



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

DSECL ZG 522: Big Data Systems

Session 14: Spark -Part 2

Janardhanan PS

Professor

janardhanan.ps@wilp.bits-pilani.ac.in

Topics for today

- **RDD operations**
- Standalone application
- Cluster architecture



Types of operations

- Transformations: Produce new RDD from existing RDD.
 - ✓ Create a lineage of transformations - DAG.
 - ✓ Lazy i.e. not done till an action is performed.
 - ✓ e.g. map(), filter()
- Actions: Actually work on the data but new RDD is not formed.
 - ✓ Non-RDD values returned to driver program or put on storage
 - ✓ Trigger transformations to create lineage of RDDs required
 - ✓ e.g. count()



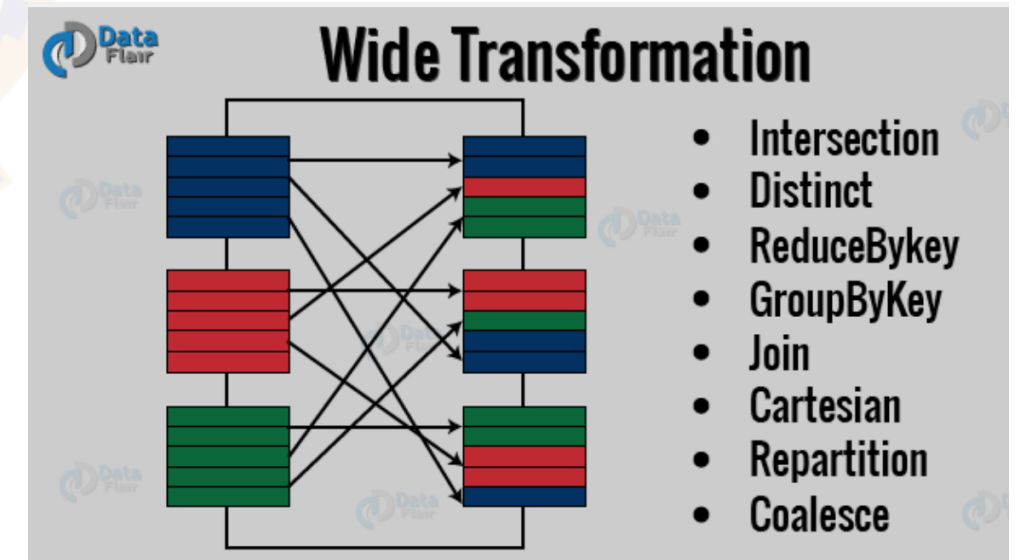
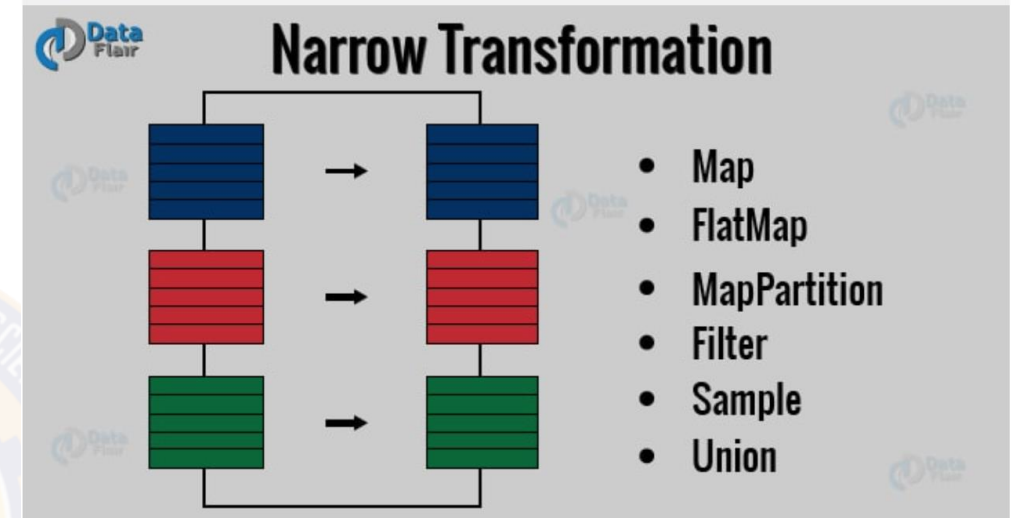
Transformations

- Narrow

- ✓ All elements required to compute a partition of result RDD are in the same partition in parent RDD.
- ✓ No shuffles across nodes.
- ✓ Examples:- `map()`, `mapPartition()`, `flatMap()`, `filter()`, `union()`

- Wide :

- ✓ Elements required to compute a partition of result RDD are in multiple partitions in parent RDD.
- ✓ Need shuffle across nodes.
- ✓ Examples:- `groupByKey()`, `aggregateByKey()`, `aggregate()`, `join()`, `repartition()`



Transformations

- `map(f)`: Apply a function `f` over every element of an RDD and returns only one element - so `N` elements can create `N` output RDDs
- `flatMap(f)`: Same as `map` but can create an RDD with a list of `N` elements
- `filter(p)`: Return elements that only satisfy a predicate
- `mapPartition(f)` / `mapPartitionWithIndex(f)`: Same as `map` but `f` is applied on each partition of the input RDD in parallel instead of `f` being called on each element. Done to avoid overheads of moving data across cluster nodes, e.g. during initialisation.
- `union()`/`intersection()`: Create new RDD with union/intersection of elements of input RDDs
- `distinct()`: Returns distinct elements
- `groupByKey()`: Shuffles data by key in `(k, v)` lists. Leads to data movement across nodes / partitions.
- `reduceByKey(f)`: Similar to a combiner in Hadoop MapReduce
- `sortByKey()`: Create RDD with sort on key where we input `(k, v)` pairs in an RDD
- `join()`: Create RDD with join on key of input RDDs which have `(k, v)` lists
- `coalesce(n)`: Reduce the number of partitions of the RDD to `n`

Actions

- `count()` : Return number of elements in RDD
- `collect()`: Return RDD to driver and check whether it fits in memory
- `take(n)`: Return any n elements from RDD and minimise number of partitions it touches.
- `top(n)`: Return first n elements considering default order
- `countByValue()`: Return number of occurrences of each value
- `reduce()`: Take the set of elements of an RDD and apply a supplied commutative and associative operation on it, e.g. add
- `fold()`: Like reduce but takes an initial value, e.g. 0 for + or 1 for *
- `aggregate()`: Different from reduce/fold because datatype of output can be different from input. One function to combine elements from RDD and then another to combine next level.
- `foreach()`: Work with each element in RDD but not return to driver. E.g. print each element.

