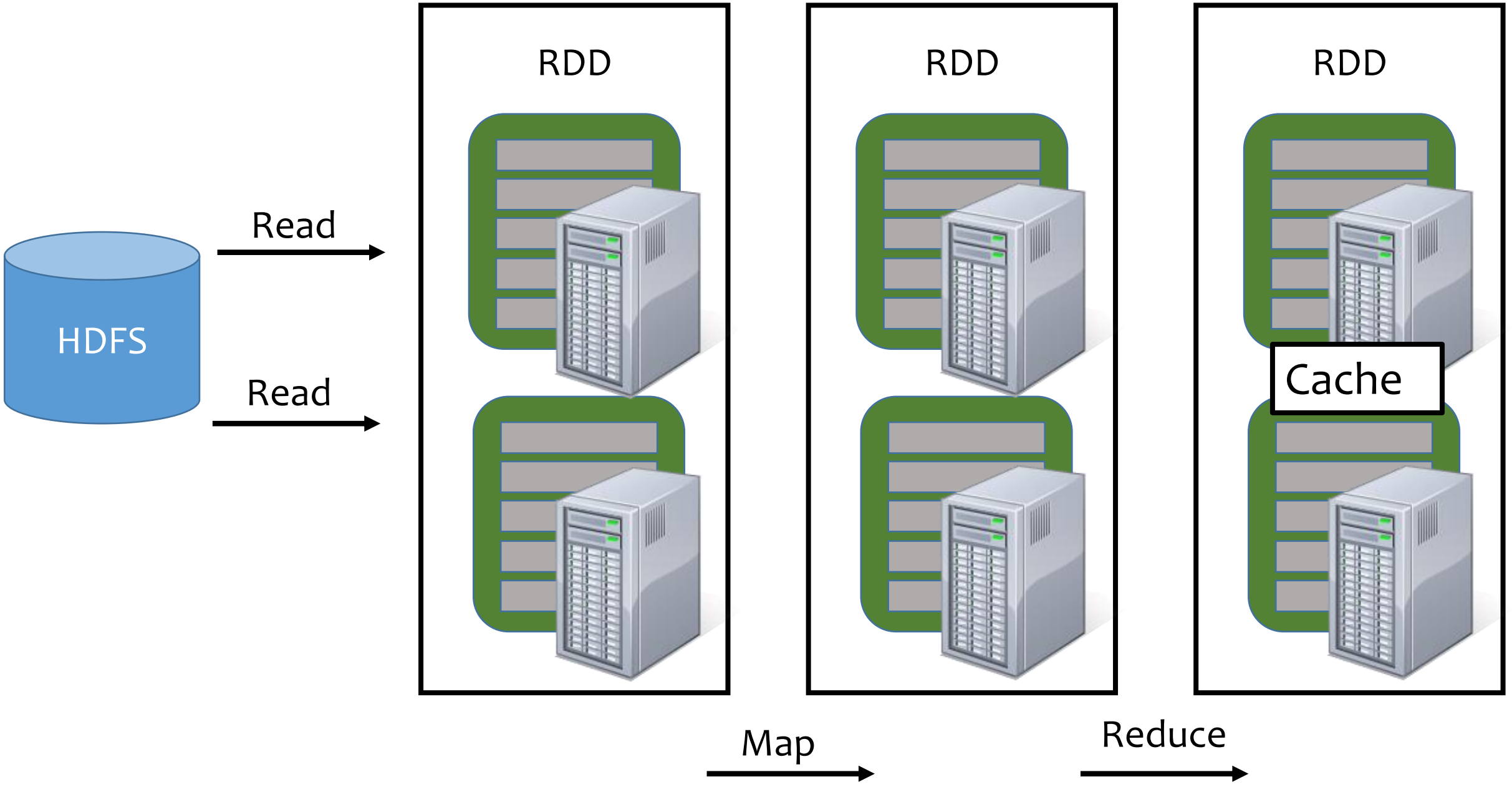
- Avoid groupByKey() unless you actually need all values per key.

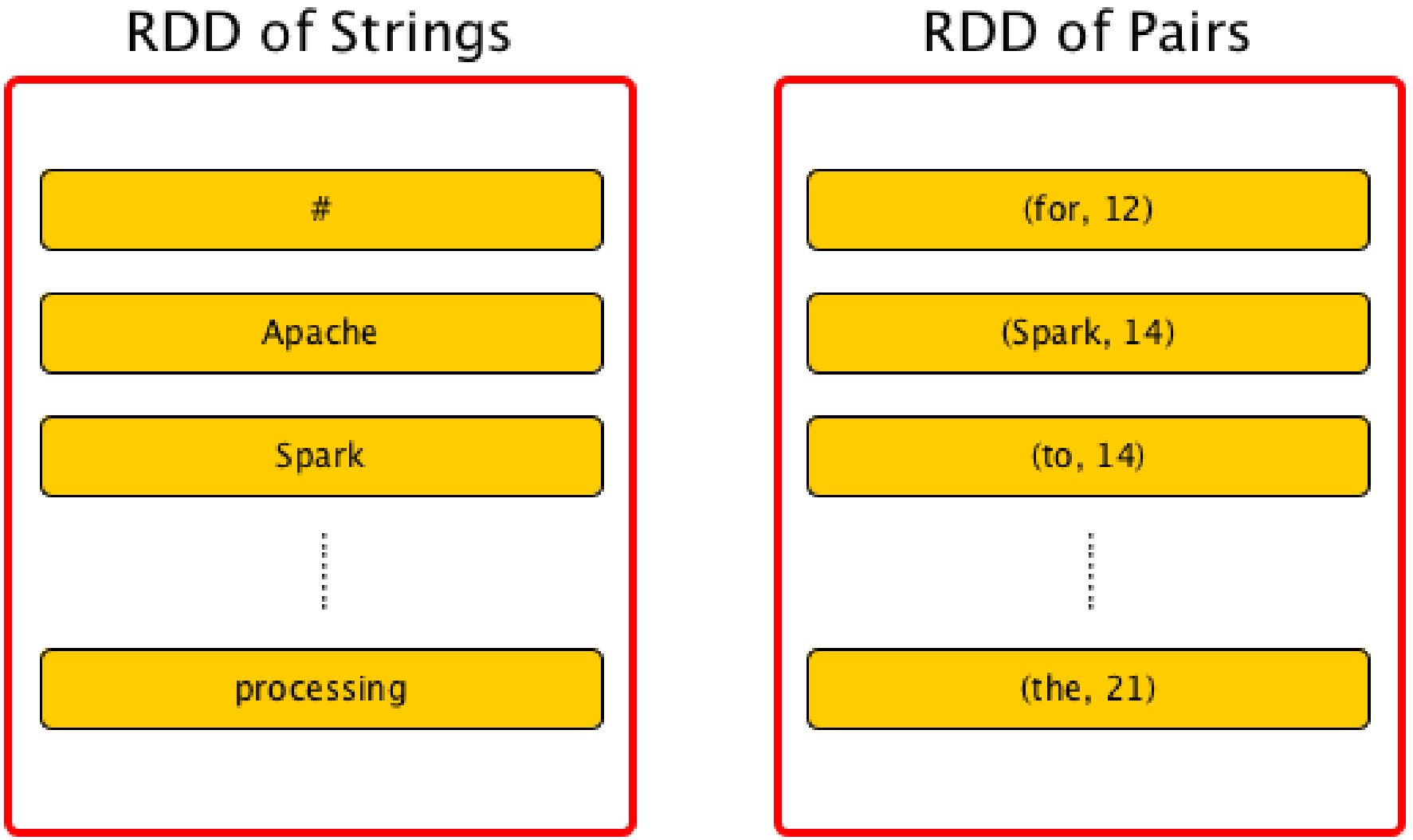
```
rdd.reduceByKey(lambda a, b: a + b).collect()
```

