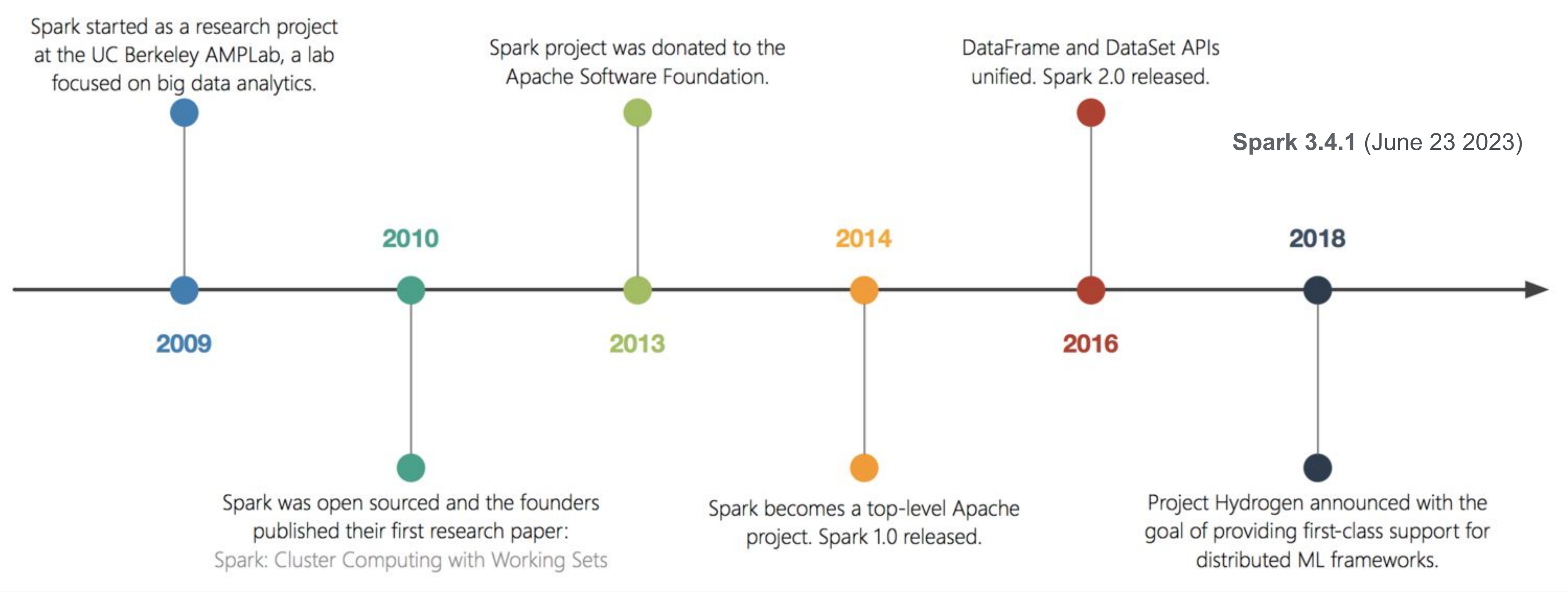
SPARK

Apache Spark is an open-source, distributed processing system used for big data workloads



2009

Spark started as a research project at the UC Berkeley AMPLab, a lab focused on big data analytics.

2010

Spark was open sourced and the founders published their first research paper: Spark: Cluster Computing with Working Sets

2013

Spark project was donated to the Apache Software Foundation.

2014

Spark becomes a top-level Apache project. Spark 1.0 released.

2016

DataFrame and DataSet APIs unified. Spark 2.0 released.

2018

Project Hydrogen announced with the goal of providing first-class support for distributed ML frameworks.

Spark 3.4.1 (June 23 2023)

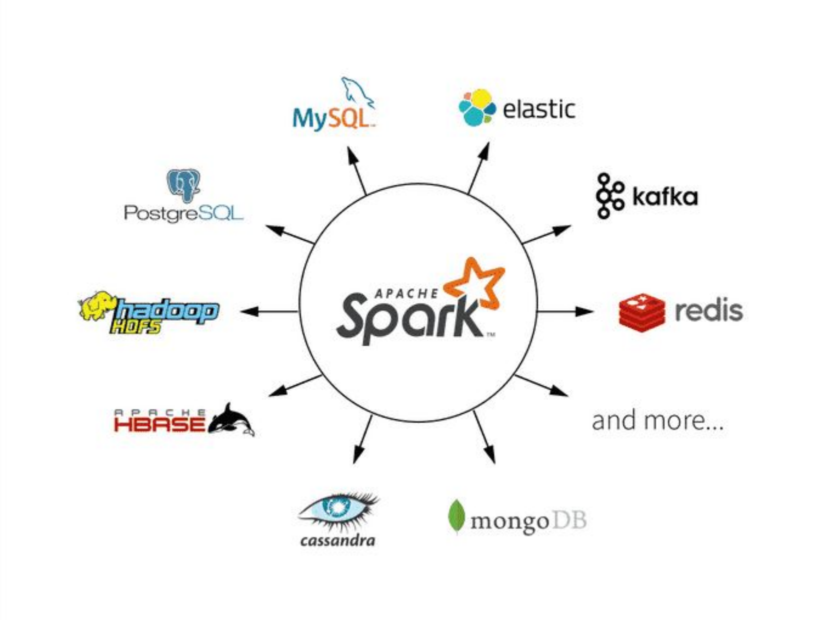
Apache Mesos is an open source cluster manager that handles workloads in a distributed environment through dynamic resource sharing and isolation.

The Amp Lab is a unique learning environment built for students of diverse backgrounds to gain real-world experience in an innovative and agile setting – creating cluster manager

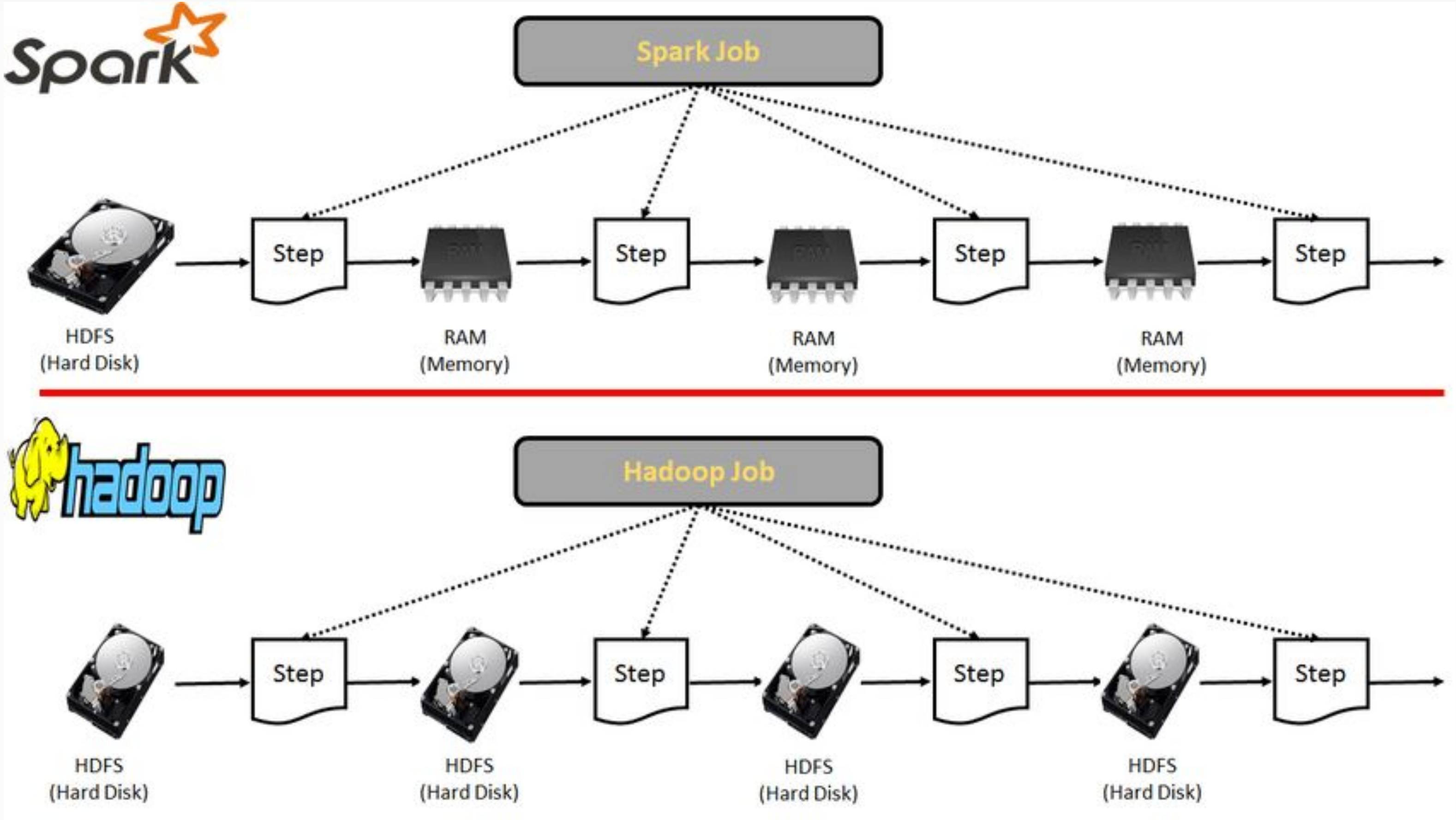
Mesos	YARN
Data-center-wide resource manager	Motivated by Mesos, but is a Hadoop resource manager
Manages all resources well except Hadoop	Manages Hadoop resources well
Can do multi-level scheduling	Pluggable schedulers for Hadoop
Excellent Docker support	Cannot handle long-lived tasks
Takes into account cpu, mem, disk, network, etc,	Can pre-empt tasks

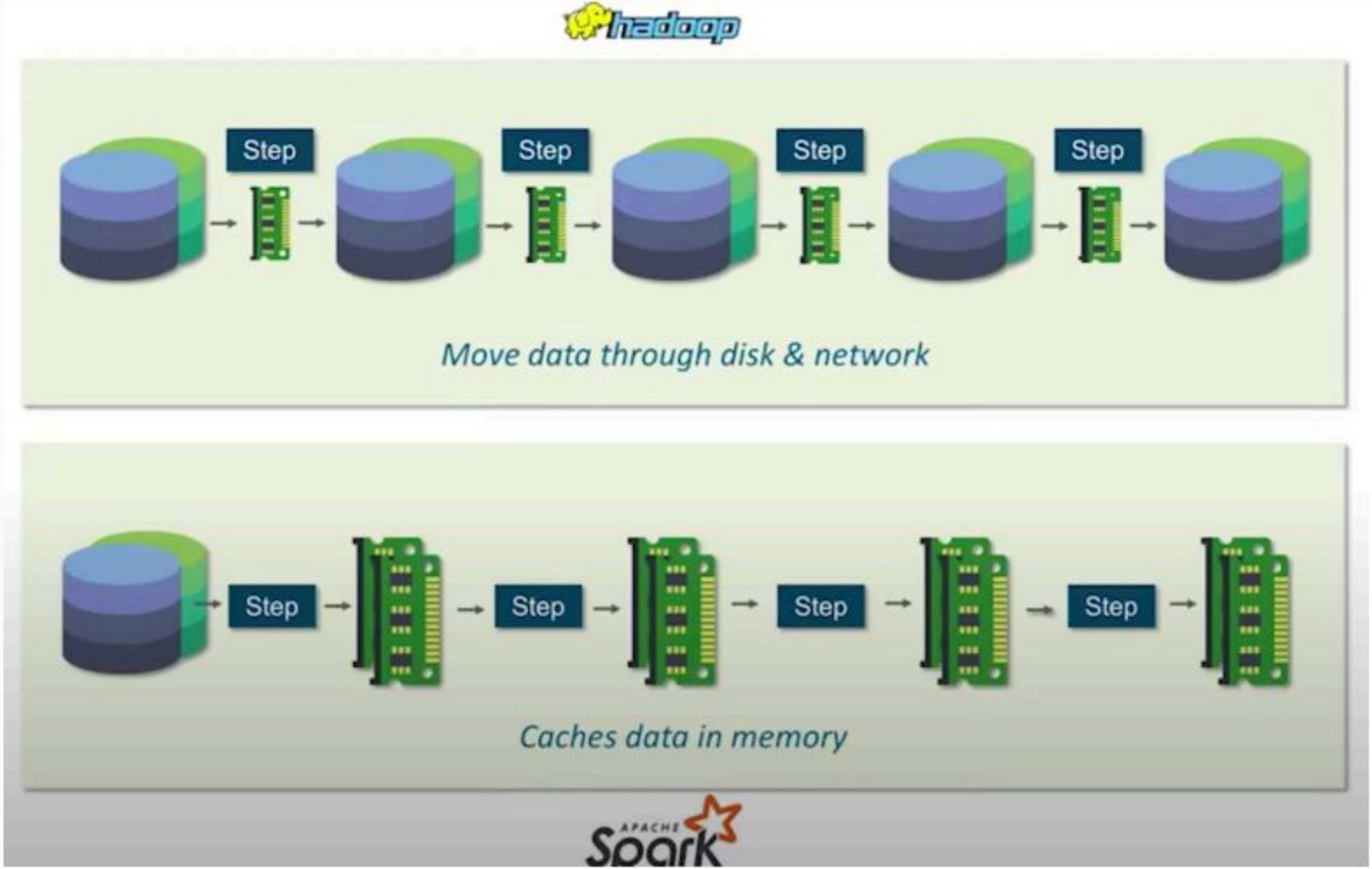
What is Apache Spark

Apache Spark is an open-source, distributed processing system used for big data workloads. It utilizes in-memory caching and optimized query execution for fast queries against data of any size. Simply put, Spark is a fast and general engine for large-scale data processing.



