Data Processing with Spark: RDD Way

Author : Amit Singh Chowdhery

Agenda

- What is the need of Spark?
- Complete View of Spark Architecture
- How to interact with Spark?
- What is RDD and what is action, job, task
- Detailing out narrow vs wide dependencies
- Few more advanced concepts like RDD Caching, Shared Variables, DAG in detail

Problem Statement :

How to do **Exploratory Data Analysis on** GBs of movie data to figure out a recipe to make profitable movies?

Questions be like::

a. Which comedy movie has the most ratings and for the same return the title and the number of rankings.

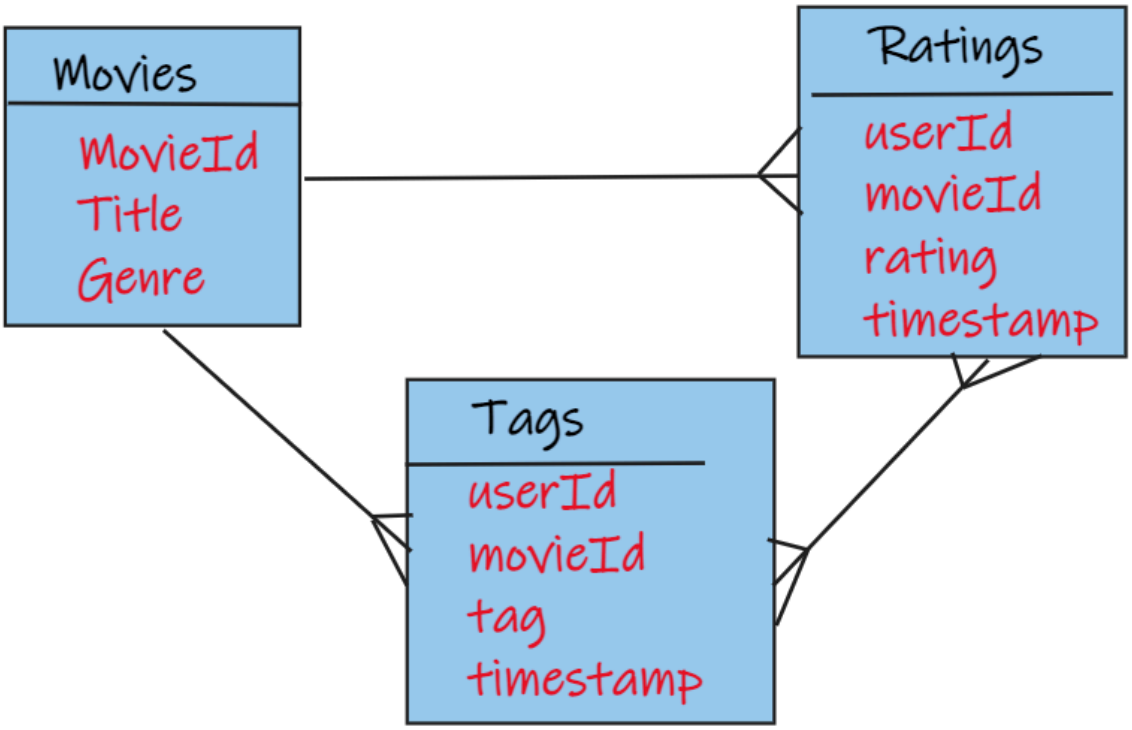
   Answer : American Beauty but how ?

---

b. Understand viewership by age group for targeted marketing, retention activities or market expansion.
c. Insight to the top 25 movies by viewership rating, indicating the type of movies or sequels that perform better.

d. Find the average rating of each movie for which there are at least 100 ratings. Order the result by average rating in decreasing order.

The **challenge** here is that weWe need to Answer this question by joining two datasets which contain almost 20M ratings given on 62K movies collected over the last 20 years(1995-2015)?
<Ask students:> Couldn't we do this using pandas?



The data is contained in 3 files.

tag.csv that contains tags applied to movies by users:

- userId
- movieId
- tag
- timestamp

rating.csv that contains ratings of movies by users:

- userId
- movieId
- rating
- timestamp

movie.csv that contains movie information:

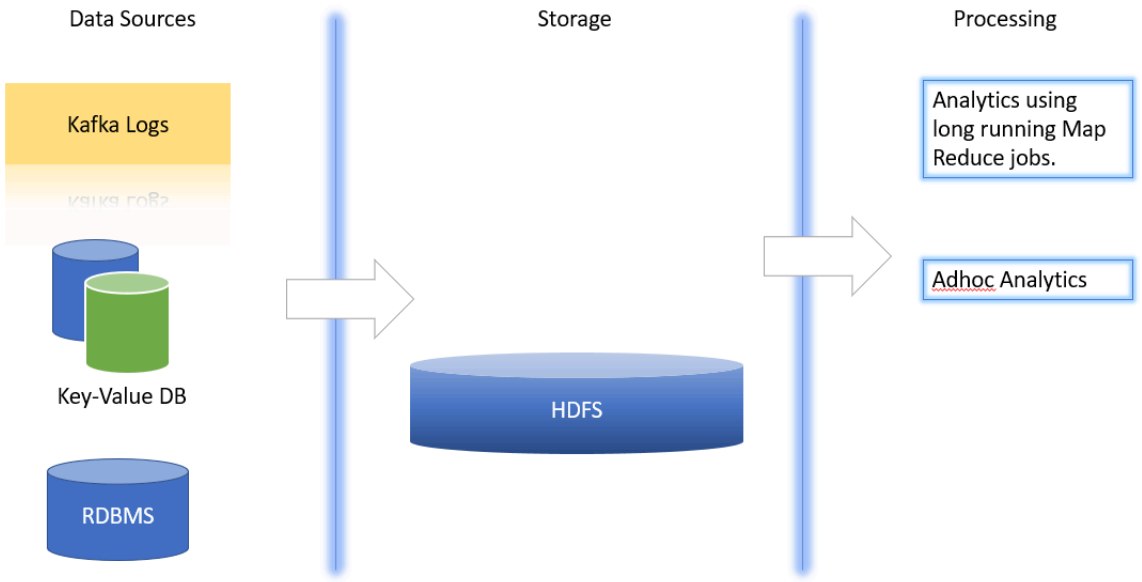
- movieId
- title
- genres

What are the Potential Solutions to deal with large amounts of data?