"a" → partial_sum(3)
"b" → partial_sum(3)

Optimization Techniques for RDD

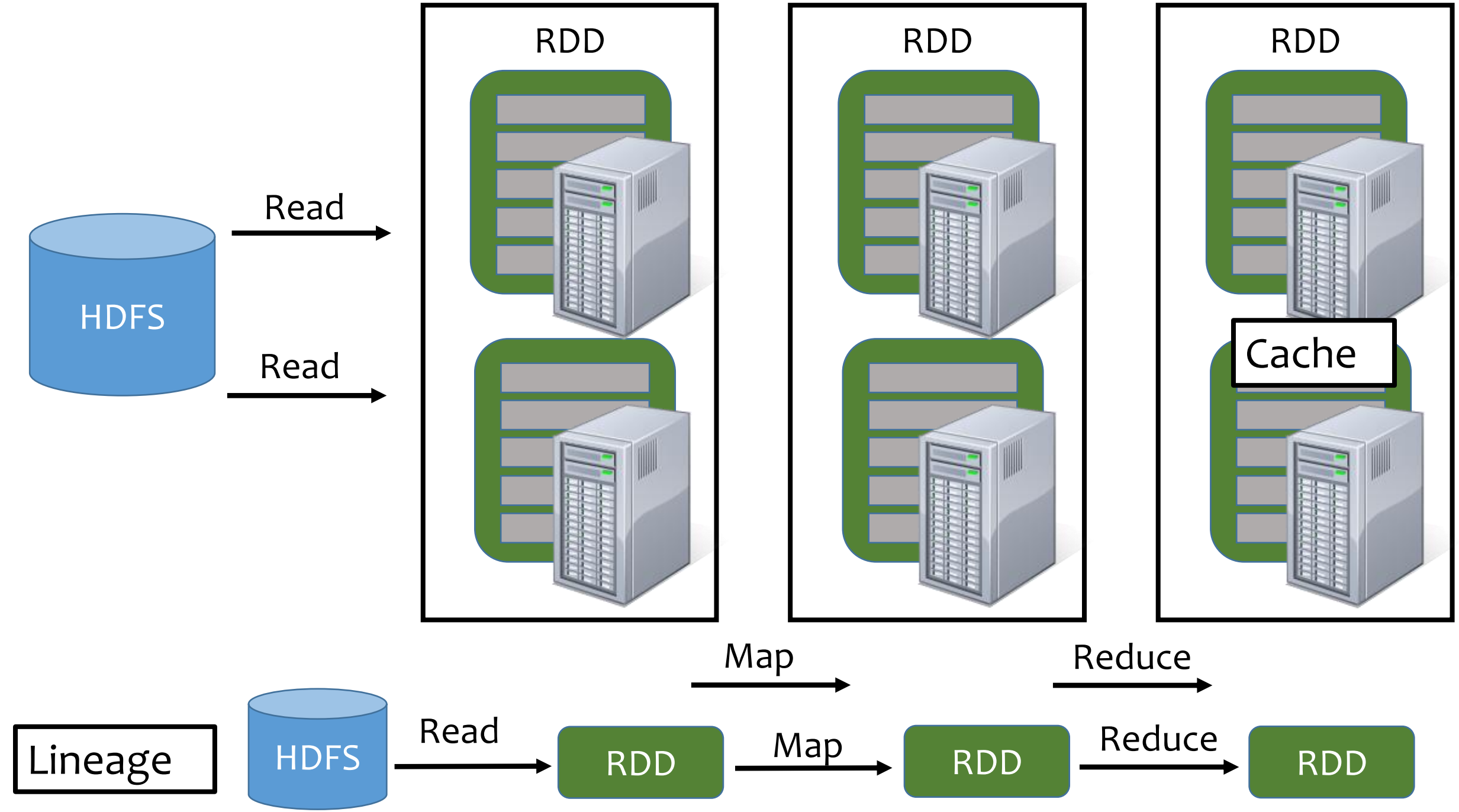
- Repartition intelligently (e.g., repartition(10) or coalesce(4))
- Use broadcast variables for small lookup data
- Avoid large collect() (can crash driver)

