Big Data Real-time Processing with Apache Spark

Outline

- Big Data Processing in Real-time
- Apache Spark

Big Data Processing in Real-time

Processing in Real-time Mode

- Real-time processing is also called **event** or **stream** processing
  - **Stream**: data arrives **continuously**
  - **Event**: data arrives at **intervals**
  - The individual event/stream datum is generally small in size, but its continuous nature results in very large datasets

- Two important concepts related to big data processing
  - **Event Stream Processing (ESP)**
    - An incoming stream of events, generally from a **single source** and ordered by time, is continuously analysed.
    - ESP focuses more on **speed** than complexity. The operations to be executed is comparatively simple to aid faster execution.
  - **Complex Event Processing (CEP)**
    - A number of real-time events often coming from **disparate sources** and arriving at different time intervals are analysed simultaneously.
    - CEP focuses more on **complexity**, providing rich analytics.

Big Data Processing in Real-time

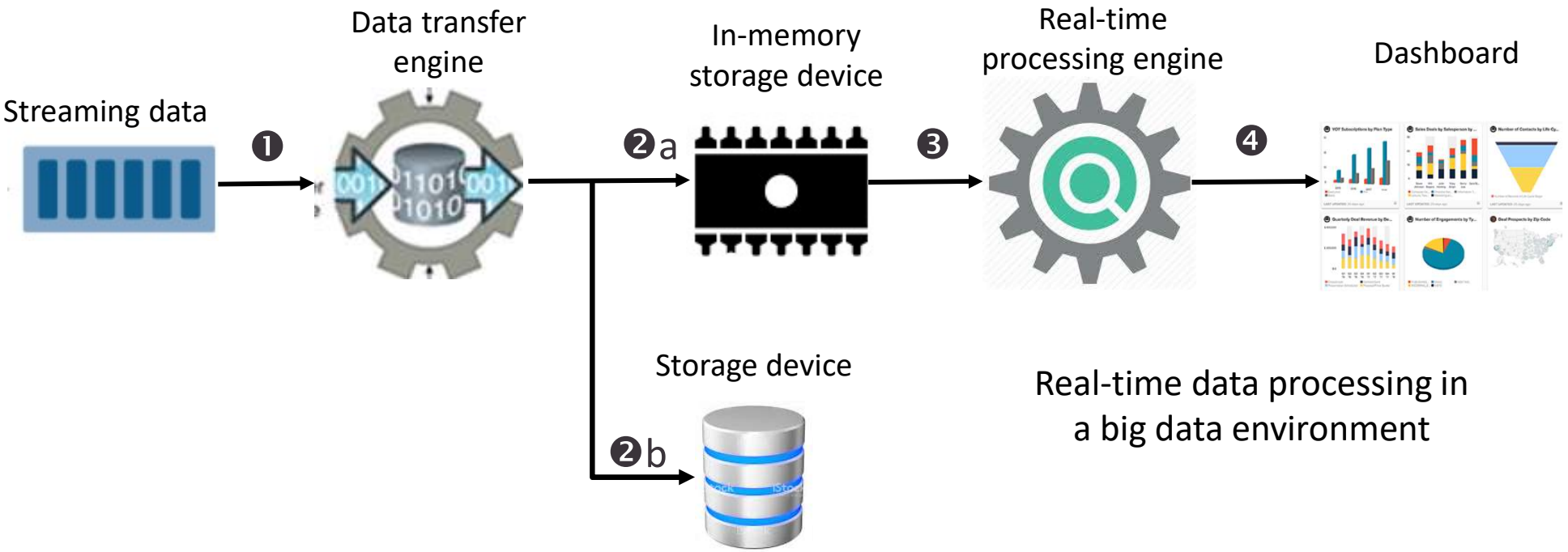
Processing in Real-time Mode

- Data is processed **in-memory** as captured bf being persisted to the **disk**
- Real-time mode address the **velocity** characteristic of big data datasets
- **Interactive mode** refers to query processing in real-time (e.g. operational BI is generally conducted in real-time mode)

What is real-time

Big Data Processing in Real-time

Processing in Real-time Mode



Real-time data processing in a big data environment:

❶ Streaming data is captured via a data transfer engine

❷ It is then simultaneously saved to an in-memory storage device (a) and an on-disk storage device (b)