Pair RDD

- RDD with (key, value) pairs
 - ✓ Common format for processing data sets
- Enables a set of transformation operations
 - ✓ groupByKey
 - ✓ reduceByKey
 - ✓ combineByKey
 - ✓ mapValues(f) : just apply f to value
 - ✓ keys()
 - ✓ values()
 - ✓ sortByKey()
- and action operations
 - ✓ countByKey()
 - ✓ collectAsMap()
 - ✓ lookup(key)



Examples (1)

```
> spark-shell
scala> val bankdata = sc.textFile("bank.csv")
scala> bankdata.take(1)
  Array[String] =
  Array(age,job,marital,education,default,balance,housing,loan,contact,day,month,duration,campaign,pdays,previous,poutcome,deposit)
scala> val bankdata2 = bankdata.flatMap(lines => lines.split("\n")).filter(value =>
value.contains("admin"))
scala> bankdata2.count()
  Long = 11334
scala> val bankdata3 = bankdata2.map(x => (x.split(",")(3), x.split(",")(0).toInt))
scala> bankdata3.foreach(println)
  (tertiary,37)
  (secondary,59)
  (secondary,56)
  (secondary,29)
  (tertiary,54)
  (secondary,28)
  (tertiary,33)
  (secondary,38)
  ...
```

convert each line to RDD and filter
#records

extract education and age

Examples (2)

```
scala> val bankdata4 = bankdata3.reduceByKey((x,y)=>x+y)
```

```
scala> bankdata4.foreach(println)
```

```
(secondary,42848)
```

```
(tertiary,5817)
```

```
(primary,2003)
```

```
(unknown,1857)
```

```
scala> val bankdata5 = bankdata4.map(_._2.swap)
```

```
scala> bankdata5.foreach(println)
```

```
(42848,secondary)
```

```
(5817,tertiary)
```

```
(2003,primary)
```

```
(1857,unknown)
```

```
scala> val sorted = bankdata5.sortByKey(numPartitions=1)
```

```
scala> sorted.foreach(println)
```

```
(1857,unknown)
```

```
(2003,primary)
```

```
(5817,tertiary)
```

```
(42848,secondary)
```

```
scala> bankdata4.lookup("primary")
```

```
res39: Seq[Int] = ArrayBuffer(2003)
```

add up all ages by education

- just want to show how numbers are added up

swap key and value

sort : limit to one partition to avoid

different sort orders since sorting is per partition

find value for a key

Examples (3)

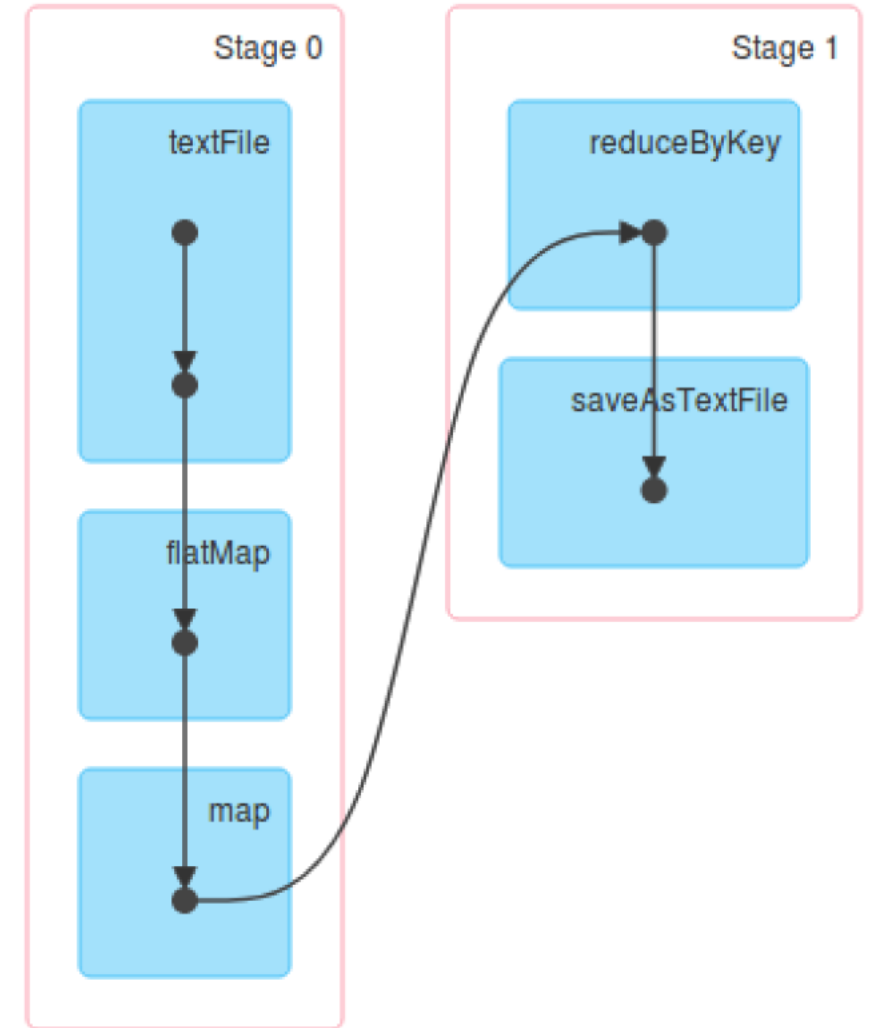
```
scala> val countsbykey = bankdata3.countByKey() # this is not an RDD because it is action operation
scala> val counts = sc.parallelize(Seq(countsbykey))
scala> counts.foreach(println)
  Map(secondary -> 1084, tertiary -> 168, primary -> 42, unknown -> 40)
scala> val counts2 = counts.flatMap(x=>x)
scala> counts2.foreach(println)
  (secondary,1084)
  (tertiary,168)
  (primary,42)
  (unknown,40)
scala> val joinrdd = bankdata4.join(counts2)
scala> joinrdd.foreach(println)
  (secondary,(42848,1084))
  (tertiary,(5817,168))
  (primary,(2003,42))
  (unknown,(1857,40))
scala> val avg = joinrdd.mapValues{case (x,y) => (1.0 * x/y)}
scala> avg.foreach(println)
  (secondary,39.52767527675277)
  (tertiary,34.625)
  (primary,47.69047619047619)
  (unknown,46.425)
```

showing some manipulations on
creating RDD from a list, joining 2
RDDs, and doing an average
calculation on 2 columns



Directed Acyclic Graph of RDDs

- Vertices are RDDs
- Edges are operations
- Operations are divided into stages
- Each stage has a set of tasks working on a partition
- Tasks may execute in parallel
- Why DAG ?
 - ✓ Avoid Hadoop MR limitation for iterative / interactive applications when each MR operation is considered independent - so wasting resources on memory / IO
 - ✓ Breaking upto into DAG and stages helps to optimise execution plan and avoid shuffling data around and sharing data between stages
 - ✓ Operations that don't need shuffling can be put in same stage and then shuffling happens across stages.
 - ✓ Achieves fault tolerance by capturing lineage of operations. So RDD partitions can be reconstructed on node failures.