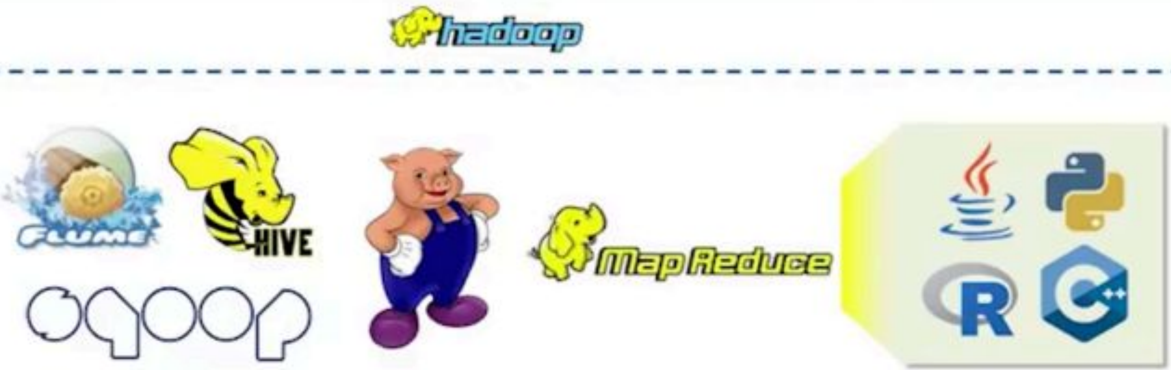
Difference Between Hadoop and Spark





Move data through disk & network

Caches data in memory



Hadoop can be integrated with multiple tools like Sqoop, Flume, Pig, Hive

Spark comes with user-friendly APIs for Scala, Java, Python, and Spark SQL

APACHE Spark

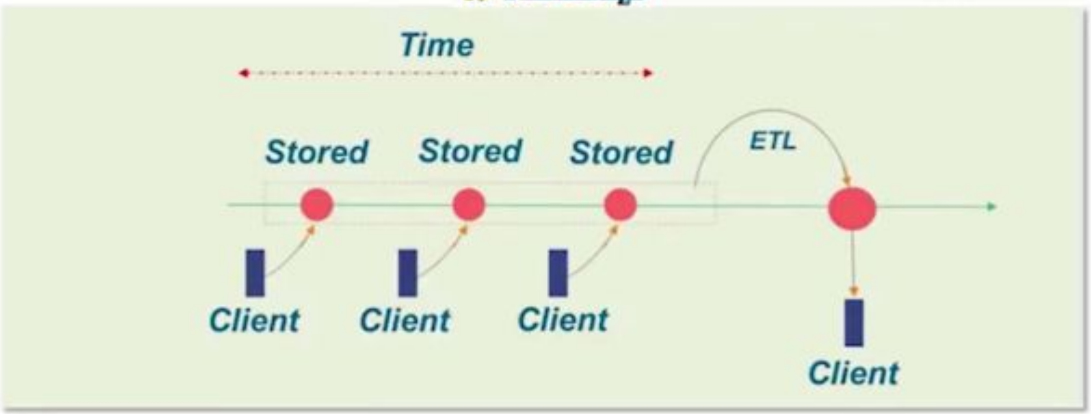
hadoop

Hadoop requires lot of disk space as well as faster disks

Spark requires large amounts of RAM for executing everything in memory

APACHE Spark

hadoop



Time

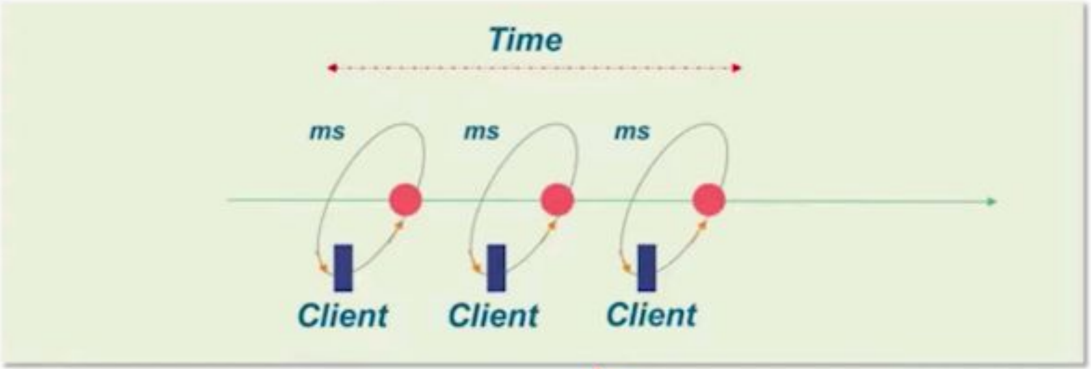
Stored Stored Stored

ETL

Client Client Client

Client

Batch Processing



Time

ms ms ms

Client Client Client

Stream Processing

APACHE Spark

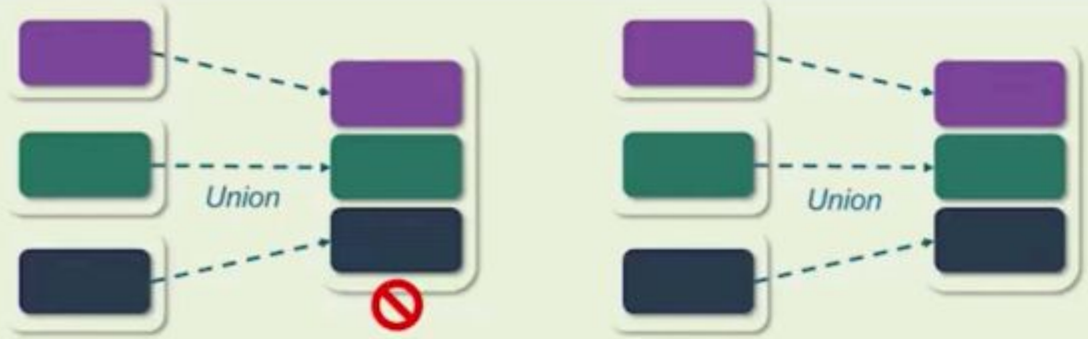
1

Replication

2

Re-Execution of Job

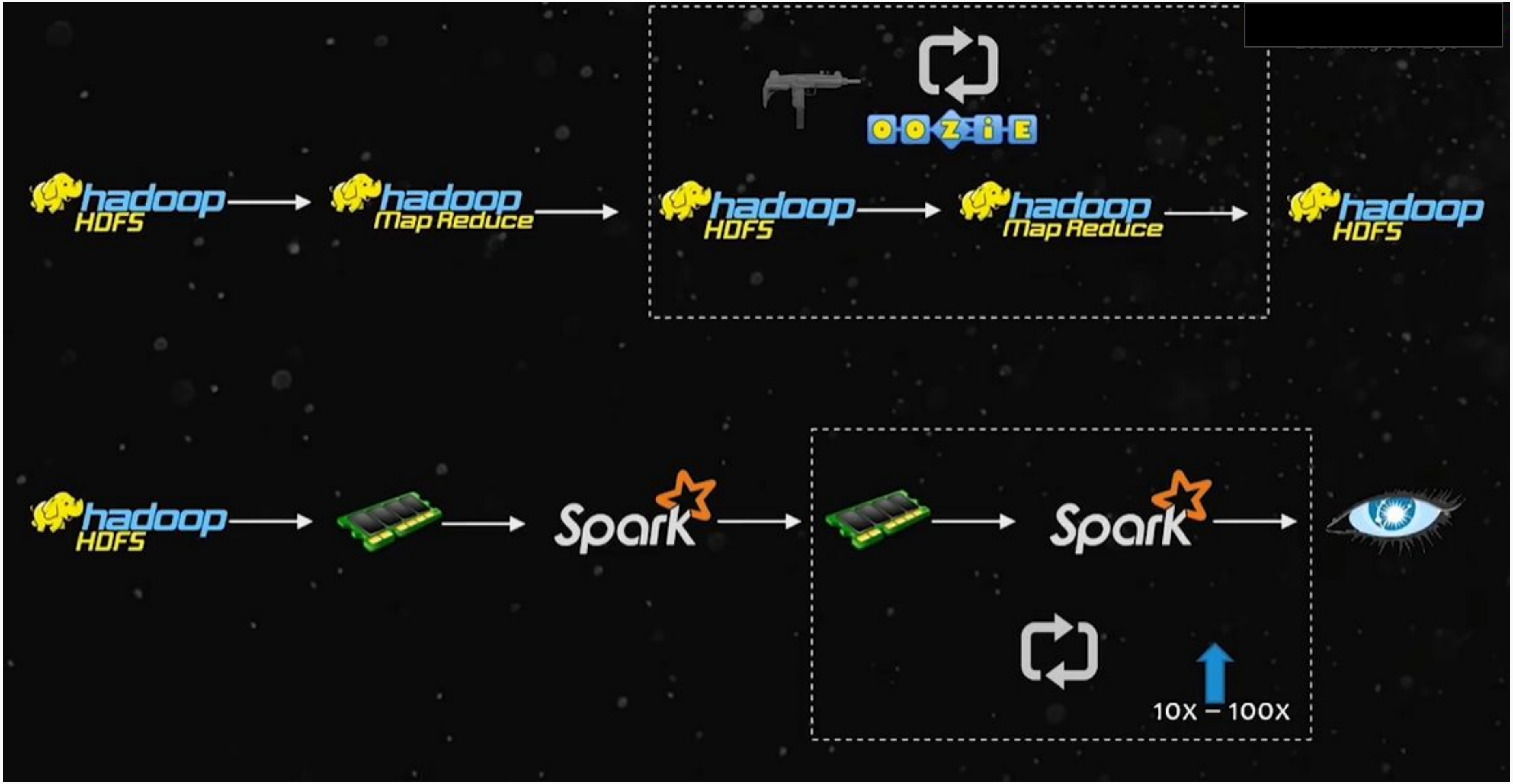
hadoop



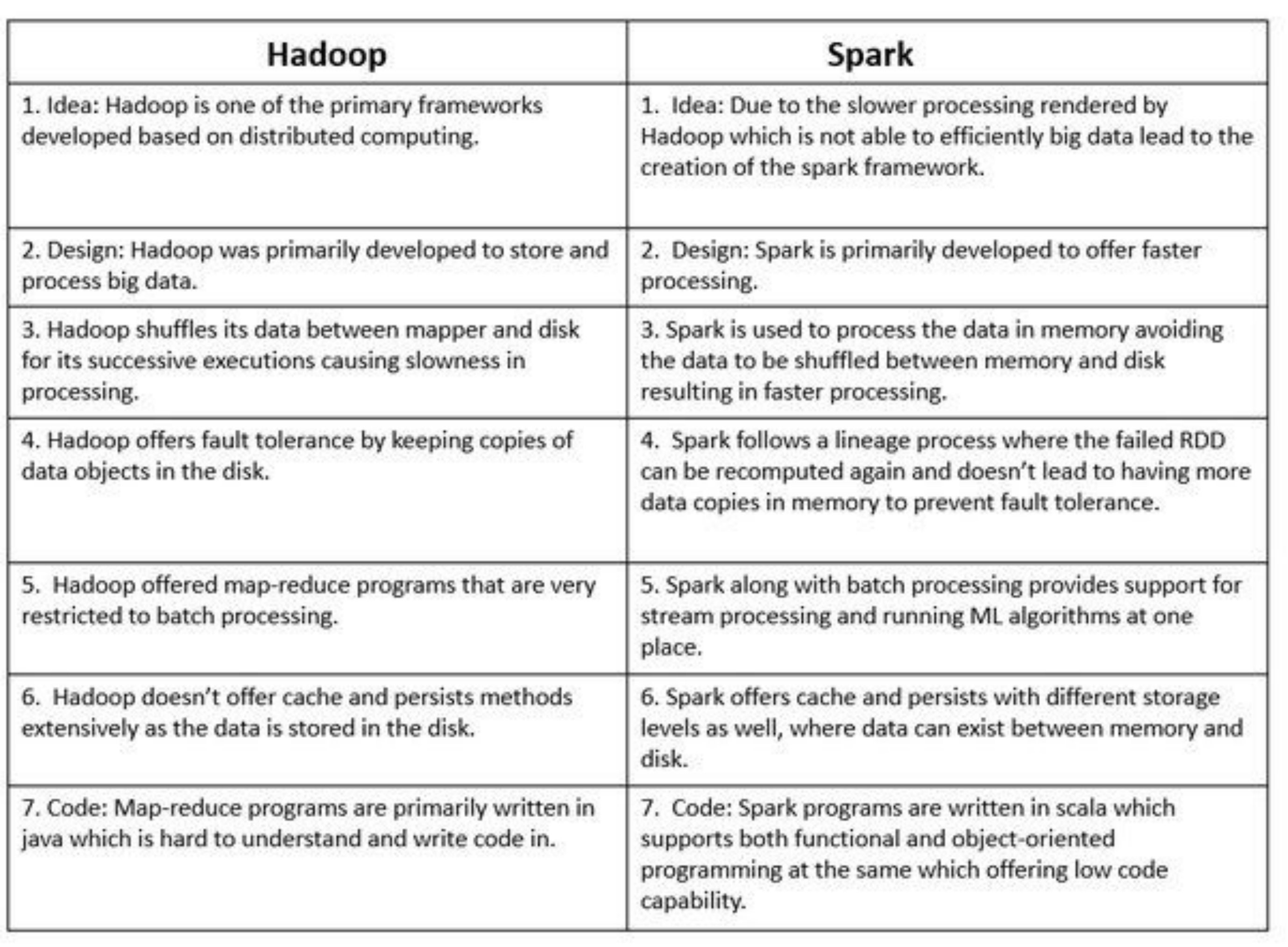
RDD is automatically recomputed by using the original transformations

APACHE Spark

[redacted]



10X – 100X



Hadoop	Spark
1. Idea: Hadoop is one of the primary frameworks developed based on distributed computing.	1. Idea: Due to the slower processing rendered by Hadoop which is not able to efficiently big data lead to the creation of the spark framework.
2. Design: Hadoop was primarily developed to store and process big data.	2. Design: Spark is primarily developed to offer faster processing.
3. Hadoop shuffles its data between mapper and disk for its successive executions causing slowness in processing.	3. Spark is used to process the data in memory avoiding the data to be shuffled between memory and disk resulting in faster processing.
4. Hadoop offers fault tolerance by keeping copies of data objects in the disk.	4. Spark follows a lineage process where the failed RDD can be recomputed again and doesn't lead to having more data copies in memory to prevent fault tolerance.
5. Hadoop offered map-reduce programs that are very restricted to batch processing.	5. Spark along with batch processing provides support for stream processing and running ML algorithms at one place.
6. Hadoop doesn't offer cache and persists methods extensively as the data is stored in the disk.	6. Spark offers cache and persists with different storage levels as well, where data can exist between memory and disk.
7. Code: Map-reduce programs are primarily written in java which is hard to understand and write code in.	7. Code: Spark programs are written in scala which supports both functional and object-oriented programming at the same which offering low code capability.

Spark Features

Fast processing

In-memory computing

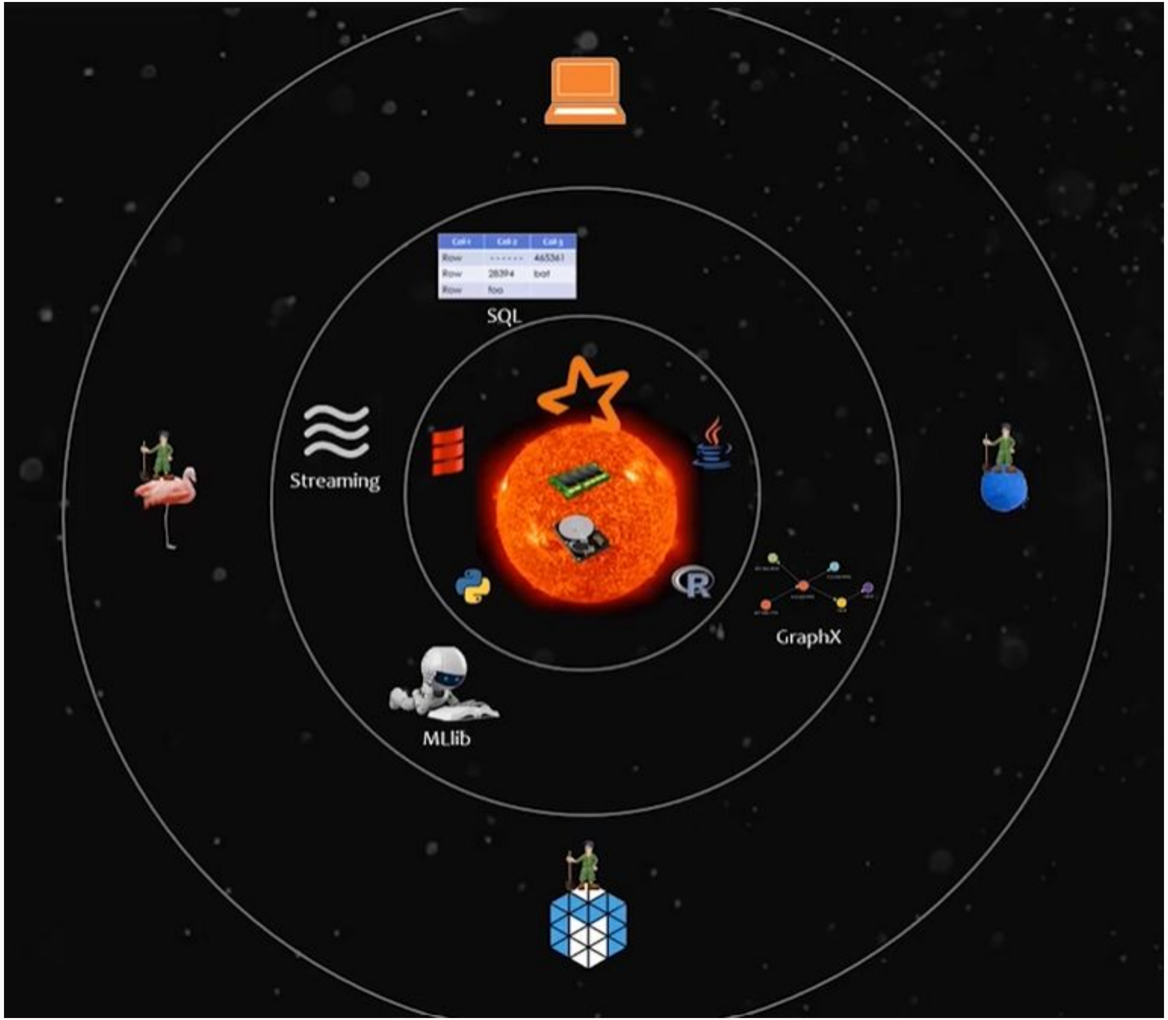
Flexible

Fault tolerance

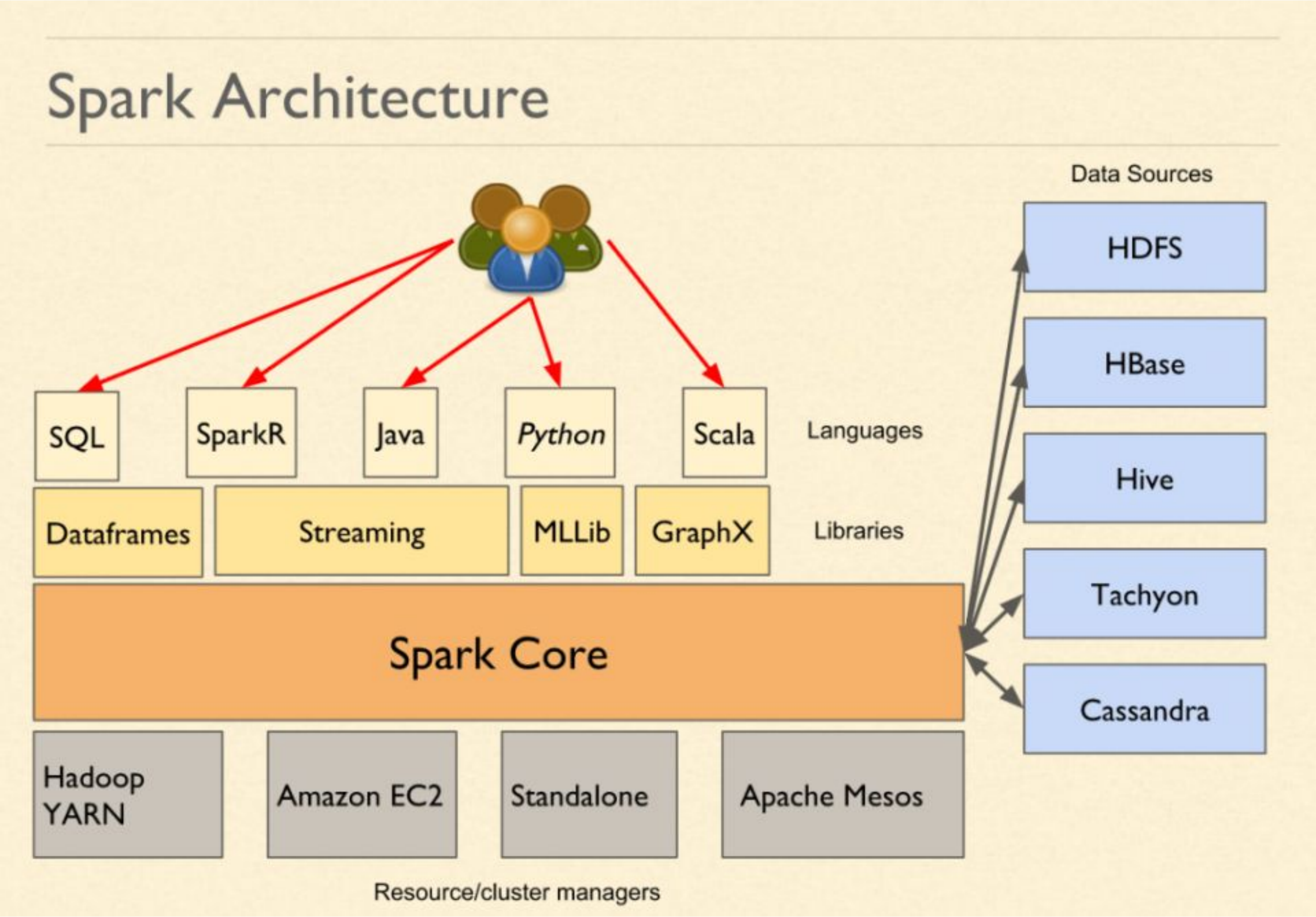
Better analytics

Spark has a rich set of SQL queries, machine learning algorithms, complex analytics, etc. With all these functionalities, analytics can be performed better

Components of Spark



Col 1	Col 2	Col 3
Row	-------	465361
Row	28394	bat
Row	foo	



Spark Architecture

Data Sources

HDFS

HBase

Hive

Tachyon

Cassandra

SQL

SparkR

Java

Python

Scala

Languages

Dataframes

Streaming

MLLib

GraphX

Libraries

Spark Core

Hadoop YARN

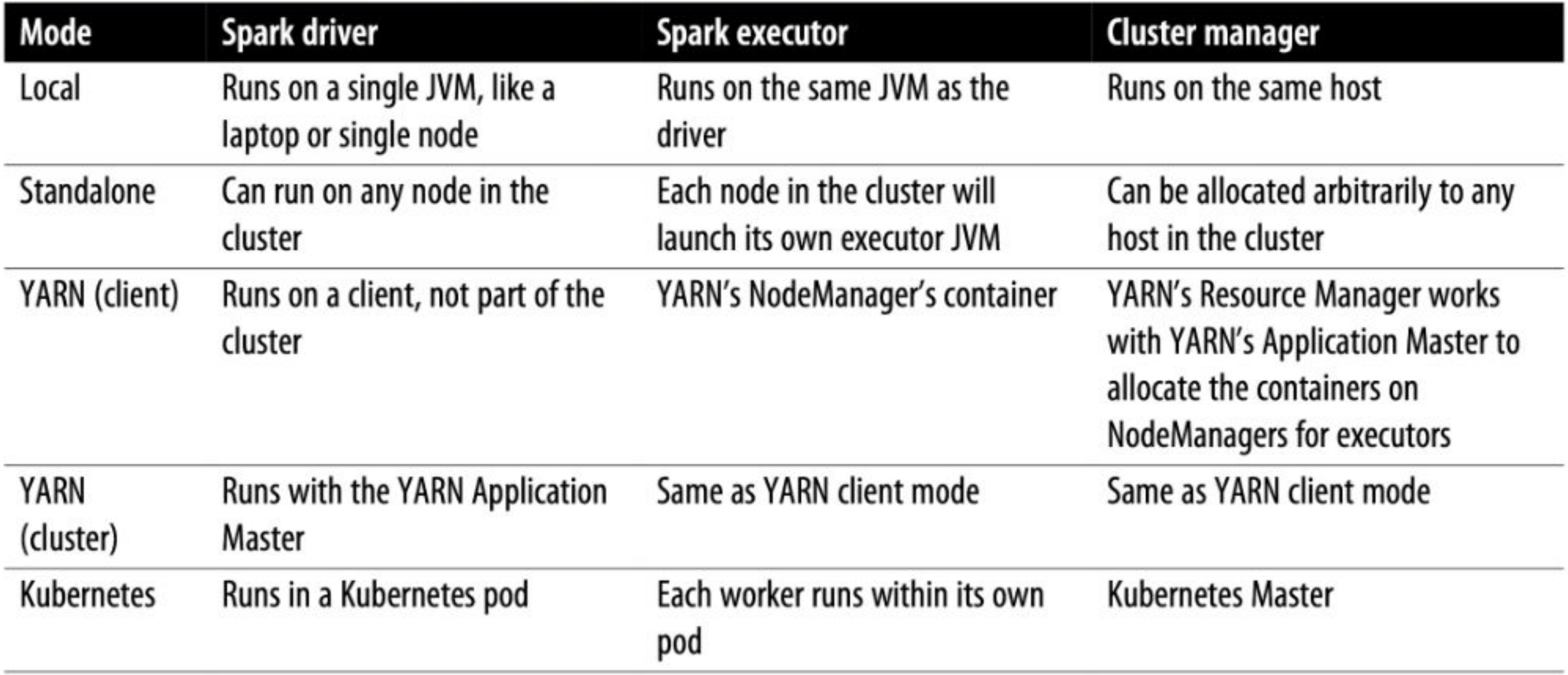
Amazon EC2

Standalone

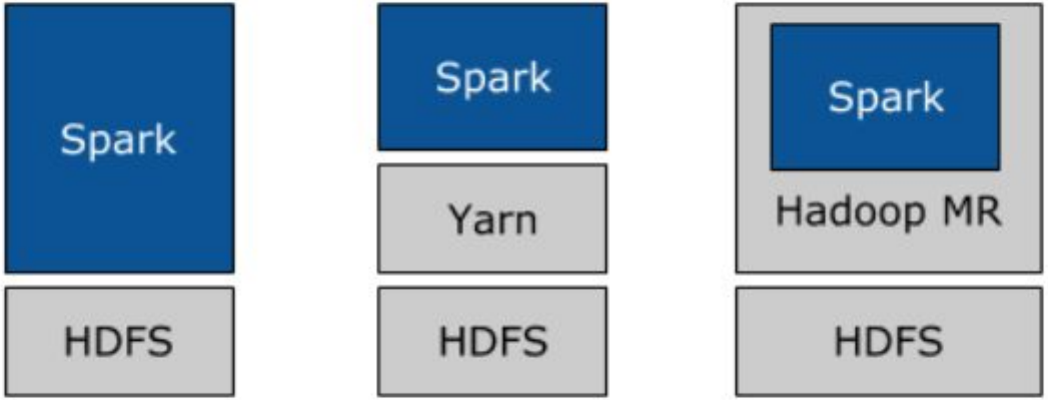
Apache Mesos

Resource/cluster managers

Table 1-1. Cheat sheet for Spark deployment modes



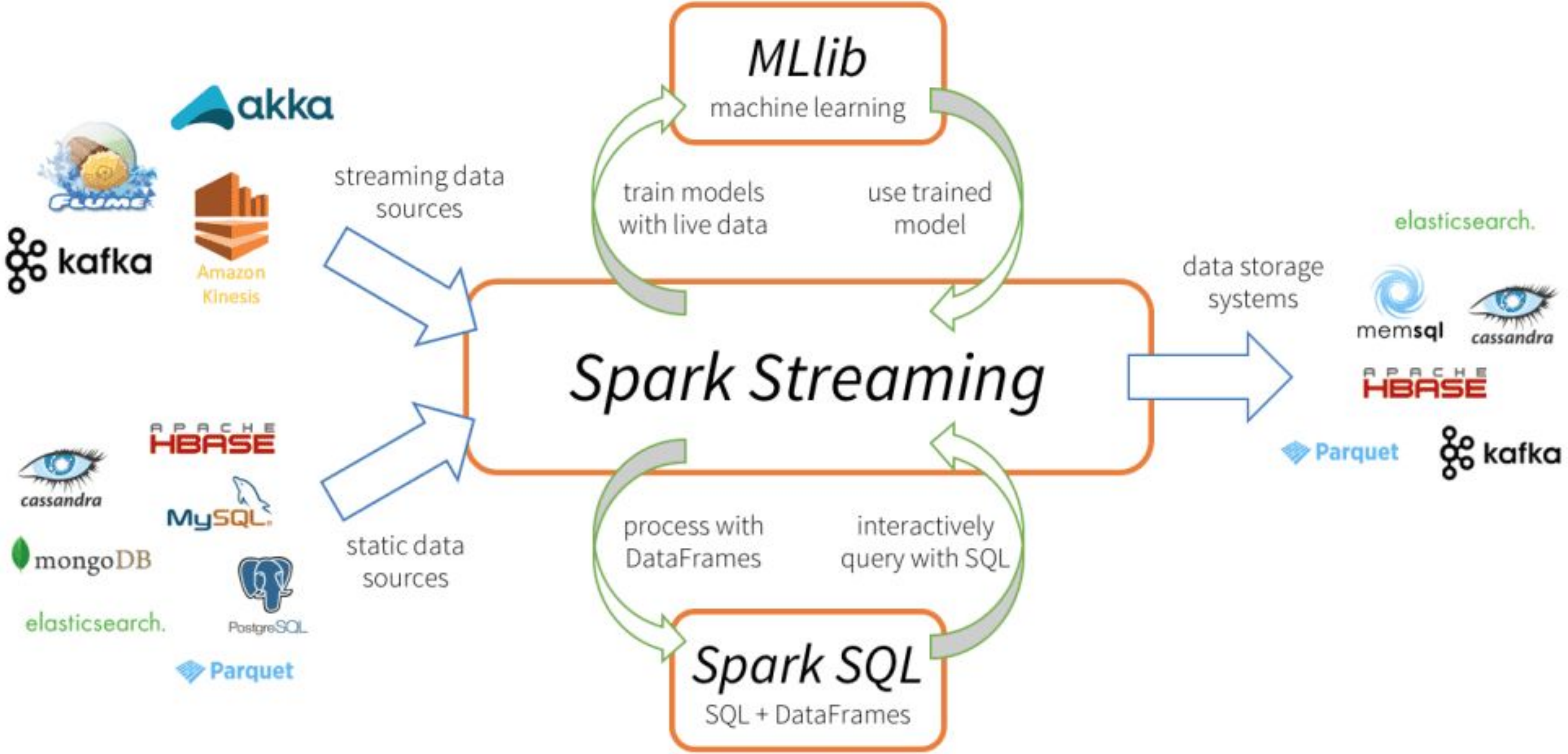
Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master



(a) Standalone

(b) Over Yarn

(c) Spark in
MR (SIMR)



streaming data sources

static data sources

MLlib
machine learning

train models with live data

use trained model

Spark Streaming

process with DataFrames

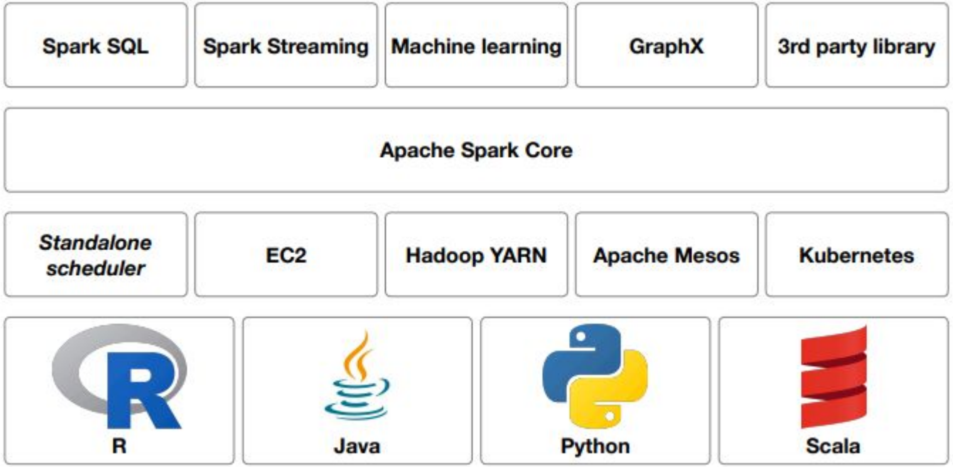
interactively query with SQL

Spark SQL
SQL + DataFrames

data storage systems

Spark ecosystem: Spark core

- Core functionalities
  - task scheduling
  - memory management
  - fault recovery
  - storage systems interaction
  - etc.
- Basic data structure definitions/abstractions
  - Resilient Distributed Data sets (RDDs)
    - main Spark data structure
  - Directed Acyclic Graph (DAG)