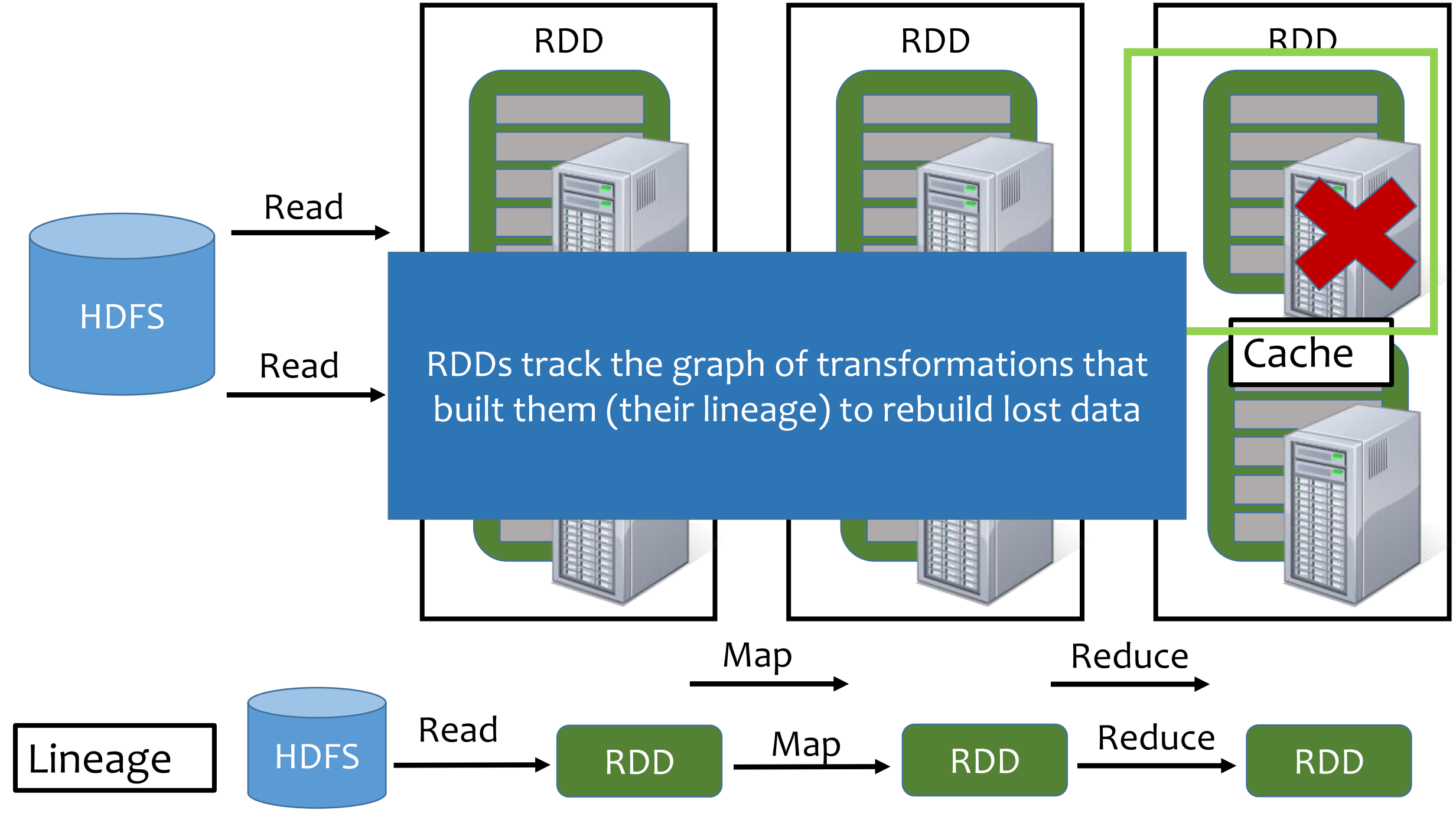

HDFS
Read
Read
RDD
RDD
RDD
Cache
Map
Reduce

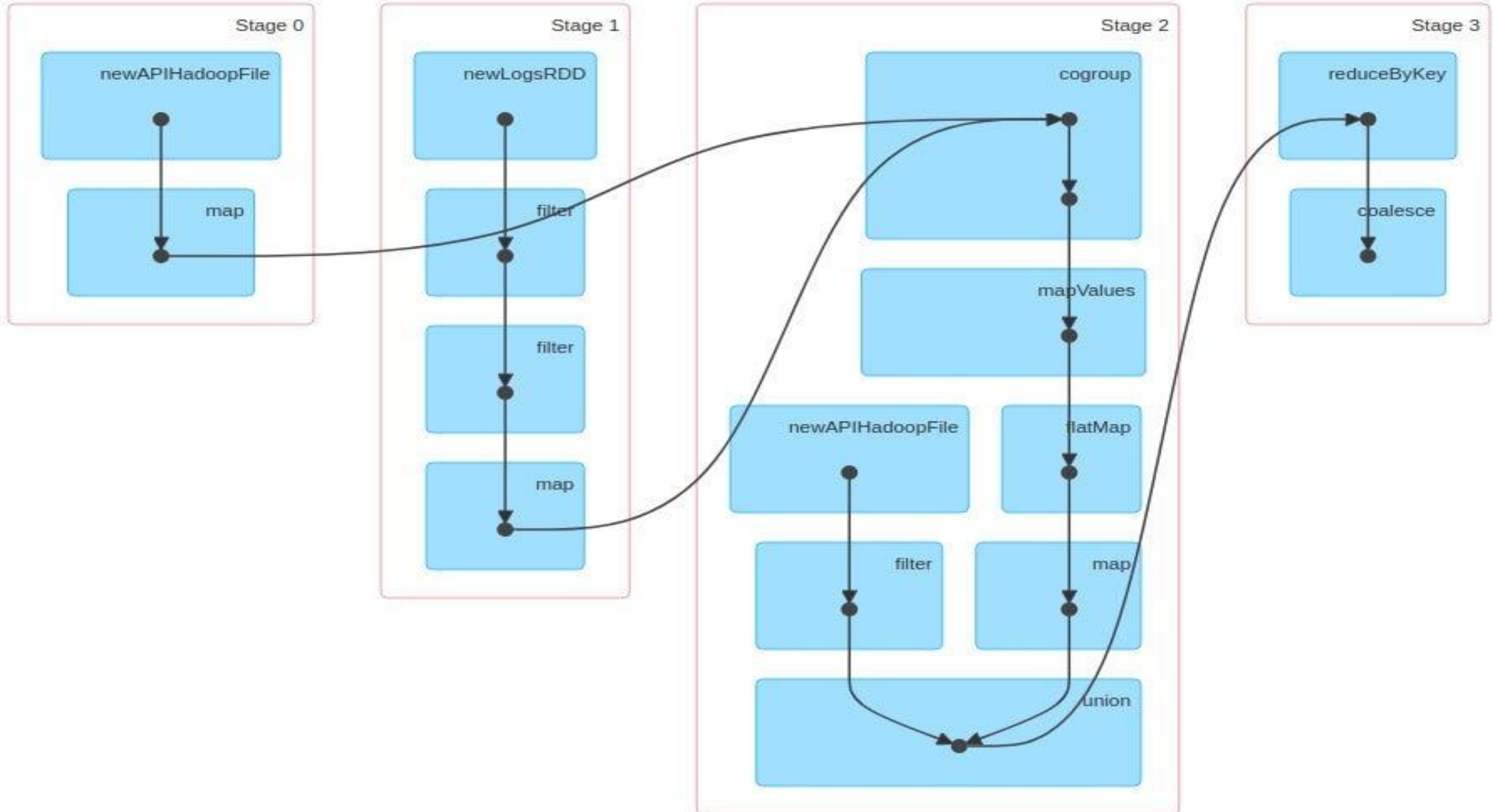

RDD of Strings
#
Apache
Spark
processing
RDD of Pairs
(for, 12)
(Spark, 14)
(to, 14)
(the, 21)

Lineage

- Log the coarse-grained operation (transformation that applies to entire dataset rather than individual records) applied to a partitioned dataset
- Simply re-compute the lost partition if failure occurs!
- No cost if no failure