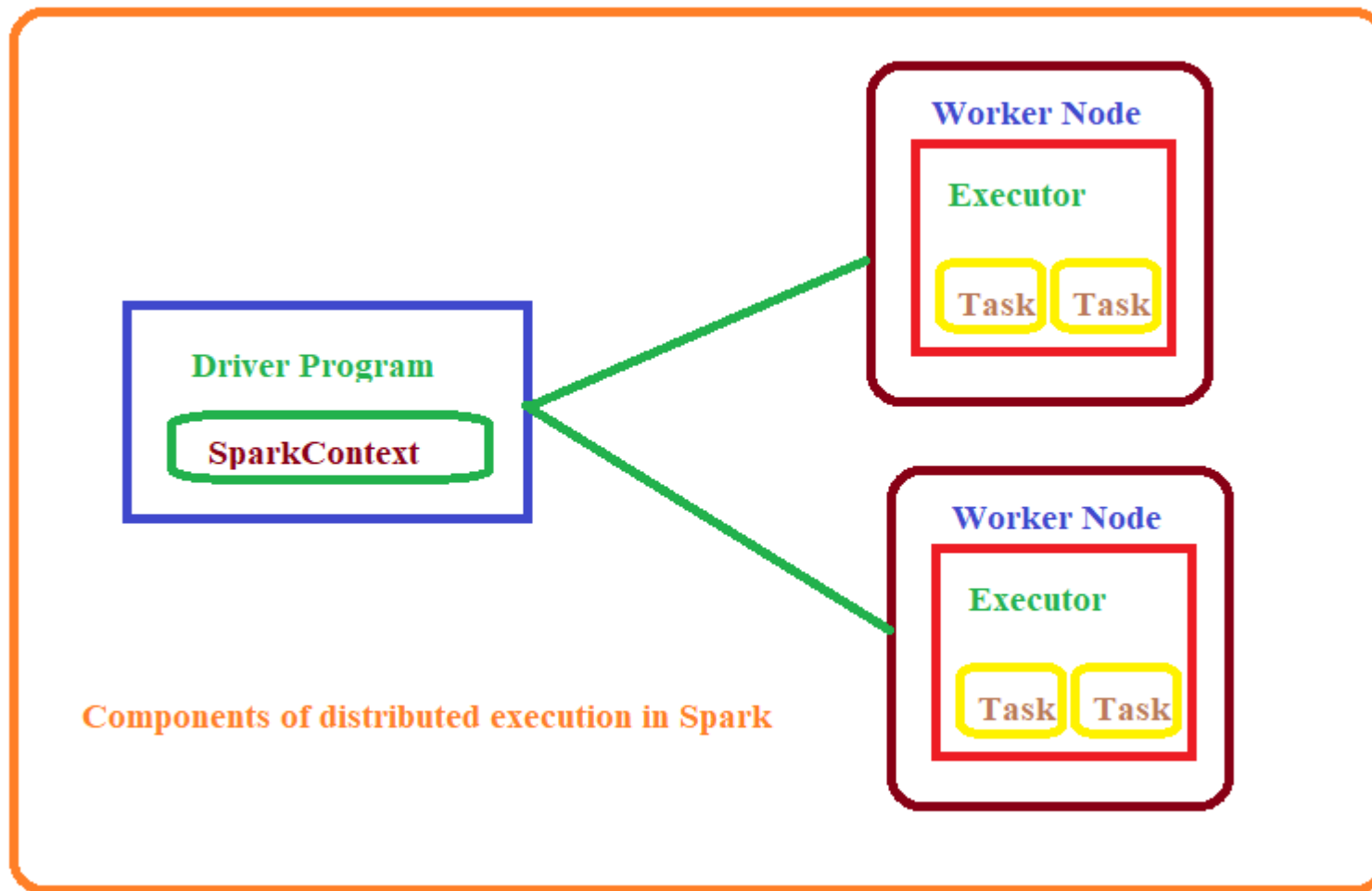
Topics for today

- RDD operations
- **Standalone application**
- Cluster architecture



Core Spark Concepts

Components of distributed execution



Core Spark Concepts (2)

Driver Program

- Every Spark application consists of driver program that launches various parallel operations on a cluster
- Driver program
 - ✓ contains main function
 - ✓ defines distributed datasets on cluster
 - ✓ then applies actions to them
 - Shell is the driver program, hence able to do operations in it
- Driver program accesses Spark through SparkContext object, which is connection to the cluster
 - ✓ In shell, SparkContext is automatically created and accessed as “sc”
 - ✓ SparkContext is used to build RDDs
- RDDs can be used to run various operations on them
- To run these operations, driver program manage number of nodes called executors
 - ✓ Executors will be present on the different nodes

Standalone Applications


Using Python

- Spark can be linked into standalone applications in Java, Scala or Python
 - ✓ Need to initialize the SparkContext in program
 - ✓ After that API is same!
- In Python,
 - ✓ Simply write applications as Python scripts
 - ✓ Run them using the bin/spark-submit script
 - ❖ Spark-submit includes the Spark dependencies in Python
 - ❖ Sets up environment for Spark's Python API to function
 - ❖ Need to have pyspark package installed on the system

Standalone Applications(2)

Initializing and Using SparkContext

- Import the packages in program
- Create SparkConf – to configure the applications
- Create SparkContext
 - ✓ A cluster URL, which tells Spark how to connect to cluster
 - ✓ Local is special value that runs Spark on one thread on local machine
 - ✓ Application name – to identify application on the cluster managers UI
- Use methods to create RDDs
- Use APIs to carry out operations on the RDDs
- Call stop() method SparkContext object to stop execution of application



```
my_script.py
from pyspark import SparkConf, SparkContext

print("Packages imported!")

conf = SparkConf().setMaster("local").setAppName("MyScript")
sc = SparkContext(conf = conf)


print('Able to contact Spark!')

lines = sc.textFile("README.md")
print('*****')
print(lines.count())
print('*****')
```

Standalone Applications(3)

Executing program

- Using spark-submit , execute the program



```
File Edit View Search Terminal Help
```

```
[csishydlab@apache-spark spark-2.4.4-bin-hadoop2.7]$ spark-submit my_script.py
```

```
20/04/15 13:31:51 INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 0 (PythonRDD[2] at count at /home/csishydlab/spark-2.4.4-bin-hadoop2.7/my_script.py:12) (first 15 tasks are for partitions Vector(0))
20/04/15 13:31:51 INFO TaskSchedulerImpl: Adding task set 0.0 with 1 tasks
20/04/15 13:31:51 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, localhost, executor driver, partition 0, PROCESS_LOCAL, 7917 bytes)
20/04/15 13:31:51 INFO Executor: Running task 0.0 in stage 0.0 (TID 0)
20/04/15 13:31:51 INFO HadoopRDD: Input split: file:/home/csishydlab/spark-2.4.4-bin-hadoop2.7/README.md:0+3952
20/04/15 13:31:52 INFO PythonRunner: Times: total = 315, boot = 266, init = 49, finish = 0
20/04/15 13:31:52 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1547 bytes result sent to driver
20/04/15 13:31:52 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 428 ms on localhost (executor driver) (1/1)
20/04/15 13:31:52 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
20/04/15 13:31:52 INFO PythonAccumulatorV2: Connected to AccumulatorServer at host: 127.0.0.1 port: 58581
20/04/15 13:31:52 INFO DAGScheduler: ResultStage 0 (count at /home/csishydlab/spark-2.4.4-bin-hadoop2.7/my_script.py:12) finished in 0.486 s
20/04/15 13:31:52 INFO DAGScheduler: Job 0 finished: count at /home/csishydlab/spark-2.4.4-bin-hadoop2.7/my_script.py:12, took 0.539315 s
105
```

```
*****
```