Spark SQL | Spark Streaming | Machine learning | GraphX | 3rd party library

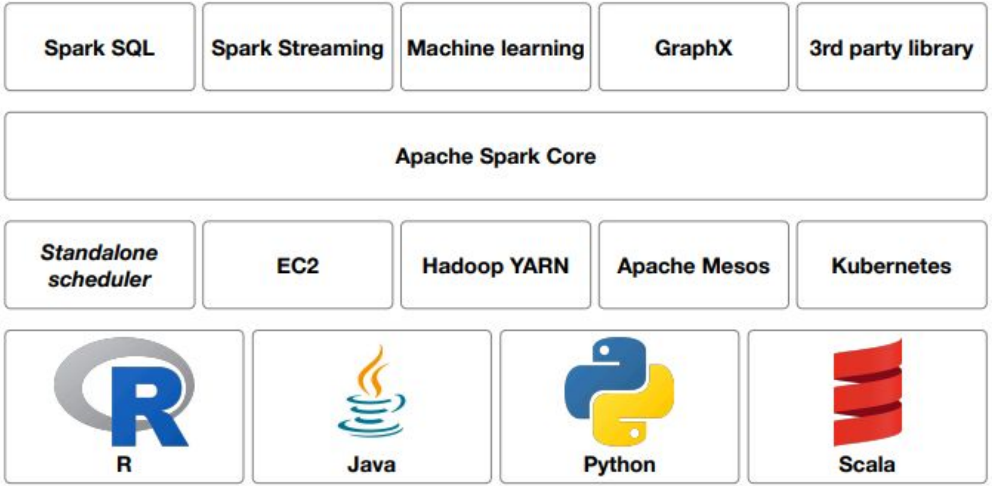
Apache Spark Core

Standalone scheduler | EC2 | Hadoop YARN | Apache Mesos | Kubernetes

R | Java | Python | Scala

Spark ecosystem: Spark SQL

- Structured data manipulation
  - Data Frames definition
- Table-like data representation
  - RDDs extension
  - Schema definition
- SQL queries execution
- Native support for schema-based data
  - Hive, Paquet, JSON, CSV



Spark SQL

Spark Streaming

Machine learning

GraphX

3rd party library

Apache Spark Core

Standalone scheduler

EC2

Hadoop YARN

Apache Mesos

Kubernetes

R

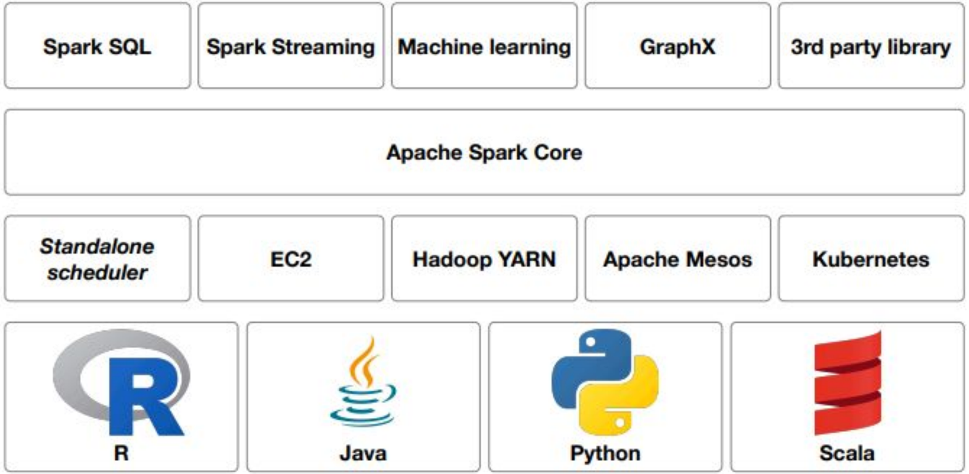
Java

Python

Scala

Spark ecosystem: Streaming

- Data analysis of streaming data
  - e.g. tweets, log messages
- Features of stream processing
  - High-troughput
  - Fault-tolerant
  - End-to-end
  - Exactly-once
- High-level abstraction of a discretized stream
  - Dstream represented as a sequence of RDDs
- Spark 2.3+ , Continuous Processing
  - end-to-end latencies as low as 1ms



Spark SQL

Spark Streaming

Machine learning

GraphX

3rd party library

Apache Spark Core

Standalone scheduler

EC2

Hadoop YARN

Apache Mesos

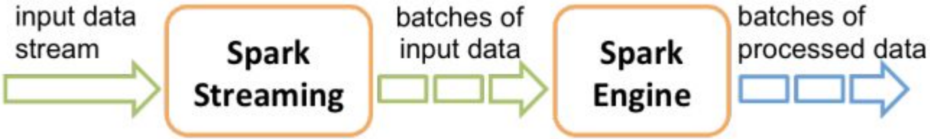
Kubernetes

R

Java

Python

Scala



input data stream

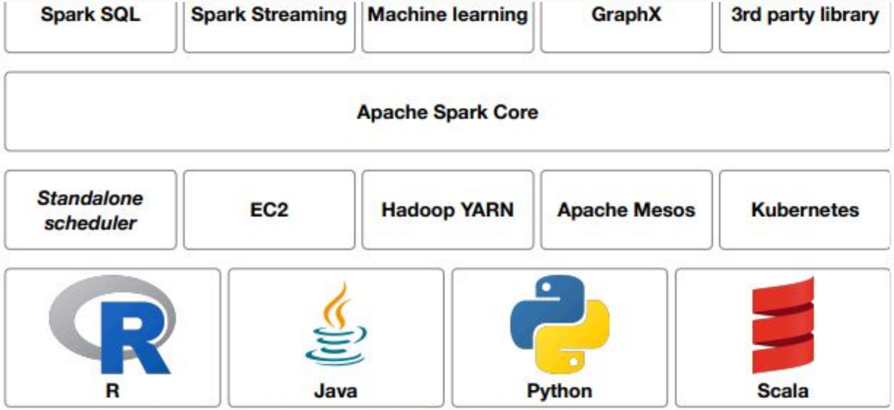
Spark Streaming

batches of input data

Spark Engine

batches of processed data

Spark ecosystem: MLlib



Spark SQL | Spark Streaming | Machine learning | GraphX | 3rd party library

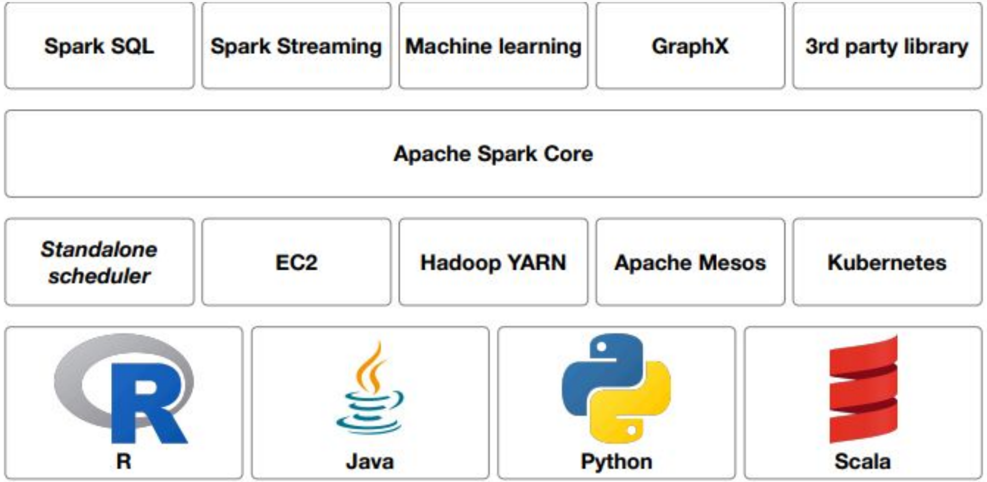
Apache Spark Core

Standalone scheduler | EC2 | Hadoop YARN | Apache Mesos | Kubernetes

R | Java | Python | Scala

- Common ML functionalities
  - ML Algorithms
    - common learning algorithms such as classification, regression, clustering, and collaborative filtering
  - Featurization
    - feature extraction, transformation, dimensionality reduction, and selection
  - Pipelines
    - tools for constructing, evaluating, and tuning ML Pipelines
  - Persistence
    - saving and load algorithms, models, and Pipelines
  - Utilities
    - linear algebra, statistics, data handling, etc.
- Two APIs
  - RDD-based API (*spark.mllib package*)
  - Spark 2.0+, DataFrame-based API (*spark.ml package*)
- Methods scale out across the cluster by default

Spark ecosystem: GraphX



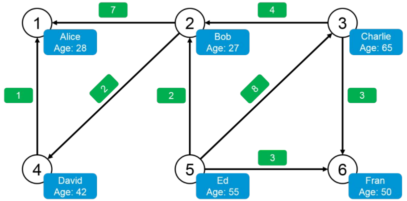
Spark SQL | Spark Streaming | Machine learning | GraphX | 3rd party library

Apache Spark Core

Standalone scheduler | EC2 | Hadoop YARN | Apache Mesos | Kubernetes

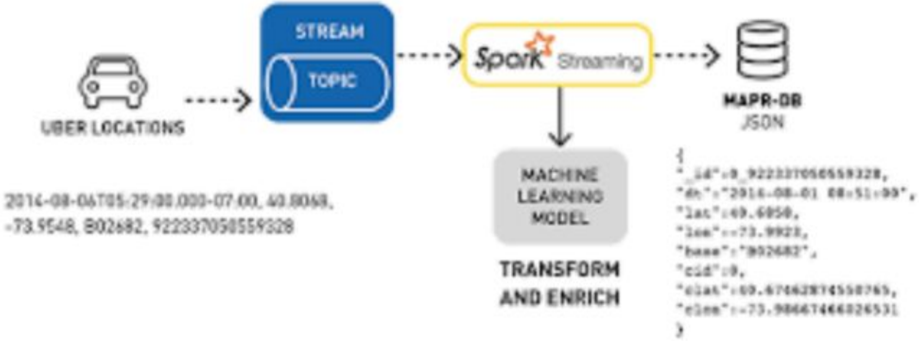
R | Java | Python | Scala

- Support for graphs and graph-parallel computation
  - Extension of RDDs (Graph)
    - directed multigraph with properties on vertices and edges
- Graph computation operators
  - subgraph, joinVertices, and aggregateMessages, etc.
  - PregelAPI support



Apache Spark Use cases

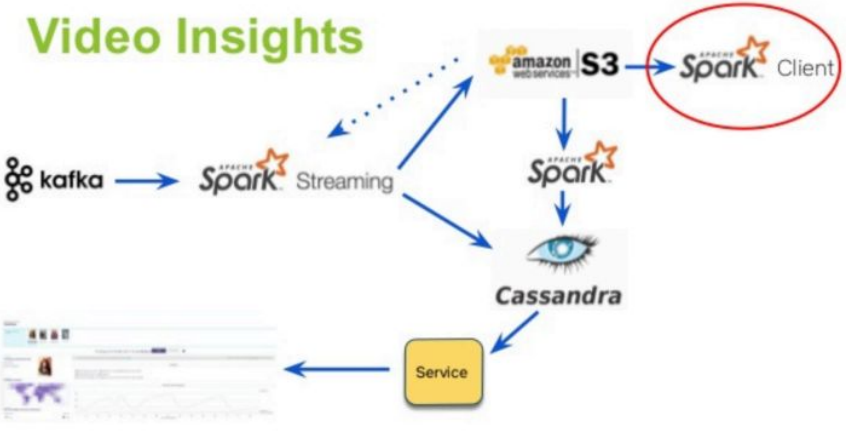
- Logs processing (Uber)
- Event detection and real-time analysis
- Interactive analysis
- Latency reduction
- Advanced ad-targeting (Yahoo!)
- Recommendation systems (Netflix, Pinterest)
- Fraud detection
- Sentiment analysis (Twitter)
- ...



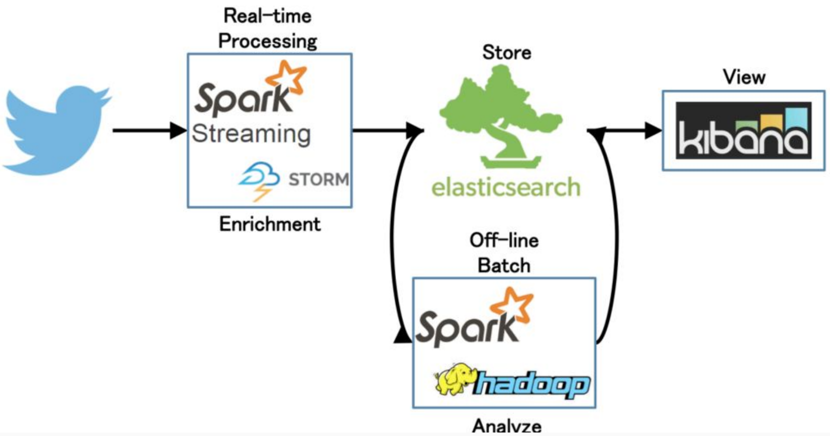
```
2014-08-06T05:29:00.000-07:00, 40.8068,
-73.9548, B02682, 9223370508559328
```

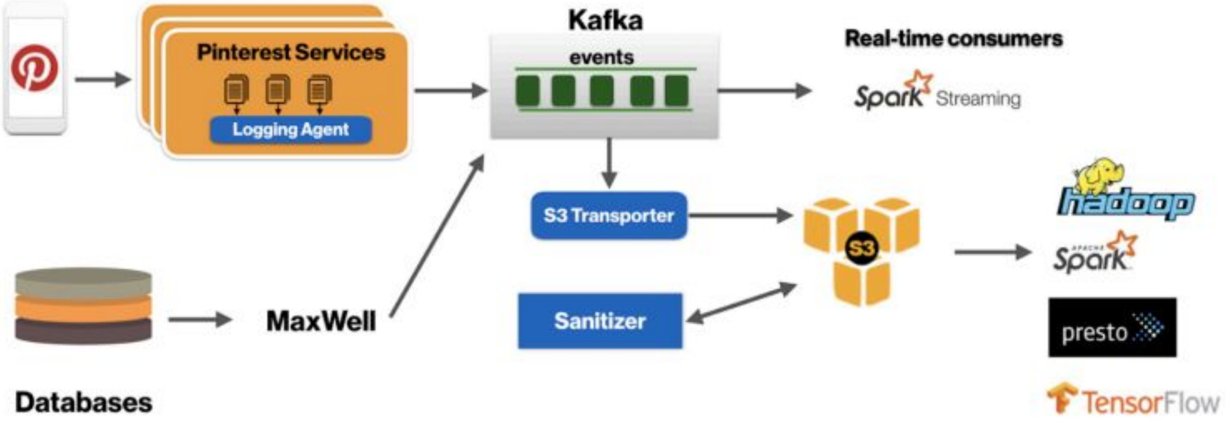
```
{
"_id":0_9223370506559328,
"dt":"2016-08-01 08:51:00",
"lat":40.6050,
"lon":-73.9923,
"base":"B02682",
"cid":0,
"clat":40.67462874558765,
"clon":-73.98667466026531
}
```

Video Insights

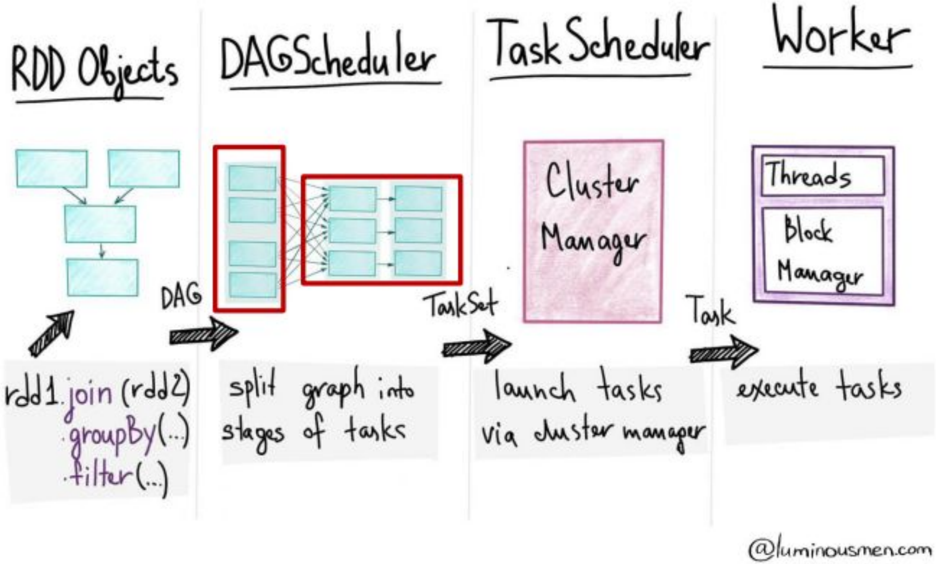


Apache Spark general setup: Twitter sentiment analysis





Spark Execution process



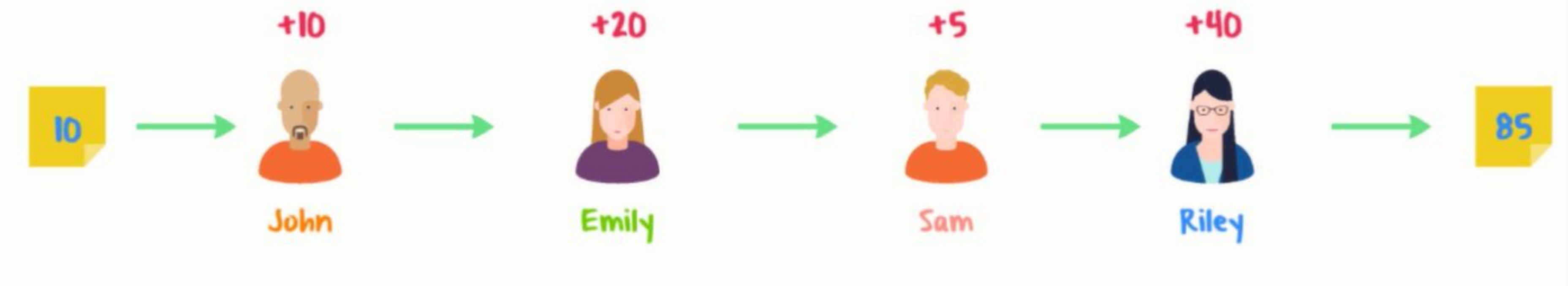
1. Data preparation/import
   - RDDs creation – i.e. parallel dataset with partitions
2. Transformations/actions definition*
   - Creation of tasks (units of work) sent to one executor
   - Job is a set of tasks executed by an action*
3. Creation of a directed acyclic graph (DAG)
   - Contains a graph of RDD operations
   - Defition of stages – set of tasks to be executed in parallel (i.e. at a partition level)
4. Execution of a program

Spark Programming concepts (Resilient distributed datasets - RDDs)

Need for RDD

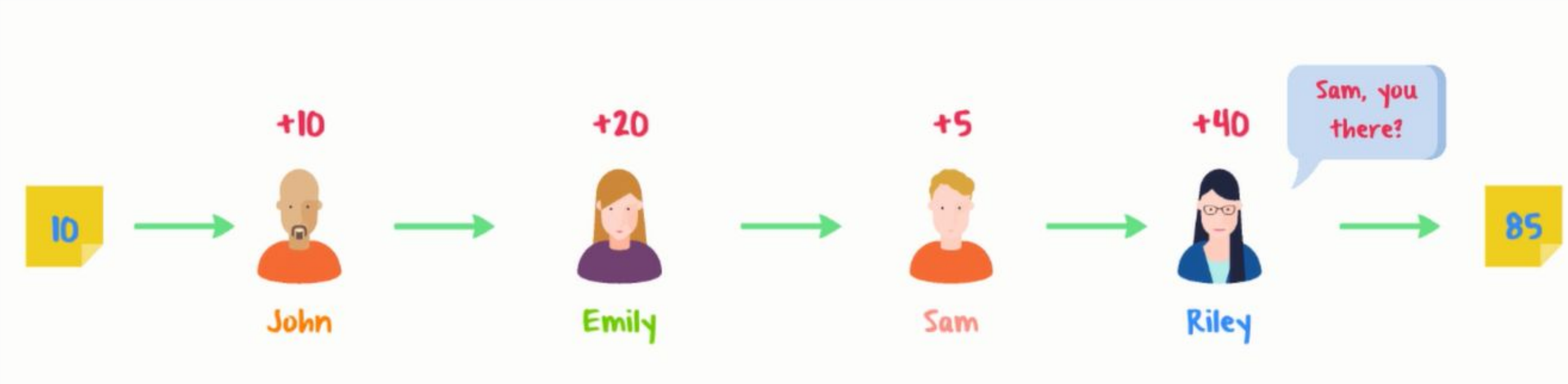
- The very first reason behind Spark’s speed is in-memory computing but in-memory computing is not a revolutionary new concept.
- In-memory computing in Spark is special because Spark does in-memory computing in a distributed scale – with hundreds, thousand or even more commodity computers.
- In-memory computing sounds like a very simple idea but to implement the Same in a distributed environment is not simple and easy. There are lot of complexities.

Complexity with In-memory Computation



John is given a piece of paper with number 10 written on it. You can think of this as an input dataset. So John reads 10 adds 10 and memorize the result 20, Emily asks John for the result, John answers 20, Emily adds 20 and memorize 40 as the result. Sam asks Emily for the result, Emily says 40, Sam adds 5 and memorize 45 as the result, finally Riley will ask Sam for the result and will add 40 to 45 and memorize 85 as the result.

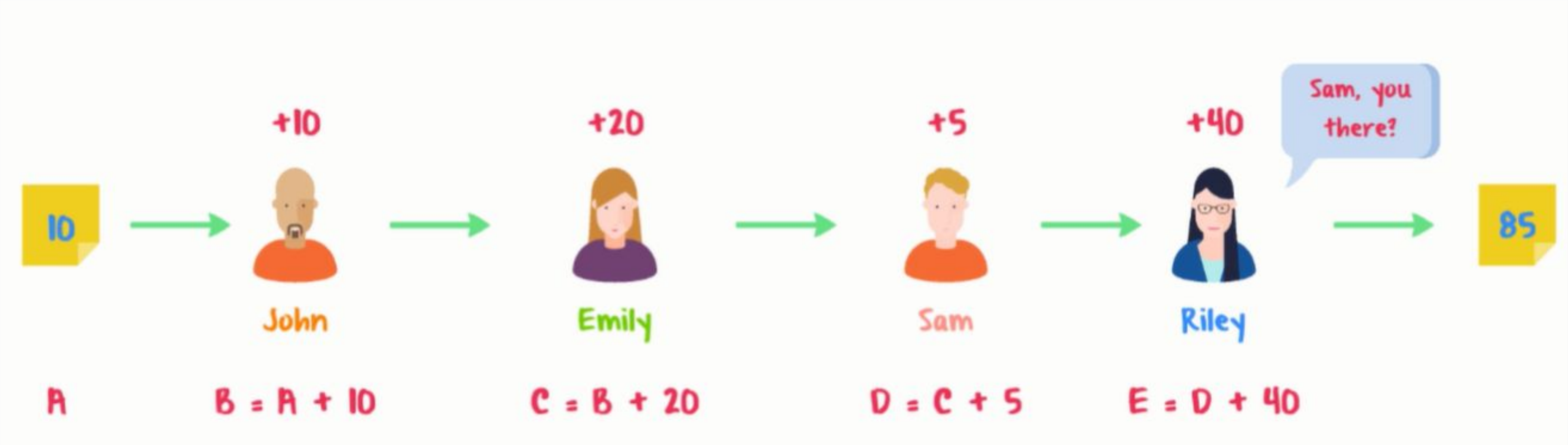
Note that each person is memorizing the result of their calculation. This is analogous to individual nodes keeping their intermediate output in memory.