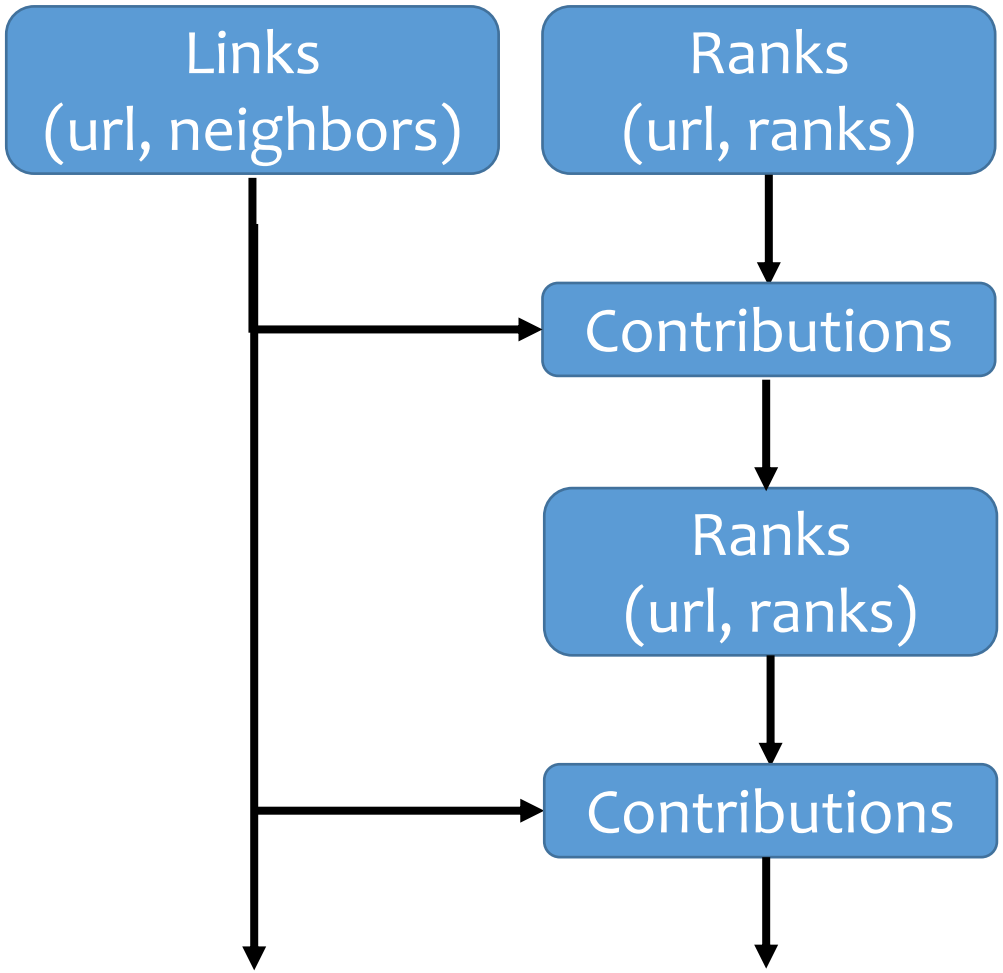

RDD
RDD
RDD
Cache
HDFS
Read
Read
Map
Reduce
Lineage
HDFS
Read
RDD
Map
RDD
Reduce
RDD


RDD
RDD
RDD
Read
HDFS
Read
RDDs track the graph of transformations that built them (their lineage) to rebuild lost data
Cache
Map
Reduce
Lineage
HDFS
Read
RDD
Map
RDD
Reduce
RDD


Stage 0
newAPIHadoopFile
map
Stage 1
newLogsRDD
filter
filter
map
Stage 2
cogroup
mapValues
newAPIHadoopFile
flatMap
filter
map
union
Stage 3
reduceByKey
coalesce

