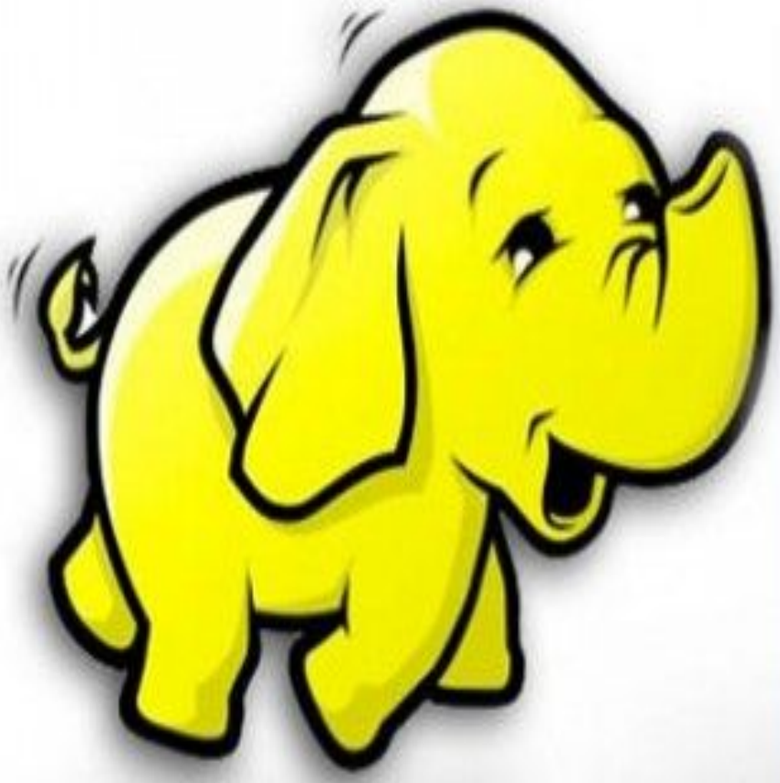
Spark Frameworks and Architecture



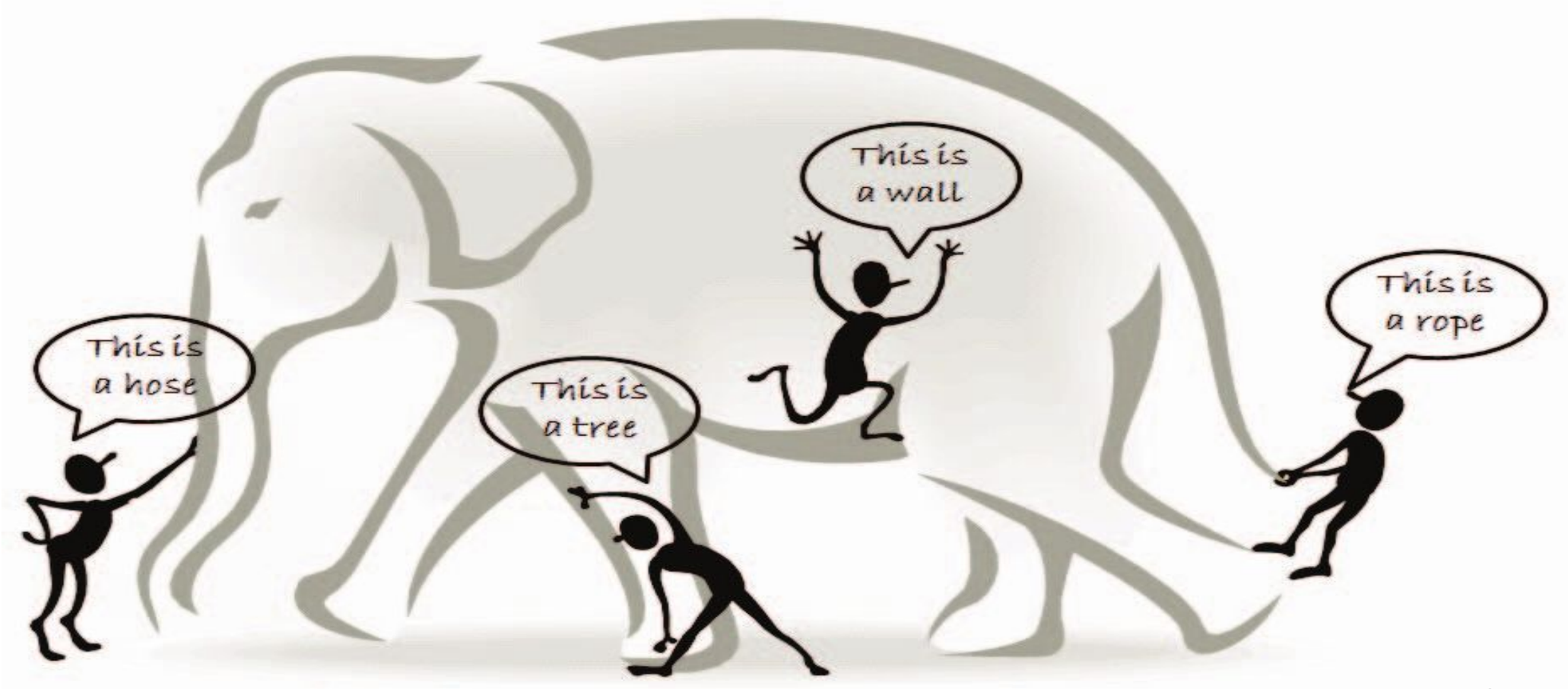
Jai Prakash Verma

Agenda

- Big Data
- Big Data Analytics: Open Source Solutions
- Introdution: Spark
- Spark Vs MapReduce
- Spark Essentials
- Spark Architecture
- Spark Components
- GraphFrames
- Advanced Spark Programming

Big Data Introduction

- The localized (limited) view of each blind man leads to a biased conclusion
- Big Data (Volume, Velocity, Variety, Value)
-