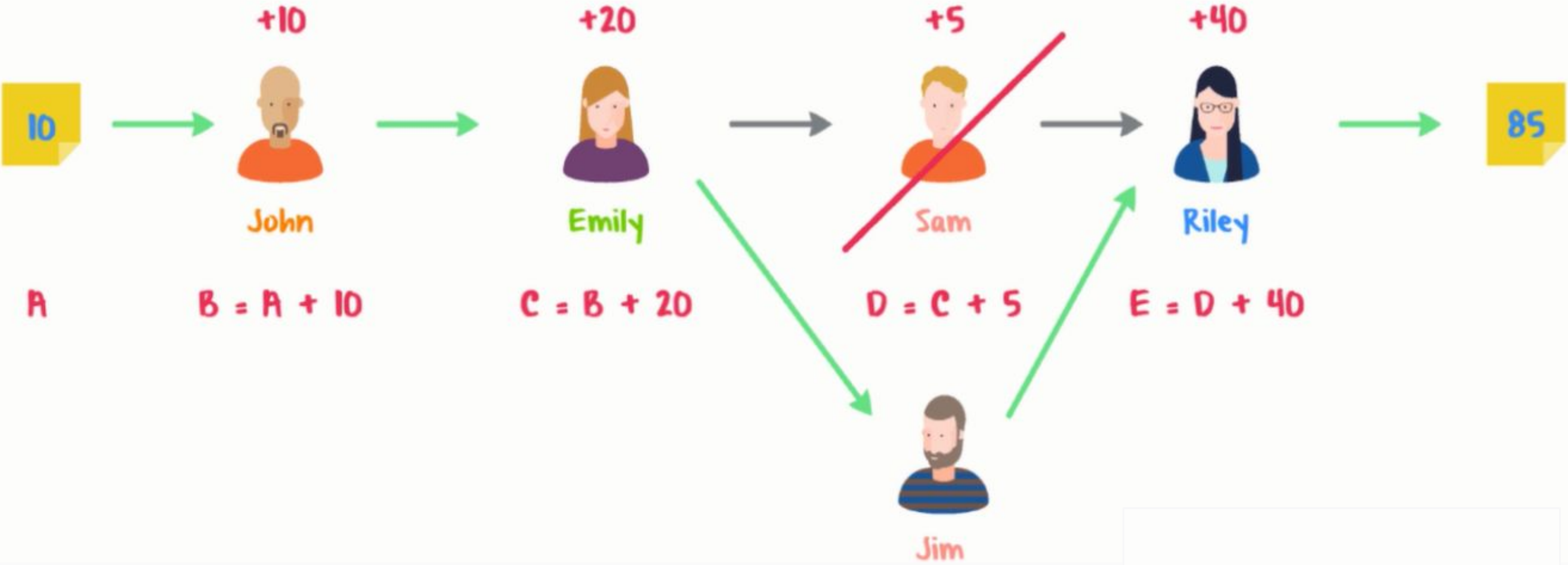
Let’s now say Sam finished the calculation he is supposed to do and keeps the result 45 in his memory. It is now Riley’s turn to do the calculation. Riley reaches out to Sam for the number, Sam is not responding. May be Sam is thinking of something important or may be he is having a brain freeze :-). Whatever the reason may be, Sam is not responding to Riley. Now this is a problem, without the input from Sam, Riley cannot proceed with the calculation.

If you take out the individuals from this illustration and substitute nodes or hosts, you will get the same problem. When the node goes down everything that is stored in its main memory or RAM is lost. So what is the solution?

Lineage & Fault Tolerance



Let’s call the input that was given to John as A. John takes A adds 10 and produces 20 and lets to this intermediate result as B. Emily takes B adds 20 and produces 40 and lets call this C, Sam takes C adds 5 produces D which is 45, finally Riley takes D adds 40 and produces E which is 85. Here we have clear track of how our data is getting transformed. We know B is A plus 10, D is C plus 5 and finally E is D plus 40.


10
+10
John
+20
Emily
+5
Sam
+40
Riley
85
Jim
A
B = A + 10
C = B + 20
D = C + 5
E = D + 40

With this in mind, let's walk through a failure scenario. It's now Riley's turn to do the calculation. Riley know that she should take D and add 40 to it. So she reaches out to Sam for D. Sam is not responding. Sam memorized D so if Sam is not responding how would Riley know the value of D? We know what Sam did to calculate D. He took C from Emily and add 5 to it to come up with D. Now Sam is not available but we know how to get D, we can ask someone else to calculate D. Let's call Jim to calculate D, Jim will ask Emily for C add 5 to it and will calculate D which is 45.

Now Riley will ask Jim instead of Sam for D and will complete the calculation. Since we kept track of all the transformations that happened to our data we were able to tolerate Sam's failure and that is our solution to fault tolerance – keep track of all transformations that happens to our dataset.

Spark does exactly this, it keeps tracks of every single operation or transformation that happens to your dataset and it is referred to as **lineage**, this way even when there is a failure Spark knows how to recover from that failure. Here is the million dollar question. How does spark keep track of everything you do with your data? The answer is RDDs. RDD stands for Resilient Distributed Dataset.

What is RDD?

- Spark tightly controls what you do with the dataset. You cannot work with data in Spark without RDD.
- To refer your dataset in Spark you will use the functions provided by Spark and those functions will create a RDD behind the scenes.
- Can you join two datasets in spark? Of course you can using the join function provided by Spark and internally the join function will result in a RDD.
- Can you do aggregation in your dataset? Of course you can, Spark provides functions for that too and spark will create a RDD behind the scenes any time you attempt to transform the data.
- In short, Spark controls any operations that you do with your dataset, to perform any meaningful operation with your data you need to use functions provided by Spark and it will result in RDD.
- This way Spark can keep track of everything that you are trying to do with the dataset and this is referred to as lineage and this helps Spark deal with tolerating failures effectively, which by the way is a very big deal in in-memory computing.

RDDs - Resilient Distributed Datasets

What is RDD?

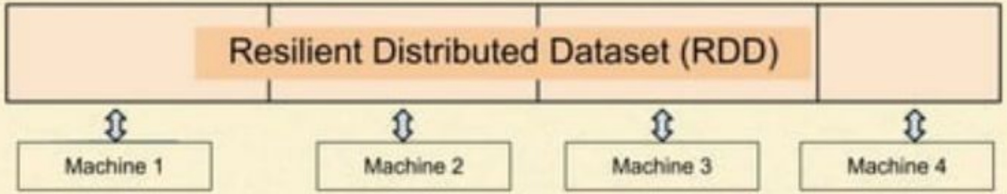
Dataset:
Collection of data elements.
e.g. Array, Tables, Data frame (R), collections of mongodb

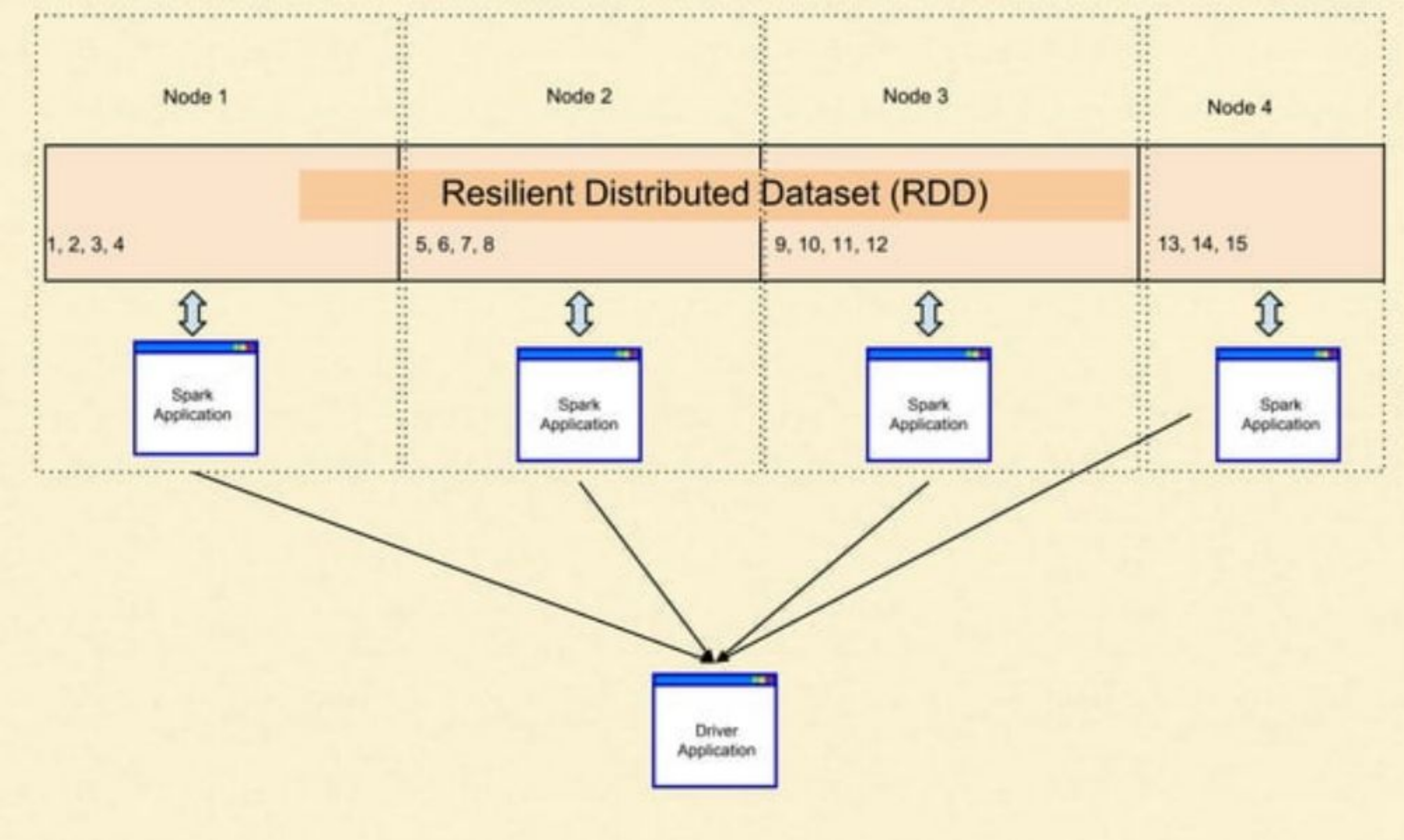
Distributed:
Parts Multiple machines

Resilient:
Recovers on Failure

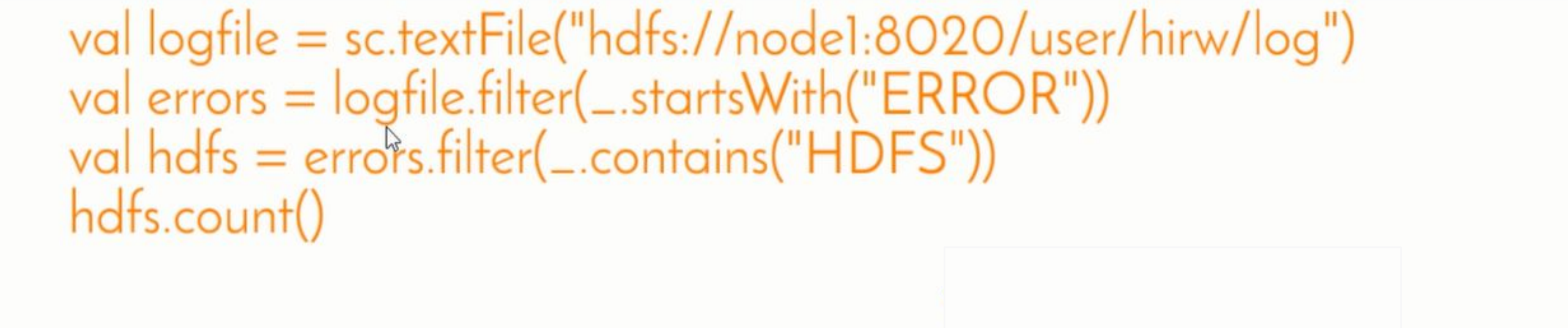
SPARK - CONCEPTS - RESILIENT DISTRIBUTED DATASET

A collection of elements partitioned across cluster





Node 1
Node 2
Node 3
Node 4
Resilient Distributed Dataset (RDD)
1, 2, 3, 4
5, 6, 7, 8
9, 10, 11, 12
13, 14, 15
Spark Application
Spark Application
Spark Application
Spark Application
Driver Application

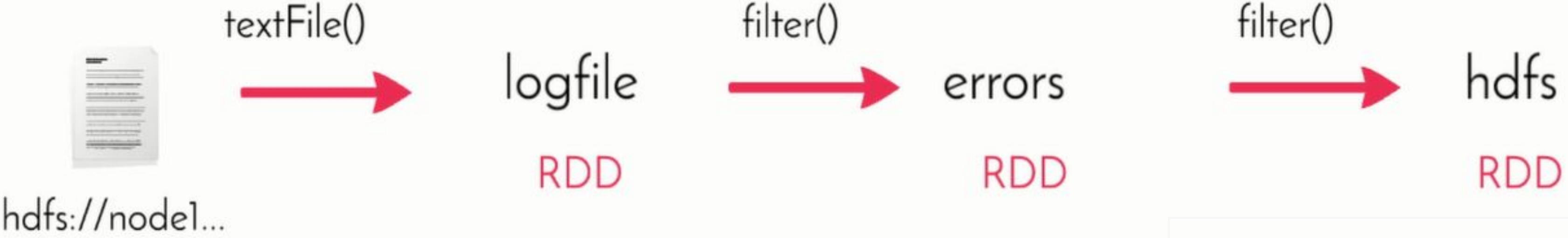


```
val logfile = sc.textFile("hdfs://node1:8020/user/hirw/log")
val errors = logfile.filter(_.startsWith("ERROR"))
val hdfs = errors.filter(_.contains("HDFS"))
hdfs.count()
```

- Let’s understand what is RDD with a simple use case.
- We are going to count the number of HDFS related errors in a log file.
- First line – we are referring to a text file in HDFS and we are assigning it to logfile, in the next line, we are filtering all lines in logfile with the word errors in it and calling it errors, in the next line, we are filtering all lines in errors with the word HDFS in it and we are referring the result hdfs.
- Finally in the 4th line we are counting the number of lines in hdfs.

```
val logfile = sc.textFile("hdfs://node1:8020/user/hirw/log")
val errors = logfile.filter(_.startsWith("ERROR"))
val hdfs = errors.filter(_.contains("HDFS"))
hdfs.count()
```

Lineage



textFile()

filter()

filter()

logfile

errors

hdfs

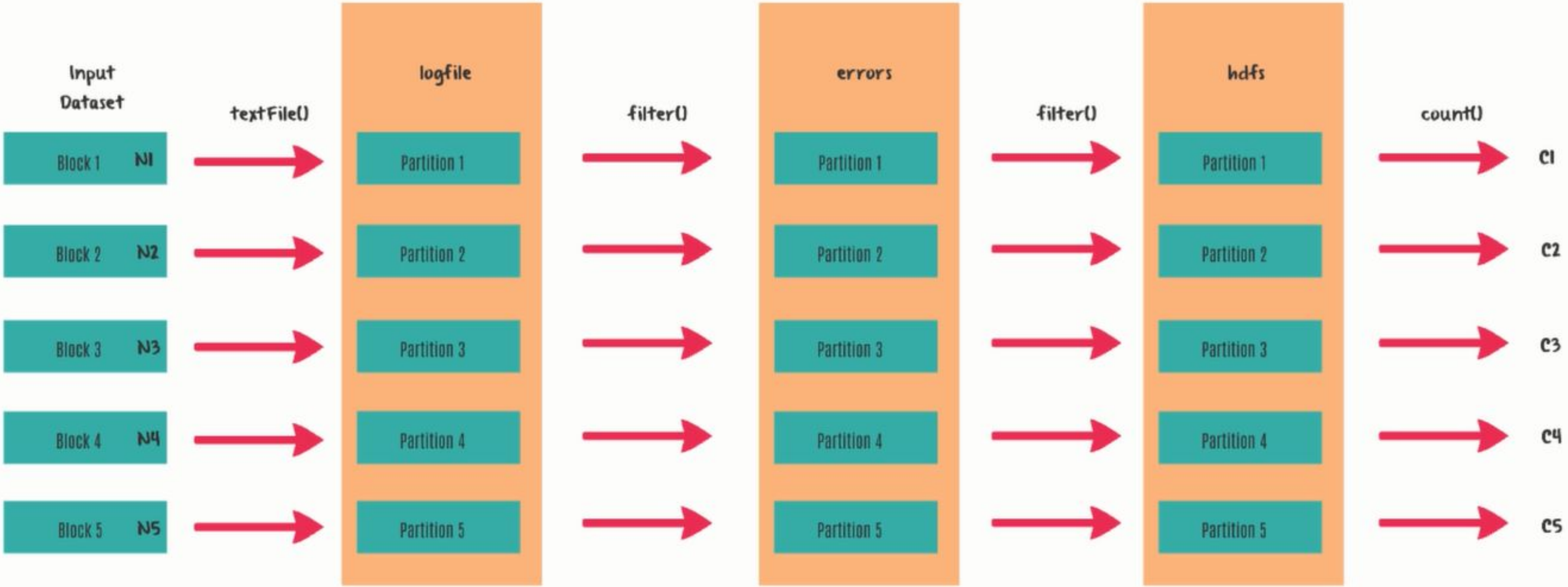
RDD

RDD

RDD

hdfs://node1...

- First we have logfile, then we applied a filter function on logfile to get errors and then we applied another filter function on errors this time to get hdfs.
- We can say hdfs is dependent on errors and errors is dependent on logfile Spark refers to this dependency chain as lineage.
- Like your family lineage for instance, you are a result of your father and mother. Your father and mother are result of their respective parents. textfile and filter functions are functions from the Spark API.

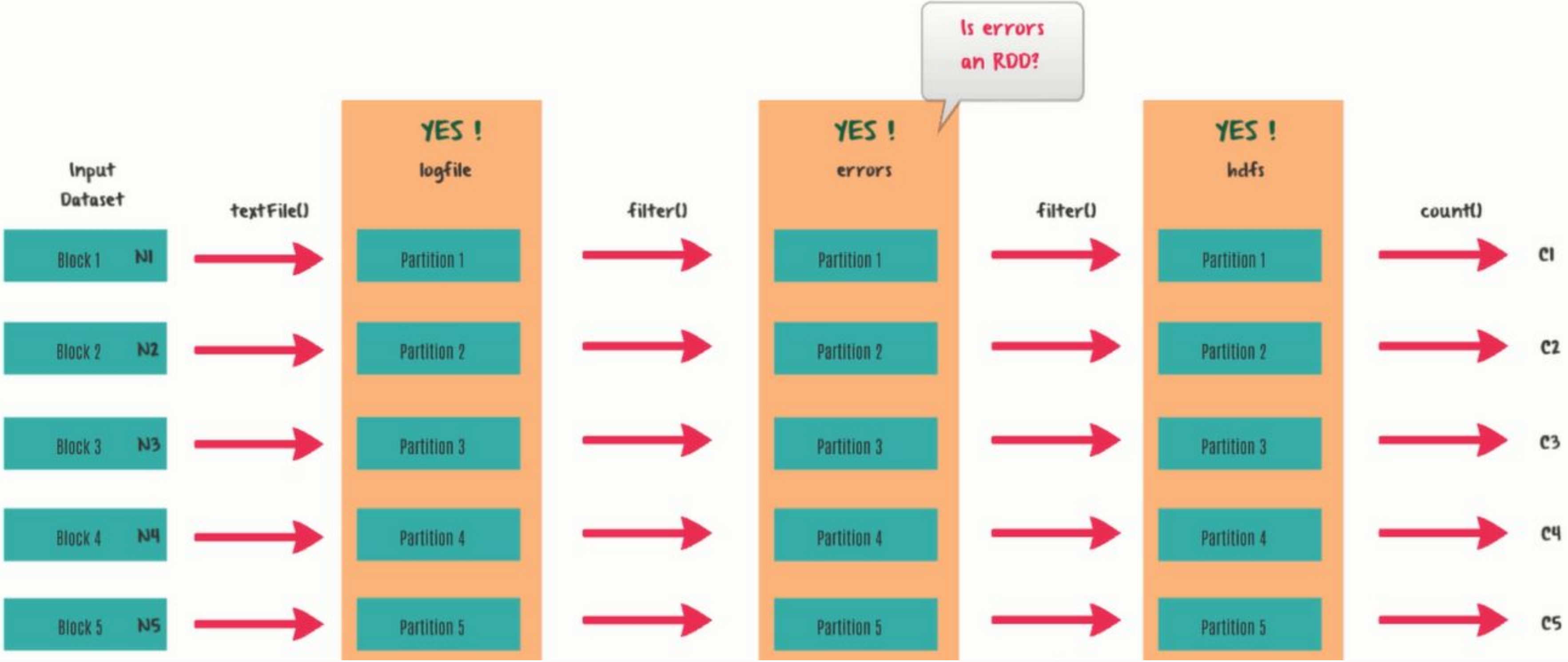


Assume our dataset is in HDFS and is divided in to 5 blocks. When we referred to the dataset using the textFile function the resulting logfile will have 5 blocks and each of these 5 blocks can technically be on a different node. Spark calls a block a data as partition. So logfile will refer to partitions from different nodes. Since we are referring to a text file in HDFS each partition will have several lines or records. In Spark we refer to each record as element.

- So each partition will have several elements.
- Next, we apply filter function on logfile to filter only lines with the word ERROR in it and this operation results in errors.
- Filter function is applied on each element or in simple terms each record in each partition from logfile.
- The result will be errors, which will again have 5 partitions and will only have elements with the word ERROR in it.
- Next we apply filter operation on errors to filter only lines with the word HDFS in it and this operation results in hdfs.
- Filter function is applied on each element in each partition from errors RDD resulting in HDFS.
- Finally we call the count function on hdfs which counts all the elements in each partition. Spark will sum up all the individual counts and send it to the user.

R in RDD stands for Resilient meaning resilience to failure; that is being fault tolerant. DD in RDD stands for Distributed Dataset. So RDD refers to a distributed dataset which is tolerant to failure. Now with that in mind, let’s pick errors from the illustration.

R D D
Resilient Distributed Dataset



Is errors an RDD?