Topics for today

- RDD operations
- Standalone application
- **Cluster architecture**

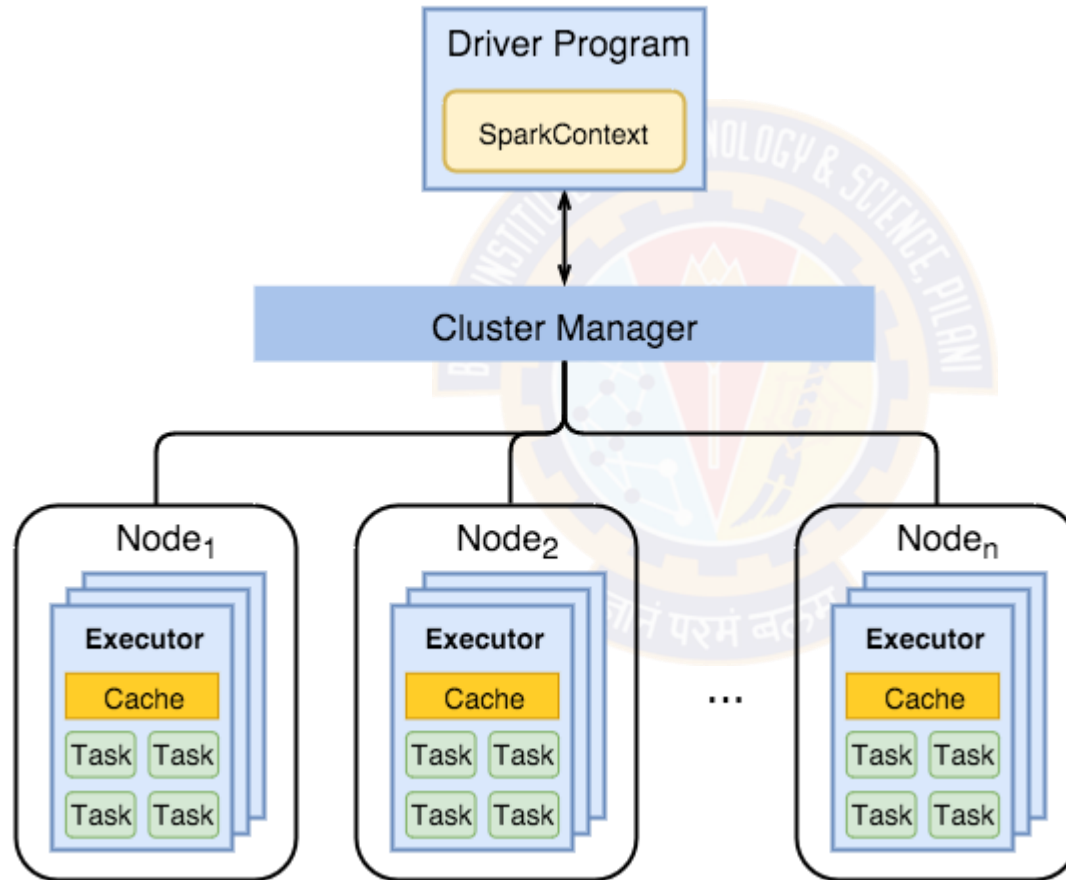


Motivation

- Till now executed applications on the local (standalone) mode
- How to move to the cluster?
 - ✓ If same application needs to be run on the cluster, nothing needs to be changed!
 - ✓ Just need to add more machines and run it in the cluster mode!
 - ✓ All this possible because of high level API of Spark!
- Can rapidly prototype applications on smaller dataset locally
- Run unmodified code on very large clusters !

Spark Runtime Architecture

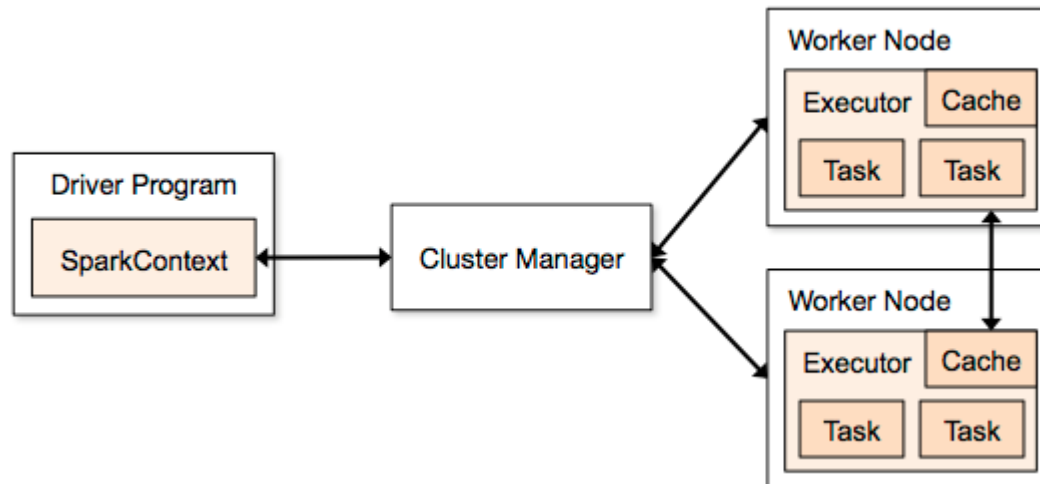
Components of Distributed Spark Application



Source : [Spark intro blog](#)

Spark Runtime Architecture (2)

- Uses Master/Slave architecture with one central coordinator and many distributed workers
 - ✓ Central coordinator – driver
 - ✓ Driver communicates with large number of workers – executors
 - ✓ Both run in their separate Java processes
 - ✓ Driver and Executor together termed as Spark Application!
- Spark application launched on set of machines using external service called cluster manager
 - ✓ Spark has built-in standalone cluster manager
 - ✓ Also supports Hadoop YARN, Apache Mesos as cluster managers



Driver

- Driver is the central coordinator for Spark application
 - ✓ Runs in separate Java process
 - ✓ It's the process where main() method of program runs
- It's the process running the user code that
 - ✓ Creates SparkContext
 - ✓ Creates RDDs
 - ✓ Performs transformations and actions
- Once the driver terminates, application ends
- Two responsibilities
 - ✓ Converting user program into tasks
 - ✓ Scheduling tasks on executors



Driver (2)