❸ A processing engine is then used to process data in real-time

❹ Finally the results are fed to a dashboard for operational analysis

Big Data Processing in Real-time

In-Memory Storage Devices

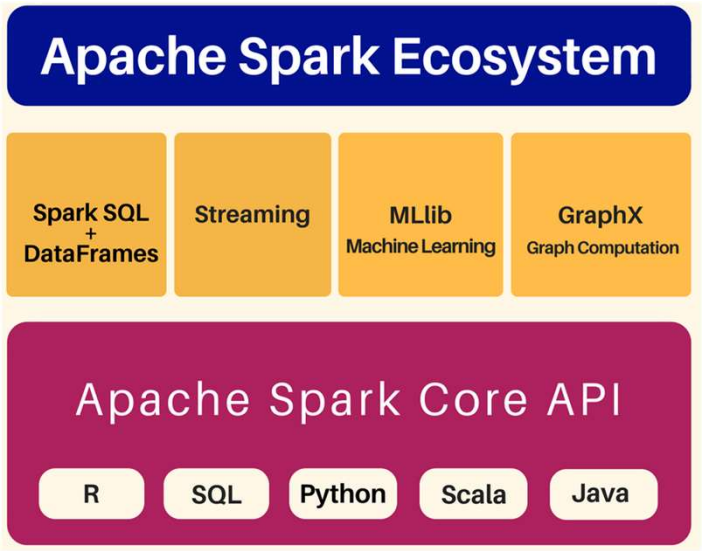
- Real-time big data analytics requires the use of in-memory storage devices
  - In-memory storage device generally utilizes RAM as its storage medium to provide fast data access
  - Reading data from in-memory storage is approx. 80 times faster than on-disk storage
  - Growing capacity and decreasing cost of RAM make it possible to develop in-memory data storage solution

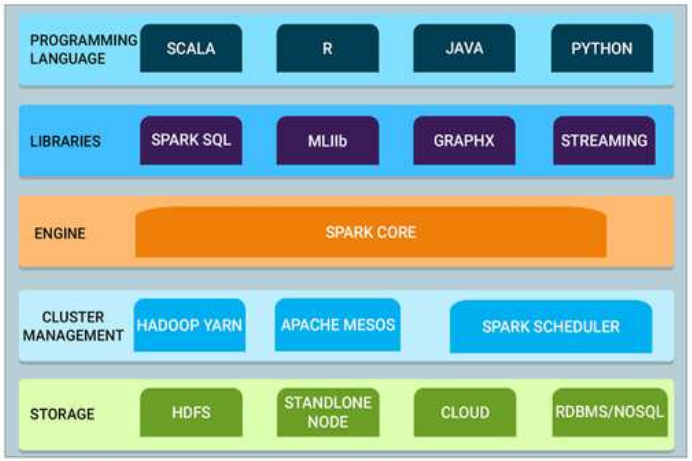
Origin

- **Apache Spark** is an open-source distributed general-purpose cluster-computing framework.
- Lightning fast, in-memory data analytics engine for large-scale data processing
- Originally developed in 2009 in UC Berkeley; Fully open sourced in 2010 – now a Top Level Project at the Apache Software Foundation
- Currently the largest open source project in data processing.
- Initial release: May 2014; Stable release: version **3.1.1** as of March 2021
- Written in Scala language