Input Dataset

Block 1 N1

Block 2 N2

Block 3 N3

Block 4 N4

Block 5 N5

textFile()

YES !

logfile

Partition 1

Partition 2

Partition 3

Partition 4

Partition 5

filter()

YES !

errors

Partition 1

Partition 2

Partition 3

Partition 4

Partition 5

filter()

YES !

hdfs

Partition 1

Partition 2

Partition 3

Partition 4

Partition 5

count()

C1

C2

C3

C4

C5

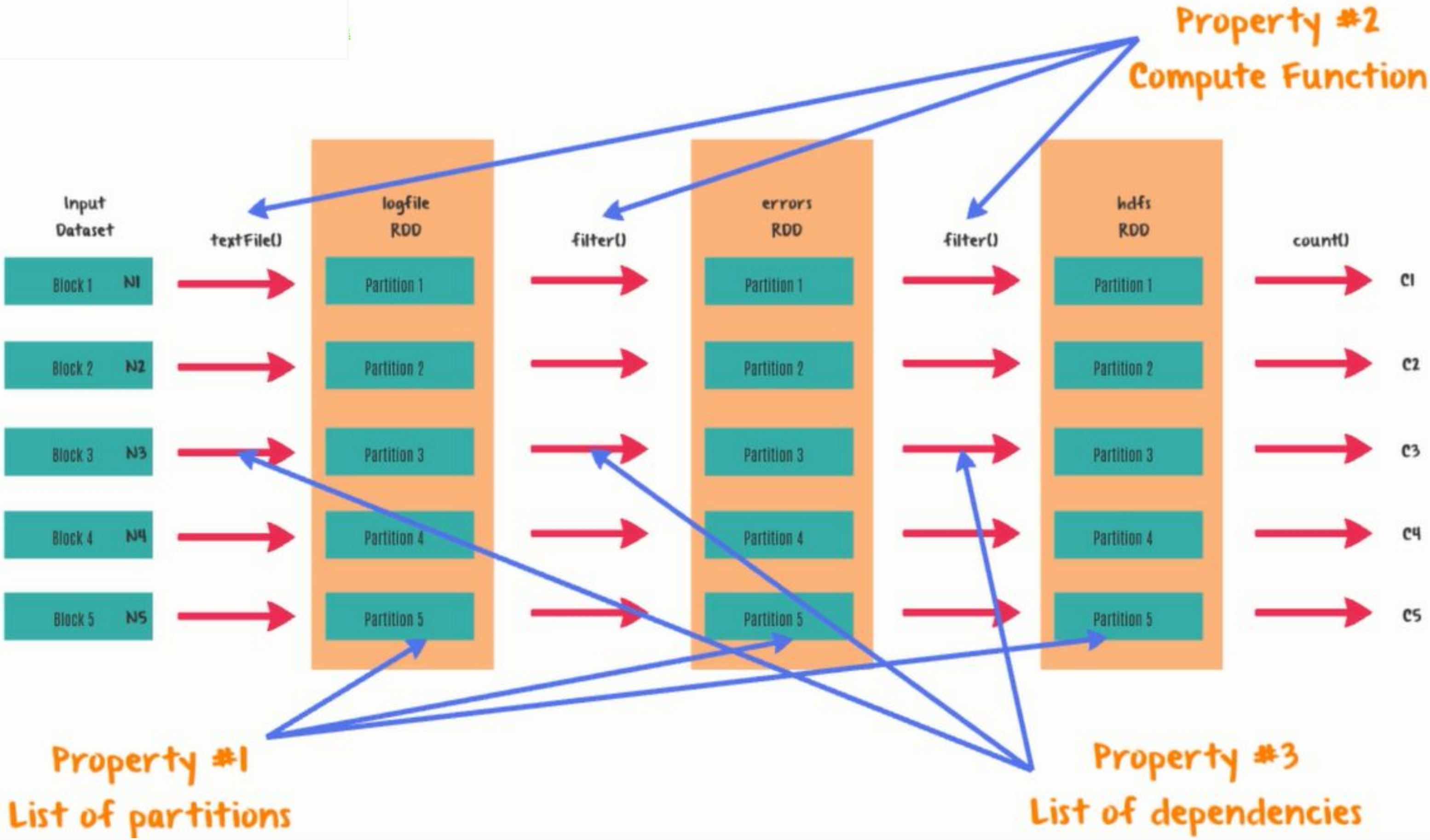
Let’s first check whether errors is resilient or not.

Assume partition 2 in errors resides in node 2 and node 2 for some hardware failure went down. Which means we lost partition number 2 in errors. If we are able to recover partition 2 we can say errors is resilient correct? True.

We know exactly how to recover partition 2. To get partition 2 in errors simply run the filter function on partition 2 of logfile and we will get partition 2 of errors. This is possible because we know the lineage and because of lineage we know errors is dependent on logfile and errors can be derived from logfile by applying the filter function. So errors is resilient.

DD in RDD stands for distributed dataset. Is errors a distributed dataset? errors has multiple partitions distributed in different nodes which means it is a distributed dataset. So errors is both resilient and a distributed dataset and hence an RDD.

Same is true for logfile and hdfs. So logfile is an RDD, errors is an RDD and hdfs is an RDD. But wait a minute! In our instructions we did not specify any where that we are creating RDDs correct? So is Spark really treating logfile, errors and hdfs as RDDs?


Property #2
Compute Function
Input
Dataset
textFile()
logfile
RDD
filter()
errors
RDD
filter()
hdfs
RDD
count()
Block 1 N1
Block 2 N2
Block 3 N3
Block 4 N4
Block 5 N5
Partition 1
Partition 2
Partition 3
Partition 4
Partition 5
Partition 1
Partition 2
Partition 3
Partition 4
Partition 5
Partition 1
Partition 2
Partition 3
Partition 4
Partition 5
C1
C2
C3
C4
C5
Property #1
List of partitions
Property #3
List of dependencies

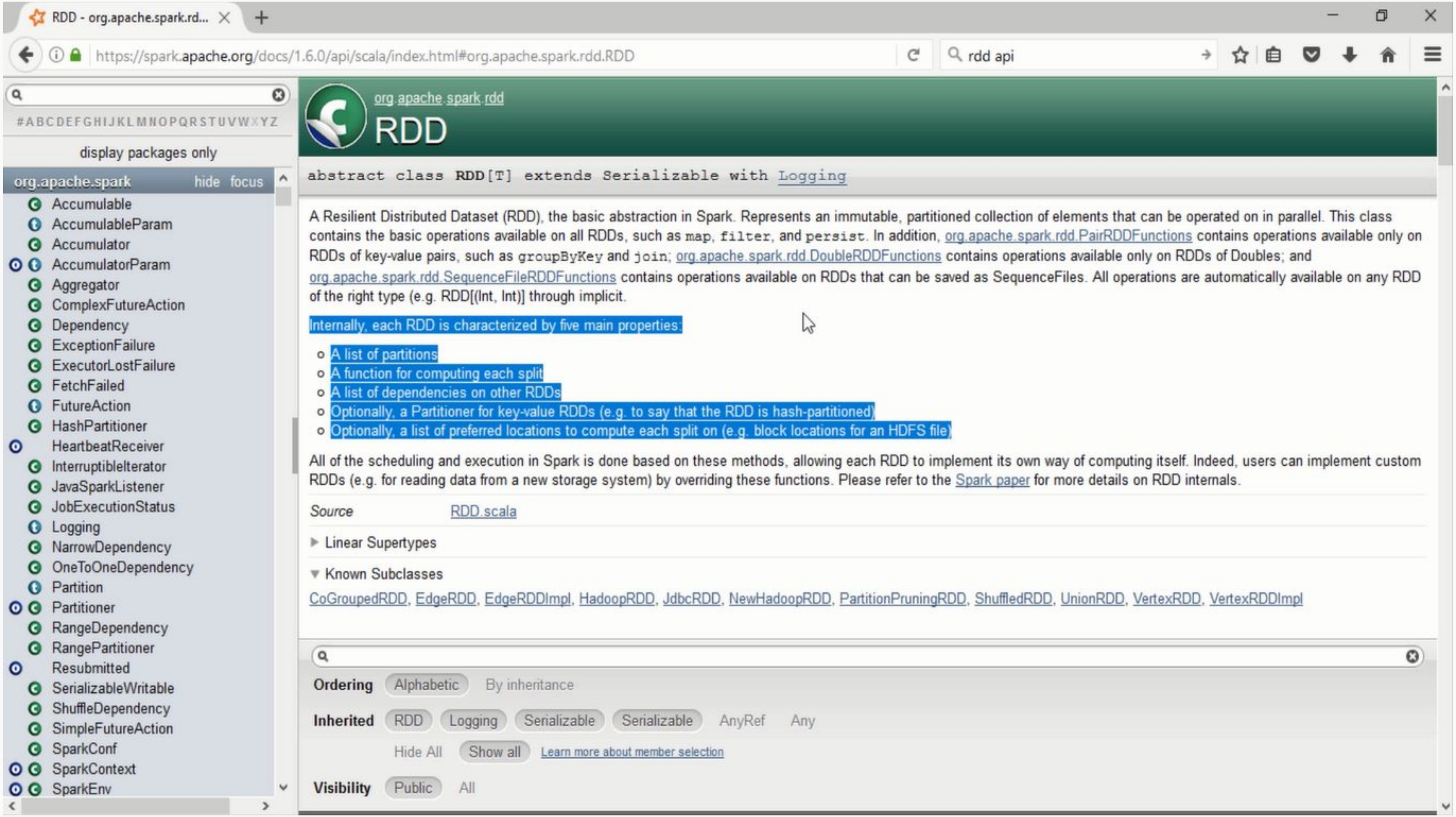
SPARK - CONCEPTS - RESILIENT DISTRIBUTED DATASET

A collection of elements partitioned across cluster

- An immutable distributed collection of objects.
- Split in partitions which may be on multiple nodes
- Can contain any data type:
    - Python,
    - Java,
    - Scala objects
    - including user defined classes

SPARK - CONCEPTS - RESILIENT DISTRIBUTED DATASET

- RDD Can be persisted in memory
- RDD Auto recover from node failures
- Can have any data type but has a special dataset type for key-value
- Supports two type of operations:
  - Transformation
  - Action



```
abstract class RDD[T] extends Serializable with Logging
```

A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable, partitioned collection of elements that can be operated on in parallel. This class contains the basic operations available on all RDDs, such as `map`, `filter`, and `persist`. In addition, org.apache.spark.rdd.PairRDDFunctions contains operations available only on RDDs of key-value pairs, such as `groupByKey` and `join`; org.apache.spark.rdd.DoubleRDDFunctions contains operations available only on RDDs of Doubles; and org.apache.spark.rdd.SequenceFileRDDFunctions contains operations available on RDDs that can be saved as SequenceFiles. All operations are automatically available on any RDD of the right type (e.g. RDD[(Int, Int)] through implicit.

Internally, each RDD is characterized by five main properties:

- A list of partitions
- A function for computing each split
- A list of dependencies on other RDDs
- Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
- Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)

All of the scheduling and execution in Spark is done based on these methods, allowing each RDD to implement its own way of computing itself. Indeed, users can implement custom RDDs (e.g. for reading data from a new storage system) by overriding these functions. Please refer to the Spark paper for more details on RDD internals.

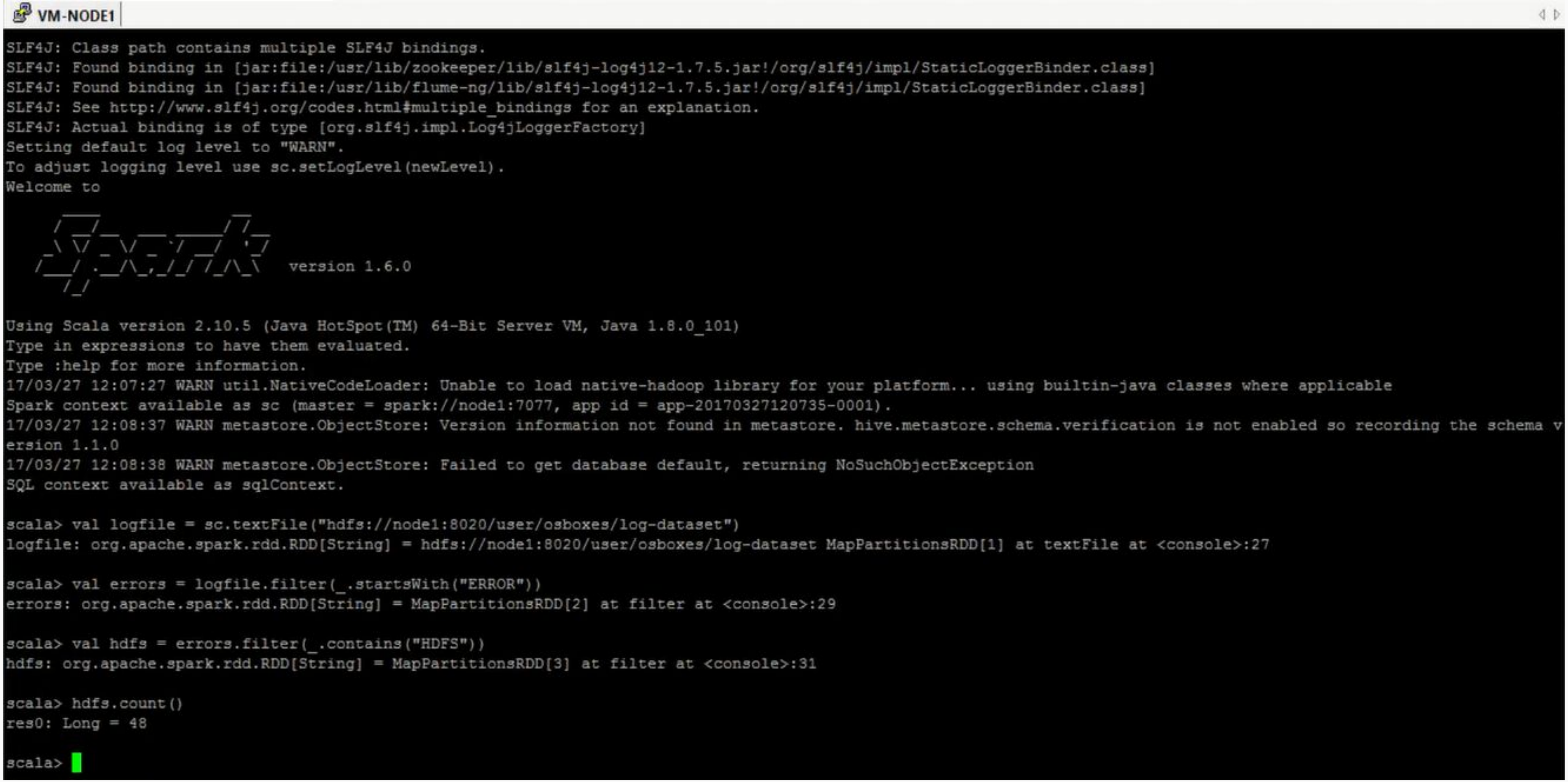
Source RDD.scala

▶ Linear Supertypes

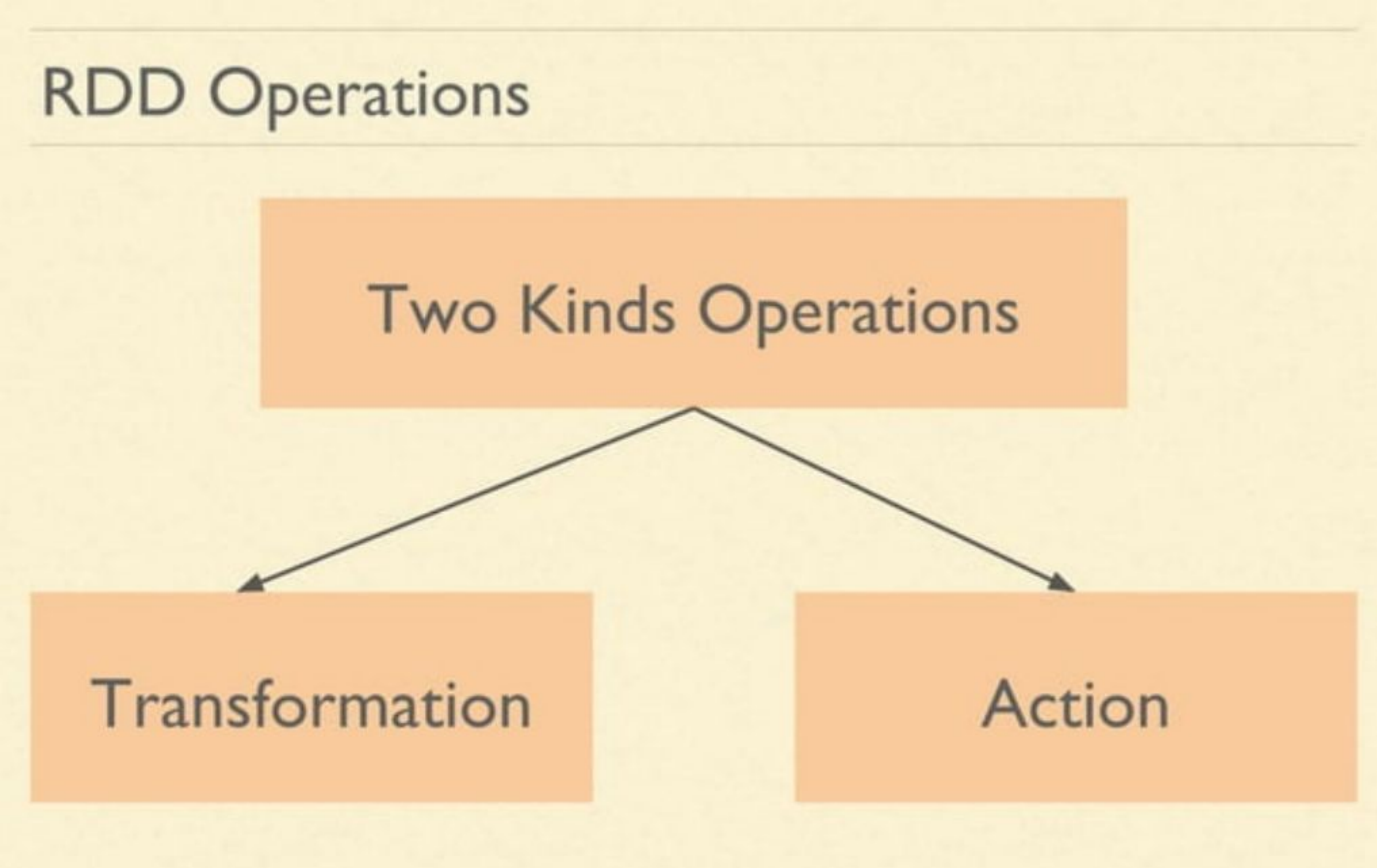
▼ Known Subclasses

CoGroupedRDD, EdgeRDD, EdgeRDDImpl, HadoopRDD, JdbcRDD, NewHadoopRDD, PartitionPruningRDD, ShuffledRDD, UnionRDD, VertexRDD, VertexRDDImpl

Ordering Alphabetic By inheritance
Inherited RDD Logging Serializable Serializable AnyRef Any
Hide All Show all Learn more about member selection
Visibility Public All



```
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/flume-ng/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_101)
Type in expressions to have them evaluated.
Type :help for more information.
17/03/27 12:07:27 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Spark context available as sc (master = spark://node1:7077, app id = app-20170327120735-0001).
17/03/27 12:08:37 WARN metastore.ObjectStore: Version information not found in metastore. hive.metastore.schema.verification is not enabled so recording the schema v
ersion 1.1.0
17/03/27 12:08:38 WARN metastore.ObjectStore: Failed to get database default, returning NoSuchObjectException
SQL context available as sqlContext.

scala> val logfile = sc.textFile("hdfs://node1:8020/user/osboxes/log-dataset")
logfile: org.apache.spark.rdd.RDD[String] = hdfs://node1:8020/user/osboxes/log-dataset MapPartitionsRDD[1] at textFile at <console>:27

scala> val errors = logfile.filter(_.startsWith("ERROR"))
errors: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at filter at <console>:29

scala> val hdfs = errors.filter(_.contains("HDFS"))
hdfs: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at filter at <console>:31

scala> hdfs.count()
res0: Long = 48

scala> 
```

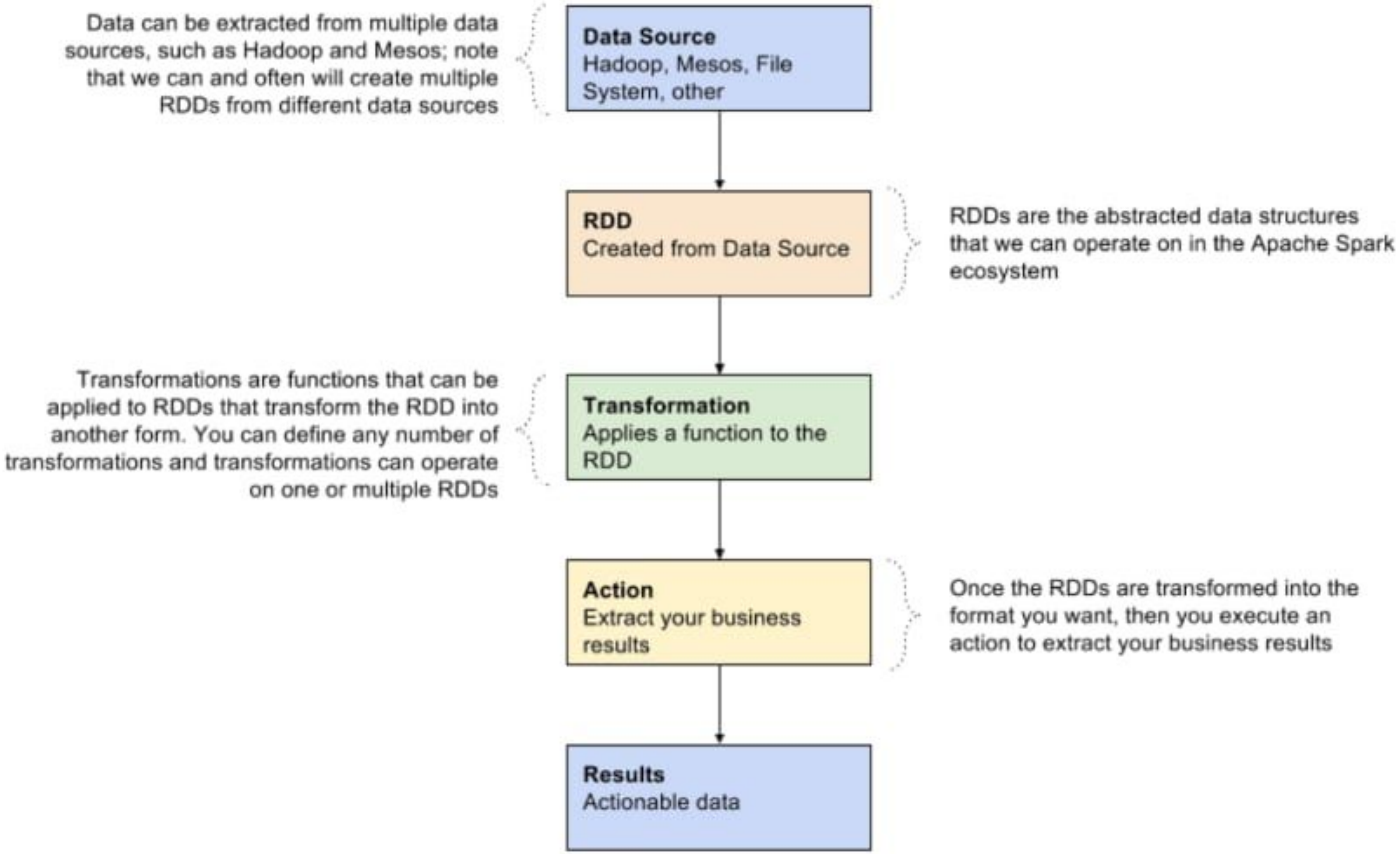


RDD Operations

Two Kinds Operations

Transformation

Action



Data can be extracted from multiple data sources, such as Hadoop and Mesos; note that we can and often will create multiple RDDs from different data sources