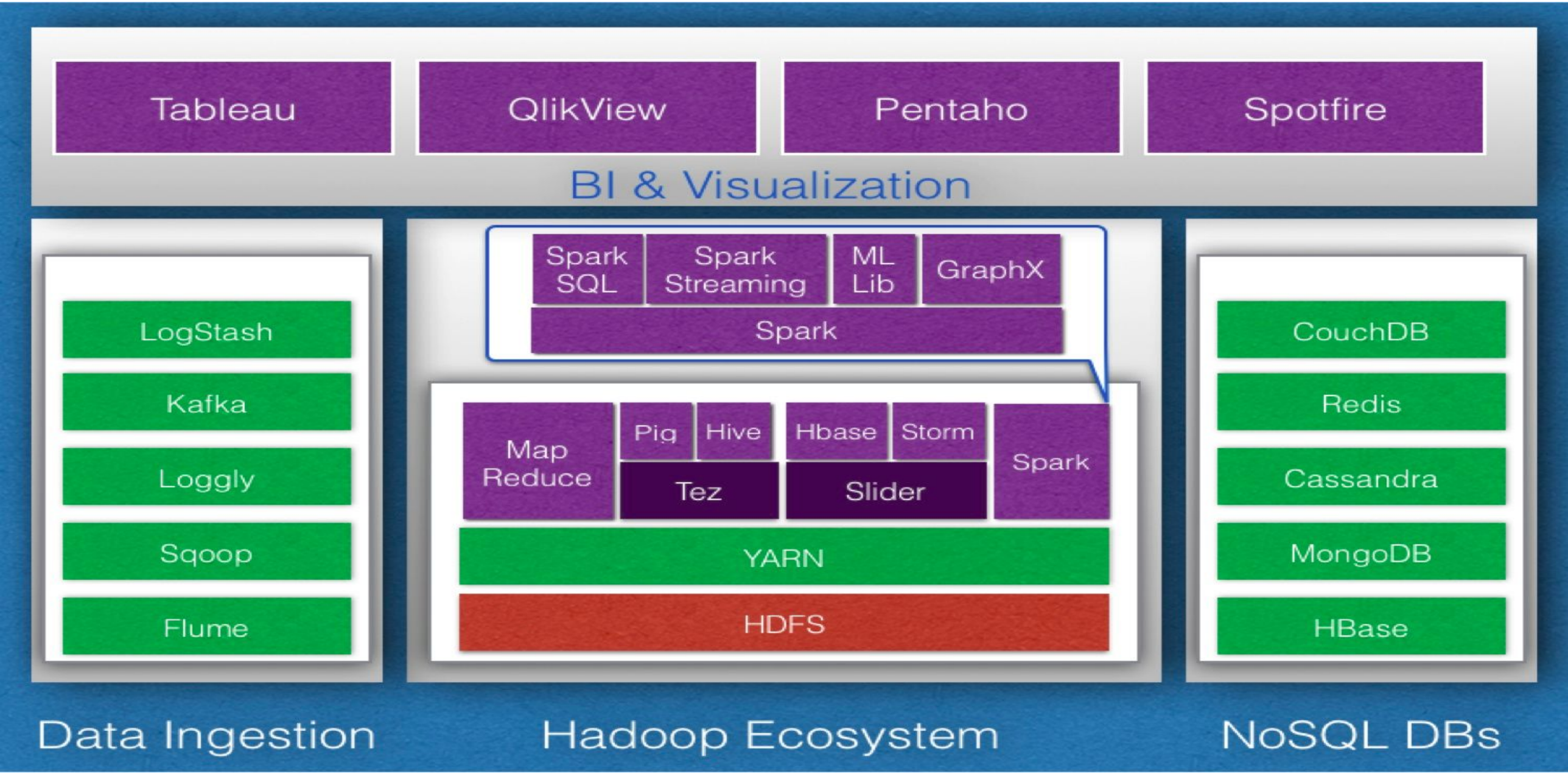


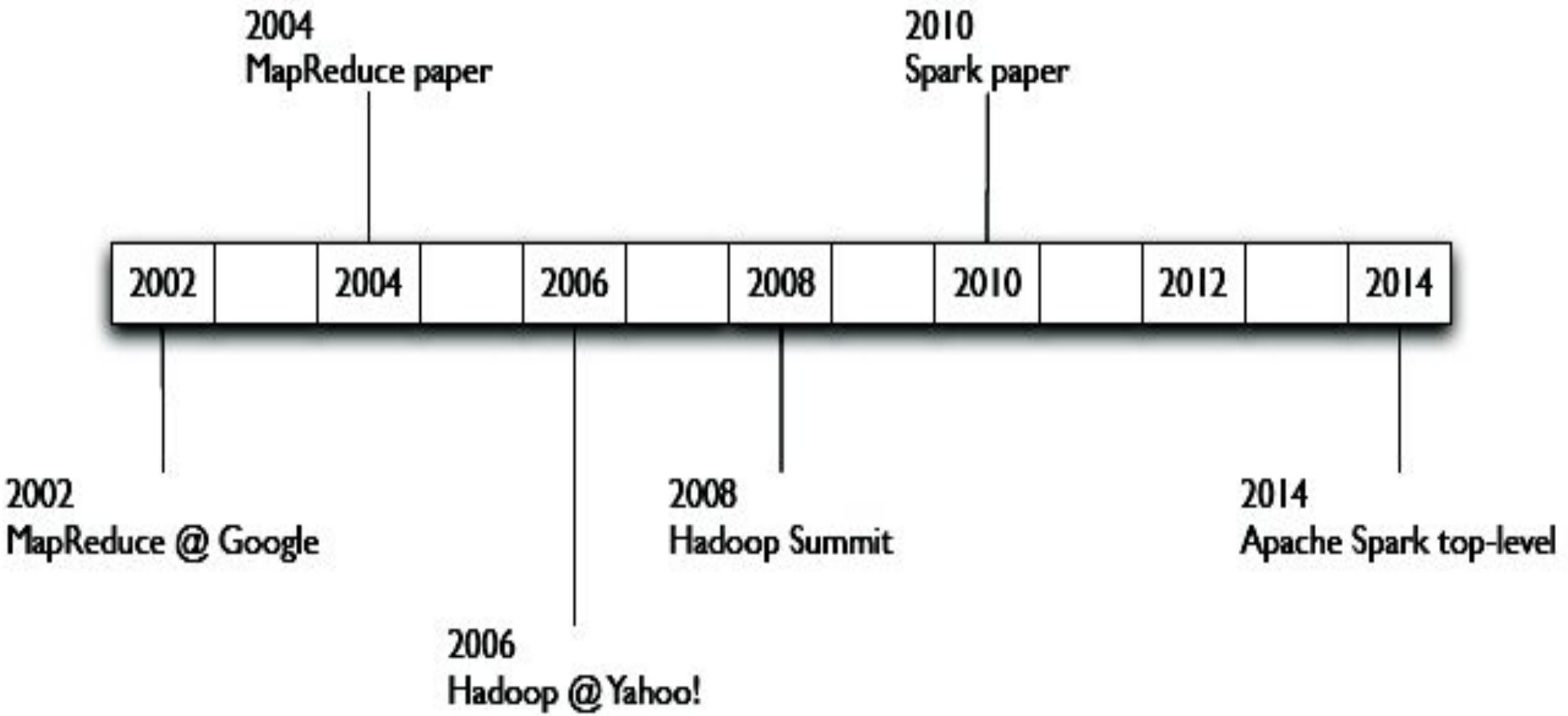
Big Data in five domains

- Health care
- Public sector administration
- Retail Transaction Data
- Global manufacturing
- Personal location data
-

- James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, Angela Hung Byers, Big data: The Next Frontier for Innovation, Competition, and Productivity, McKinsey Global Institute, 2012.