Spark

Spark ecosystem



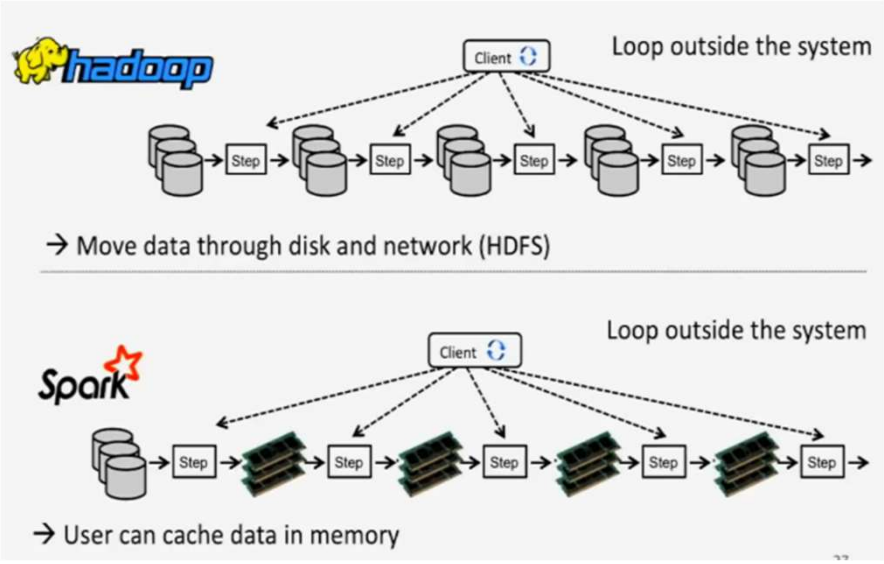


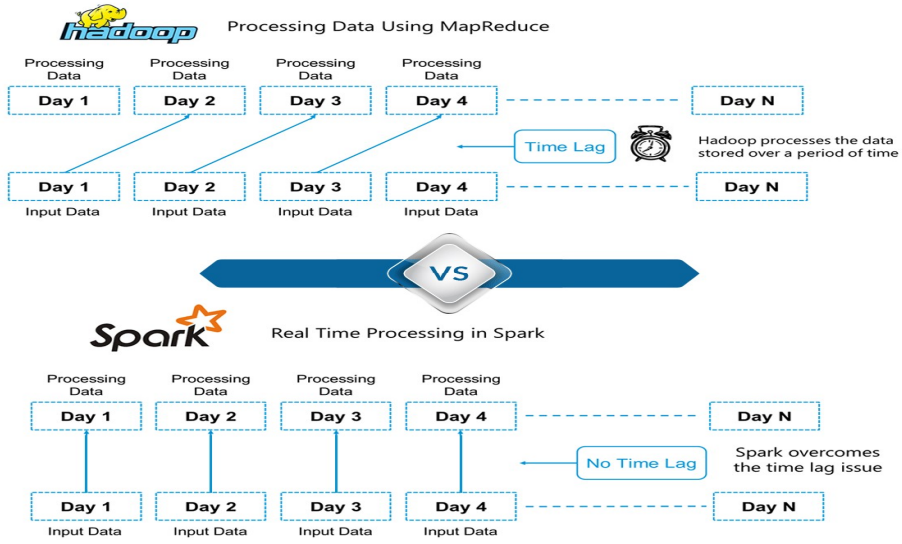
- **Spark core** – execution engine of Spark platform, providing memory management, task scheduling, fault recovery, etc.
- **Spark SQL** – Spark SQL performs the query on data through SQL and HQL (Hive Query Language) based on DataFrame model.
- **Spark Streaming** – live stream data processing
- **Spark MLlib** – in-built library of Spark that contains the functionality of Machine Learning, providing various ML algorithms
- **Spark GraphX** – graph computation engine built on top of Apache Spark that enables to process graph data at scale.
- **Spark API** – Spark provides API in Scala, Java, Python and R.
  - **PySpark** is Python package for Apache Spark.

Spark

Features

- Swift processing
  - In-memory storage and computation
  - Real-time data processing, process data as that data is being generated
  - Usually 10 times faster on disk / 100 times faster in memory than MapReduce





Spark

Features

- Advanced analytics
  - Real-time stream processing
  - SQL query
  - Graph computation
  - Machine learning