Data Source
Hadoop, Mesos, File System, other

RDD
Created from Data Source

RDDs are the abstracted data structures that we can operate on in the Apache Spark ecosystem

Transformations are functions that can be applied to RDDs that transform the RDD into another form. You can define any number of transformations and transformations can operate on one or multiple RDDs

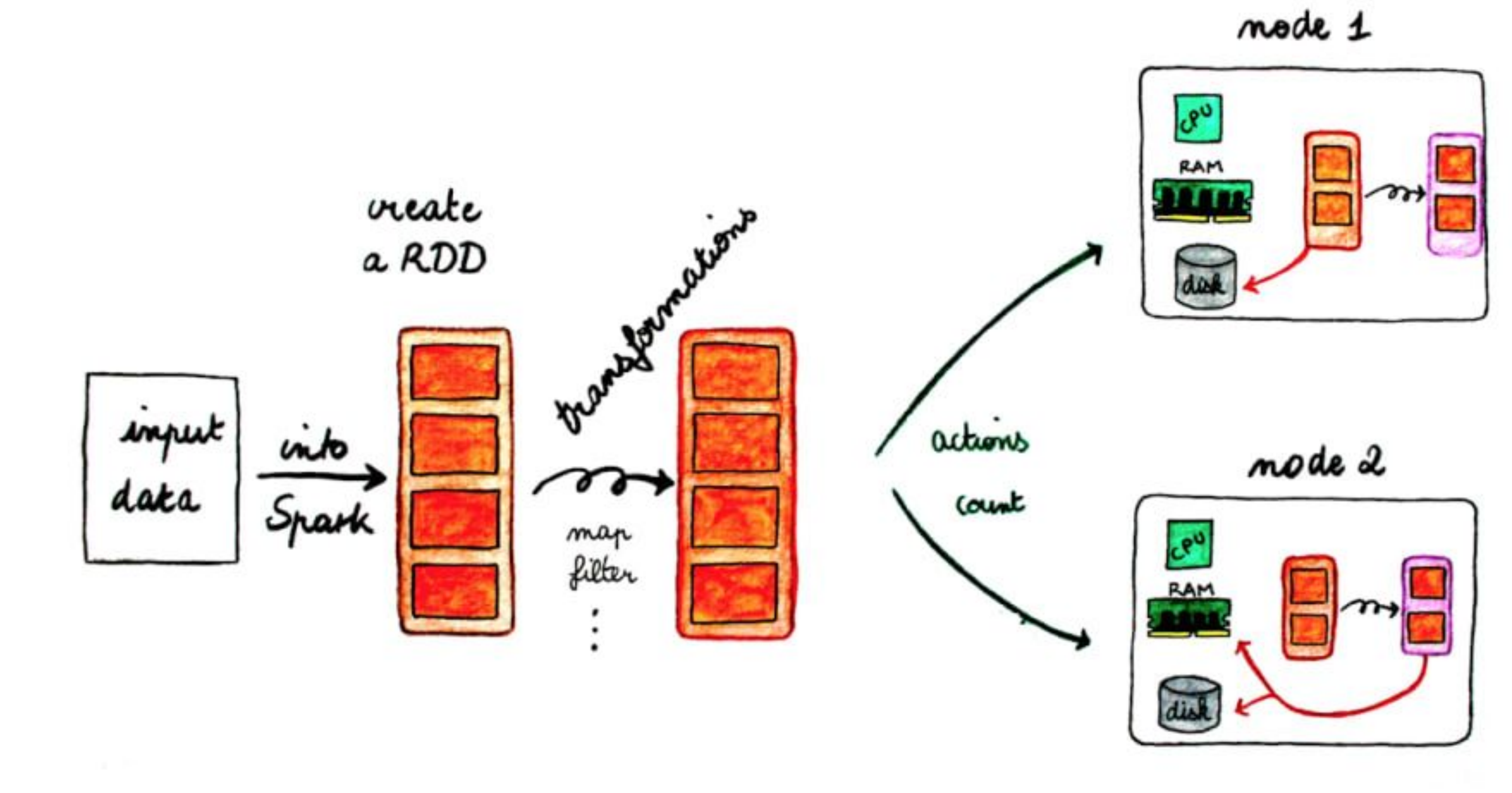
Transformation
Applies a function to the RDD

Action
Extract your business results

Once the RDDs are transformed into the format you want, then you execute an action to extract your business results

Results
Actionable data

Figure 1. Apache Spark Application Process Flow


input data
into Spark
create a RDD
transformations
map
filter
⋮
actions
count
node 1
node 2
CPU
RAM
disk
CPU
RAM
disk

Spark SQL

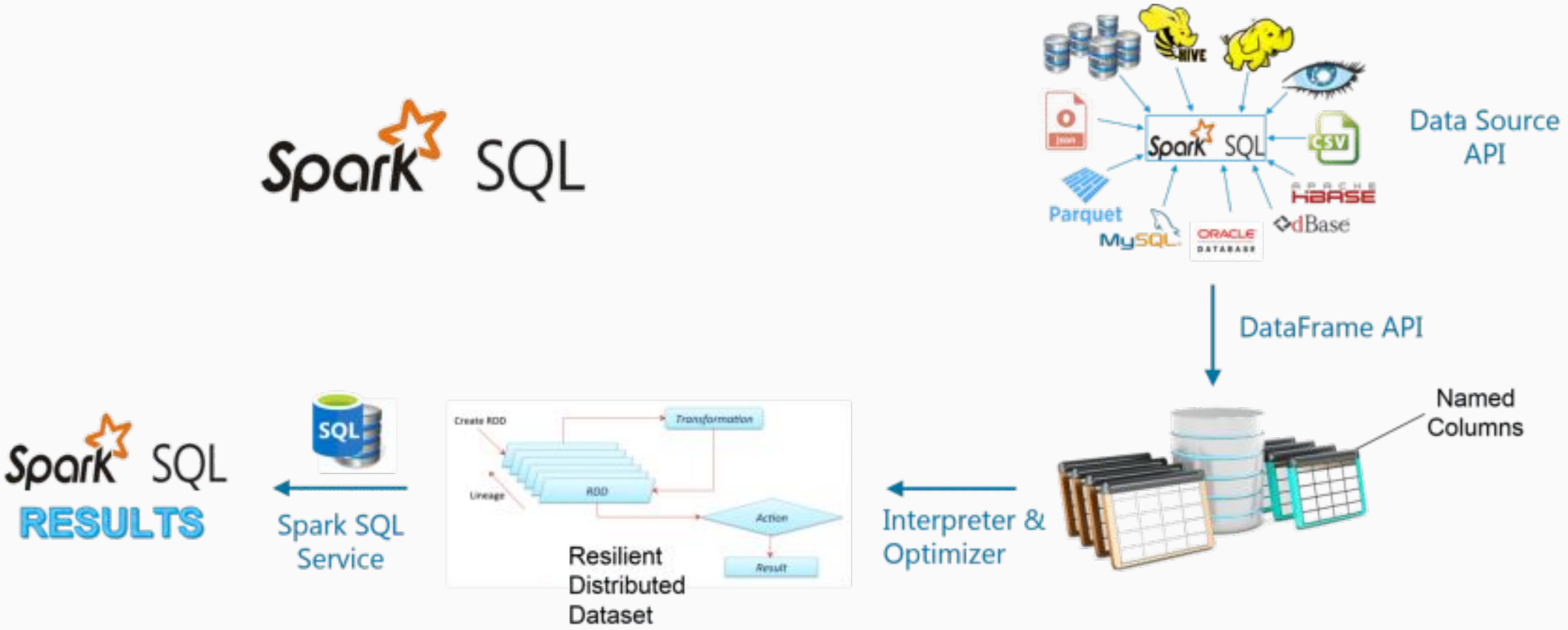
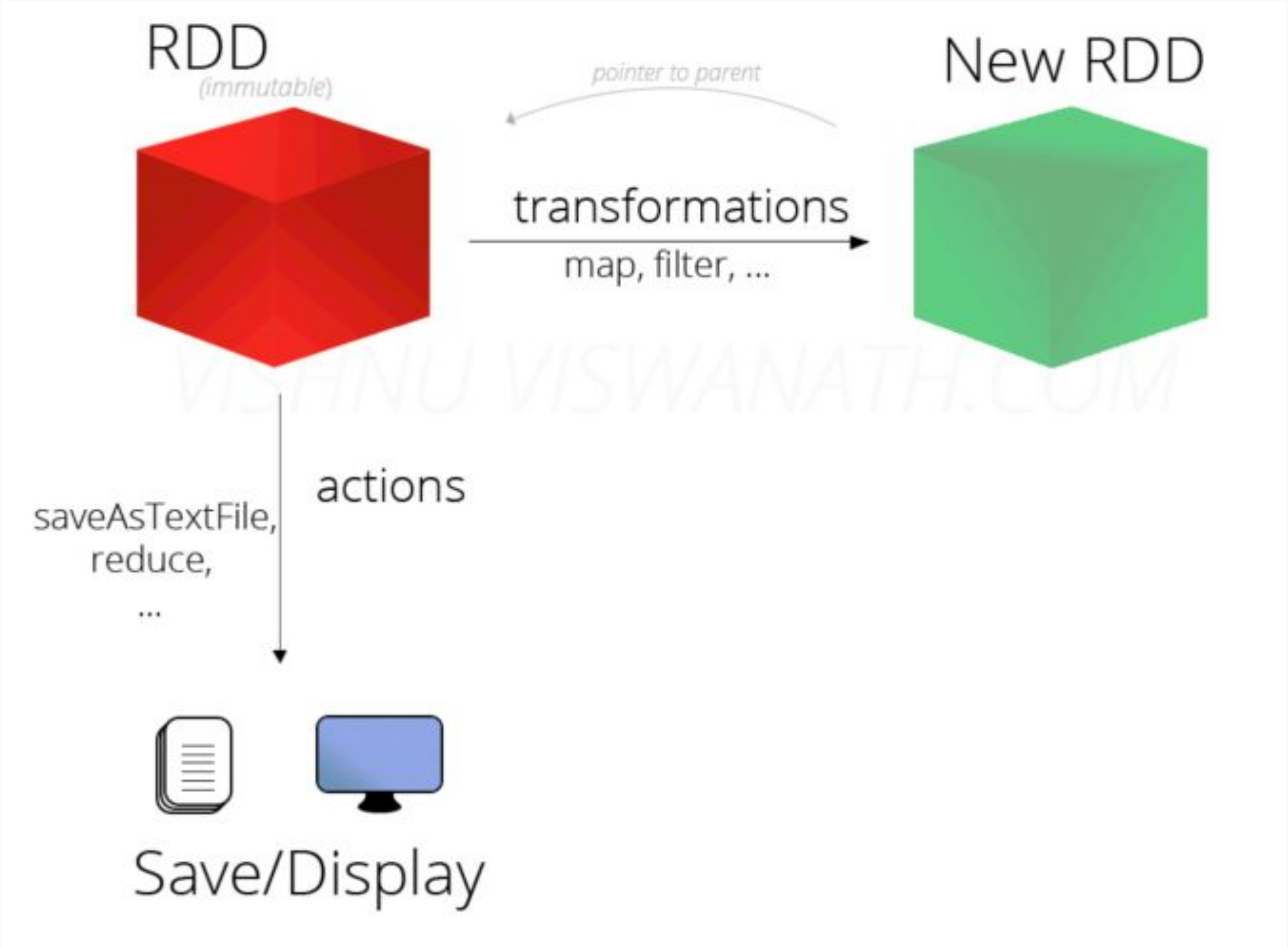
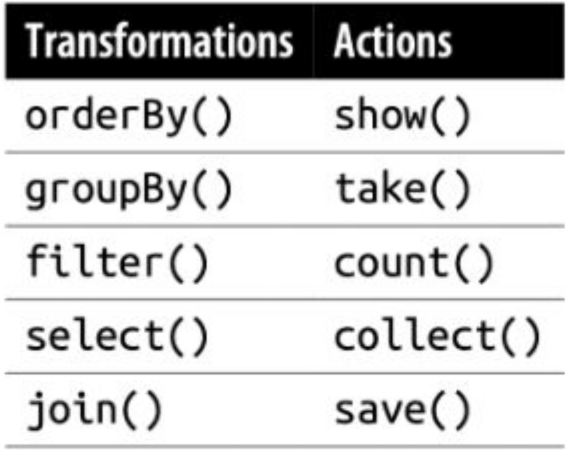


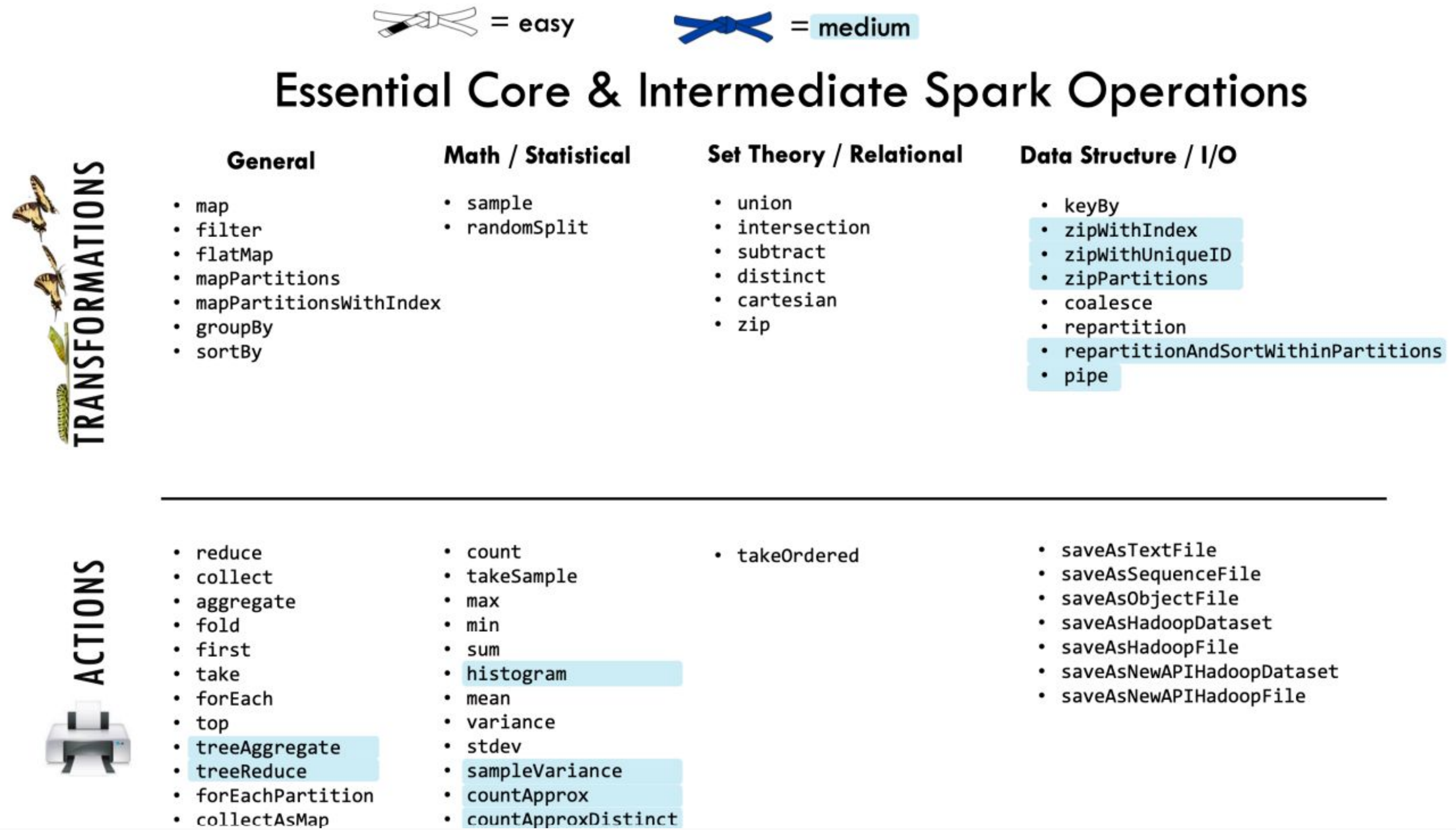
Figure: *The flow diagram represents a Spark SQL process using all the four libraries in sequence*


RDD
(immutable)
pointer to parent
New RDD
transformations
map, filter, ...
actions
saveAsTextFile,
reduce,
...
Save/Display



Transformations	Actions
orderBy()	show()
groupBy()	take()
filter()	count()
select()	collect()
join()	save()

RDD Transformations vs. actions



= easy

= medium

Essential Core & Intermediate Spark Operations

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

ACTIONS

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

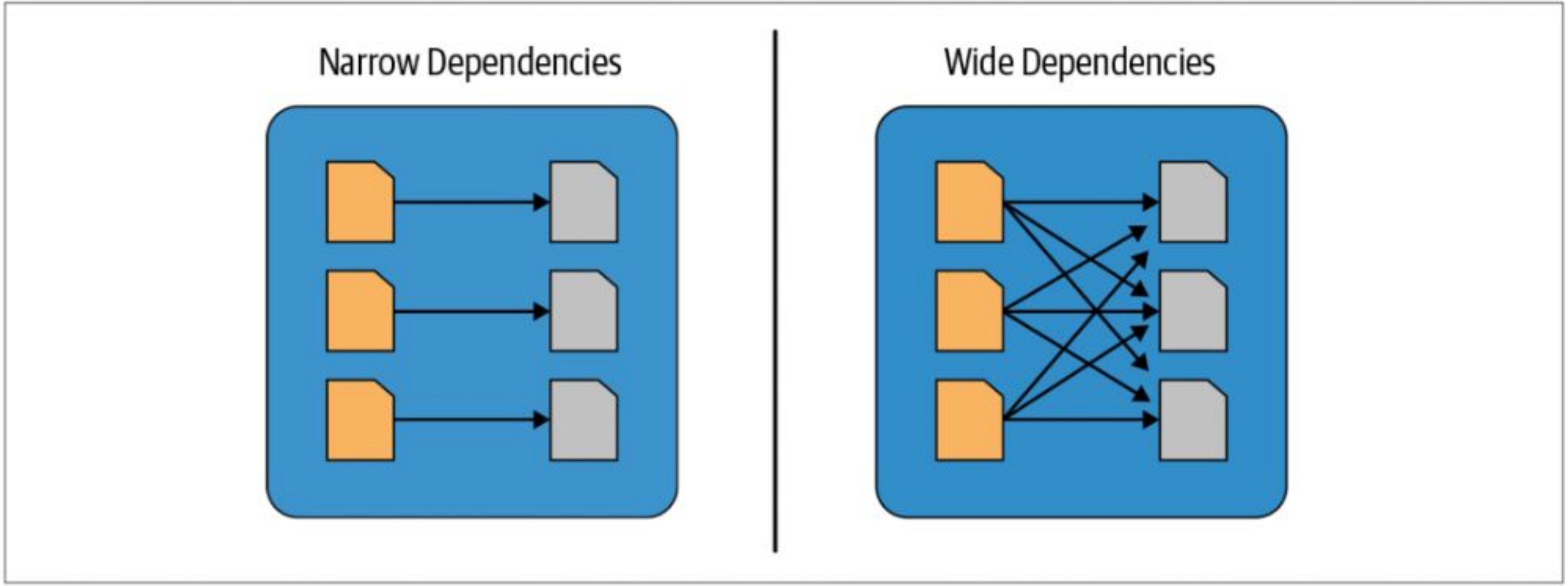
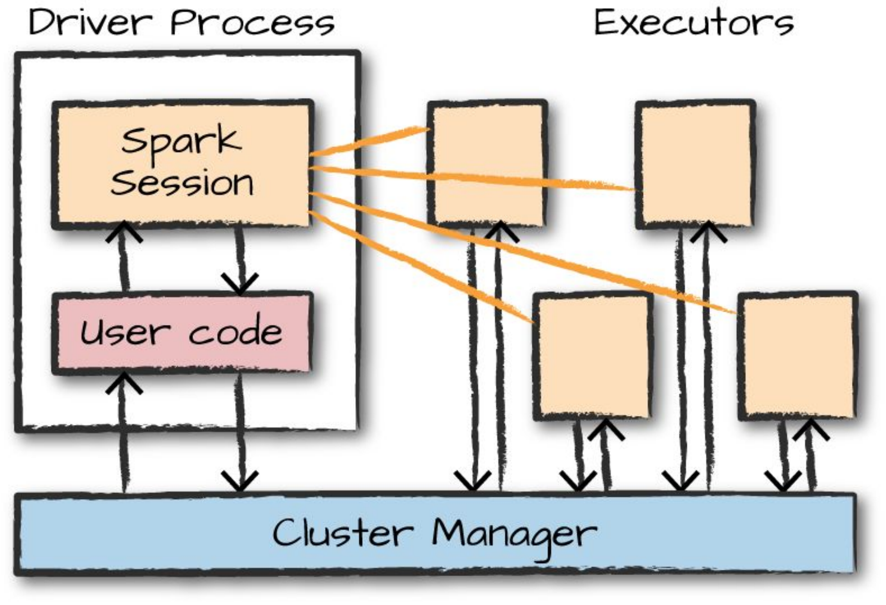


Figure 2-7. Narrow versus wide transformations

spark session

A SparkSession is a central entry point for working with Apache Spark, which is a powerful open-source distributed computing framework for big data processing. SparkSession was introduced in Apache Spark 2.0 to simplify the interaction with Spark and consolidate various Spark contexts (like SparkContext, SQLContext, and HiveContext) into a single unified interface.



why SparkSession is important

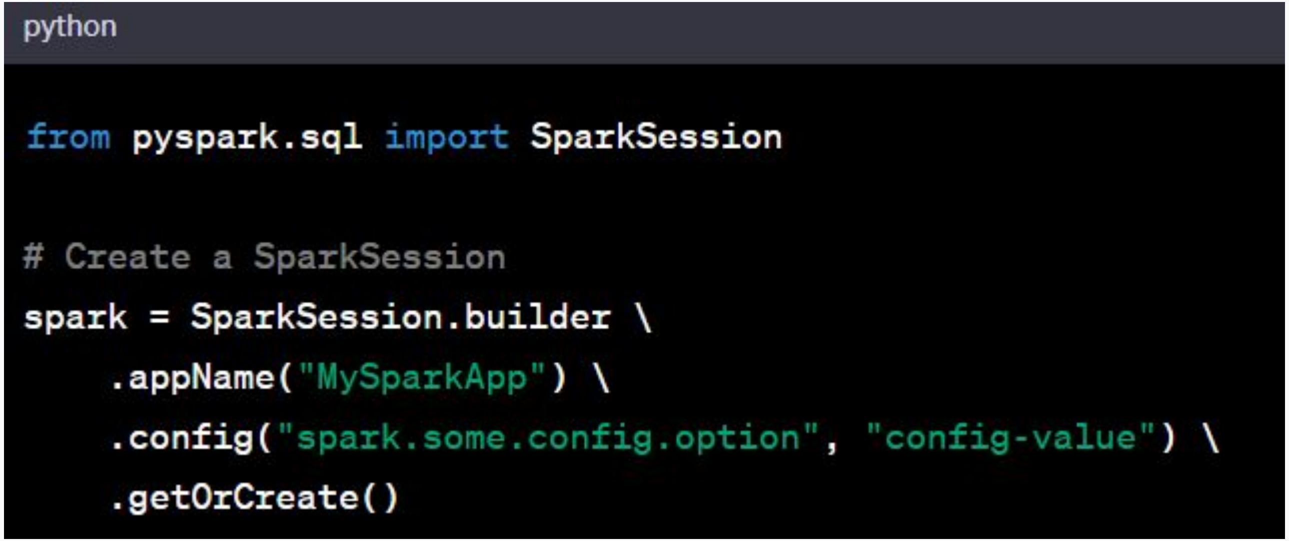
Unified Entry Point: SparkSession provides a single entry point for all Spark functionality, which makes it easier for developers to work with Spark. You don't need to create separate contexts for different Spark components anymore.