Partitioning

- PageRank

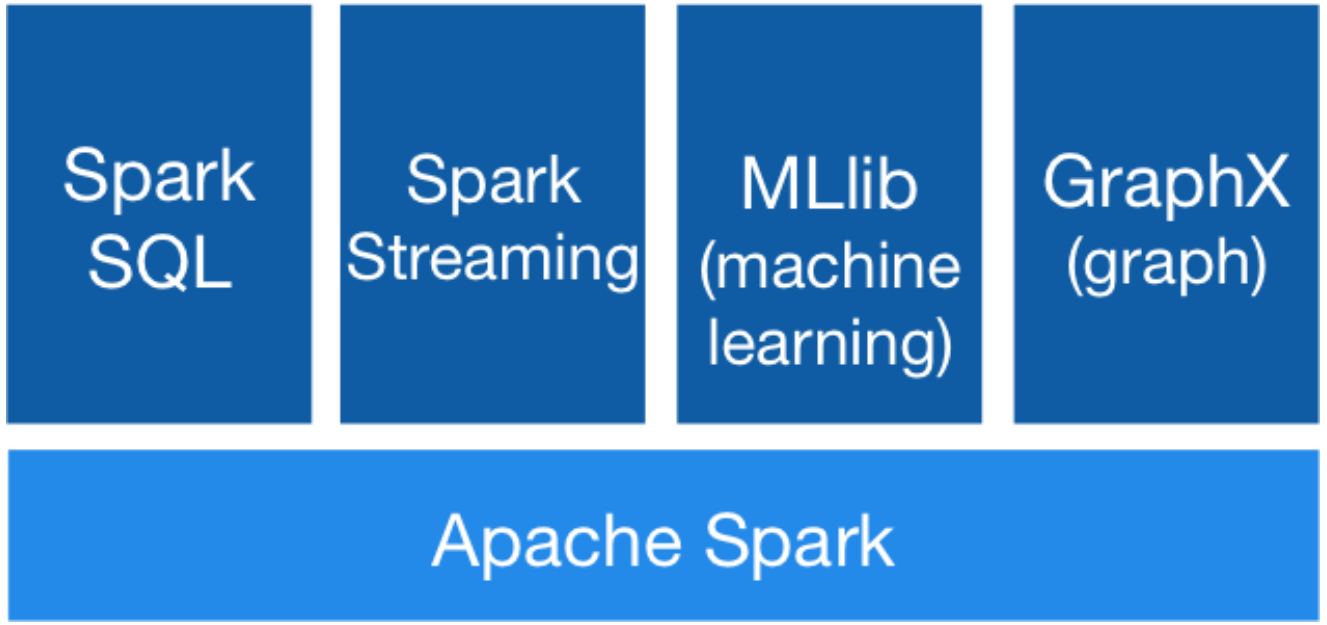


Joins take place repeatedly

Good partitioning reduces shuffles

Generality

- RDDs allow unification of different programming models
  - Stream Processing
  - Graph Processing
  - Machine Learning …..



Spark example #1 (Python)

```
#  Estimate π (compute-intensive task).
# Pick  random points in the unit square ((0, 0) to (1,1)),
# See how many fall in the unit circle. The fraction should be π / 4
# Note that "parallelize" method creates an RDD
def sample(p):
   x, y = random(), random()
   return 1 if x*x + y*y < 1 else 0

count = spark.parallelize(xrange(0, NUM_SAMPLES)).map(sample) \
          .reduce(lambda a, b: a + b)
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

Source: https://spark.apache.org/docs/latest/quick-start.html

Spark example #2 (Scala)

```
// “sc” is a “Spark context” – this transforms the file into an RDD
val textFile = sc.textFile("README.md")
// Return number of items (lines) in this RDD; count() is an action
textFile.count()
// Demo filtering.  Filter is a transform.  By itself this does no real work
val linesWithSpark = textFile.filter(line => line.contains("Spark"))
// Demo chaining – how many lines contain “Spark”?  count() is an action.
textFile.filter(line => line.contains("Spark")).count()
// Length of line with most words.  Reduce is an action.
textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
// Word count – traditional map-reduce.  collect() is an action
val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b)
wordCounts.collect()
```

Source: https://spark.apache.org/docs/latest/quick-start.html

Sample Spark transformations

- map(func): Return a new distributed dataset formed by passing each element of the source through a function func.
- filter(func): Return a new dataset formed by selecting those elements of the source on which func returns true
- union(otherDataset): Return a new dataset that contains the union of the elements in the source dataset and the argument.
- intersection(otherDataset): Return a new RDD that contains the intersection of elements in the source dataset and the argument.
- distinct([numTasks])): Return a new dataset that contains the distinct elements of the source dataset
- join(otherDataset, [numTasks]): When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
- Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

Source: https://spark.apache.org/docs/latest/programming-guide.html

Sample Spark Actions

- reduce(func): Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- collect(): Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- count(): Return the number of elements in the dataset.

Remember: *Actions* cause calculations to be performed; *transformations* just set things up (lazy evaluation)

Source: https://spark.apache.org/docs/latest/programming-guide.html

Summary of Common Coarse-Grained Transformations

Transformation	Description
`map`	Applies a function to each element.
`flatMap`	Produces multiple elements per input.
`filter`	Selects elements based on a condition.
`distinct`	Removes duplicates.
`union`	Combines two RDDs.
`intersection`	Returns common elements between RDDs.
`cartesian`	Produces cross-product of two RDDs.
`groupBy`	Groups elements by a function.
`reduceByKey`	Aggregates by key.
`join`	Inner join on keys.
`sortBy`	Sorts elements by a function.

Fine-Grained Transformation: operates on each individual element of an RDD - Each operation works element-by-element, often producing a new RDD with modified content or structure.

Most fine-grained operations are narrow (e.g., map, filter), meaning no data shuffling occurs across partitions.

`map`	Applies a function to each element.	`sc.parallelize([1, 2, 3]).map(lambda x: x * 2).collect()`	`[2, 4, 6]`
`filter`	Selects elements satisfying a condition.	`sc.parallelize([1, 2, 3]).filter(lambda x: x % 2 == 0).collect()`	`[2]`
`flatMap`	Maps elements, allowing multiple outputs per input.	`sc.parallelize(["a b", "c"]).flatMap(lambda x: x.split()).collect()`	`['a', 'b', 'c']`
`distinct`	Removes duplicate elements.	`sc.parallelize([1, 2, 2, 3]).distinct().collect()`	`[1, 2, 3]`
`mapPartitions`	Applies a function to each partition of RDD.	`sc.parallelize([1, 2, 3]).mapPartitions(lambda x: [sum(x)]).collect()`	`[6]`
`union`	Combines two RDDs, allowing duplicates.	`sc.parallelize([1, 2]).union(sc.parallelize([2, 3])).collect()`	`[1, 2, 2, 3]`
`intersection`	Returns common elements from two RDDs.	`sc.parallelize([1, 2, 3]).intersection(sc.parallelize([2, 3, 4])).collect()`	`[2, 3]`
`sample`	Randomly selects elements from the RDD.	`sc.parallelize([1, 2, 3, 4]).sample(withReplacement=False, fraction=0.5).collect()`	Varies (e.g., `[2, 4]`)
`cartesian`	Computes the Cartesian product of two RDDs.	`sc.parallelize([1, 2]).cartesian(sc.parallelize(['a', 'b'])).collect()`	`[(1, 'a'), (1, 'b'), (2, 'a'), ...]`
`zip`	Zips two RDDs together element-wise.	`sc.parallelize([1, 2]).zip(sc.parallelize(['a', 'b'])).collect()`	`[(1, 'a'), (2, 'b')]`

Spark – RDD Persistence

- Memory_Only:
  - Stores the RDD or DataFrame in memory
  - If the data does not fit into memory, the excess data is recomputed when needed.
- Use Case:
  - Suitable for datasets where recomputation is cheaper than storing on disk.
  - Best for iterative algorithms where data fits into memory.

Spark – RDD Persistence

- Memory_And_Disk:
  - Tries to store the data in memory.
  - If the data doesn’t fit, the remaining data is spilled to disk.
  - When accessing spilled data, it is read back from disk.
- Use Case:
  - Default storage level.
  - Useful for larger datasets that do not fit entirely into memory.

Spark – RDD Persistence

- Disk_Only:
  - Stores the RDD or DataFrame only on disk.
  - Data is read back into memory when needed.
- Use Case:
  - Suitable for very large datasets that cannot fit into memory
  - Useful when recomputation is expensive.

Spark Example #3 (Python)