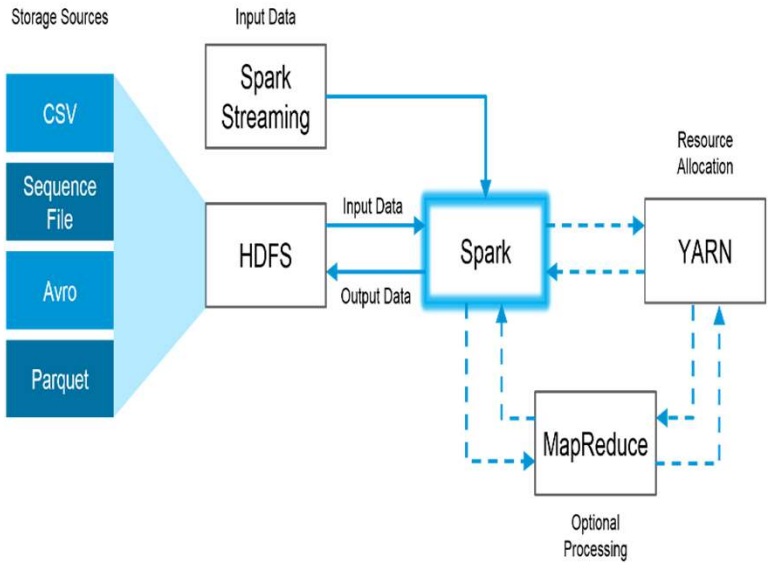
- Support multiple language programming
  - Rich API in Scala, Python, R and Java
  - Interactive shell
  - 2-5 times less code than MapReduce

```
mbo@mbo-ubuntu-vbox:~/mbo/spark$ MASTER=spark://localhost:7077 ./spark-shell
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 0.9.0-SNAPSHOT
      /_/

Using Scala version 2.9.3 (Java HotSpot(TM) Client VM, Java 1.6.0_45)
Initializing interpreter...
Creating SparkContext...
Spark context available as sc.
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Understanding Spark

Spark

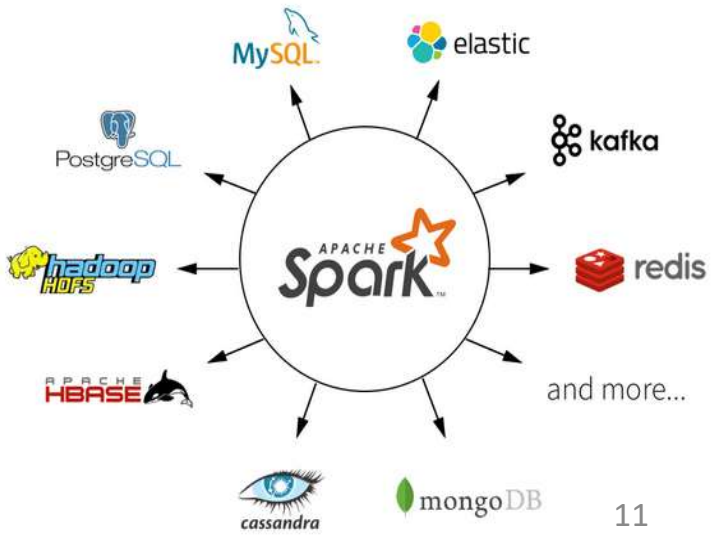
Spark vs Hadoop



- Spark can run on the Hadoop ecosystem and works smoothly with ***Hadoop Distributed File System (HDFS)***, Apache Hive, and others.

- Spark can also work standalone and access data from diverse data sources

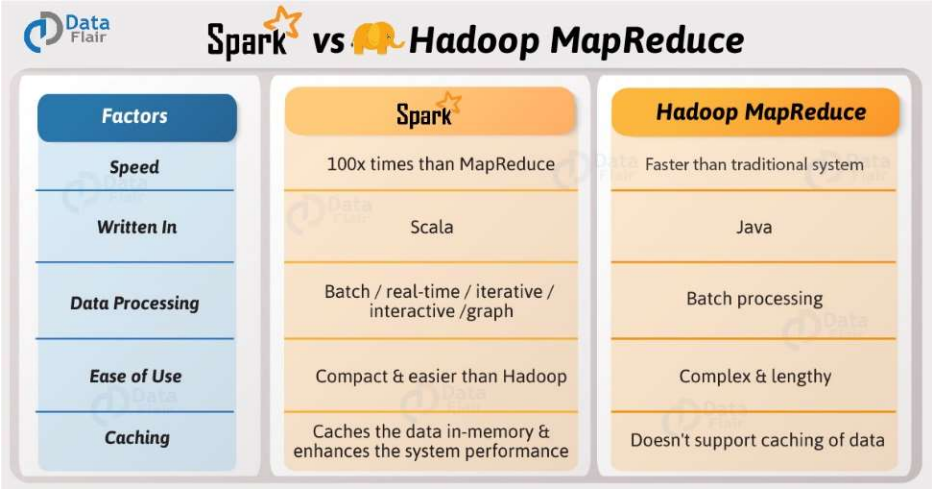
- Spark can also run on the clouds (e.g. Amazon Web Services)



Spark

Spark vs MapReduce

- MapReduce is intended for batch processing
- Spark delivers fast performance, iterative processing, real-time analytics, graph processing, machine learning and more
- These big data frameworks to be **complementary**, using them **together** to solve a broader business challenge.



Spark

Spark vs MapReduce

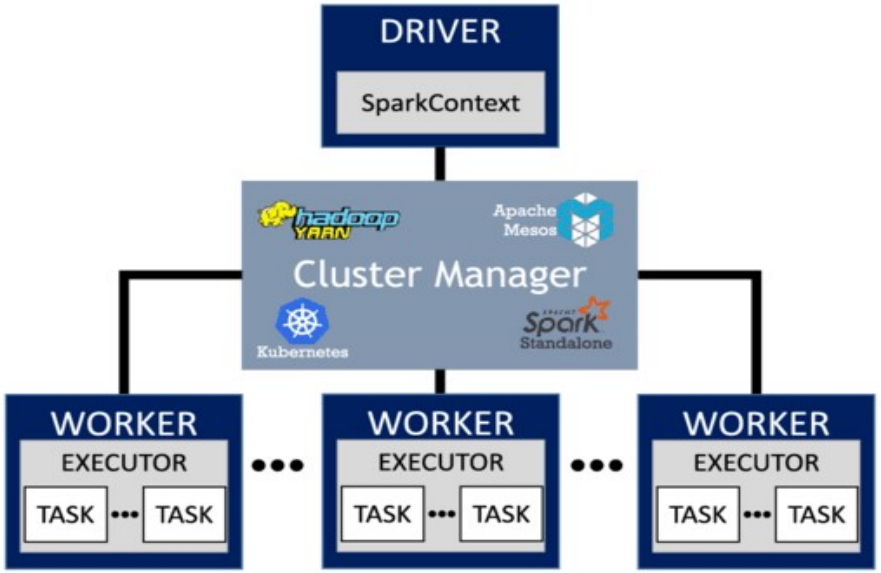
Apache MapReduce	Apache Spark
Difficult to program and requires abstractions.	Easy to program and does not require any abstractions.
It is used for generating reports that help find answers to historical queries.	Programmers can perform streaming, batch processing and machine learning, all in the same cluster.