1. **API/Libraries(python,java) :** API like pandas, Dask can process data but to limits .Our TB,PB of datasets will force them to have OOM errors.
2. **RDBMS(MySQL,Oracle,MSSQL,Postgres):** Another alternative can be any RDBMS , but here speed will become a bottleneck and lots of RTO's(Request Timed Out) errors will be there.
3. **Hadoop(MapReduce):** Yes!, it can process huge TB/PB scale of data easily, but Hadoop MR has below constraints:
    a. Since I/O are more and getting information is very time consuming.
    b. Moreover only Batch Processing support is there.

Example : A typical Data Processing System using MR

Most of the time organizations collect the data from multiple different sources and store the data in the Hadoop ecosystem. They run long running ETL jobs and solve various KPIs using Hadoop MapReduce framework.



With the change in data access patterns rapidly,Map Reduce started falling apart due to below mentioned reasons:

- Only Batch Processing Supported: like doing processing on csv,tsv,avro,parquet files,no streaming support.
- Slow Processing Speed: for sorting 100Gb of data, 72 mins are taken
- No Real time data Processing: Real events from twitter, clickstream apps, facebook can’t be processed.
- Lengthy Line of code: Everytime a core developer needs to write at least 100-200 lines of code to process even the simplest aggregation like groupby, orderby.

What can be the best solution in terms of flexibility and speed?

It's Apache Spark !!!

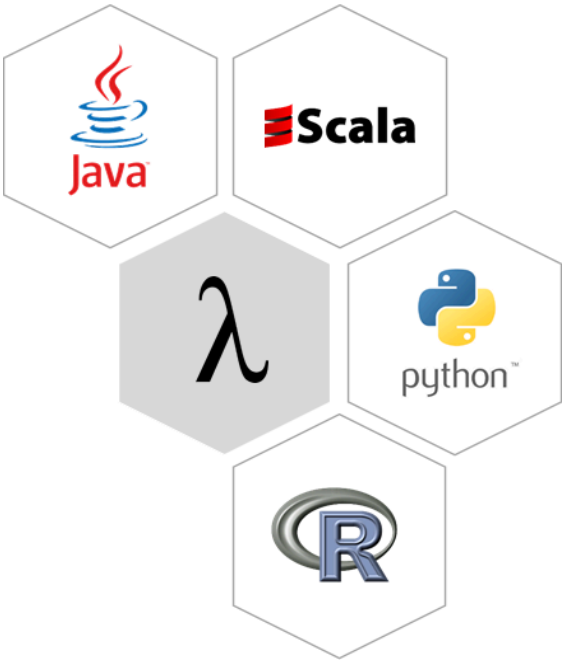
What is Apache Spark?

Apache Spark is an **open-source unified analytics engine** for **large-scale data processing** and is **100 times faster in memory** and **10 times faster on disk** when compared to **Apache Hadoop.**

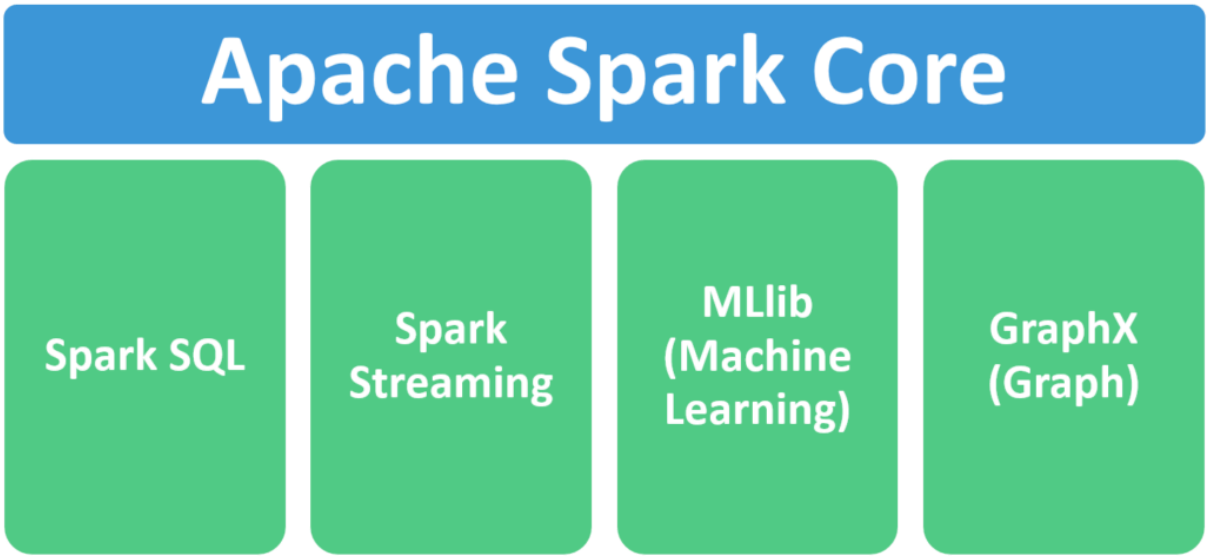
Spark provides an interface for programming clusters with implicit data parallelism and fault tolerance.