Big Data Analytics: Open Source Solutions




2004
MapReduce paper
2010
Spark paper
2002
2004
2006
2008
2010
2012
2014
2002
MapReduce @ Google
2006
Hadoop @ Yahoo!
2008
Hadoop Summit
2014
Apache Spark top-level

- Apache Spark is an open source big data processing framework built around speed, ease of use, and sophisticated analytics. It was originally developed in 2009 in UC Berkeley's AMPLab, and open sourced in 2010 as an Apache project.
-
- Spark as an alternative to Hadoop MapReduce rather than a replacement to Hadoop. It's not intended to replace Hadoop but to provide a comprehensive and unified solution to manage different big data use cases and requirements.

Spark - Introduction

- Not a modified version of Hadoop
- Separate, fast, MapReduce-like engine

–In-memory data storage for very fast iterative queries

–General execution graphs and powerful optimizations

–Up to 40x faster than Hadoop MapReduce

- Compatible with Hadoop's storage APIs

–Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, etc

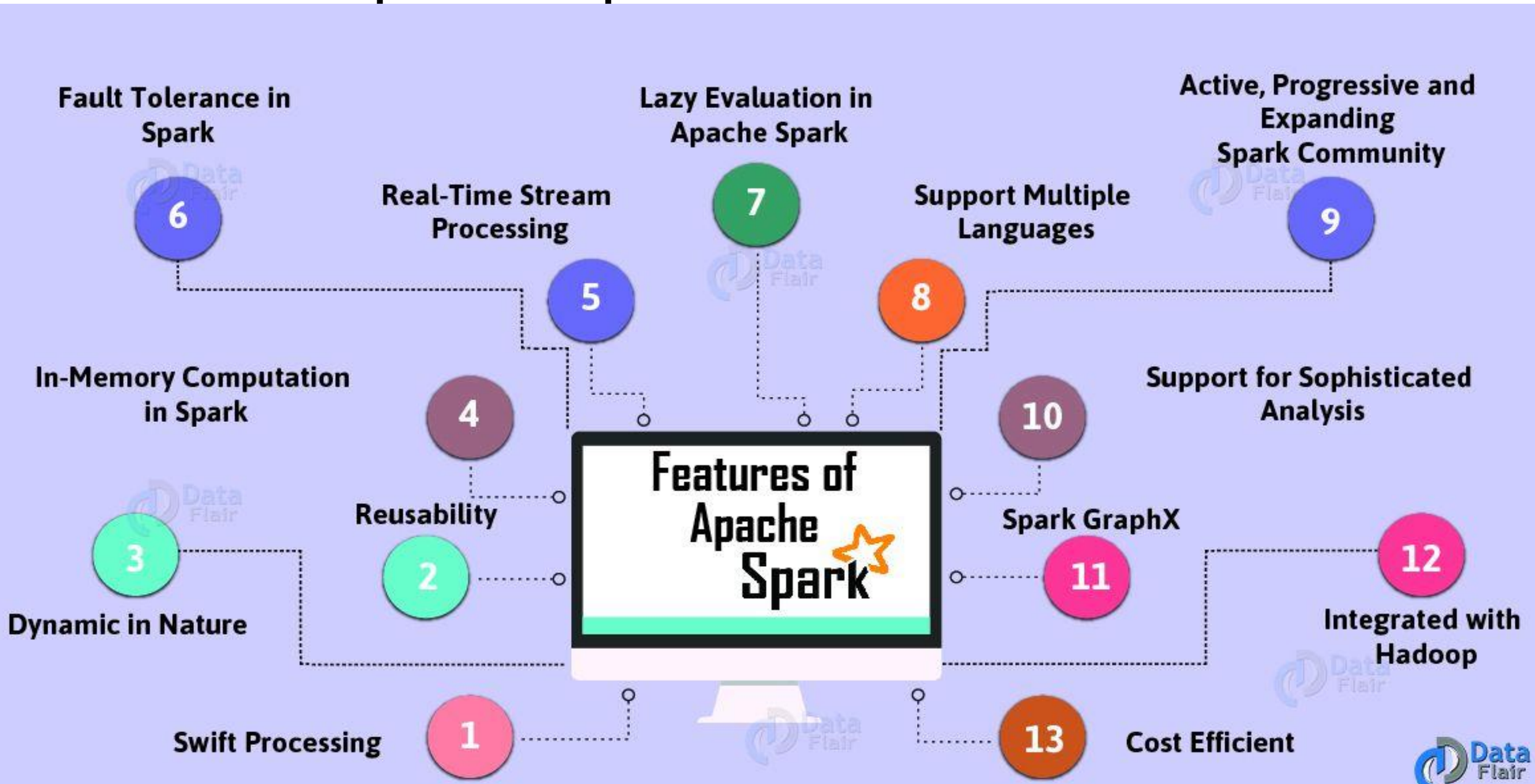
Spark - Introduction

- Spark is written in Scala Programming Language and runs on Java Virtual Machine (JVM) environment.
- Currently supports the following languages for developing applications:
  - Scala
  - Java
  - Python
  - Clojure
  - R

Spark Vs Hadoop

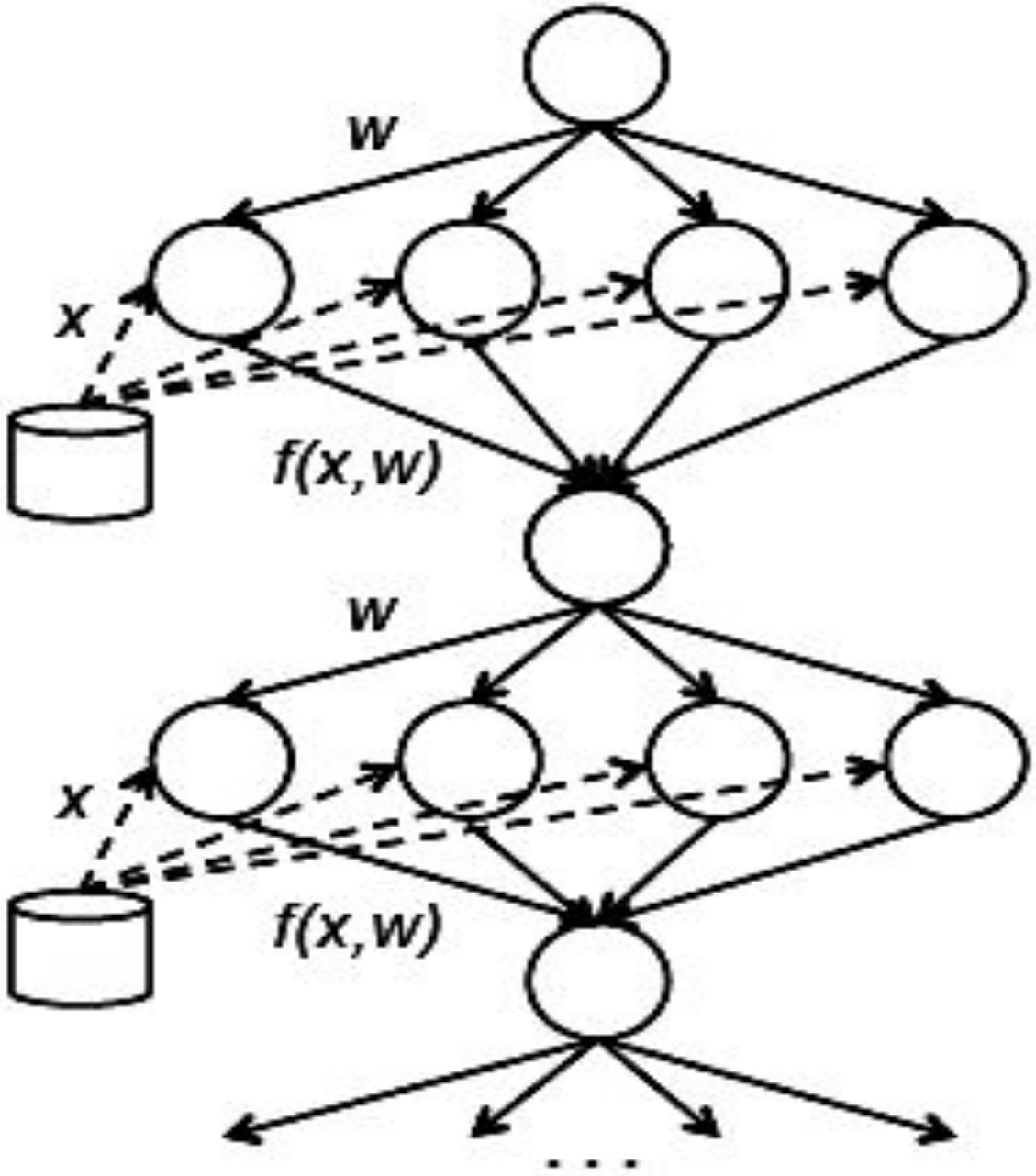
- Spark is not a modified version of Hadoop and is not, really, dependent on Hadoop because it has its own cluster management.
- Spark uses Hadoop in two ways – one is storage and second is processing. Since Spark has its own cluster management computation, it uses Hadoop for storage purpose only.
- The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application.