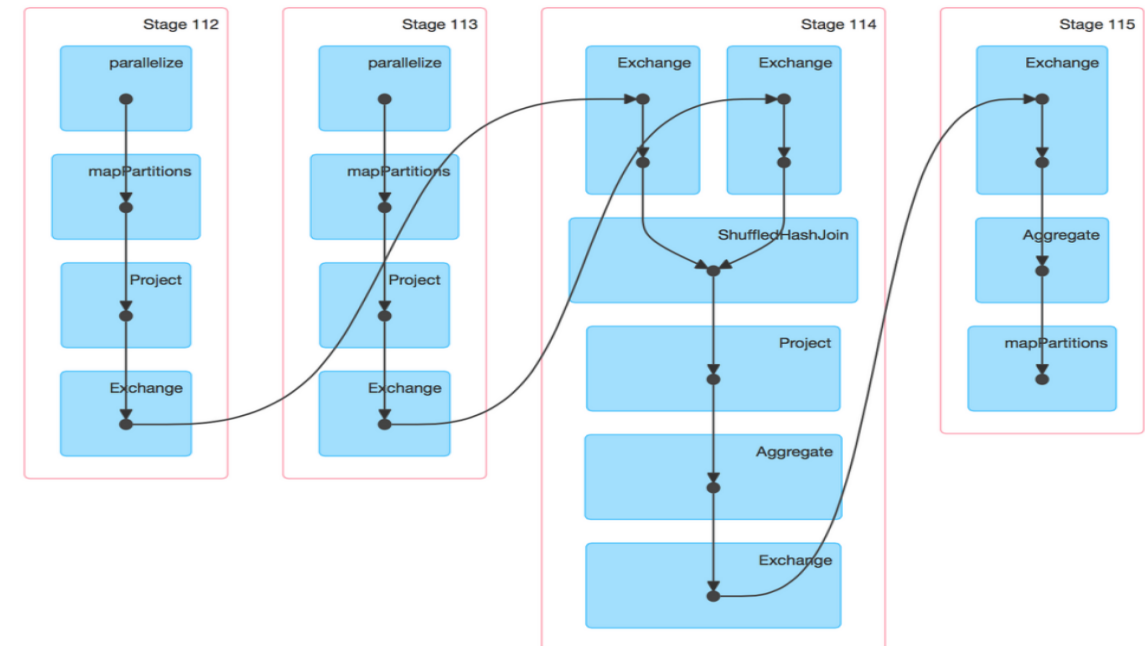
Two Duties - Converting user program into tasks

- Converting user program into tasks
 - ✓ Physical execution unit is called as task
 - ✓ Smallest unit of work – task!
 - ✓ Programs creates SparkContext, Create RDDs and perform transformations, actions on RDDs
 - ✓ Implicitly a logical directed acyclic graph (DAG) of operations is created
 - ✓ When driver runs, it converts the DAG into physical execution plan
 - ✓ Converts execution plan into stages which in turn consists of tasks
 - ✓ Tasks are bundled up and sent to the cluster

Details for Job 8

Status: SUCCEEDED
Completed Stages: 4

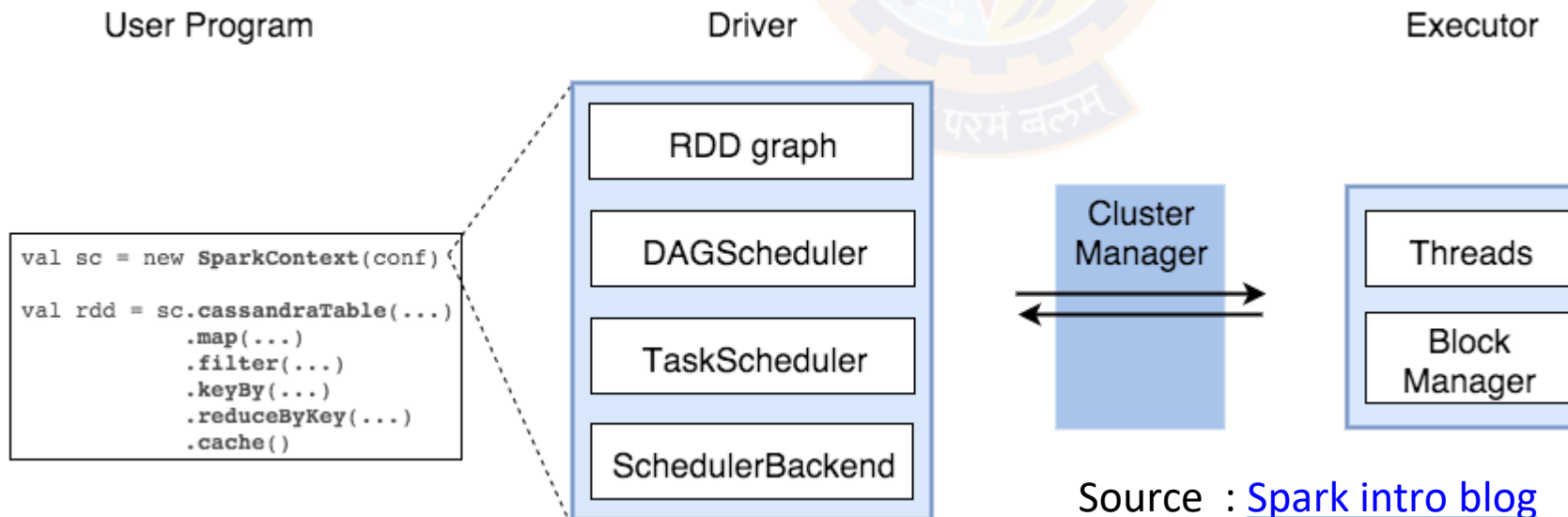
► Event Timeline
▼ DAG Visualization



Driver (3)

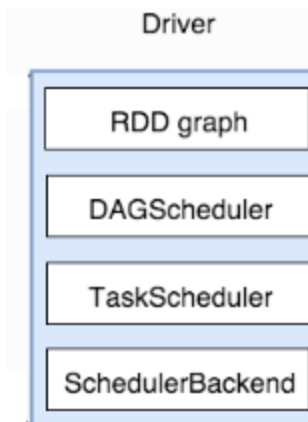
Two Duties – Tasks Scheduling

- Scheduling the tasks on executors
 - ✓ With execution plan, Driver needs to coordinate scheduling of tasks on executors
 - ✓ When executor starts, it gets registered with driver, so driver has complete info about them
 - ✓ Executor is process capable of running tasks and storing RDD data
- Based on data placement, Spark will identify the executors to schedule the tasks
 - ✓ When tasks completed on executor, they might have cached data
 - ✓ Driver uses that cached data to schedule future tasks based on that data