What kind of flexibility can be expected from Spark?

1. Spark **supports** popular development languages like **Java, Scala, Python and R**. So one a developer need not be bound to only 1 parent kind of language.



2. By default, spark internally has **five** major **components** that gives flexibility to process any kind of workload like **batch,streaming, graph or ML**. Components include:



1. **Spark Core** : Spark Core is a central point of Spark. Basically, it provides an execution platform for all the Spark applications.

2. **Spark SQL :** On the top of Spark, Spark SQL enables users to run SQL/HQL queries. We can process structured as well as semi-structured data, by using Spark SQL.

Moreover, it offers to run unmodified queries up to 100 times faster on existing deployments.

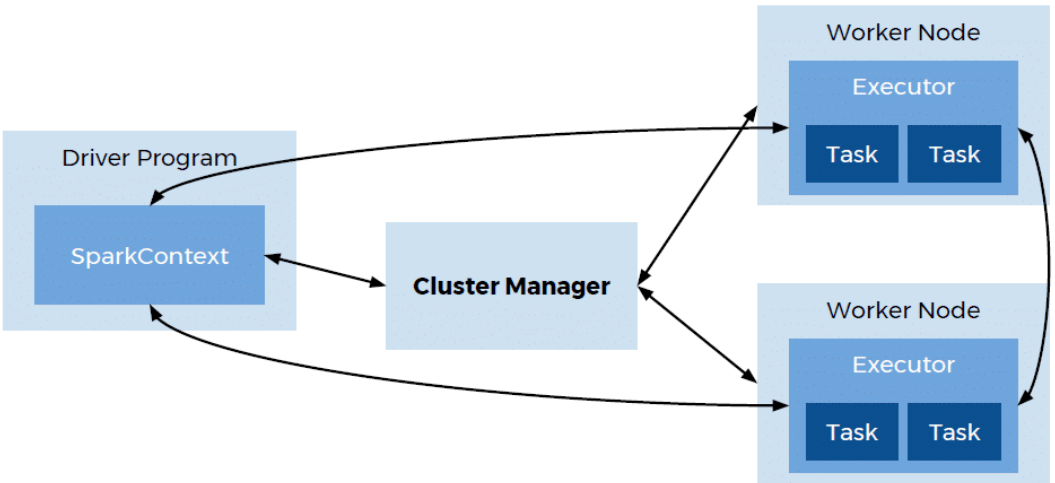
3. **Spark Streaming:** Spark Streaming leverages Spark Core's capability to perform streaming analytics. It ingests data in mini-batches and performs transformations on those mini-batches of data.

4. **Spark MLlib :** MLlib is a distributed machine learning framework above Spark. Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:
    - ML Algorithms like classification, regression, clustering, and collaborative filtering
    - Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
    - Persistence: saving and load algorithms, models, and Pipelines
    - Utilities: linear algebra, statistics, data handling, etc.

5. **Spark GraphX :** Spark GraphX is the graph computation engine built on top of Apache Spark that enables it to process graph data at scale.

But the question remains open: what makes Spark faster?

All credit goes to its Architecture.



Apache Spark is a Master-Slave architecture.

In the master node, we have a driver program, which drives the application. The code you are writing behaves as a driver program or if you are using the interactive shell, the shell acts as the driver program.

Driver program

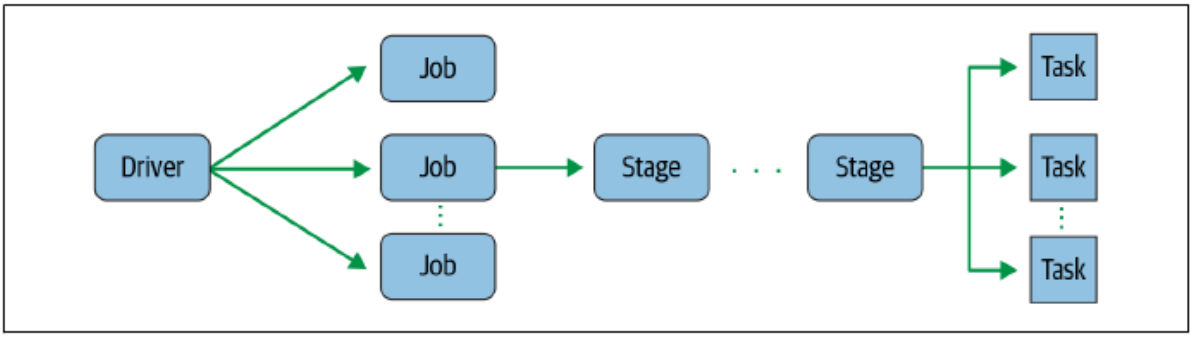
- “Main” process coordinated by the SparkContext object
- Allows to configure any spark process with specific parameters
- Spark actions are executed in the Driver
- If we are using the interactive shell, the shell acts as the driver program.

SparkContext

- SparkContext is the main entry point for Spark functionality
- SparkContext represents the connection to a Spark cluster
- Tells Spark how & where to access a cluster
- Can be used to create RDDs, accumulators and broadcast variables on the cluster

Worker

- Any node that can run application code in the cluster
- Key Terms
    - **Executor**: A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
    - **Job**: A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect).
        - jobs are work submitted to Spark.
        - Jobs are divided into "stages" based on the shuffle boundary.
        - Each stage is further divided into tasks based on the number of partitions in the RDD. So tasks are the smallest units of work for Spark.
    - **Task**: Unit of work that will be sent to one executor



Cluster Manager

- Cluster Manager is an external service for acquiring resources on the cluster
- Spark supports 3 cluster manager:

    - Standalone cluster manager
        - It is a part of spark distribution and available as a simple cluster manager to us.
    - Hadoop Yarn
        - Most famous cluster manager. It's a part of Apache Hadoop Ecosystem.
        - YARN can manage resources for Map Reduce as well as Spark applications.
        - As most organizations working on Big Data have a Hadoop ecosystem and this YARN, so it becomes the most famous choice.
    - Apache Mesos
        - Another cluster manager of Spark like YARN, no so famous though.
        - Like yarn, it is also highly available and manages resources per application.
        - Mesos can manage resources for Map Reduce as well as Spark applications.