No in-built interactive mode except tools like Pig and Hive.	Has in-built interactive mode.
Hadoop MapReduce does not leverage the memory of the hadoop cluster to the maximum.	Executes jobs 10 to 100 times faster than Hadoop MapReduce.
Allows you to just process a batch of stored data.	Programmers can modify the data in real-time through Spark streaming.
Written in Java	Written in Scala
Store data on disk	Store data in memory

Spark

Spark architecture

- **Driver program** - drive your application on master node. Your Spark program behaves as a driver program or if you are using the interactive shell, the shell acts as the driver program.

- Inside the driver program, the first thing you do is, you *create* a ***Spark Context*** (a gateway to all the Spark functionalities).



Apache Spark Architecture

- Spark Context takes the job, breaks the job in tasks. The driver talks to the ***cluster manager*** and negotiates the resources. Cluster manager launches executors in worker nodes on behalf of the driver.

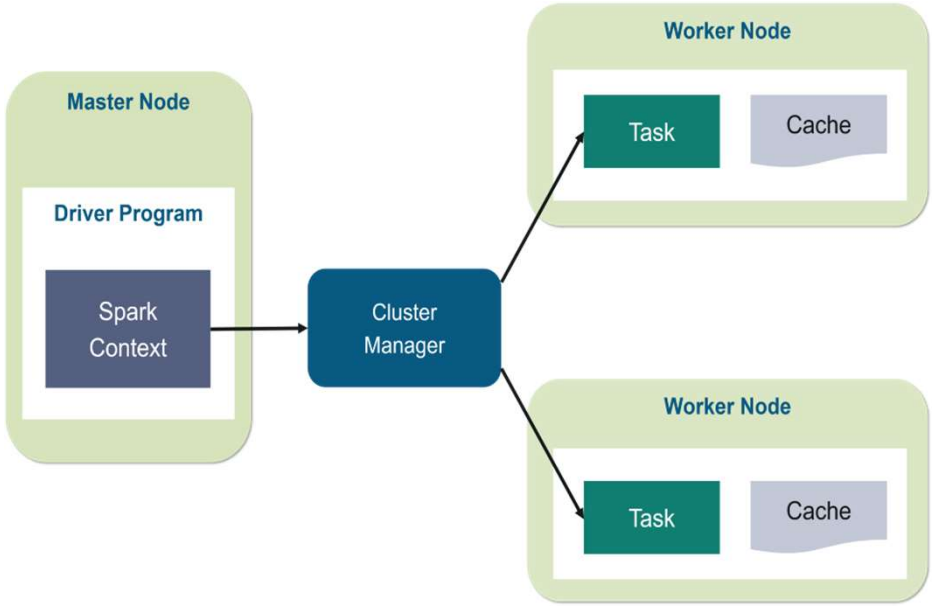
- These tasks work on the partitioned datasets, perform operations, collect the results and return to the main Spark Context.

Spark

Spark architecture

- The driver will send the tasks to the executors based on data placement. When executors start, they register themselves with drivers. So, the driver will have a complete view of executors that are executing the task

- During the course of execution of tasks, driver program will monitor the set of executors that runs.