```
# Logistic Regression - iterative machine learning algorithm
# Find best hyperplane that separates two sets of points in a
# multi-dimensional feature space.  Applies MapReduce operation
# repeatedly to the same dataset, so it benefits greatly
# from caching the input in RAM

points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
   gradient = points.map(
      lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
   ).reduce(lambda a, b: a + b)
   w -= gradient
print "Final separating plane: %s" % w
```

Source: https://spark.apache.org/docs/latest/quick-start.html

Spark Example #3 (Scala)

```
// Same thing in Scala
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}
println("Final separating plane: " + w)
```

Source: https://spark.apache.org/docs/latest/quick-start.html

Spark Example #3 (Java)

```
// Same thing in Java
class ComputeGradient extends Function<DataPoint, Vector> {
  private Vector w;
  ComputeGradient(Vector w) { this.w = w; }
  public Vector call(DataPoint p) {
    return p.x.times(p.y * (1 / (1 + Math.exp(w.dot(p.x))) - 1));
  }
}
JavaRDD<DataPoint> points = spark.textFile(...).map(new ParsePoint()).cache();
Vector w = Vector.random(D); // current separating plane
for (int i = 0; i < ITERATIONS; i++) {
  Vector gradient = points.map(new ComputeGradient(w)).reduce(new AddVectors());
  w = w.subtract(gradient);
}
System.out.println("Final separating plane: " + w);
```

Source: https://spark.apache.org/docs/latest/quick-start.html

Integrating Spark with Hadoop

Hadoop primarily used for…

- Storage (HDFS)
- Resource management (YARN)
- Metadata and SQL interoperability (Hive / HCatalog)
- Data formats (Parquet, ORC, Avro)
- Security (Kerberos, Hadoop ACLs)

- And Spark can leverage Hadoop – it can be built with Hadoop in 4 major ways.

- Spark was introduced by Apache Software Foundation for speeding up the Hadoop computational computing software process.
- As against a common belief, Spark is not a modified version of Hadoop and is not, really, dependent on Hadoop because it has its own cluster management. Hadoop is just one of the ways to implement Spark.
- Spark uses Hadoop in two ways – one is storage and second is processing. Since Spark has its own cluster management computation, it uses Hadoop for storage purpose only.

1. Spark on Hadoop HDFS (Storage Layer Only)

- Spark as compute engine
- Hadoop HDFS as the storage layer
- Spark reads and writes data **from/to HDFS** using Hadoop InputFormats and OutputFormats
- Spark doesn’t need the full Hadoop stack — only the **HDFS client libraries**
- rdd = sc.textFile("hdfs://namenode:9000/data/input.txt")
- Spark uses Hadoop APIs to connect to HDFS:

1. Spark on Hadoop HDFS (Storage Layer Only)

- Fast parallel data access.
- Compatible with Hadoop's file formats (Parquet, ORC, Avro, SequenceFiles).
- Works even if YARN is not present.
- But… you must manage cluster resources manually (e.g., standalone cluster or Kubernetes).

2. Spark on Hadoop YARN (Cluster Manager)

- Spark uses YARN (Hadoop's resource manager)
- Optionally uses HDFS for storage
- Spark jobs are submitted through YARN (spark-submit --master yarn)
- YARN allocates containers for Driver (ApplicationMaster) and Executors
- YARN manages CPU, memory, and node usage dynamically.