Deploy modes when the Cluster Manager is YARN

There are two deploy modes that can be used to launch Spark applications on YARN.

- **Cluster Mode** : In cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster, and the client can go away after initiating the application.
- **Client Mode** : In client mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN.

The development workflow is that you start in local mode and transition to cluster mode. During the transition from local to cluster mode, no code change is necessary.

How to Interact with Spark?

Currently spark supports 2 ways to interact :

1. **Interactive way :** Spark comes with interactive shells that enable ad-hoc data analysis and they are of 2 types:
    a. **Spark-shell:** Spark Shell is a Spark Application written in Scala.
    b. **Pyspark:** shell is the Python-based command line tool to develop Spark's interactive applications in Python.
    c. **SparkR:** SparkR is an R package that provides a light-weight frontend to use Apache Spark from R.

2. **Application way :**

    The Spark application is a self-contained computation that runs user-supplied code to compute a result. Here we create file conationgin logic in java, python ,scala and then deploy it to the cluster.

Right now we will restrict with Working with pyspark shell and that too with loading data into RDD , so now let's take our question.

Qn : Which comedy movie has the most ratings and for the same return the title and the number of rankings. We need to Answer this question by joining two datasets which contain almost 20M ratings given on 62K movies collected over the last 20 years(1995-2015)?

Objective : To load data from csv file

Solution :

a. Create a sparksession object . This is the main entry point of any spark program , similar to the **main() program** in any programming language. It is one of the very first objects we create.

```
sc: SparkContext = SparkSession \
        .builder \
        .appName("Demo") \
        .getOrCreate() \
        .sparkContext
```

b. Read the file from the HDFS/s3 file location:

```
wordsRDD = sc.textFile("../data-files/movielens/movies.csv")
```

c. Now let's see if we have content loaded into wordsRDD or not.
```
wordsRDD.collect()
```

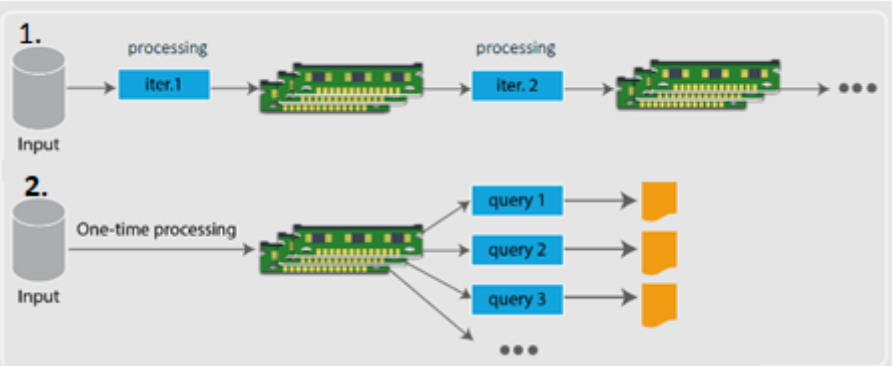
d. Volla!! you see data here but a question is what exactly is **WordsRDD** and **collect()** doing here.

WordsRDD is a RDD ,actual fundamental data Structure that **supports** in-memory processing computation and is the main reason for making Spark so faster.

collect() is an Action which we will study next.

What prompted the creation of RDD ?

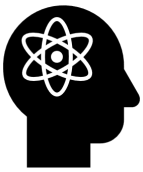
To understand the concept of RDD, let's re-iterate the challenges in existing computing methods.



There are several problems with Map Reduce.

- The intermediate data is stored in the file system as shown in the first half of the diagram(**1.**).
- Multiple I/O operations makes the overall computation slower.

So, the goal of programmers is to reduce the no. of I/O operations. This can be achieved through In-Memory data sharing (which is 10-100 times faster than Network/Disk sharing) shown in the second part of Diagram(**2.**).



The challenge is how to design a distributed memory abstraction which is Fault tolerant, distributed, and efficient?

Programmers created RDD to solve all above mentioned problems by enabling tolerant, distributed in-memory computations.

What exactly is RDD?

RDD is the most basic building block in Apache Spark. RDD stands for Resilient Distributed Dataset.

“RDD is the bread and butter of Spark, and mastering the concept is of utmost importance to become a Spark pro”