Features of Apache Spark

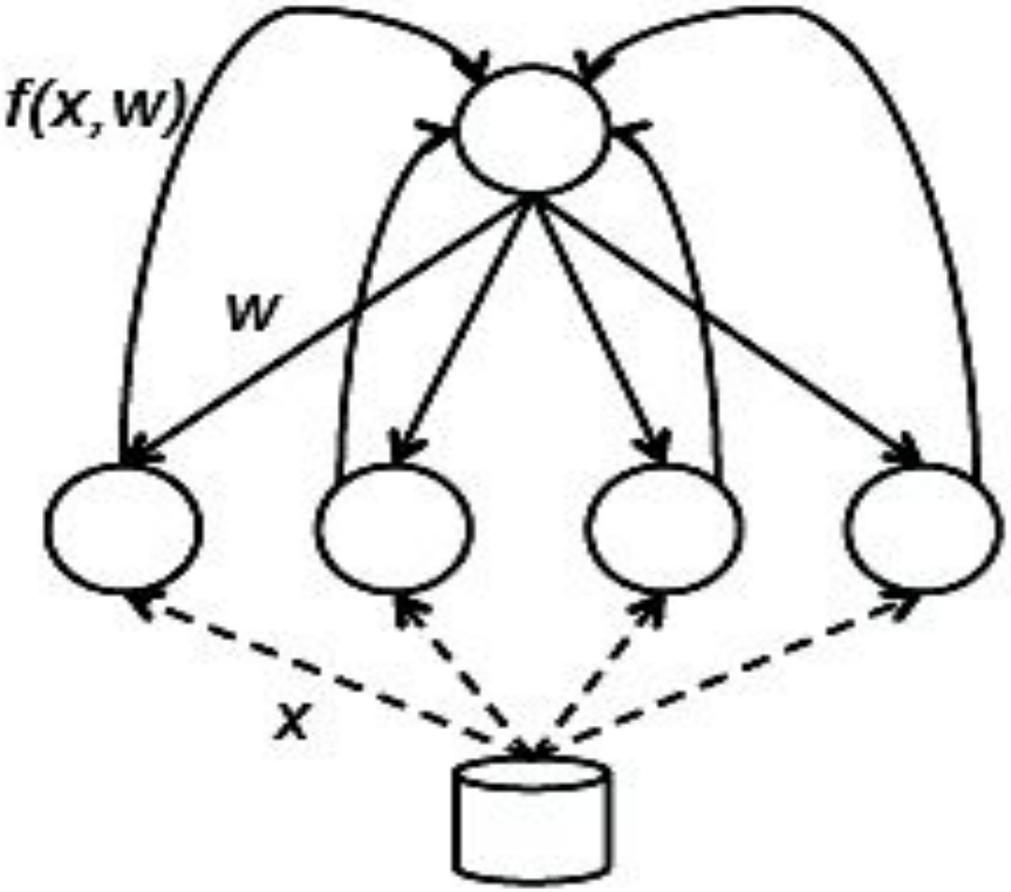


Other Spark features includes

- Supports more than just Map and Reduce functions.
- Optimizes arbitrary operator graphs.
- Lazy evaluation of big data queries which helps with the optimization of the overall data processing workflow.
- Provides concise and consistent APIs in Scala, Java and Python.
- Offers interactive shell for Scala and Python. This is not available in Java yet.