```
spark-submit \
  --master yarn \
  --deploy-mode cluster \
  app.py
```

2. Spark on Hadoop YARN (Cluster Manager)

- Full resource management & fault tolerance from YARN
- Coexists easily with Hadoop MapReduce, Hive, and Hbase
- Ideal for multi-tenant Hadoop clusters.
- But:
  - Slightly more configuration overhead (needs Hadoop setup)
  - Slightly more startup latency compared to standalone

3. Spark with Hive / HCatalog (SQL and Metadata Layer)

- Spark SQL engine
- Hive Metastore + HCatalog (metadata)
- Optionally YARN + HDFS underneath
- Spark uses Hive Metastore for table schemas
- Allows Spark SQL to read/write Hive-managed tables
- Supports ORC/Parquet formats and ACID tables

```
spark = SparkSession.builder \
    .enableHiveSupport() \
    .getOrCreate()
spark.sql("SELECT * FROM sales WHERE amount > 1000").show()
```

3. Spark with Hive / HCatalog (SQL and Metadata Layer)

- Seamless interoperability between Spark and Hive
- Shared catalog: Hive tables visible in Spark SQL
- Spark can run HiveQL queries directly
- But:
  - Hive metastore setup required.
  - Performance depends on Hive warehouse configuration.

4. Spark with the Entire Hadoop Stack

- HDFS (storage)
- YARN (cluster management)
- Hive/HCatalog (metadata and SQL)
- HBase (NoSQL integration)
- Hadoop Security (Kerberos)
- Spark runs *inside* the Hadoop ecosystem — using all of its services.

4. Spark with the Entire Hadoop Stack

```
spark-submit \
  --master yarn \
  --deploy-mode cluster \
  --conf spark.hadoop.yarn.resourcemanager.address=yarn-master:8032
\
  --conf spark.sql.catalogImplementation=hive \
  my_spark_job.py
```

DataFrames and DataSets

A tale of multiple APIs

- Spark now has three sets of APIs—RDDs, DataFrames, and Datasets
  - RDD – In Spark 1.0 release, "lower level"
  - DataFrames – Introduced in Spark 1.3 release
  - Dataset – Introduced in Spark 1.6 release
- Each with pros/cons/limitations

DataFrame & Dataset

- DataFrame:
  - Data organized into named columns, e.g. a table
  - Imposes a structure onto a distributed collection of data, allowing higher-level abstraction
- Dataset:
  - Extension of DataFrame API which provides type-safe, object-oriented programming interface (compile-time error detection)

Both built on Spark SQL engine & use Catalyst to generate optimized logical and physical query plan; both can be converted to an RDD

https://data-flair.training/blogs/apache-spark-rdd-vs-dataframe-vs-dataset/
https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

API distinction: typing

- Python & R don't have compile-time type safety checks, so *only* support DataFrame
  - Error detection only at runtime
- Java & Scala support compile-time type safety checks, so support *both* DataSet and DataFrame
  - Dataset APIs are all expressed as lambda functions and JVM typed objects
  - any mismatch of typed-parameters will be detected at compile time.

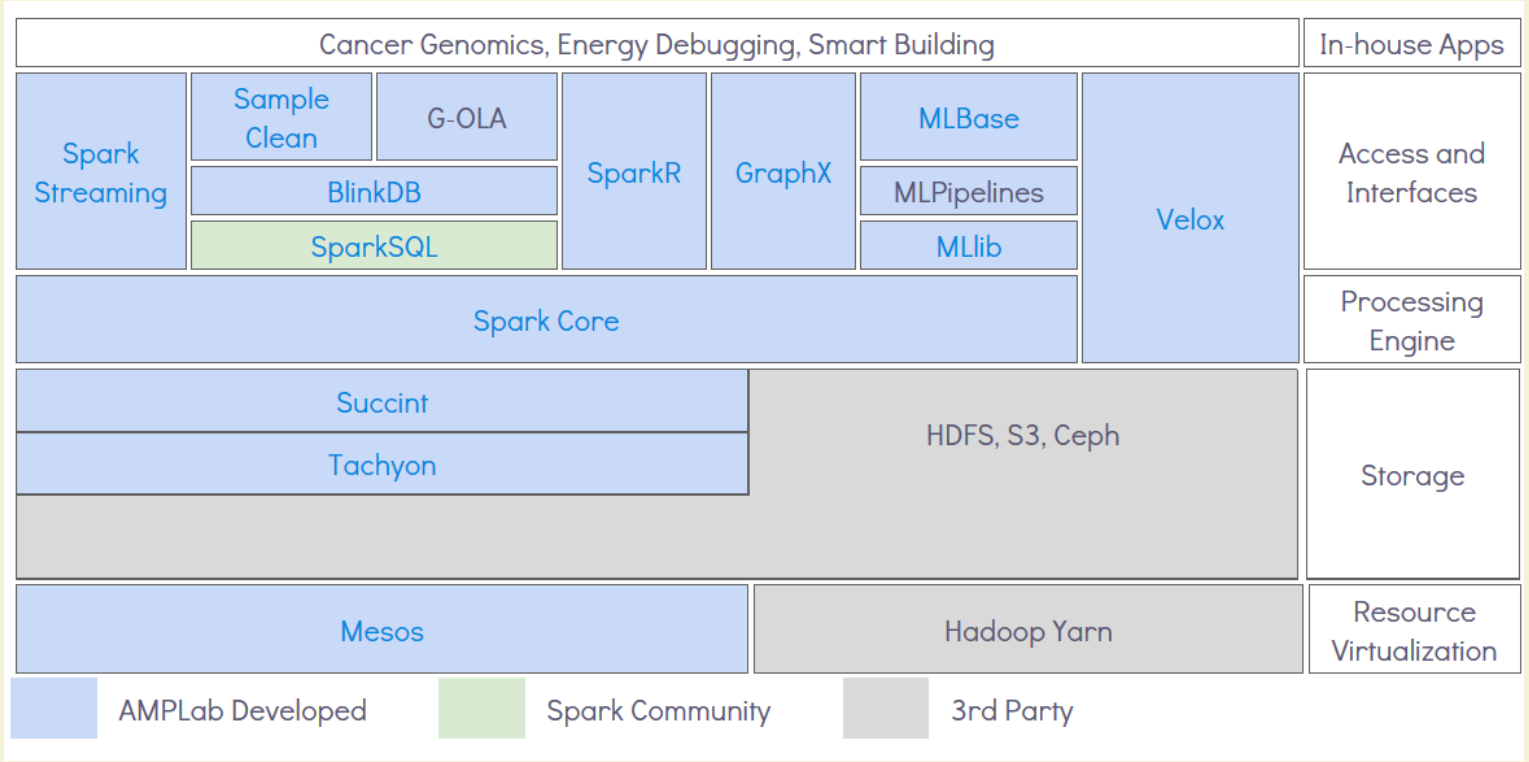
https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Spark vs. Hadoop MapReduce

- Performance: Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it often needs lots of memory to do well; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- Ease of use: Spark is easier to program
- Data processing: Spark more general
- Maturity: Spark maturing, Hadoop MapReduce mature

"Spark vs. Hadoop MapReduce" by Saggi Neumann (November 24, 2014)
https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/

AMPLab's Berkeley Data Analytics Stack (BDAS)



Don't need to memorize this figure – the point is to know that components can be combined to solve big data problems

Source: https://amplab.cs.berkeley.edu/software/

DataFrames & SparkSQL

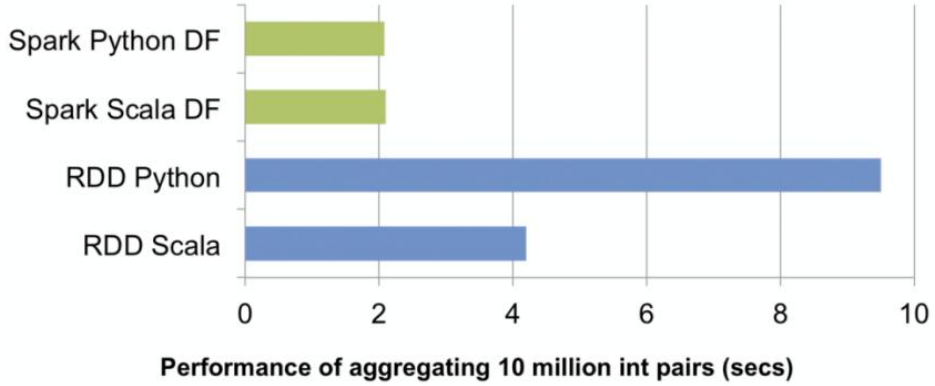
- DataFrames (DFs) is one of the other distributed datasets organized in named columns
- Similar to a relational database, Python Pandas Dataframe or R's DataTables
  - Immutable once constructed
  - Track lineage
  - Enable distributed computations
- How to construct Dataframes
  - Read from file(s)
  - Transforming an existing DFs(Spark or Pandas)
  - Parallelizing a python collection list
  - Apply transformations and actions

DataFrame example