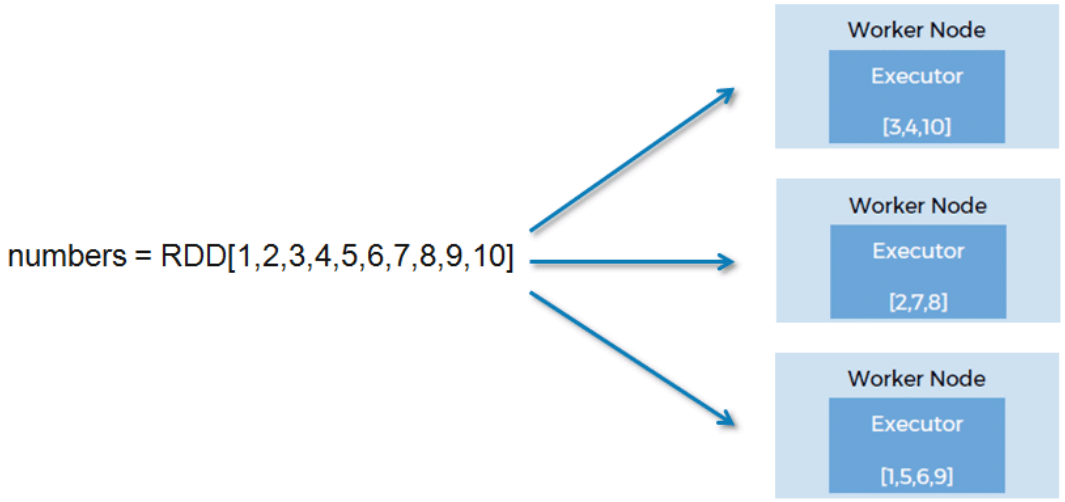
- **Resilient** : Fault tolerant and is capable of rebuilding data on failure
- **Distributed** : Distributed data among the multiple nodes in a cluster
- **Dataset** : Collection of partitioned data with values

Let’s understand each of the above in detail below –

R means Resilient

- The property of RDD is that it can recover from failures.
- In case, multiple tasks are running in the cluster and a node goes down.
- Spark will re-compute the failed tasks on other worker nodes which are up and running.

D means distributed dataset



- A collection of objects that is partitioned and distributed across nodes in a cluster.
- Spark partitions the RDD and distributes the data across multiple Worker nodes.

RDD is Immutable

```
wordsRDD = sc.textFile("../data-files/movielens/movies.csv")
finalRDD = wordsRDD.map(lambda word: (word.split(",")[2], word.split(",")[1])
```

wordsRDD = Parent RDD

finalRDD = derivedRDD

map = a sample transformation

- When a transformation is called on wordsRDD , a new RDD(finalRDD) is returned.
- None of the Spark operations modify an existing wordsRDD.
- Instead, they create a new finalRDD.

Immutable RDDs allow Spark to rebuild an RDD from the previous RDD in the pipeline if there is a failure.

RDD supports Lazy evaluation :

Lazy evaluation in Spark means that the execution will not start until an action is triggered. In Spark, the picture of lazy evaluation comes when Spark transformations occur.

Example :

```
finalRDD = wordsRDD.map(lambda word: (word.split(",")[2], word.split(",")[1])
```

This will not give the output there and then only. It requires action which will cover in next 10 mins or so.

Difficult to digest ?

let's take an real time example to have more clarity

Let's say we have to make an omelette so to build that we need a recipe , which are nothing but steps needed to get the final product. Below are the steps for same :

a. Wash the veggies
b. Add the eggs
c. Beat until frothy
d. Add oil to Pan and heat it to some level
e. When it is hot enough, pour the eggs mixture
f. Wait until base firms up then flip it to other side
g. switch off the stove
h. **Serve it with rice , roti or bread. (Action : last step)**

Yuppy!!!! omelette is done. but is it actually done? We just listed the steps , we have not gone ahead and cooked .

- This is exactly what a RDD behaves like, until and unless an action has been taken,
- RDD is just a recipe where things are known to be done , steps are created but not executed.
- Once an action has been called then only execution will start.
- Here step **h is dependent for g to execute and this too till step a**.

How can we create RDD?

To create a RDD , we have 3 methods:

a. From **Texfile**
b. From **Collections**
c. From **Transformation**

Sample code for each of them :

a. **TexFile** :

```
wordsRDD = sc.textFile("../data-files/movielens/movies.csv")
```

In our problem statement , we loaded data into RDD by reading through a movies.csv which can be present on local or HDFS or any cloud storage(bucket) and textFile does not only conform to csv , any file(txt,csv,tsv,) with textFile()

b. **Collections**:

```
dataRDD = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

If we have data in collections like map,set,List, array and we want to create RDD from that, spark provides a parallelize method.

c. **Transformation:**

```
finalRDD = wordsRDD.map(lambda word: (word.split(",")[2], word.split(",")[1])
```

when we work RDD and perform any type of transformation (which will be covered later), we get RDD back(finalRDD)

finalRDD is the new RDD that we will get after applying split transformation on wordsRDD RDD.

What operations are supported by RDD ?

- **Transformations**
- **Actions**

1. **Transformations** are spark operations which result in a single or multiple new RDD's like