- ***Worker nodes*** are the slave nodes whose job is to basically execute the tasks. These tasks are then executed on the partitioned datasets in the worker node and hence returns back the result to the Spark Context.

Spark

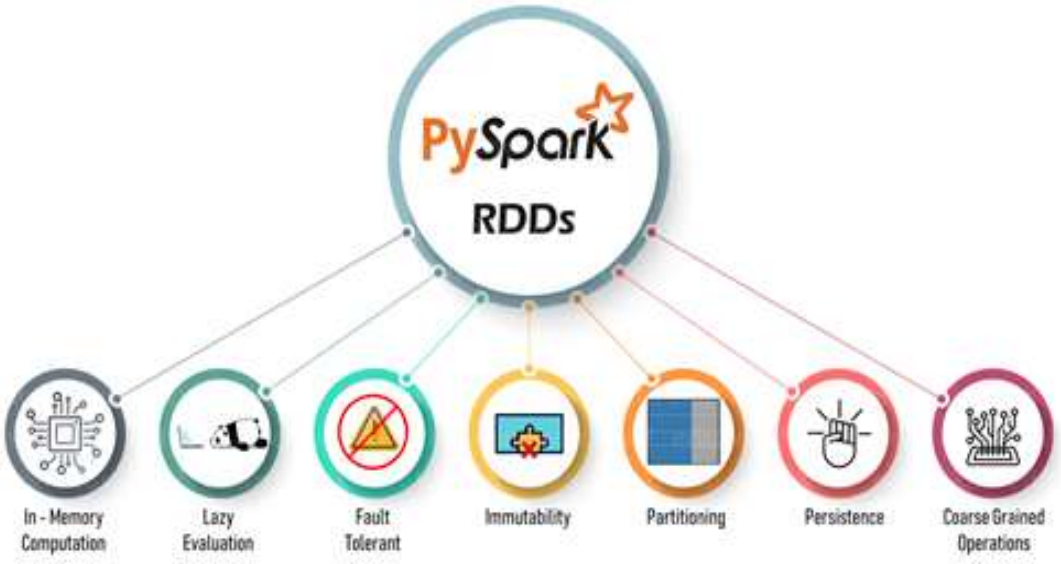
Spark programming model – data structure

- Spark supports following data structures:
  - **RDD** - Spark RDD (Resilient Distributed Dataset) is distributed collection of data over nodes in the cluster. It is the **fundamental** data structure of Spark.
  - **Dataframe** - The Dataframe is next data structures in Spark which is again a distributed collection of data and here data is organized into columns. This data structure is used with SparkSQL and provides SQL like operations over the data.
  - **Dataset** - Dataset is strongly typed, relational and column based data structure in Spark used with SparkSQL
  - **TUNGSTEN** - New data structure introduced with the Spark 2.x (SparkSQL) to provide high performance processing of data. The Tungsten operations works on the byte-code level and by optimizing Spark Jobs for CPU/Memory efficiency.
  - **Graphframe** - GraphFrames are DataFrames based package for Apache Spark, which is used for Graph Processing in Apache Spark.

Spark

Spark programming model – RDD

- Resilient Distributed Datasets (RDD) is the fundamental data structure of Spark

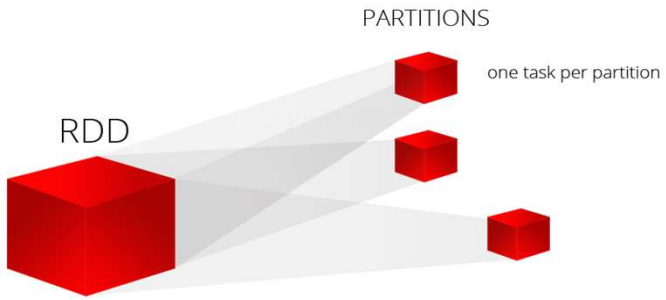


- **In-Memory Computations**: It improves the performance by an order of magnitudes.
- **Lazy Evaluation**: All transformations in RDDs are lazy, i.e, doesn't compute their results right away.