```
// Create a new DataFrame that contains “students”
students = users.filter(users.age < 21)

//Alternatively, using Pandas-like syntax
students = users[users.age < 21]

//Count the number of students users by gender
students.groupBy("gender").count()

// Join young students with another DataFrame called logs
students.join(logs, logs.userId == users.userId,
“left_outer")
```

RDDs vs. DataFrames

- RDDs provide a low level interface into Spark
- DataFrames have a schema
- DataFrames are cached and optimized by Spark
- DataFrames are built on top of the RDDs and the core Spark API



Example: performance

Python RDD API Examples

- Word count

```
text_file = sc.textFile("hdfs://usr/godil/text/book.txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
          .map(lambda word: (word, 1)) \
         .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://usr/godil/output/wordCount.txt")
```

- Logistic Regression

```
# Every record of this DataFrame contains the label and
# features represented by a vector.
df = sqlContext.createDataFrame(data, ["label", "features"])
# Set parameters for the algorithm.
# Here, we limit the number of iterations to 10.
lr = LogisticRegression(maxIter=10)
# Fit the model to the data.
model = lr.fit(df)
# Given a dataset, predict each point's label, and show the results.
model.transform(df).show()
```

Examples from http://spark.apache.org/

RDD Persistence and Removal

- RDD Persistence
  - RDD.persist()
  - Storage level:
    - MEMORY_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER, DISK_ONLY,…….

- RDD Removal
  - RDD.unpersist()

Spark’s Main Use Cases

- Streaming Data
- Machine Learning
- Interactive Analysis
- Data Warehousing
- Batch Processing
- Exploratory Data Analysis
- Graph Data Analysis
- Spatial (GIS) Data Analysis
- And many more

Spark Use Cases

- Fingerprint Matching
  - Developed a Spark based fingerprint minutia detection and fingerprint matching code
- Twitter Sentiment Analysis
  - Developed a Spark based Sentiment Analysis code for a Twitter dataset

Spark in the Real World (I)

- Uber – the online taxi company gathers terabytes of event data from its mobile users every day.
  - By using Kafka, Spark Streaming, and HDFS, to build a continuous ETL pipeline
  - Convert raw unstructured event data into structured data as it is collected
  - Uses it further for more complex analytics and optimization of operations

- Pinterest – Uses a Spark ETL pipeline
  - Leverages Spark Streaming to gain immediate insight into how users all over the world are engaging with Pins—in real time.
  - Can make more relevant recommendations as people navigate the site
  - Recommends related Pins
  - Determine which products to buy, or destinations to visit

Spark in the Real World (II)

Here are Few other Real World Use Cases:

- Conviva – 4 million video feeds per month
  - This streaming video company is second only to YouTube.
  - Uses Spark to reduce customer churn by optimizing video streams and managing live video traffic
  - Maintains a consistently smooth, high quality viewing experience.

- Capital One – is using Spark and data science algorithms to understand customers in a better way.
  - Developing next generation of financial products and services
  - Find attributes and patterns of increased probability for fraud

- Netflix – leveraging Spark for insights of user viewing habits and then recommends movies to them.
  - User data is also used for content creation

Spark: when not to use

- Even though Spark is versatile, that doesn't mean Spark's in-memory capabilities are the best fit for all use cases:
  - For many simple use cases Apache MapReduce and Hive might be a more appropriate choice
  - Spark was not designed as a multi-user environment
  - Spark users are required to know that memory they have is sufficient for a dataset
  - Adding more users adds complications, since the users will have to coordinate memory usage to run code

Spark In Memory Benefits

Feature	Spark (In-Memory)	Hadoop MapReduce
Intermediate Data Storage	RAM (optional disk fallback)	Disk (HDFS)
Iterative Processing	Fast, keeps data in memory	Slow, disk I/O per iteration
Execution Model	DAG, lazy evaluation	Linear Map → Shuffle → Reduce, eager execution
Caching	cache() / persist()	None
Interactive Queries	Supported (Spark SQL)	Limited (Hive/Tez needed)
Performance	High for iterative and real-time	High latency due to disk writes