Simplified Code: SparkSession simplifies the code and reduces boilerplate. It automatically configures Spark and sets up various contexts, so you can focus on writing your data processing logic rather than managing different contexts and configurations.

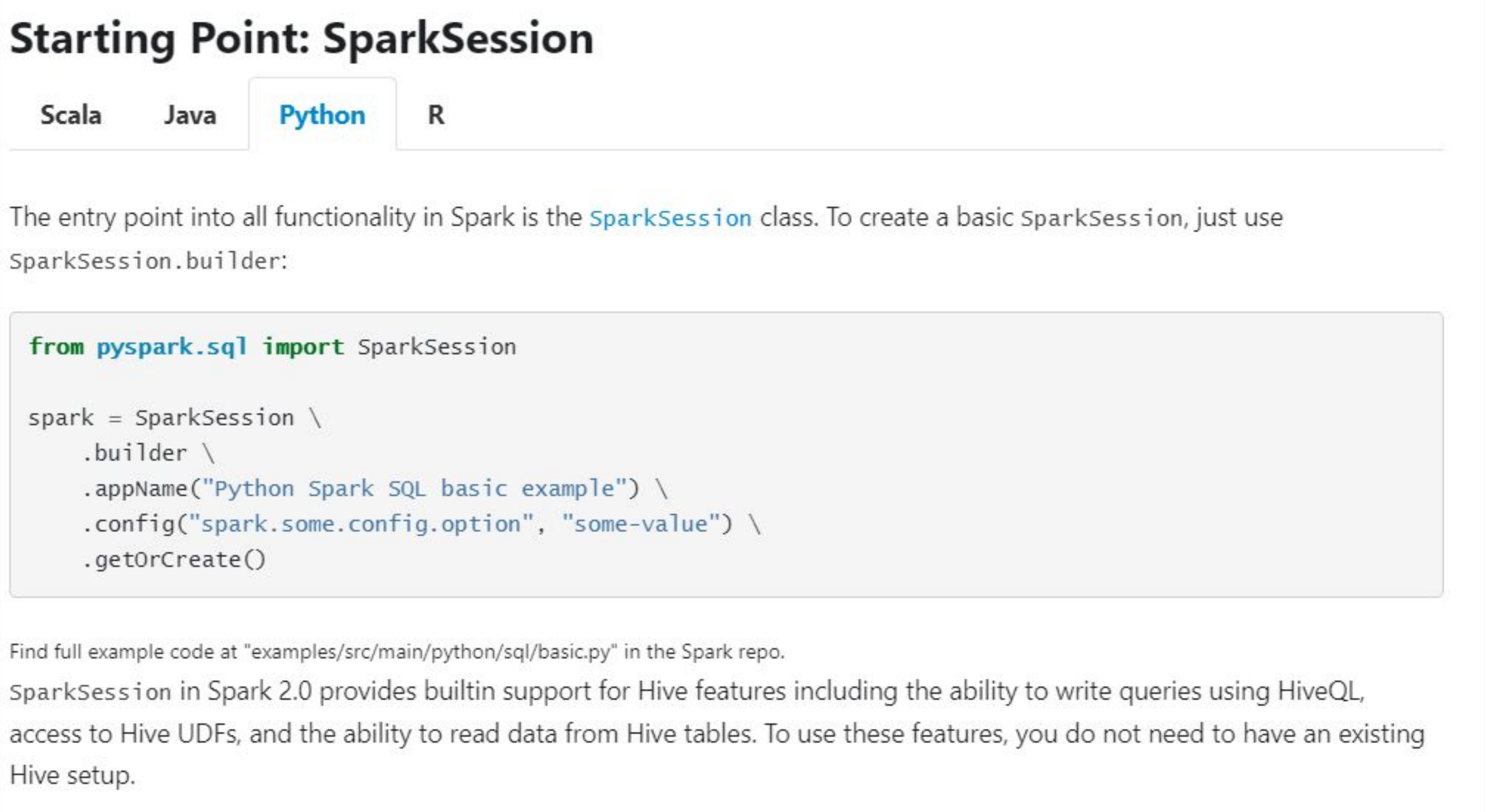
Integration with Spark Ecosystem: It provides seamless integration with various Spark components, including Spark SQL, DataFrame API, Dataset API, and structured streaming, allowing you to work with structured and semi-structured data efficiently.

Optimization: SparkSession includes optimizations for Spark's query engine. It can perform optimizations like predicate pushdown and column pruning to improve the performance of your Spark applications.

SQL Queries: With SparkSession, you can run SQL queries directly on your data using Spark SQL, making it easier to work with structured data and leverage SQL skills.



```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder \
    .appName("MySparkApp") \
    .config("spark.some.config.option", "config-value") \
    .getOrCreate()
```



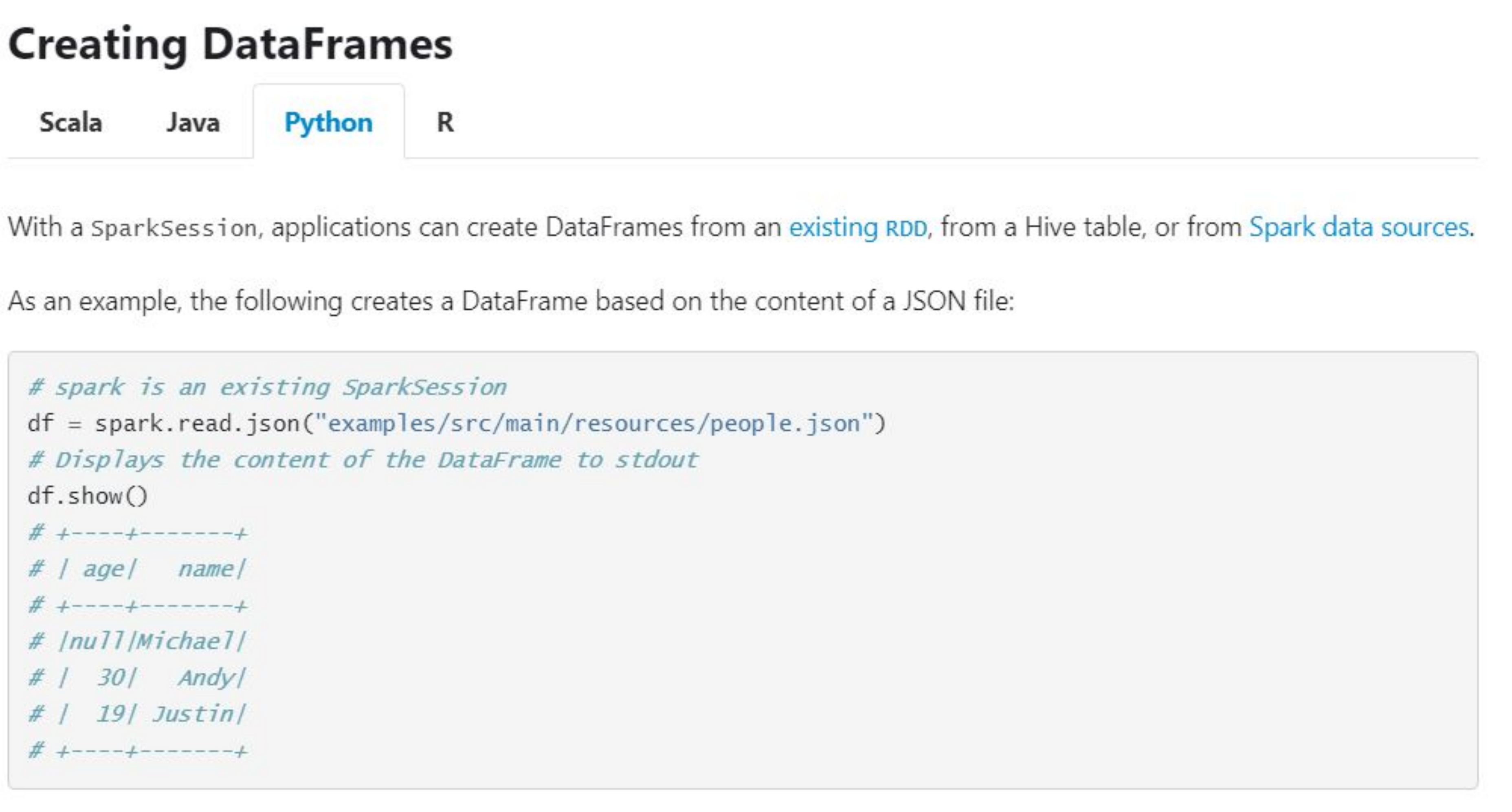
Starting Point: SparkSession

The entry point into all functionality in Spark is the `SparkSession` class. To create a basic `SparkSession`, just use `SparkSession.builder`:

```python
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Find full example code at "examples/src/main/python/sql/basic.py" in the Spark repo.

`SparkSession` in Spark 2.0 provides builtin support for Hive features including the ability to write queries using HiveQL, access to Hive UDFs, and the ability to read data from Hive tables. To use these features, you do not need to have an existing Hive setup.

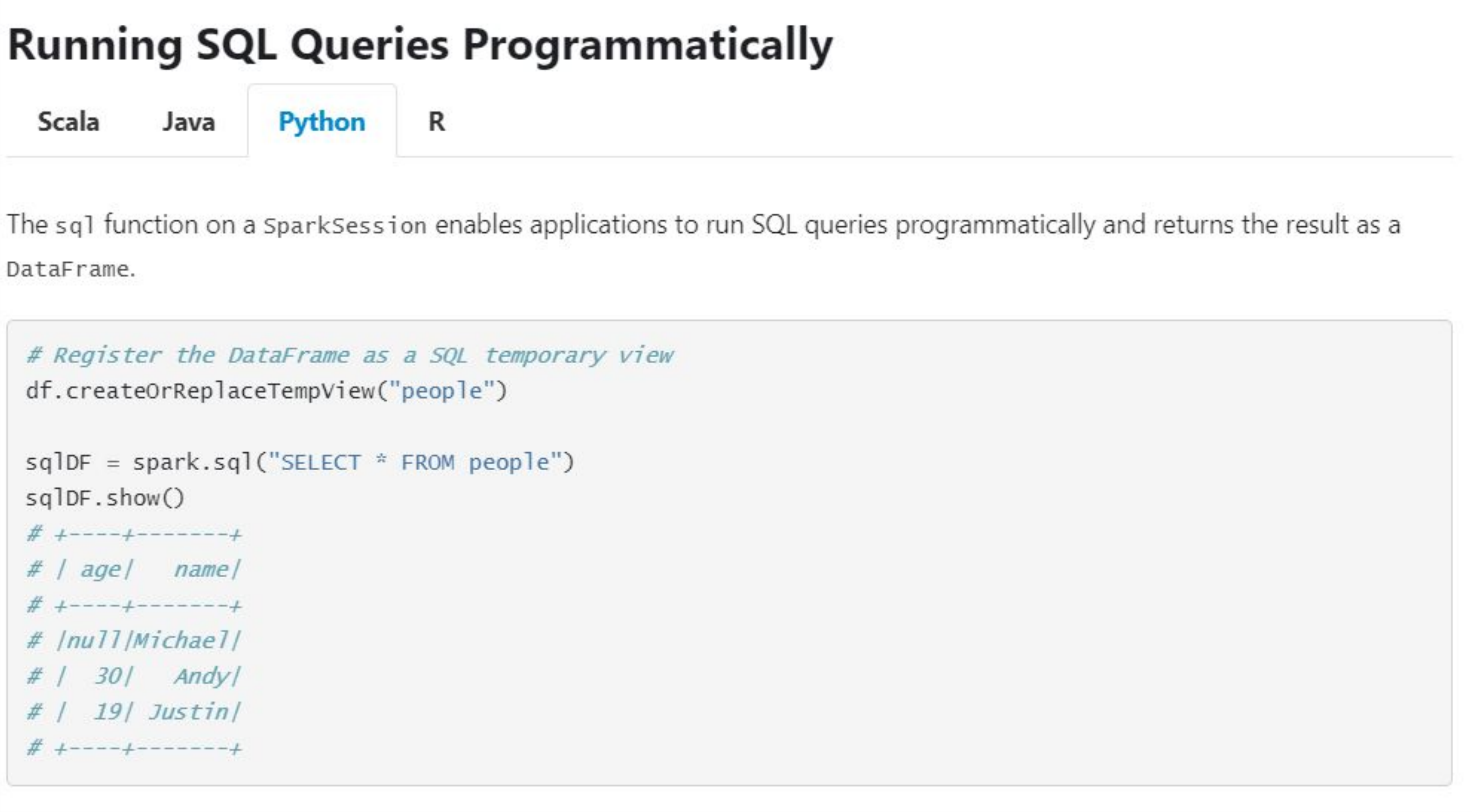


Creating DataFrames

With a `SparkSession`, applications can create DataFrames from an existing `RDD`, from a Hive table, or from Spark data sources.

As an example, the following creates a DataFrame based on the content of a JSON file:

```
# spark is an existing SparkSession
df = spark.read.json("examples/src/main/resources/people.json")
# Displays the content of the DataFrame to stdout
df.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```