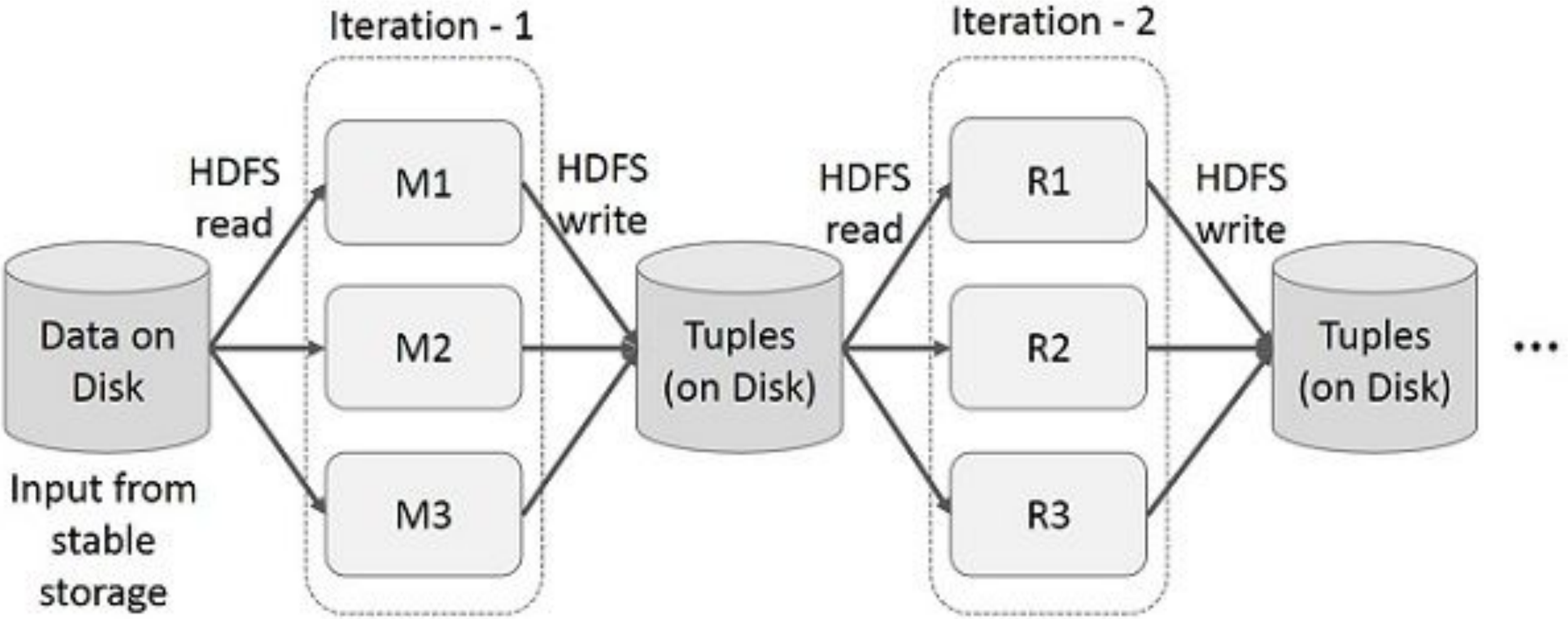


MapReduce

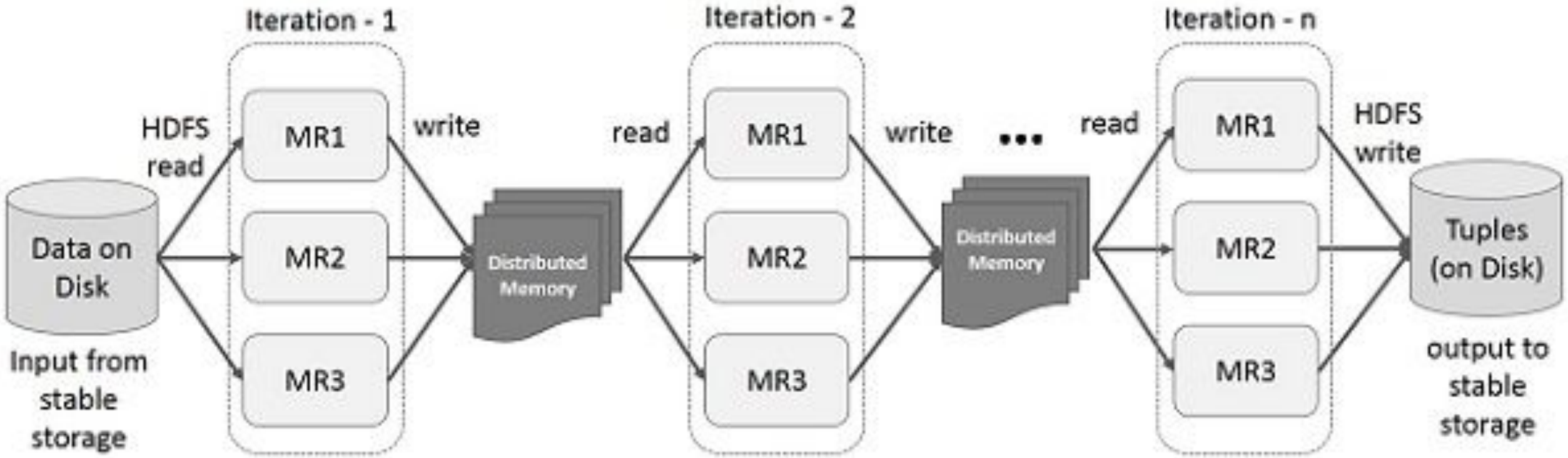


Spark

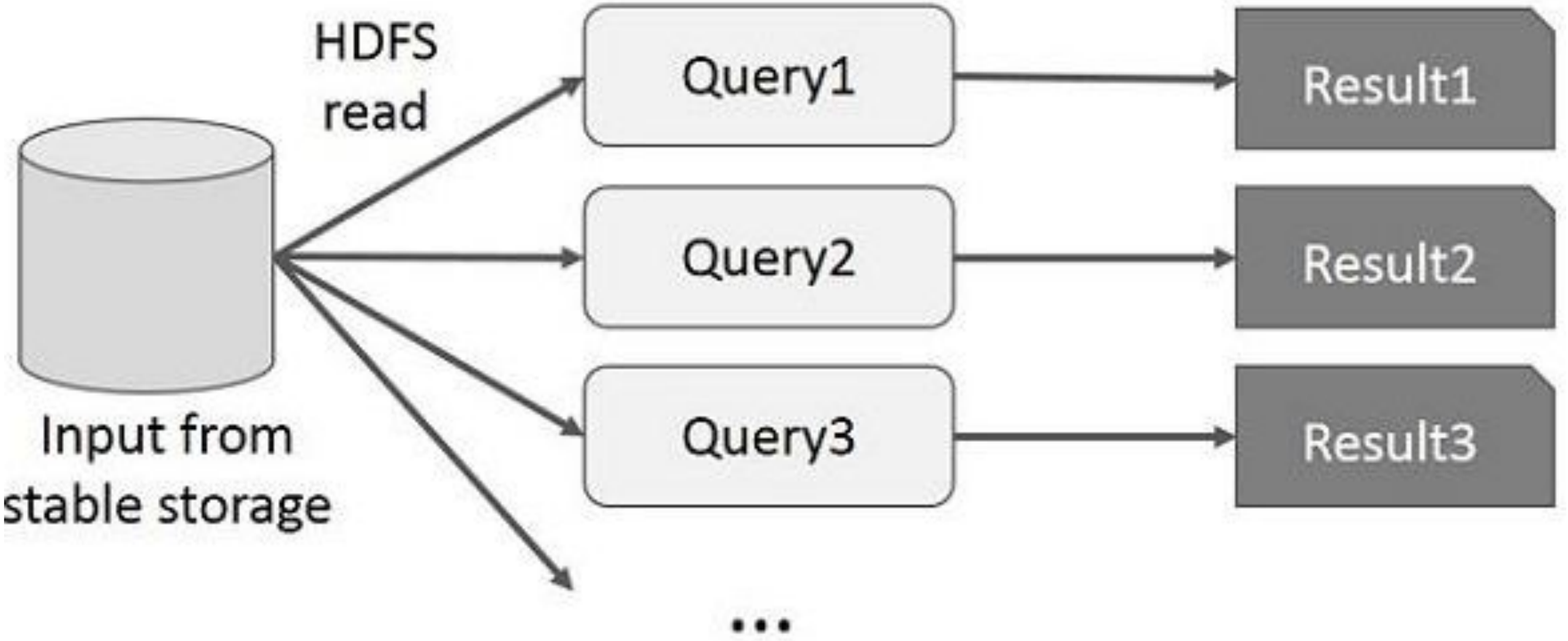
Iterative Operations on MapReduce



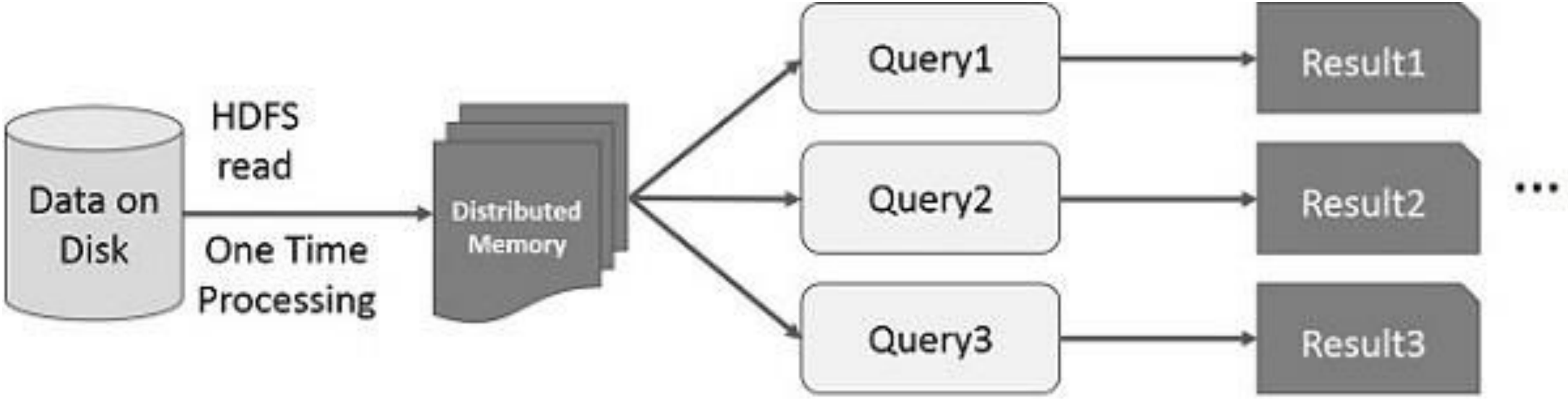
Iterative Operations on Spark RDD



Interactive Operations on MapReduce



Interactive Operations on Spark RDD



Hadoop Streaming Vs Spark – Wordcount Example

- Hadoop streaming is a utility that comes with the Hadoop distribution. The utility allows you to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer. For example:
- /home/jaiprakash/mapper.py
- /home/jaiprakash/reducer.py
- hadoop jar hadoop-streaming-1.2.0.jar -file /home/jaiprakash/mapper.py -mapper /home/jaiprakash/mapper.py -file /home/jaiprakash/reducer.py -reducer /home/jaiprakash/reducer.py -input /user/bible -output /user/bible-output3
- /home/spark/bin/spark-submit wordcount.py /user/bible /user/demo_ouput

Performance Evaluation Spark Vs MapReduce on Hadoop

Sr. No.	Text File Size	Text File Size (in Bytes)	MapReduce Framework (in s)	Spark Framework (in s)	%
1	153.3 kB	156979.2	3.15	1.053078	2.991231
2	260.1 kB	266342.4	3.67	1.110998	3.303336
3	358.4 kB	367001.6	3.52	1.132921	3.107013
4	769.3 kB	787763.2	4.5	1.240024	3.628962
5	1.0 MB	1048576	5.06	1.38958	3.641388
6	2.8 MB	2936012.8	7.08	1.770073	3.999835
7	5.1 MB	5347737.6	8.83	2.123277	4.158666
8	10.5 MB	11010048	10.46	3.132935	3.338722
9	15.6 MB	16357785.6	12.5	4.874238	2.564503
10	26.1 MB	27367833.6	16.81	5.329568	3.154102
11	55.1 MB	57776537.6	26.83	9.652699	2.779533
12	100.3 MB	105172173	54.86	20.168503	2.720083