- **Fault Tolerant**: RDDs track data lineage information to rebuild lost data automatically.

Spark

Spark programming model – RDD

- Resilient Distributed Datasets (RDD) is the fundamental data structure of Spark
  - **Immutability**: Data can be created or retrieved anytime and once defined, its value can't be changed.
  - **Partitioning**: A RDD can be automatically divided into Partitions which are distributed across different nodes of a Spark cluster. It is the fundamental unit of parallelism in Spark RDD (When executing some action, a task is launched per partition **(**the more the number of partitions, the more the parallelism)
  - **Persistence**: Users can reuse Spark RDDs and choose a storage strategy for them. RDD can also be cached for future reuse
  - **Coarse-Grained Operations**: These operations are applied to all elements in data sets through maps or filter or group by operation.



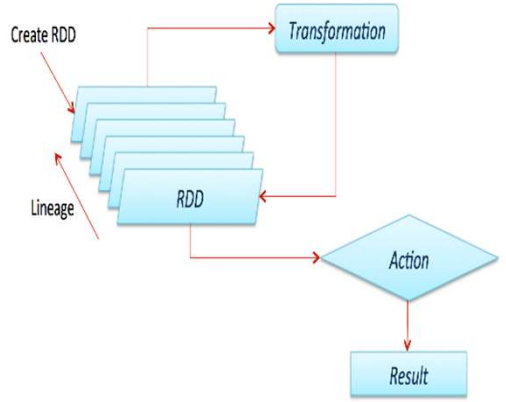
e.g. (in Scala)
val rdd = sc.textFile("some_file",3)

Create a RDD from an file with 3 partitions

Spark

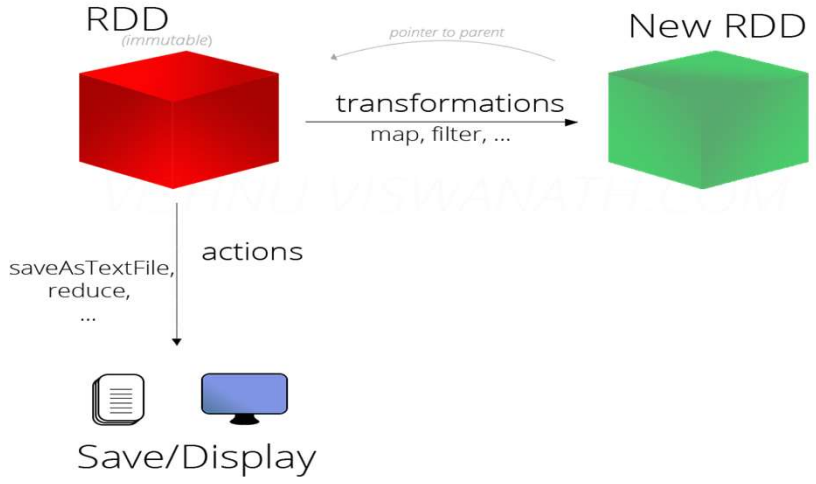
Spark programming model – RDD

Two types of operations can be performed on an RDD:



- **Transformation**: create a new RDD from an existing one.
  - New RDD keeps a pointer to it's parent RDD.
  - When you call a transformation, Spark does not execute it immediately (lazy evaluation), instead it creates a **lineage** which keeps track of what all transformations has to be applied on that RDD.

- **Action**: either save result to some location or to display it



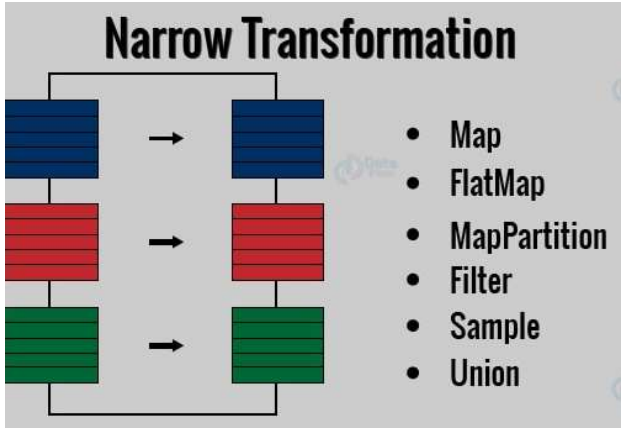
e.g. (in Scala)