```
finalRDD = wordsRDD.map(lambda word: (word.split(",")[2], word.split(",")[1])
```

➔ **wordsRDD** is parent RDD
➔ **map** is transformation
➔ **finalRDD** is new RDD created

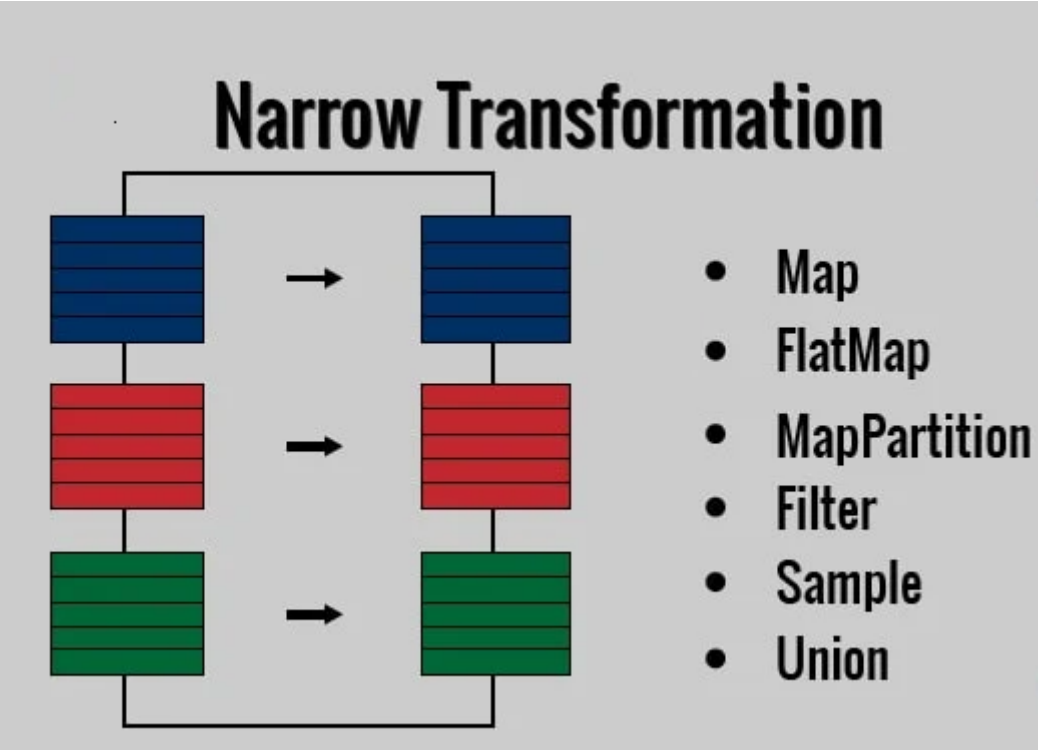
- As we know, RDDs are partitioned and distributed across multiple server nodes.
- A parent partition may have a dependency on one or more child partitions.
- An RDD models the relationships between an RDD and its partitions and the partition which it was derived from.
- This relationship introduces some new concepts like Narrow and Wide dependencies.

Two types are transformations/dependencies exists:

a. Narrow dependencies
b. Wide dependencies

Narrow Dependencies

- When each partition of parent RDD is used by at most one partition of the child RDD, then we have a narrow dependency.
- Narrow transformations do not require any **data shuffling over the cluster network,** so they are efficient.
- Examples: map, filter, union

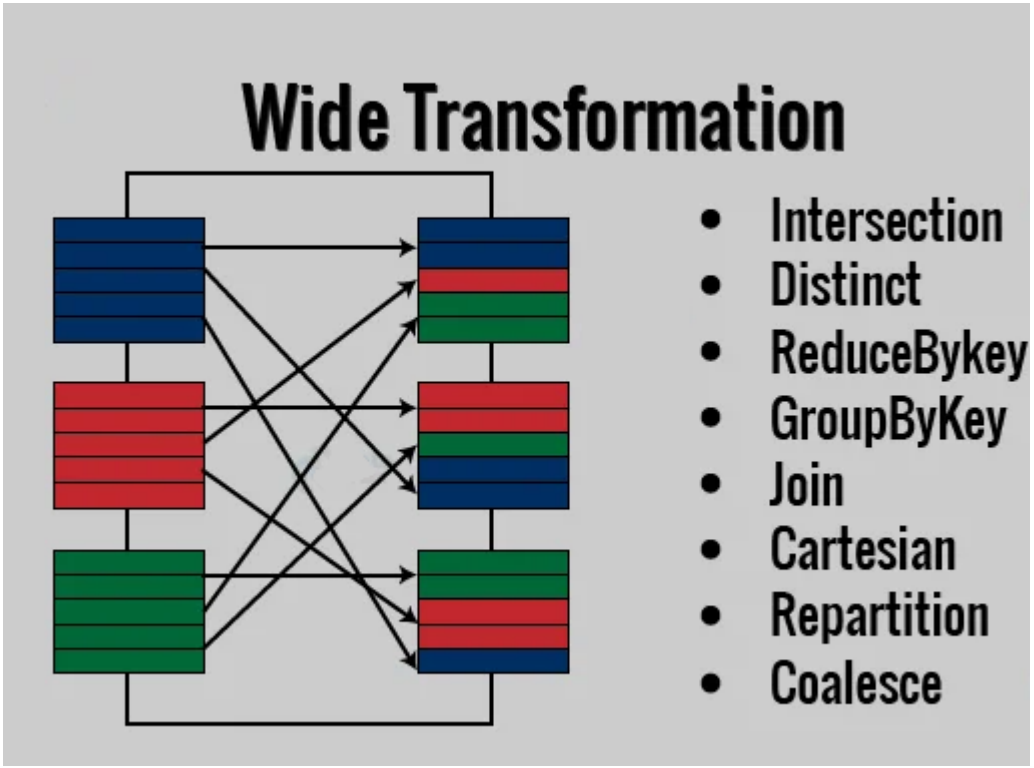


Wide dependencies

- When each partition of the parent RDD may be depended on by multiple child partitions.
- The wide dependencies are costlier in terms of performance as they cause network shuffling.

Example:

groupByKey operations and join operations whose inputs are not co-partitioned.



How to use transformation in order to solve our problem?

Statement :

Which comedy movie has the most ratings and for the same return the title and the number of rankings.?

Solution : Spark way

To solve this , we need to first apply

Transformation 1 : group ratings by unique movie
Transformation 2 : join with the filtered comedies
Transformation 3 : map to extract title and number of rankings

```
comedyRDD = ratings.groupBy(lambda x: x[1]).keyBy(lambda x: x[0]) \
.join(movies.filter(lambda x: x[2].count('Comedy')!=0).keyBy(lambda x: x[0])) \
.map(lambda (key, (k,v)): (key, v[1], k[1].__len__())) \
.max(key=lambda x: x[2])
```

So we got the result in comedyRDD but still have no clue on how to see data either or console or dump the same into file the same way we need **ACTION operation in RDD.**

Actions

Actions are applied on an RDD to instruct Apache Spark to apply computation and pass the result back to the driver.

Few more sample of action are :

Action	Description
collect()	It returns all the elements of the dataset as an array at the driver program.
count()	It returns the number of elements in the dataset.
first()	It returns the first element of the dataset (similar to take(1)).
take(n)	It returns an array with the first n elements of the dataset.
saveAsTextFile(path)	It is used to 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

Now finally to view output we will call collect on the `comdeyRDD` and result will be printed on screen as:

```
comdeyRDD.collect();
```

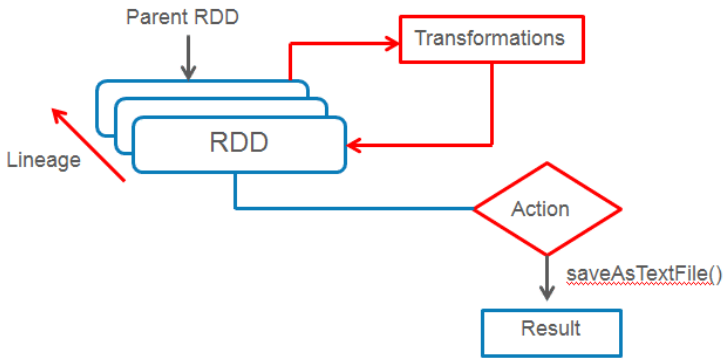
output will be :

('2858', 'American Beauty (1999)', 3428)

If you want to save this to hard disk then on same rdd call saveAsTextFile action like :

```
comdeyRDD.saveAsTextFile("/user/hadoop/output")
```

and output can be seen under /user/hadoop/output directory.



Key differences between transformations and actions.

Transformation	Actions
Creates a new dataset from an existing one.	Returns a value to the driver program after computation on the dataset.
Lazily evaluated. Spark only creates a lineage when a transformation is called on it.	Actual work only happens when an action is encountered by Spark.
Map, flatMap, reduceByKey are some examples of transformations.	Count, reduce, take, collect are some examples of actions.

Are there any different kinds of RDD too present ?

Yes they are:

a. pairRDD :

Pair RDD is just a way of referring to an RDD containing key/value pairs, i.e. tuples of data. For example: in our question we need tuples to get final output so finalRDD is a PairRDD.

```
finalRDD = wordsRDD.map(lambda word: (word.split(",")[2], word.split(",")[1])
```

b. DoubleRDD:

An RDD consists of a collection of double values.Due to this property, many statistical functions are available to use with the DoubleRDD.

Example :

rdd_one = sc.parallelize(Seq(1.0,2.0,3.0))

output : org.apache.spark.rdd.**RDD[Double]** = ParallelCollectionRDD[52] at parallelize at

How to visualize RDD workflow ?

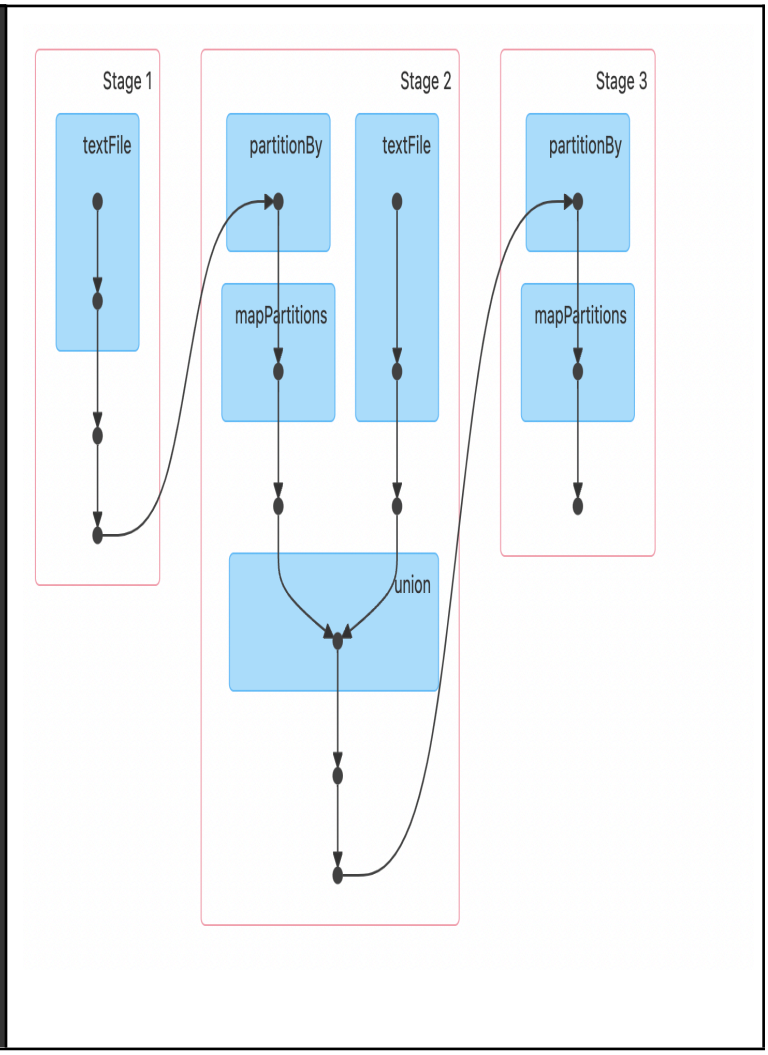
Answer : DAG(Directed Acyclic Graph)

- When we call transformations on RDD, Spark just maintains a lineage or a graph internally. The graph is called **Directed Acyclic Graph (DAG).**
- DAG is a combination of Vertices as well as Edges.
- In DAG, vertices represent the RDDs and the edges represent the Operation to be applied on RDD.
- Every edge in DAG is directed from earlier to later in a sequence.