13	255.9 MB	268330598	131.46	33.377112	3.938627
14	329.3 MB	345296077	161.76	49.030878	3.299145
15	1.2 GB	1288490189	610.65	147.019231	4.153538
16	1.6 GB	1717986918	791.09	187.673476	4.215247
17	2.8 GB	3006477107	1395.84	337.269451	4.138649
18	4.7 GB	5046586573	1731.95	550.205804	3.147822
19	7.5 GB	8053063680	3604.45	835.431703	4.314476

Clash of the Titans: Spark Vs MapReduce
(published 2015)

- Spark is about 2.5x, 3x, and 5x faster than MapReduce, for Word Count, k-means, and PageRank, respectively.
- The main causes of these speedups are the efficiency of the hash-based aggregation component for combine, as well as reduced CPU and disk overheads due to RDD caching in Spark.
- It also shows that MapReduce's execution model is more efficient for shuffling data than Spark, thus making Sort run faster on MapReduce
- Demo Spark Vs MapReduce with HDFS

Why Spark Programming Model

- There was no general purpose computing engine in the industry, since
- To perform batch processing, we were using Hadoop MapReduce.
- Also, to perform stream processing, we were using Apache Storm / S4.
- Moreover, for interactive processing, we were using Apache Impala / Apache Tez.
- To perform graph processing, we were using Neo4j / Apache Giraph.
- One engine can respond in sub-second and perform in-memory processing.

Spark

Use Cases

1. Travel Industry
2. e-commerce Industry
3. Media & Entertainment Industry
4. Finance Industry

Spark Essentials

- SparkContext
- Master
- RDD (Resilient Distributed Datasets)
- Transformations
- Actions
- Persistence