```
val rdd = sc.textFile("spam.txt")
val filtered = rdd.filter(line =>
line.contains("money"))
filtered.count()
```

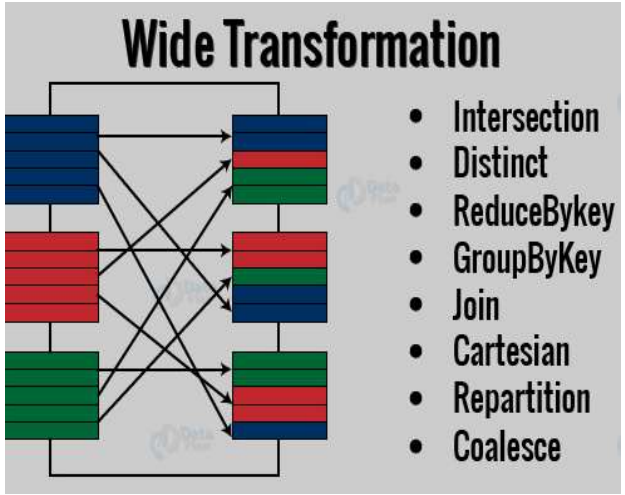
Spark

Spark programming model – RDD

Transformation: two types of transformation



- All the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result (**no partitioning**)



- All the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD (**partitioning**)

Spark

Spark programming model – RDD

Transformation

Transformation	Description
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func* (same number of elements in new dataset)
flatMap(*func*)	Similar to map, but each input item can be mapped to zero or more output items (so *func* should return a sequence rather than a single item)
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true
sample(*withReplacement, fraction, seed*)	Sample a *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 *otherDataset*
distinct()	Return a new dataset that contains the distinct elements of the source dataset

Spark

Spark programming model – RDD

Transformation

Transformation	Description
groupByKey()	When called on a dataset of *(K, V)* pairs, returns a dataset of *(K, Seq[V])* pairs
reduceByKey()	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()	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()	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()	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()	when called on datasets of types *T* and U, returns a dataset of *(T, U)* pairs (all pairs of elements)

Spark

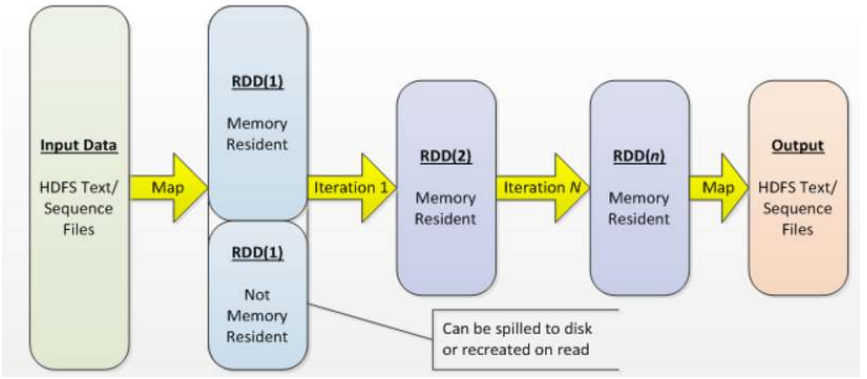
Spark programming model – RDD

Action

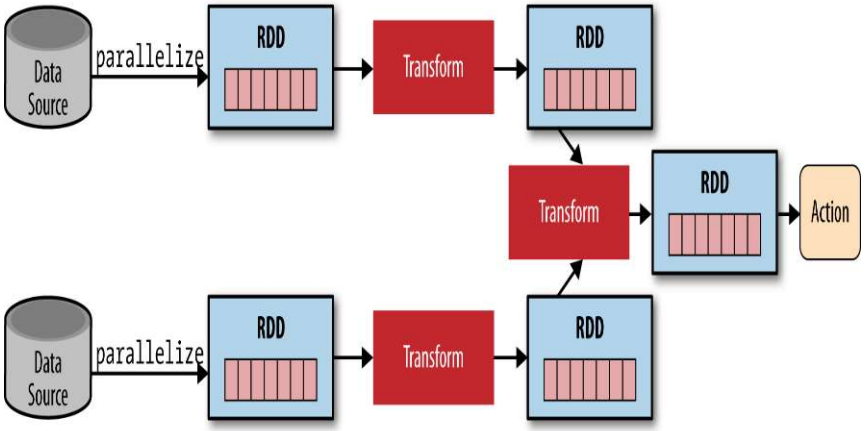
Action	Description
count()	use to count the number of elements in RDD
collect()	used to get entire RDD to driver program. If the data size is huge then its advisable to use it with care because it retrieves all the RDD data on the driver node.
take()	return *n* number of RDD on the driver machine. If you want to return 10 rows then use the function take(10).
top()	returns top *n* elements from RDD using the default ordering.
reduce()	performs aggregation on full RDD over distributed cluster
aggregate()	used when data type is different from the input type
foreach()	used to execute some operations on the each data item in RDD.
saveAsTextFile()	write the elements of the dataset as a text file

Spark

Spark programming model – life cycle of spark program based on RDD



1) Create some input RDDs from external data or parallelize a collection in your driver program.
2) Lazily transform them to define new RDDs using transformations like filter() or map()
3) Ask Spark to cache() any intermediate RDDs that will need to be reused.
4) Launch actions such as count() and collect() to kick off a parallel computation, which is then optimized and executed by Spark.



VIDEO RDD

Summary

Spark

- Spark provides a simple, efficient, scalable programming model for big data analytics
- Support *interactive* and *streaming* computations
  - In-memory, fault-tolerant storage abstraction, low-latency scheduling,...
- ***Easy*** to combine ***batch***, ***streaming***, and ***interactive*** computations
  - Spark execution engine supports all computation models
- ***Easy*** to develop ***sophisticated*** algorithms
  - Scala interface, APIs for Java, Python, R, Hive QL, …
  - Can address graph based and ML algorithms, etc
- ***Compatible*** with existing open source ecosystem