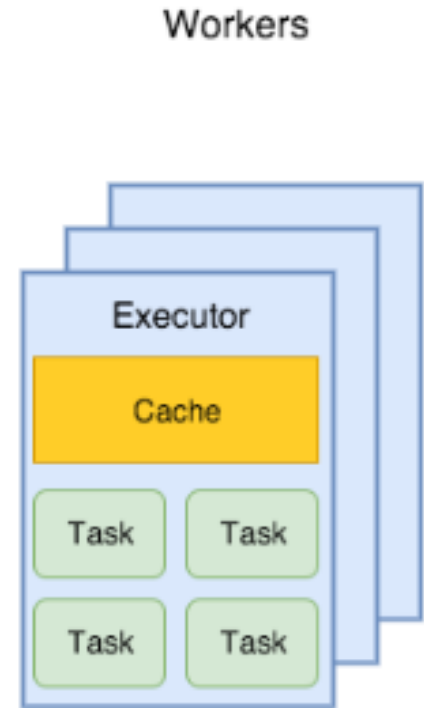
Driver: Program translation into Tasks

- DAGScheduler
 - ✓ computes a DAG of stages for each job and submits them to TaskScheduler
 - ✓ determines preferred locations for tasks (based on cache status or shuffle files locations)
 - ✓ finds minimum schedule to run the jobs
- TaskScheduler
 - ✓ responsible for
 - ✓ sending tasks to the cluster
 - ✓ running them,
 - ✓ retrying if there are failures, and mitigating stragglers
- SchedulerBackend
 - ✓ backend interface for scheduling systems that allows plugging in different implementations (Mesos, YARN, Standalone, local)
- BlockManager
 - ✓ provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk)



Executors

- Worker processes responsible for running individual tasks in given Spark job
- Launched when application is started and typically run till lifetime of application
- Even if executors fails, Spark application does not fail
- Roles of executors
 - ✓ Run the tasks and return the result to the driver
 - ✓ Provide in-memory storage for RDDs that are cached by user programs

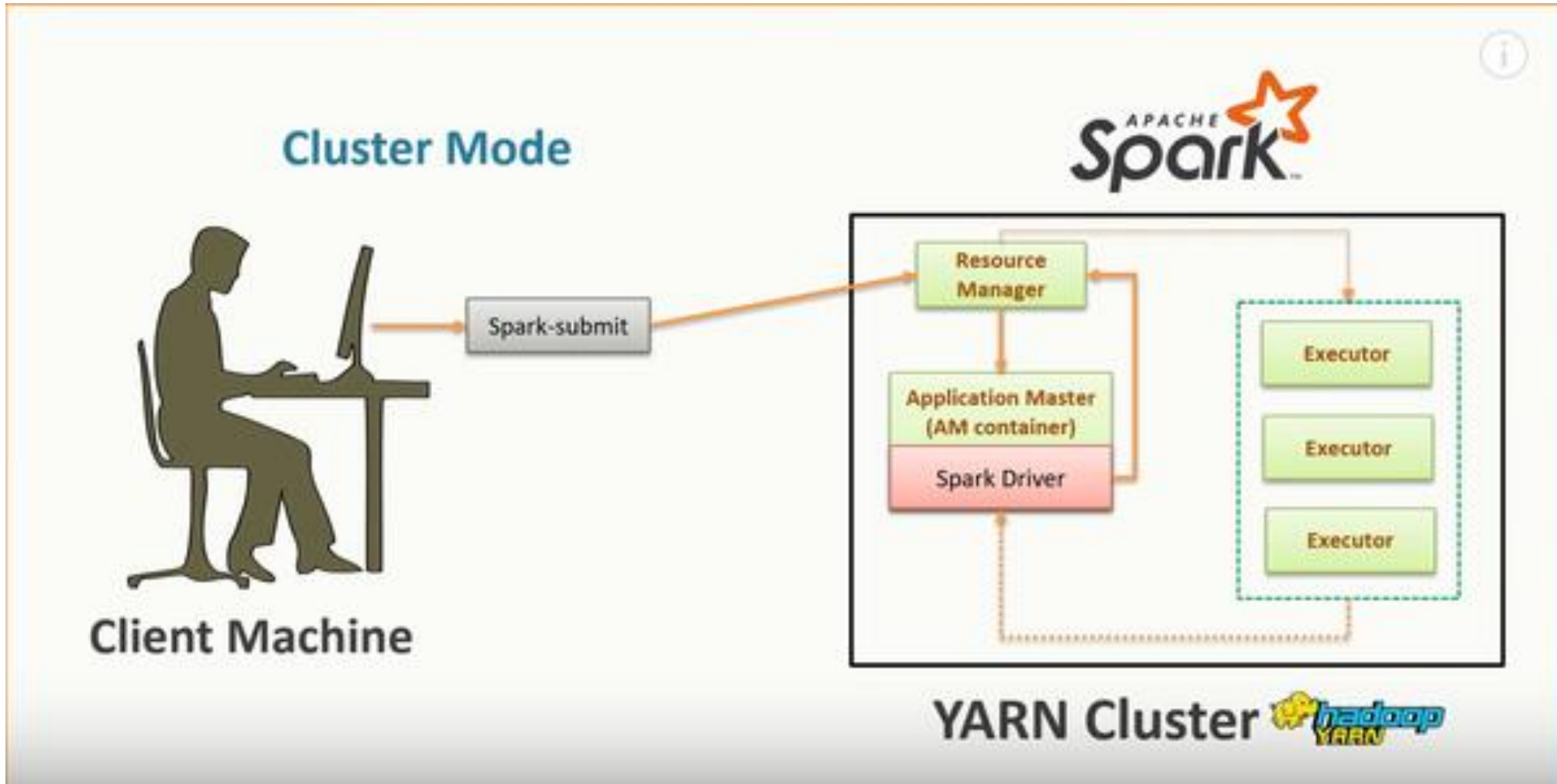


Cluster Manager

- Cluster manager is used to launch the Spark application
- Pluggable component
- Allows Spark to run on top of different external managers like YARN, Mesos as well as standalone cluster manager
- Single script to submit your application to cluster i.e. spark-submit
 - ✓ With different options, it can be used to connect to different clusters
 - ✓ Used to manage the resources allotted to the application



Application execution



Source : [Quora answer](#)