Running SQL Queries Programmatically

The `sql` function on a `SparkSession` enables applications to run SQL queries programmatically and returns the result as a `DataFrame`.

```python
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

Purpose:

SparkContext (SC): SparkContext is the entry point for Spark and is mainly focused on low-level functionality, like creating RDDs (Resilient Distributed Datasets), scheduling tasks, and managing cluster resources. It's the foundation of Spark and is used in Spark applications for tasks like data loading and distributed data processing.

SparkSession (SS): SparkSession, on the other hand, is a higher-level abstraction introduced in Spark 2.0. It's designed for working with structured data, including structured streaming, Spark SQL, DataFrames, and Datasets. SparkSession provides a unified interface to access Spark's structured data APIs and handles many of the configuration details automatically.

Data Abstractions:

SC: SparkContext primarily deals with RDDs, which are a low-level data abstraction in Spark. RDDs are typically used for unstructured or semi-structured data.

SS: SparkSession focuses on structured data processing and provides high-level abstractions like DataFrames and Datasets, which are designed for structured and tabular data. It also integrates Spark SQL for SQL-like querying.

Configuration:

SC: Configuration in SparkContext requires more manual setup. You need to configure various Spark properties directly using methods like setAppName, setMaster, and setConf.

SS: SparkSession simplifies configuration by providing a higher-level API for setting properties. You use the config method to set Spark properties, and it automatically sets up sensible defaults for many configurations.

Streaming:

SC: If you want to work with structured streaming (real-time data processing) in Spark, you typically still need to create a SparkContext, but it's usually combined with a SparkSession for structured streaming operations.

SS: SparkSession is designed to seamlessly integrate with structured streaming. It simplifies the integration of batch and streaming operations in Spark applications.

Use Cases:

SC: SparkContext is more suitable for custom or low-level Spark applications where you need fine-grained control over RDDs or specific cluster configurations.

SS: SparkSession is ideal for most Spark applications, especially those involving structured data, as it simplifies the code, provides a unified interface for structured data operations, and abstracts away many low-level details.

Spark Architecture

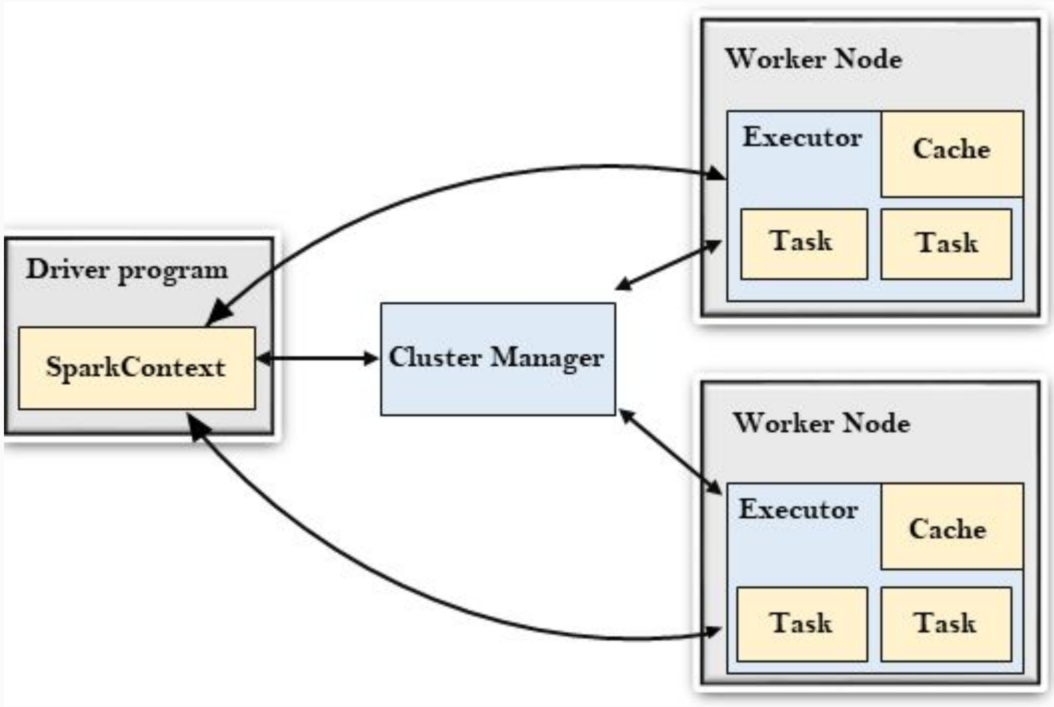
The Spark follows the master-slave architecture. Its cluster consists of a single master and multiple slaves.

The Spark architecture depends upon two abstractions:

Resilient Distributed Dataset (RDD)
Directed Acyclic Graph (DAG)

Directed Acyclic Graph (DAG)

Directed Acyclic Graph is a finite direct graph that performs a sequence of computations on data. Each node is an RDD partition, and the edge is a transformation on top of data. Here, the graph refers the navigation whereas directed and acyclic refers to how it is done.


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

Driver Program

The Driver Program is a process that runs the main() function of the application and creates the SparkContext object. The purpose of SparkContext is to coordinate the spark applications, running as independent sets of processes on a cluster.

To run on a cluster, the SparkContext connects to a different type of cluster managers and then perform the following tasks: -

- It acquires executors on nodes in the cluster.
- Then, it sends your application code to the executors. Here, the application code can be defined by JAR or Python files passed to the SparkContext.
- At last, the SparkContext sends tasks to the executors to run.

Cluster Manager

- The role of the cluster manager is to allocate resources across applications. The Spark is capable enough of running on a large number of clusters.
- It consists of various types of cluster managers such as Hadoop YARN, Apache Mesos and Standalone Scheduler.
- Here, the Standalone Scheduler is a standalone spark cluster manager that facilitates to install Spark on an empty set of machines.

Worker Node

- The worker node is a slave node
- Its role is to run the application code in the cluster.

Executor

- An executor is a process launched for an application on a worker node.
- It runs tasks and keeps data in memory or disk storage across them.
- It read and write data to the external sources.
- Every application contains its executor.

Task

- A unit of work that will be sent to one executor.

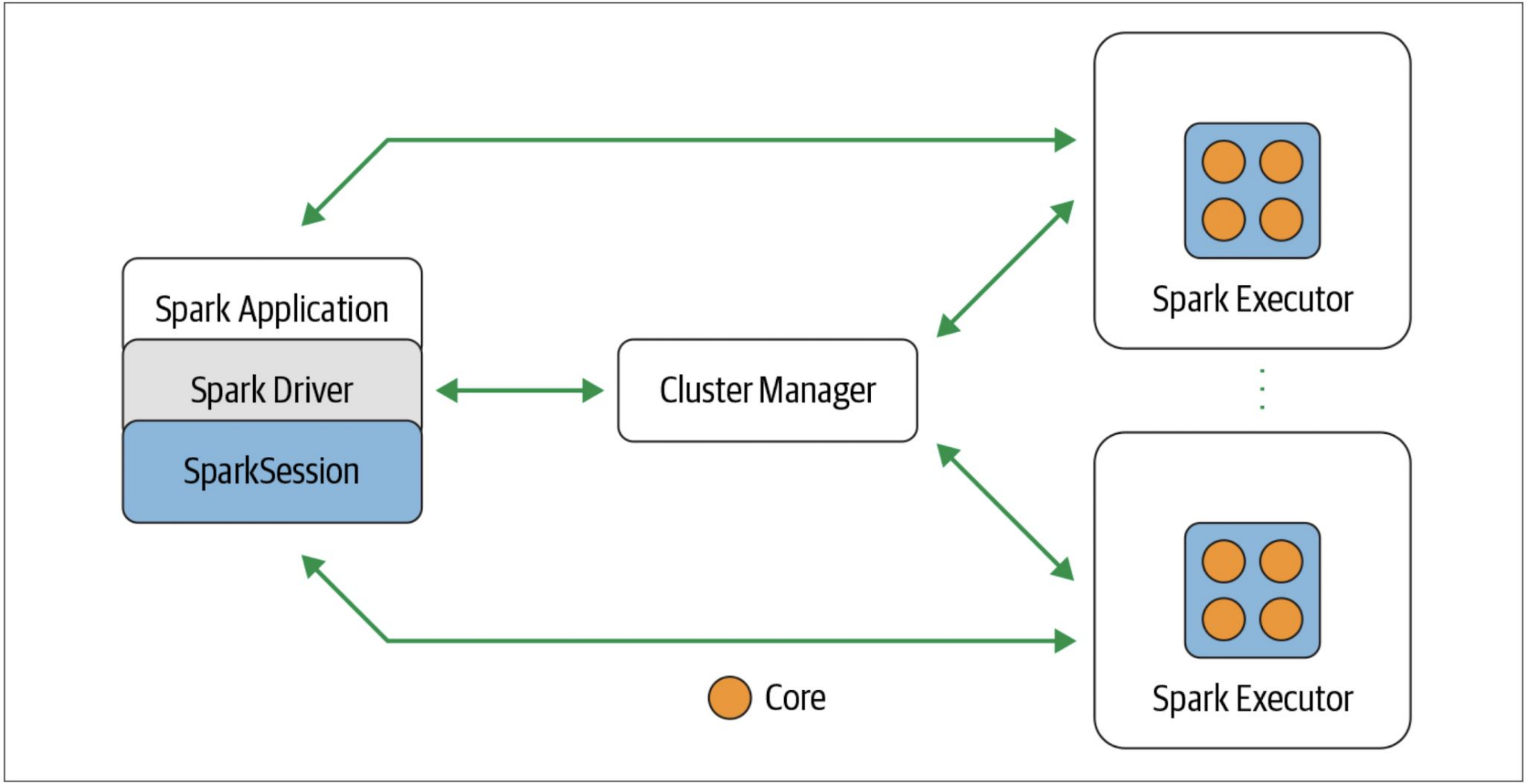


Figure 1-4. Apache Spark components and architecture

Distributed data and partitions

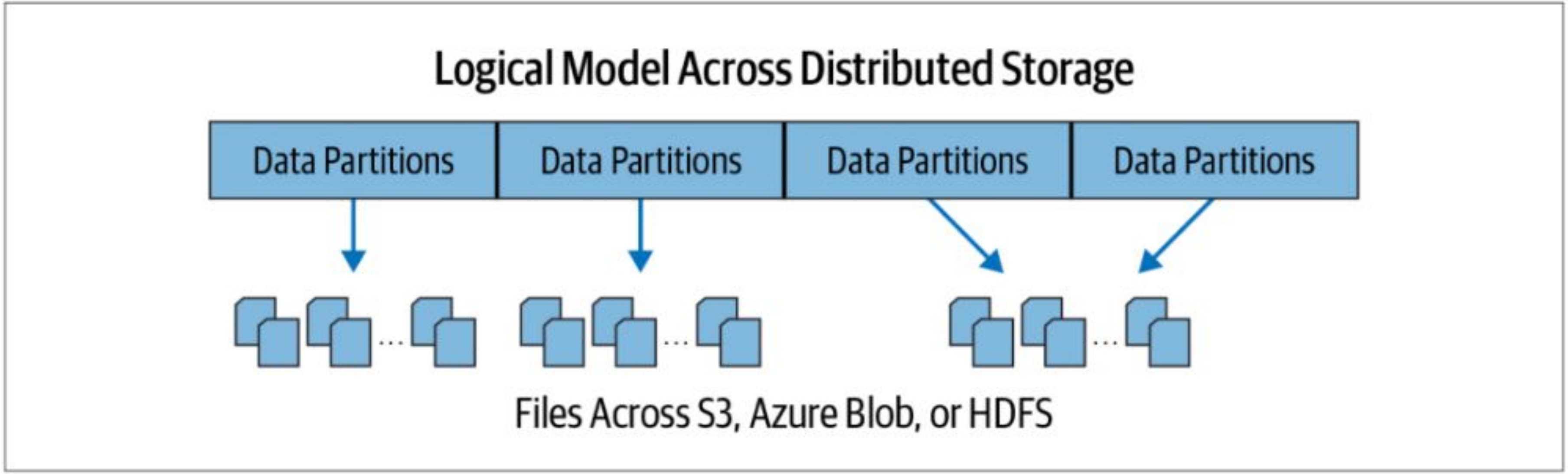

Logical Model Across Distributed Storage
Data Partitions
Data Partitions
Data Partitions
Data Partitions
Files Across S3, Azure Blob, or HDFS

Figure 1-5. Data is distributed across physical machines

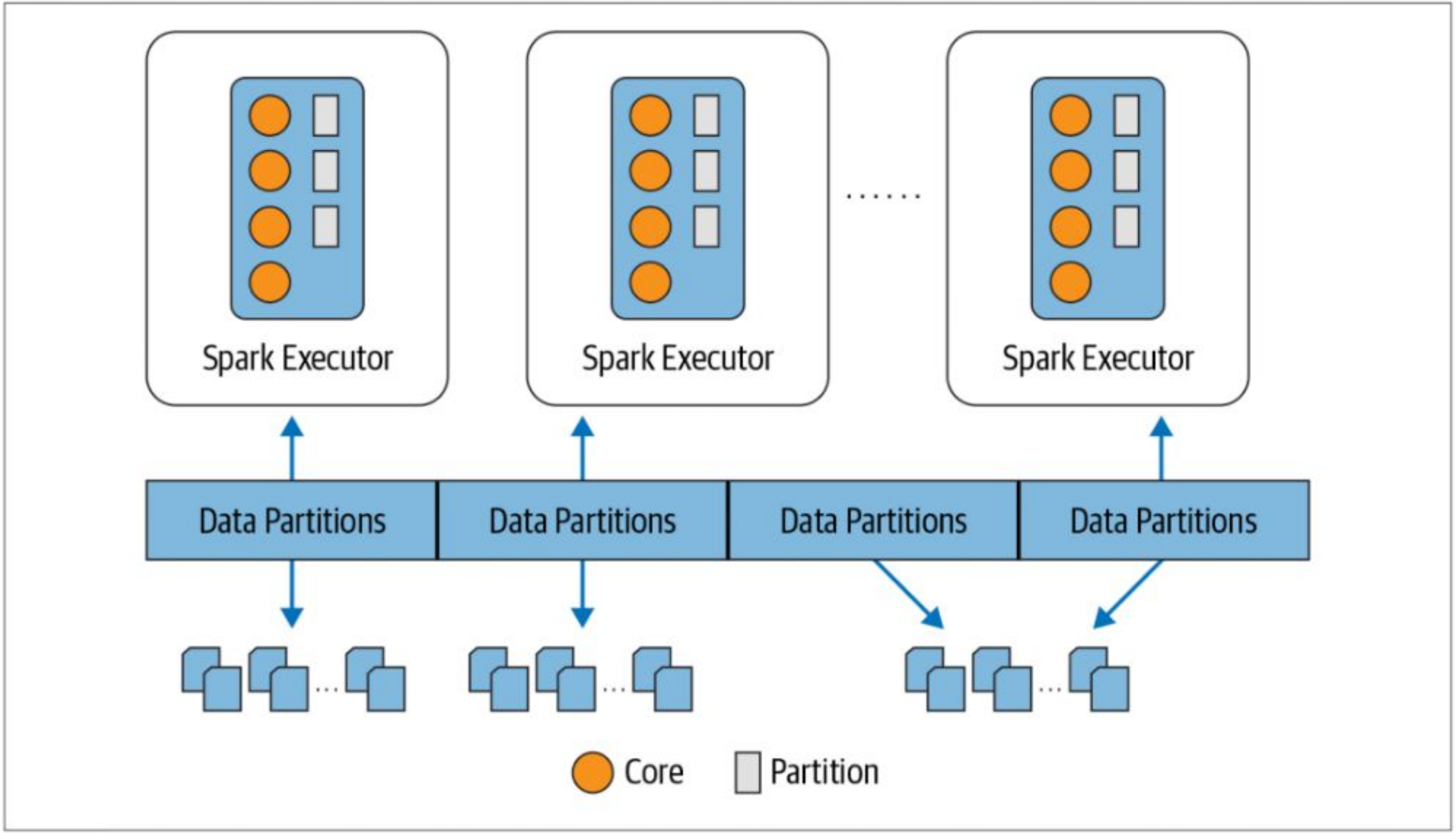


Figure 1-6. Each executor's core gets a partition of data to work on

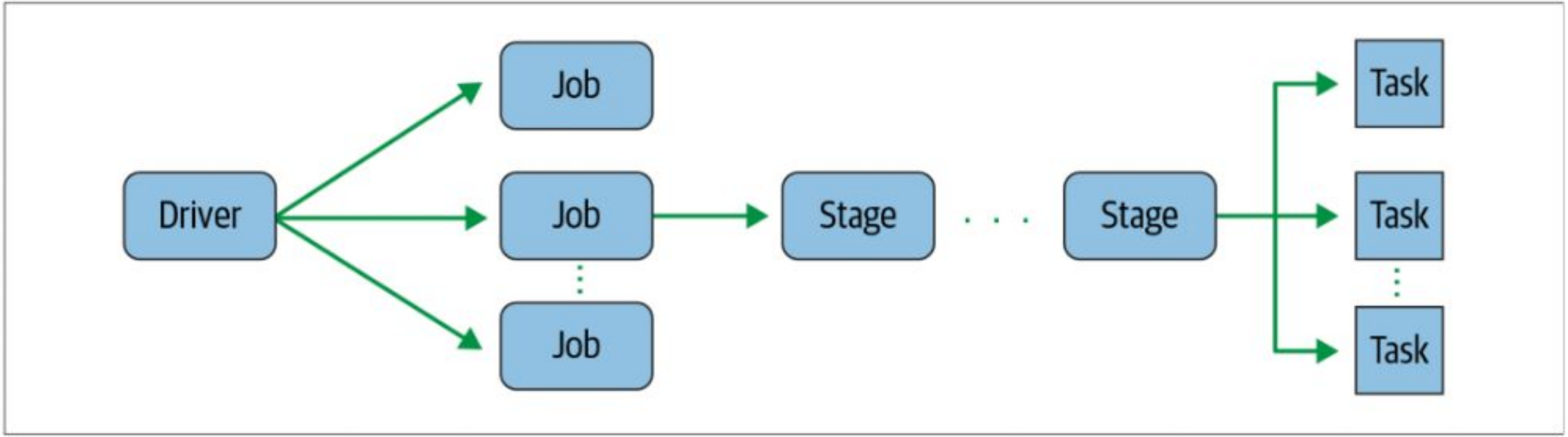


Figure 2-5. Spark stage creating one or more tasks to be distributed to executors

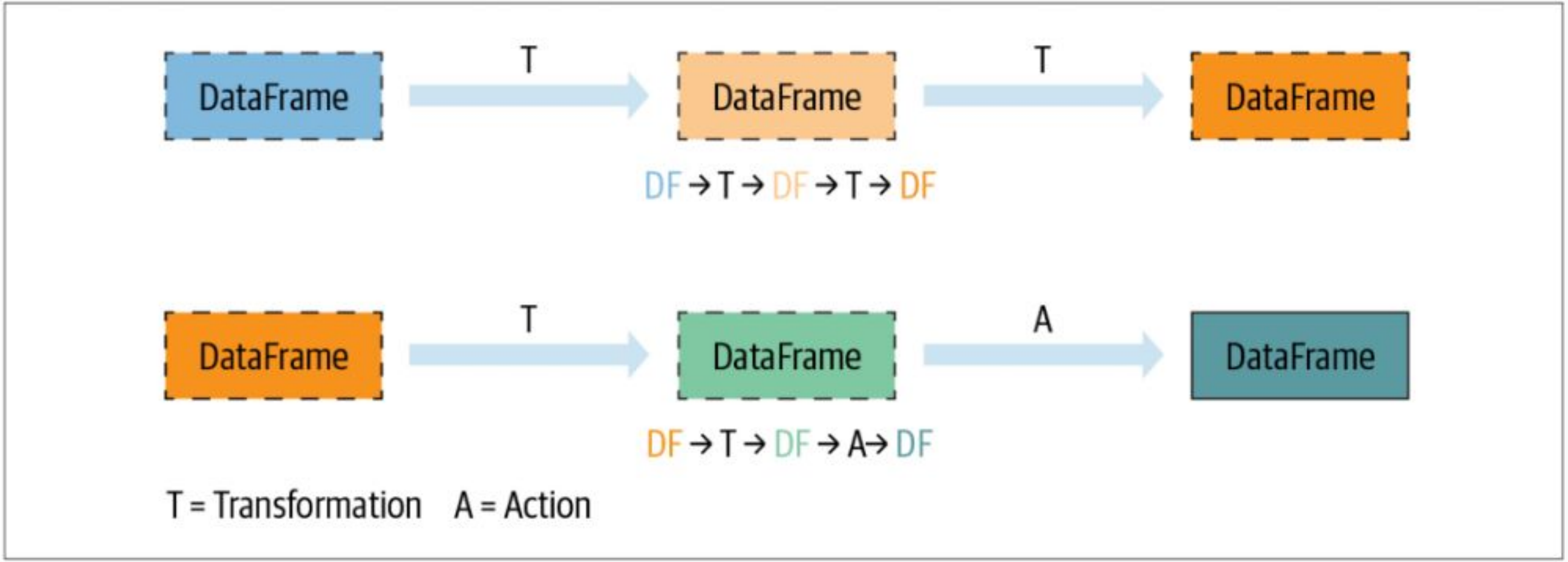
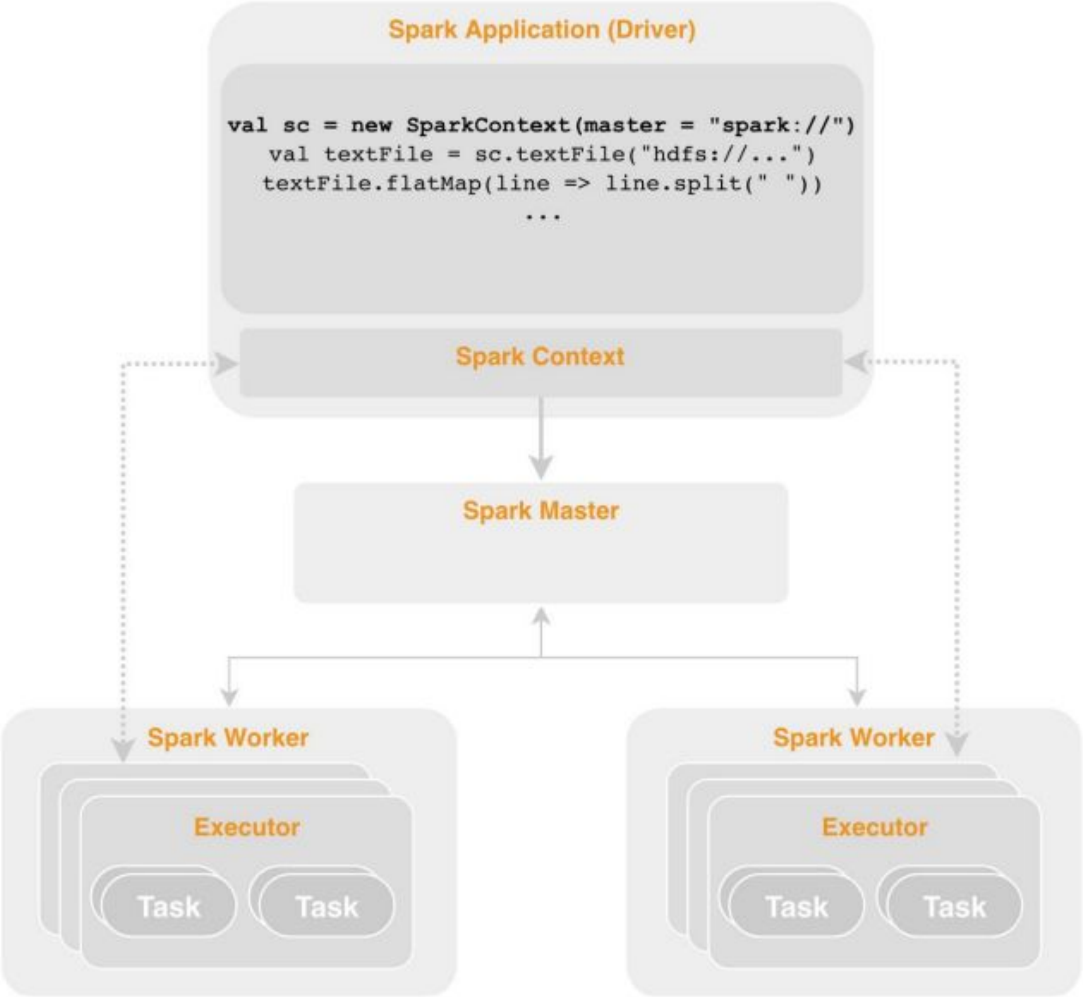


Figure 2-6. Lazy transformations and eager actions



Spark Application (Driver)

```
val sc = new SparkContext(master = "spark://")
  val textFile = sc.textFile("hdfs://...")
  textFile.flatMap(line => line.split(" "))
                    ...
```

Spark Context

Spark Master

Spark Worker

Executor

Task

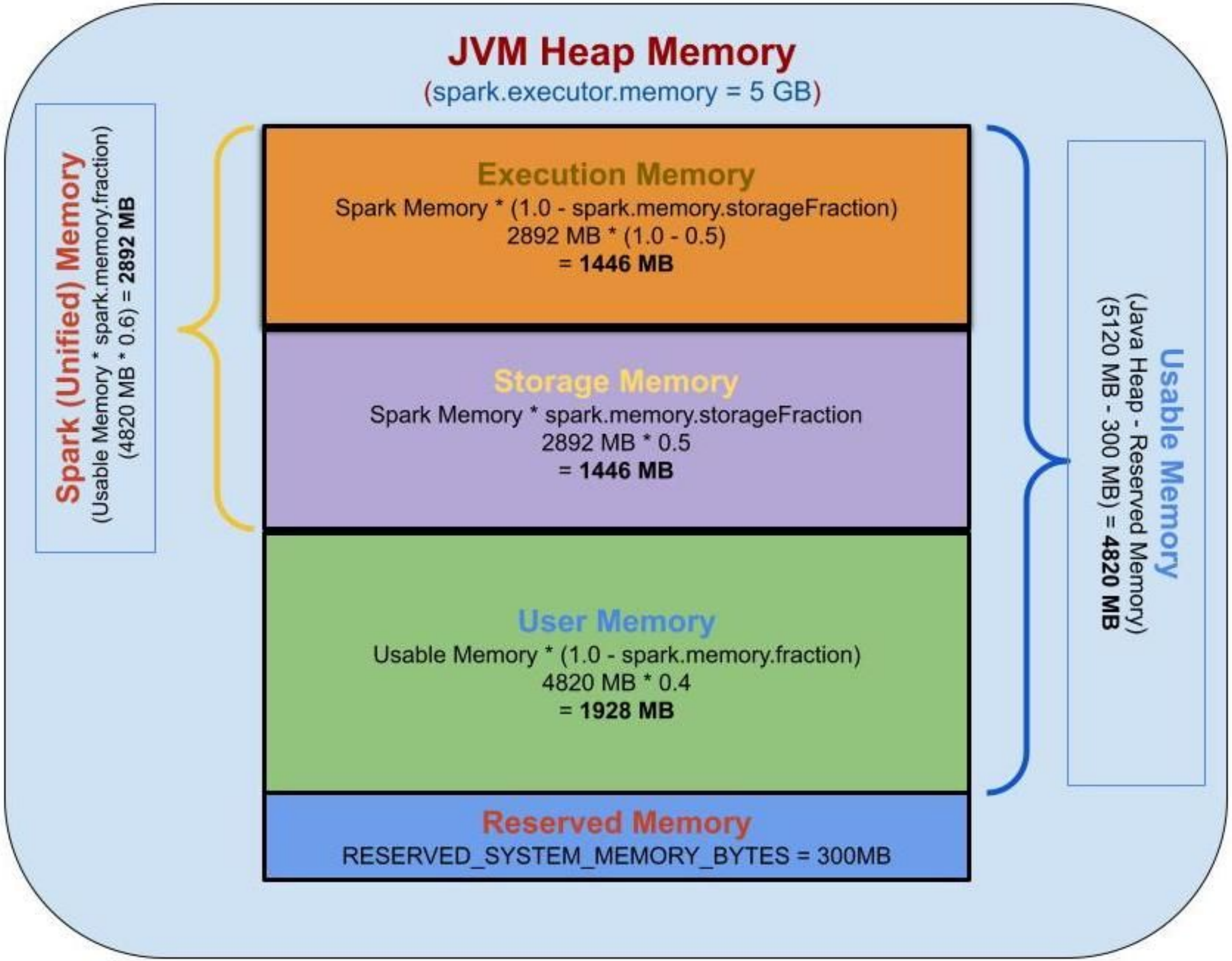
Task

Spark Worker

Executor

Task

Task



JVM Heap Memory

(spark.executor.memory = 5 GB)

Spark (Unified) Memory

(Usable Memory * spark.memory.fraction)

(4820 MB * 0.6) = **2892 MB**

Execution Memory

Spark Memory * (1.0 - spark.memory.storageFraction)

2892 MB * (1.0 - 0.5)

= 1446 MB

Storage Memory

Spark Memory * spark.memory.storageFraction

2892 MB * 0.5

= 1446 MB

User Memory

Usable Memory * (1.0 - spark.memory.fraction)

4820 MB * 0.4

= 1928 MB

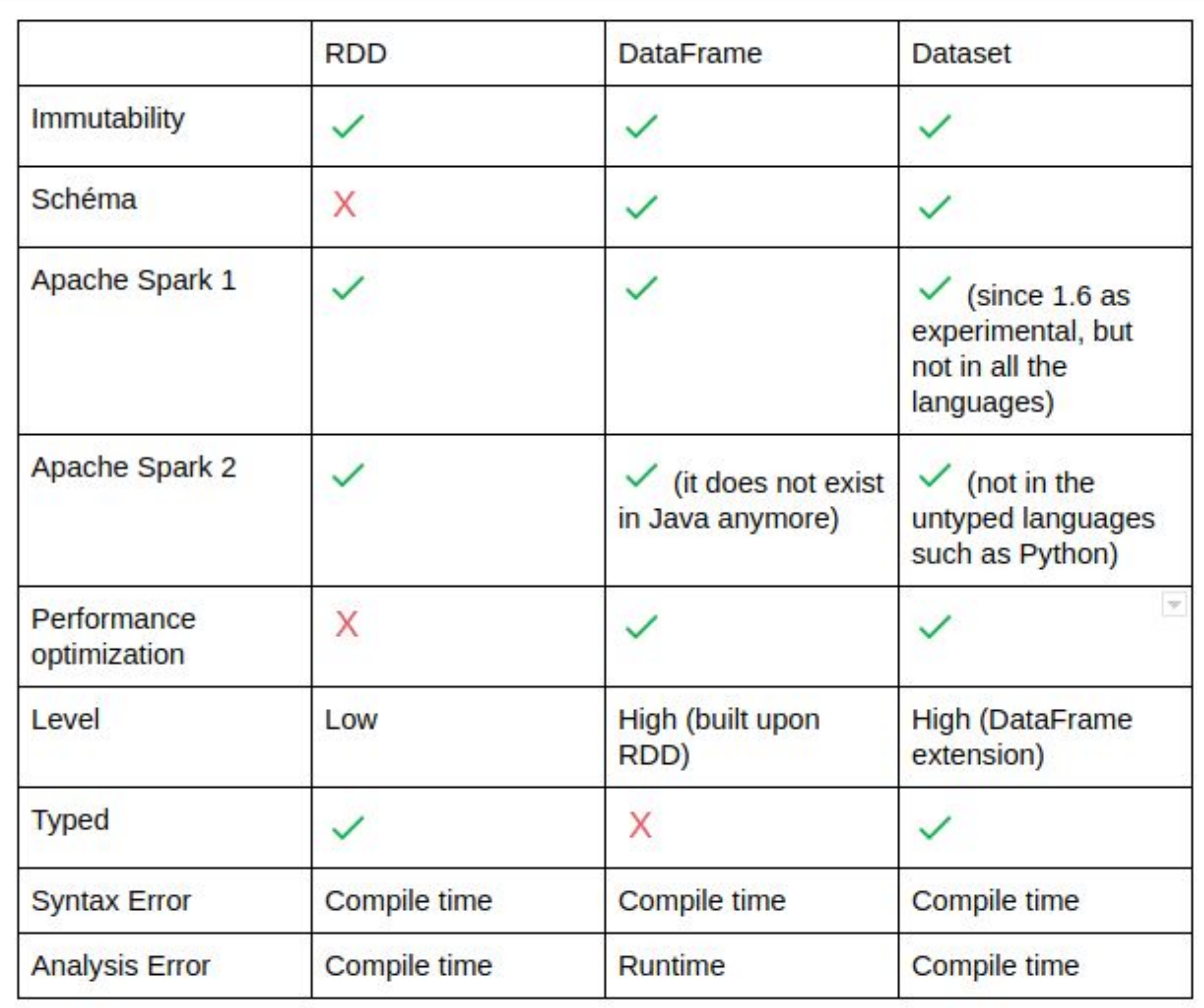
Reserved Memory

RESERVED_SYSTEM_MEMORY_BYTES = 300MB

Usable Memory

(Java Heap - Reserved Memory)

(5120 MB - 300 MB) = **4820 MB**



	RDD	DataFrame	Dataset
Immutability	✓	✓	✓
Schéma	X	✓	✓
Apache Spark 1	✓	✓	✓ (since 1.6 as experimental, but not in all the languages)
Apache Spark 2	✓	✓ (it does not exist in Java anymore)	✓ (not in the untyped languages such as Python)
Performance optimization	X	✓	✓
Level	Low	High (built upon RDD)	High (DataFrame extension)
Typed	✓	X	✓
Syntax Error	Compile time	Compile time	Compile time
Analysis Error	Compile time	Runtime	Compile time