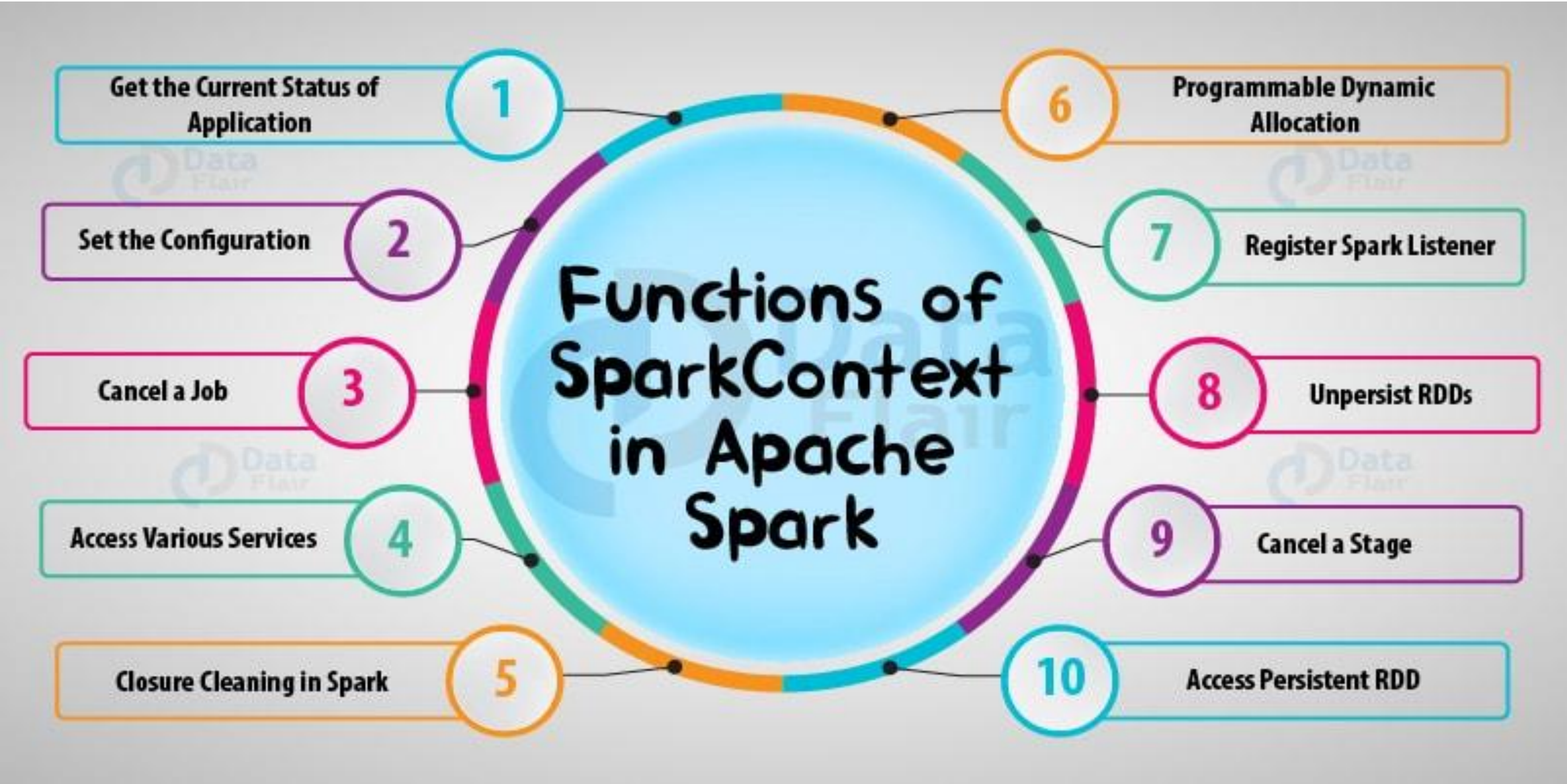
SparkContext

- It is the entry point of Spark functionality. The most important step of any Spark driver application is to generate SparkContext.
- It allows your Spark Application to access Spark Cluster with the help of Resource Manager. The resource manager can be one of these three- , YARN, Spark Standalone ,Apache Mesos.
- In the shell for either Scala or python, this is the sc variable, which is created automatically.
- Other program must use a constructor to instantiate a new SparkContext
- Then in turn SparkContext gets used to create other variables.
- it can be used to create RDDs, broadcast variable, and accumulator, ingress Spark service and run jobs. All these things can be carried out until SparkContext is stopped.

SparkContext

- 16/11/17 10:17:54 INFO storage.BlockManagerMaster: Registered BlockManager
- Welcome to
- ____ __
- / __/__ ___ _____/ /__
- _\ \/ _ \/ _ `/ __/ '_/
- /__ / .__/_,_/_/ /_/_\ version 1.6.1
- /_/
-
- Using Python version 2.7.6 (default, Jun 22 2015 17:58:13)
- SparkContext available as sc, HiveContext available as sqlContext.
- >>> sc
- <pyspark.context.SparkContext object at 0x7f1293c57ad0>
- >>>

Functions of SparkContext in Apache Spark



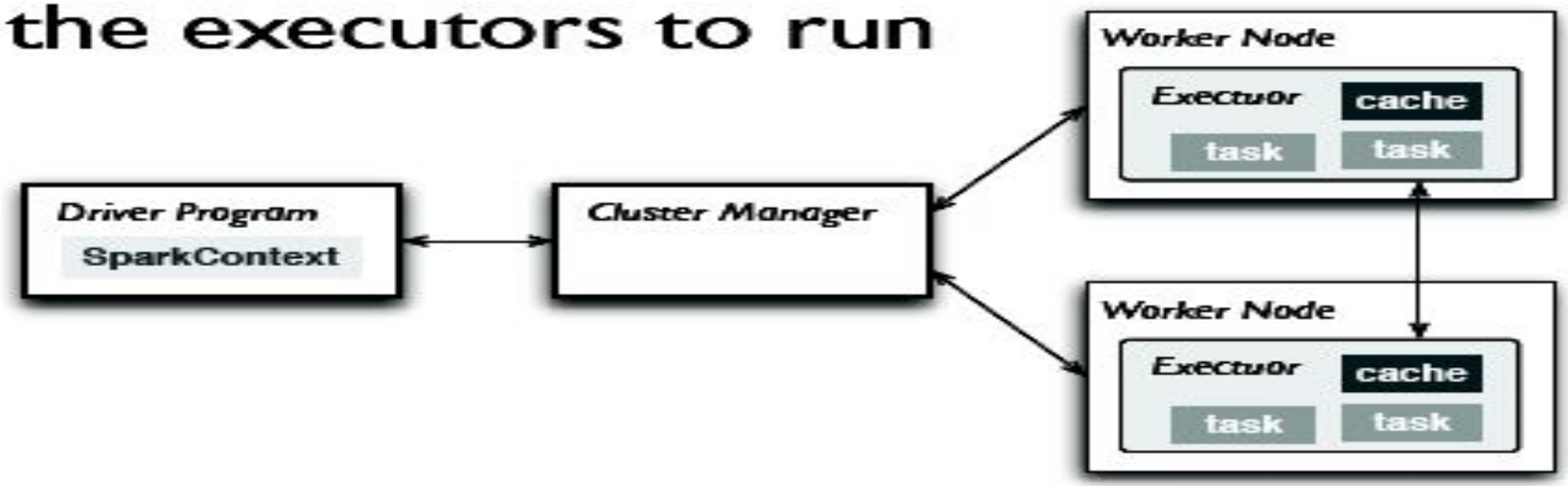
Master

- The master parameter for SparkContext determine which cluster to

master	*description*
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to # cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

Master

1. connects to a *cluster manager* which allocate resources across applications
2. acquires *executors* on cluster nodes – worker processes to run computations and store data
3. sends *app code* to the executors
4. sends *tasks* for the executors to run



Resilient Distributed Datasets (RDD)

- Think about RDD as a table in a database. It can hold any type of data. Spark stores data in RDD on different partitions.

- RDDs help with rearranging the computations and optimizing the data processing.