```
# Read a text file as an RDD.
fileRDD =
sc.textFile("../data-files/
movielens/movies.csv")

# Apply flatMap
transformation.
wordsRDD =
fileRDD.flatMap(lambda x:
x.split(","))

# Apply map transformation on
newRDD
tupleRDD =
ratings.groupBy(lambda x:
x[1]).keyBy(lambda x: x[0]) \
.join(movies.filter(lambda x:
x[2].count('Comedy')!=0).keyBy(l
ambda x: x[0])) \
.map(lambda (key, (k,v)): (key,
v[1], k[1].__len__())) \
.max(key=lambda x: x[2])
```



```
# Calculate word count using reduceByKey
tupleRDD.collect()
```

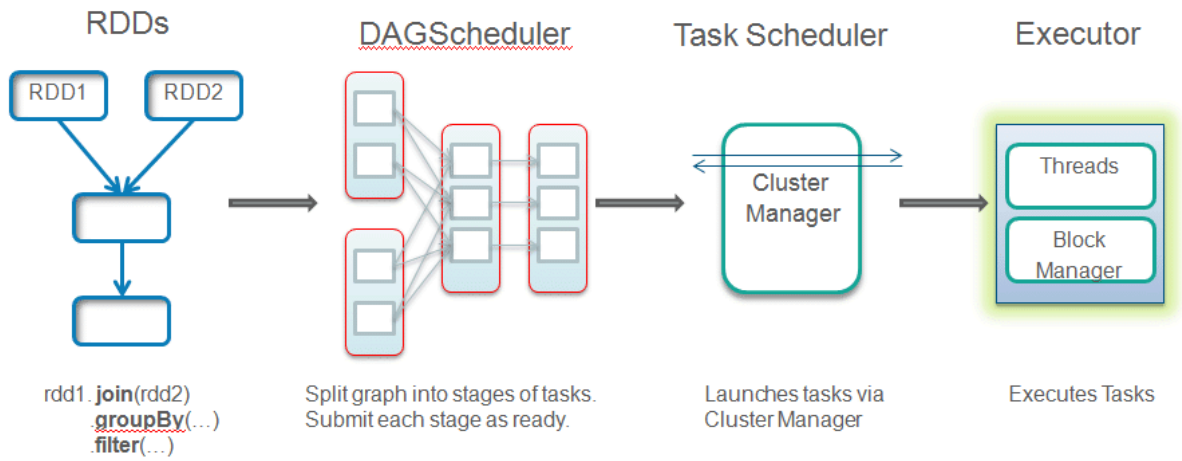
In the above example, we called transformations join, map and keyBy on the original RDD. As you can see Spark created an RDD or lineage.

The purpose of DAG is

- enables optimizations
- Fault-recovery in case of task failures.

Connecting all dots in single Picture

Lastly, let's talk about the execution flow of a Spark program. Whatever program you write is further divided into tasks which run in isolation and parallel. The Spark execution workflow is shown below:



Can I make my existing solutions more optimized?

In industry there are multiple ways to optimize spark RDD program few common ways are :

a. **Calling Cache() or Persist() on RDD**

```
movies = sc.textFile("../data-files/movielens/movies.csv")
ratings = sc.textFile("../data-files/movielens/ratings.csv")

#group ratings by unique movie
groupRDD = ratings.groupBy(lambda x: x[1]).keyBy(lambda x: x[0])

#caching the RDD for optimizing
groupRDD.cache()
#join with the filtered comedies
joinRDD = groupRDD.join(movies.filter(lambda x:
x[2].count('Comedy')!=0).keyBy(lambda x: x[0]))

#count ratings per comedy

finalRDD = joinRDD.map(lambda (key, (k,v)): (key, v[1], k[1].__len__()))
.max(key=lambda x: x[2])



# Show the final comedy movie having the most ratings
finalRDD.collect()
```

How is Caching achieved in RDD?

- RDD is recomputed every time it is materialized.
- It is a good idea to put an RDD in memory if we are doing some iterative analysis on top of an RDD.
- **Cache() and Persist()** methods are used for the same.
- To Persist an RDD in memory, use **groupRDD.cache()**
- To unpersist an RDD, use **groupRDD.unpersist(blocking=False)**

What are different Storage Levels?

Storage Level	Meaning
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 (Java and Scala)	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 (Java and Scala)	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.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

b. Shared Variables

- When a function passed to a Spark operation is executed on a remote cluster node, it works on separate copies of all the variables used in the function.

- These variables are copied to each machine, and no updates to the variables on the remote machine are propagated back to the driver program.

However, Spark does provide two limited types of *shared variables* for two common usage patterns

a. Broadcast variables
b. Accumulators.

Broadcast Variables

- Allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.
- Are created from a variable v by calling SparkContext.broadcast(v).

Accumulators

- Accumulators are equivalent to distributed counters that are shared between multiple tasks.
- Tasks can update the shared variable.
- Only the driver can see the current value of an accumulator.

Key Points

- When a job is submitted, the driver implicitly converts user code that contains transformations and actions into a logically Directed Acyclic Graph called DAG.

- DAG Scheduler converts the graph into stages. A new stage is created based on the shuffling boundaries.

- Now the driver talks to the cluster manager and negotiates the resources. Cluster manager launches executors in worker nodes on behalf of the driver. At this point, 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.

- While the job is running, the driver program will monitor and coordinate the running tasks. Driver node also schedules future tasks based on data placement.

Summary:

In this module we were able to focus on :

- Need Spark with a case study.
- Complete View of Spark Architecture
- Concepts of RDD , action,transformation, job, task
- Detailing out narrow vs wide dependencies
- Finally Working demo of spark with python