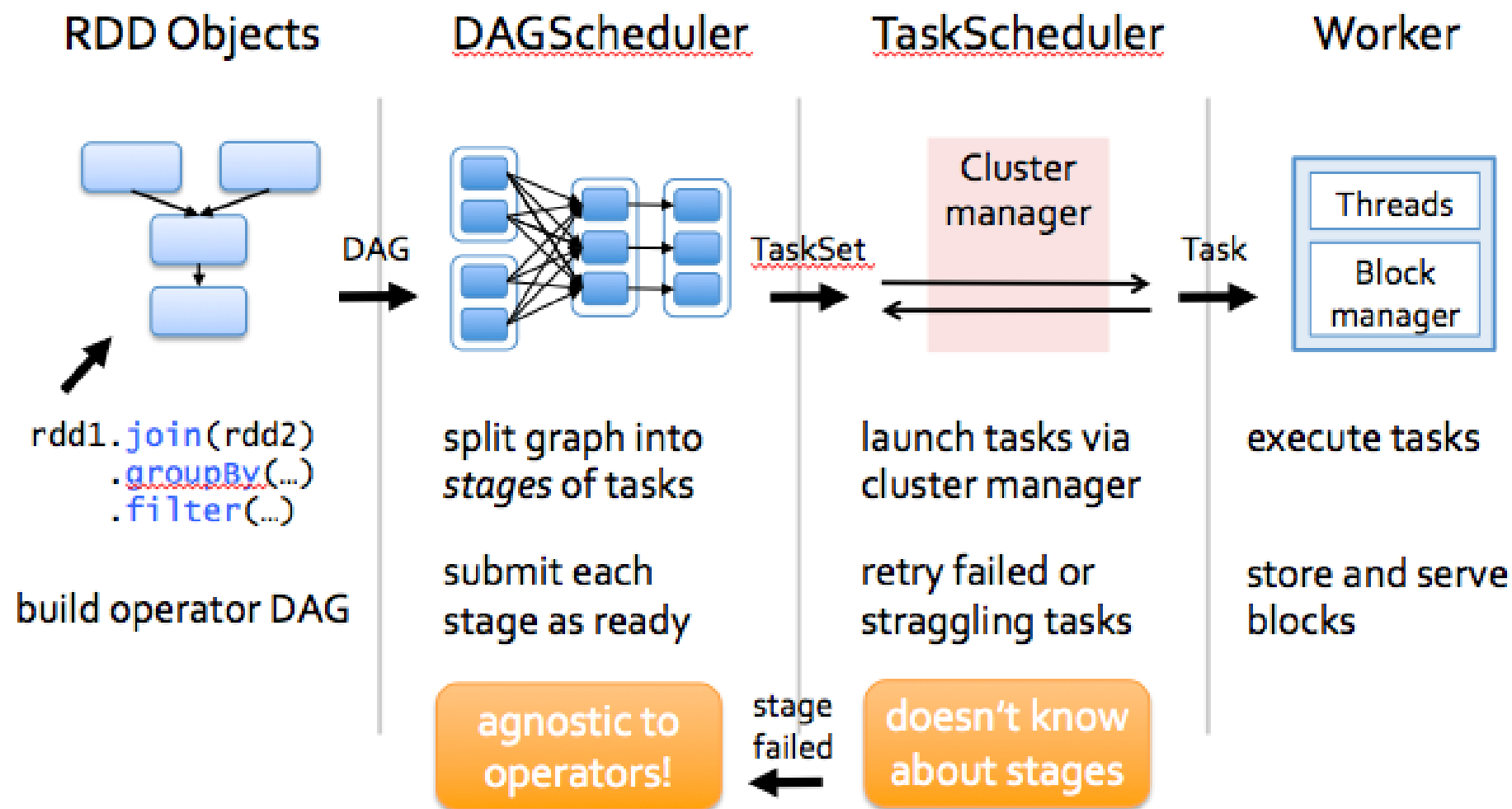
Application execution (2)

Detailed

1. User submits the application using `spark-submit`
2. `Spark-submit` launches the driver program and invokes the `main()` method of program
3. Driver program contacts the cluster manager to ask for resources to launch executors
4. Cluster manager launches the executors on the behalf of driver program
5. Driver process runs through user application
 - Based on RDD transformations and actions in the program, driver sends work to executors in form of tasks
6. Tasks are run on executor processes to compute and save results
7. If the driver's main method exits or calls `SparkContext.stop()`, it will terminate all executors and release resources from the cluster manager

Application execution (3)

Application flow



Cluster concepts

Application	User program built on Spark. Consists of a <i>driver program</i> and <i>executors</i> on the cluster.
Application jar	A jar containing the user's Spark application. In some cases users will want to create an "uber jar" containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime.
Driver program	The process running the <code>main()</code> function of the application and creating the <code>SparkContext</code>
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
Deploy mode	Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
Worker node	Any node that can run application code in the cluster
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.
Task	A unit of work that will be sent to one executor
Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. <code>save</code> , <code>collect</code>); you'll see this term used in the driver's logs.
Stage	Each job gets divided into smaller sets of tasks called <i>stages</i> that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

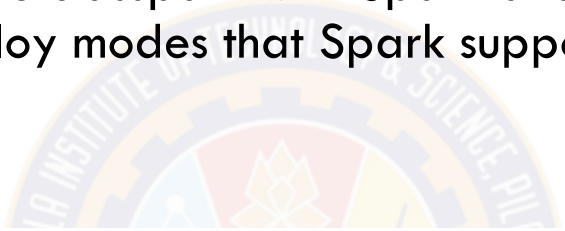
Submitting Applications

Spark-submit

- Used to launch applications on a cluster
- Can use all of Spark's supported cluster managers through a uniform interface
 - ✓ don't have to configure application especially for each one
- Bundling Application's Dependencies
 - If code depends on other projects, need to package them alongside application in order to distribute the code to a Spark cluster
 - ✓ Need to create an assembly jar (or "uber" jar) containing code and its dependencies
 - ✓ Both sbt and Maven have assembly plugins
 - ✓ Spark and Hadoop as provided dependencies – provided at runtime
 - ✓ Call the bin/spark-submit script while passing jar
 - For Python,
 - ✓ Can use the --py-files argument of spark-submit to add .py, .zip files to be distributed with application
 - ✓ If depend on multiple Python files recommend packaging them into a .zip or .egg.

Launching Applications with spark-submit

- Once a user application is bundled, it can be launched using the bin/spark-submit script
- This script takes care of setting up the classpath with Spark and its dependencies, and can support different cluster managers and deploy modes that Spark supports



```
./bin/spark-submit \  
  --class <main-class> \  
  --master <master-url> \  
  --deploy-mode <deploy-mode> \  
  --conf <key>=<value> \  
  ... # other options  
<application-jar> \  
[application-arguments]
```

Launching Applications with spark-submit (2)

Common options

- **--class:**
 - ✓ The entry point for application (e.g. `org.apache.spark.examples.SparkPi`)
- **--master:**
 - ✓ The master URL for the cluster (e.g. `spark://23.195.26.187:7077`)
- **--deploy-mode:**
 - ✓ Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (client) (default: client)
- **--conf:**
 - ✓ Arbitrary Spark configuration property in `key=value` format. For values that contain spaces wrap `"key=value"` in quotes
- **application-jar:**
 - ✓ Path to a bundled jar including application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an `hdfs://` path or a `file://` path that is present on all nodes.
- **application-arguments:**
 - ✓ Arguments passed to the main method of your main class, if any

Launching Applications with spark-submit (3)

On Standalone cluster

```
# Run application locally on 8 cores
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100

# Run on a Spark standalone cluster in client deploy mode
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000

# Run on a Spark standalone cluster in cluster deploy mode with supervise
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

Launching Applications with spark-submit (4)

On YARN cluster

```
# Run on a YARN cluster
export HADOOP_CONF_DIR=XXX
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000
```

ज्ञानं परमं बलम्

Launching Applications with spark-submit (4)

On Mesos and Kubernetes cluster

```
# Run on a Mesos cluster in cluster deploy mode with supervise
```

```
./bin/spark-submit \  
  --class org.apache.spark.examples.SparkPi \  
  --master mesos://207.184.161.138:7077 \  
  --deploy-mode cluster \  
  --supervise \  
  --executor-memory 20G \  
  --total-executor-cores 100 \  
  http://path/to/examples.jar \  
  1000
```