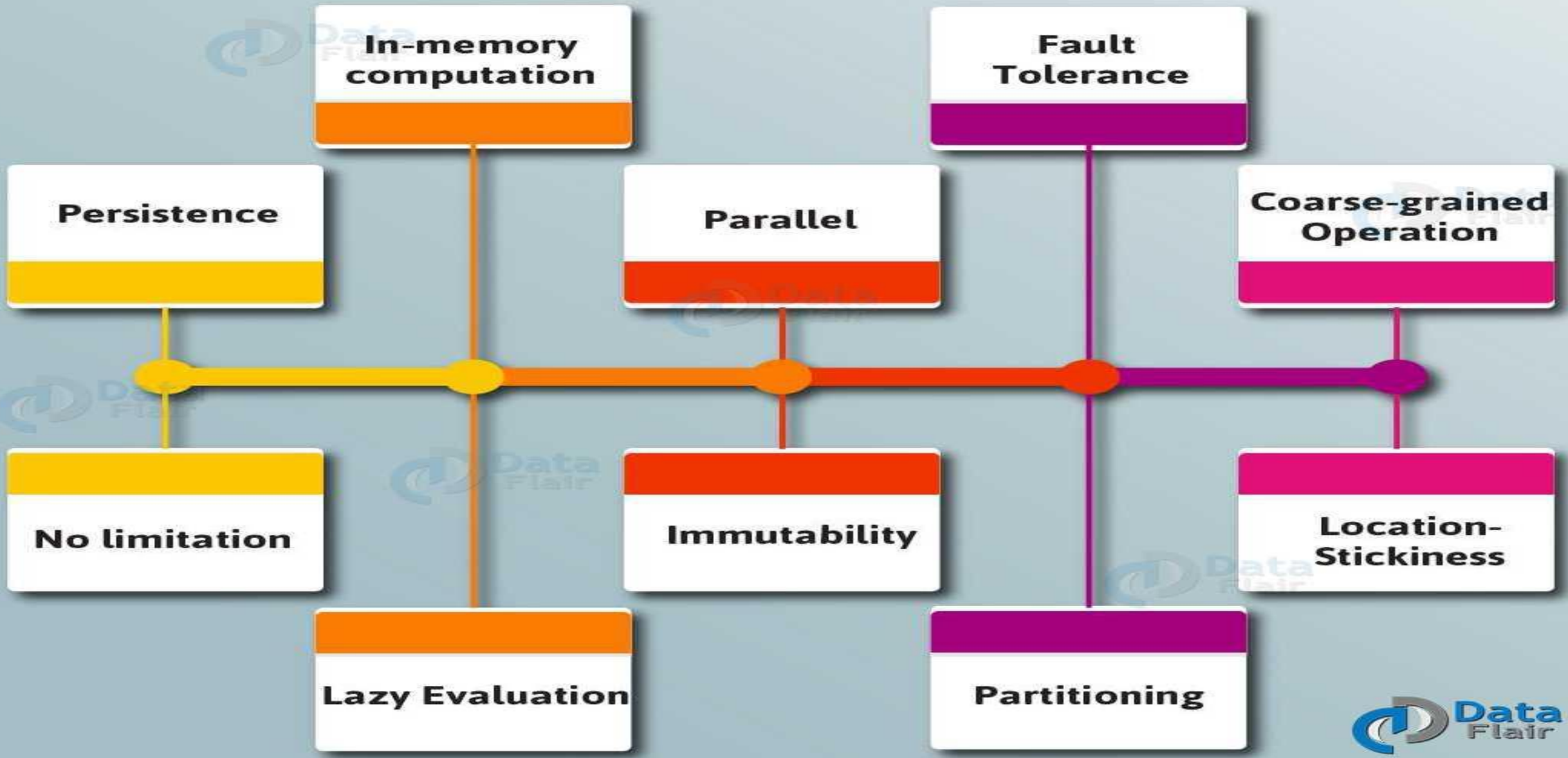
- RDDs are also fault tolerance because an RDD know how to recreate and recompute the datasets.

- RDD supports two types of operations:

- Transformation : Some of the Transformation functions are map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, pipe, and coalesce.

- Action: Some of the Action operations are reduce, collect, count, first, take, countByKey, and foreach.

Features of RDD



RDD continue....

- Python code;

```
>>> data=[1,2,3,4,5]
>>> data
[1, 2, 3, 4, 5]
>>> distData = sc.parallelize(data)
>>> distData
ParallelCollectionRDD[0] at parallelize at
  PythonRDD.scala:423
>>>
```

- *Note: Minimum no. of partitions in the RDD is 2 (by default). When we create RDD from HDFS file then a number of blocks will be equals to the number of partitions.*

Trasformation

- Trasformation create a new dataset from an existing one.
- All trasformations in spark are lazy: they do not compute their results right away, instead they remember the trasformation applied to some base dataset:
  - Optimize the required calculations
  - Recover from lost data partitions

Transformation continue...

transformation	*description*
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
flatMap(*func*)	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed*)	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed
union(*otherDataset*)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset

transformation	*description*
groupByKey([*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
reduceByKey(*func*, [*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*])	when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
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
cogroup(*otherDataset*, [*numTasks*])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called groupWith
cartesian(*otherDataset*)	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

Trasformation continue...

- Python code:

```
text_file = sc.textFile("hdfs://input/input.txt")

counts = text_file.map(lambda line: line.split(" ")).collect

counts = text_file.flatMap(lambda line: line.split(" ")).collect

    .
```

Actions

- Action operation evaluates and returns a new value. When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.
- Some of the Action operations are reduce, collect, count, first, take, countByKey, and foreach.

Action continue...

action	description
`reduce(func)`	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also 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 – 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
`first()`	return the first element of the dataset – similar to *take(1)*
`take(n)`	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
`takeSample(withReplacement, fraction, seed)`	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

Action continue...

action	*description*
`saveAsTextFile(`*`path`*`)`	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
`saveAsSequenceFile(`*`path`*`)`	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
`foreach(`*`func`*`)`	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Action continue...

- Python Code

```
from operator import add

f = sc.textFile("hdfs://input/input.txt")

words = f.flateMap(lambda x: x.split(' ')).map(lambda x: (x, 1))

words.reduceByKey(add).collect()

words.saveAsTextFile("hdfs://output")
```

Persistence

- Spark can persist (or cache) in memory across operations
- Each node stores in memory any slices of it that it computes and reuse them in other actions on that dataset - often making future actions more than 10 x faster
- The cache is fault-tolerant: if any partition of an RDD is lost, it will automatically be recomputed using the trasformations that originally created it.

Persistence continue…

transformation	*description*
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.

Persistence continue...

- Python Code

```
from operator import add

f = sc.textFile("hdfs://input/input.txt")

words = f.flateMap(lambda x: x.split(' ')).map(lambda x: (x, 1))

words.reduceByKey(add).collect().cache()

words.saveAsTextFile("hdfs://output")
```

Limitations of Apache Spark

- Questions