```
# Run on a Kubernetes cluster in cluster deploy mode
```

```
./bin/spark-submit \  
  --class org.apache.spark.examples.SparkPi \  
  --master k8s://xx.yy.zz.ww:443 \  
  --deploy-mode cluster \  
  --executor-memory 20G \  
  --num-executors 50 \  
  http://path/to/examples.jar \  
  1000
```

Launching Applications with spark-submit (5)

Submitting Python Code

```
# Run a Python application on a Spark standalone cluster  
./bin/spark-submit \  
  --master spark://207.184.161.138:7077 \  
  examples/src/main/python/pi.py \  
  1000
```



Master URLs

Master URL	Meaning
<code>local</code>	Run Spark locally with one worker thread (i.e. no parallelism at all).
<code>local[K]</code>	Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine).
<code>local[*]</code>	Run Spark locally with as many worker threads as logical cores on your machine.
<code>spark://HOST:PORT</code>	Connect to the given Spark standalone cluster master. The port must be whichever one your master is configured to use, which is 7077 by default.
<code>spark://HOST1:PORT1,HOST2:PORT2</code>	Connect to the given Spark standalone cluster with standby masters with Zookeeper . The list must have all the master hosts in the high availability cluster set up with Zookeeper. The port must be whichever each master is configured to use, which is 7077 by default.
<code>mesos://HOST:PORT</code>	Connect to the given Mesos cluster. The port must be whichever one your is configured to use, which is 5050 by default. Or, for a Mesos cluster using ZooKeeper, use <code>mesos://zk://...</code> . To submit with <code>--deploy-mode cluster</code> , the HOST:PORT should be configured to connect to the MesosClusterDispatcher .
<code>yarn</code>	Connect to a YARN cluster in <code>client</code> or <code>cluster</code> mode depending on the value of <code>--deploy-mode</code> . The cluster location will be found based on the <code>HADOOP_CONF_DIR</code> or <code>YARN_CONF_DIR</code> variable.

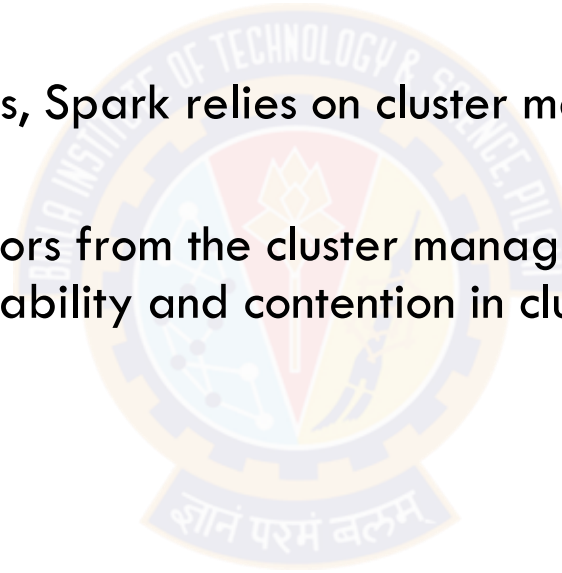
Loading Configuration from a File

Configuration file

- The spark-submit script can load default Spark configuration values from a properties file and pass them on to application
 - ✓ By default, it will read options from conf/spark-defaults.conf in the Spark directory
- Loading default Spark configurations this way can obviate the need for certain flags to spark-submit.
 - ✓ For instance, if the spark.master property is set, you can safely omit the --master flag from spark-submit.
- In general, configuration values precedence order
 - ✓ Directly set values on a SparkConf
 - ✓ then flags passed to spark-submit
 - ✓ then values in the defaults file.
- If you are ever unclear where configuration options are coming from, you can print out fine-grained debugging information by running spark-submit with the --verbose option.

Scheduling Spark Applications

- Many clusters are shared between multiple users/programs
 - ✓ Challenge is how to schedule the jobs?
 - ✓ What happens if two users launch applications that each want to use entire clusters resources?
- For scheduling in multi-tenant clusters, Spark relies on cluster managers to share resources between spark applications
- When a application asks for executors from the cluster manager, it may receive more or fewer resources depending upon the availability and contention in clusters
- Variety of cluster managers
 - ✓ Standalone
 - ✓ Hadoop YARN
 - ✓ Apache Mesos



Standalone Cluster Manager

Submitting applications

- Simple way to run applications on cluster
- Consists of master and multiple workers , each with configured amount of memory and CPU cores
- When app is submitted, user specifies
 - ✓how much memory its executors will use
 - ✓Total number of cores across all executors
- Submitting applications
 - ✓Need to set master argument as `sparkL//masternode:7077`
 - ✓Web UI can be accessed at same cluster URL

Standalone Cluster Manager (2)

Deployment modes

- Two deploy modes – where driver program of application runs
 - ✓ Client mode (default)
 - ✓ Cluster mode
- Client mode
 - ✓ Driver runs on machine where spark-submit is executed
 - ✓ Directly can see the output of the program or sent input to it
 - ✓ Machine needs to have fast connectivity to the workers
- Cluster mode
 - ✓ Driver is launched within the cluster, as another process on one of worker nodes
 - ✓ Spark-submit is “fire and forget” – close the laptop still applications run!
 - ✓ Will be able to access logs through cluster managers Web UI

Standalone Cluster Manager (3)

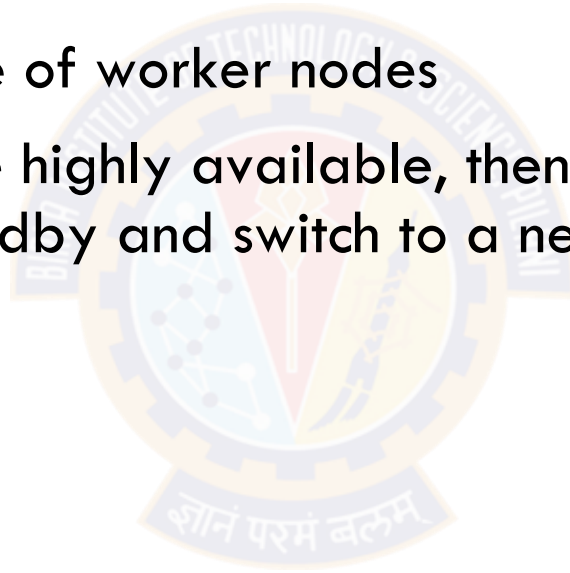
Resource configurations

- Needs to decide how to allocate resources between the executors
- Managed through two settings
- Executor memory
 - ✓ Configured using `–executor-memory` argument to `spark-submit` (default : 1 GB)
 - ✓ Each application will have at most one executor on each worker node
 - ✓ This setting controls how much of that workers memory the application will claim
- Maximum number of total cores
 - ✓ Total number of cores used across all executors for an application (default : unlimited)
 - ✓ Application will launch executors on every available node of cluster
 - ✓ Set through `–total-executor-cores` argument to `spark-submit`

Standalone Cluster Manager (4)

High Availability

- Cluster needs to be available even if the worker nodes fails
- Gracefully support the failure of worker nodes
- If the master also needs to be highly available, then zookeeper can be used to keep multiple masters on standby and switch to a new one when any of them fails



Hadoop YARN

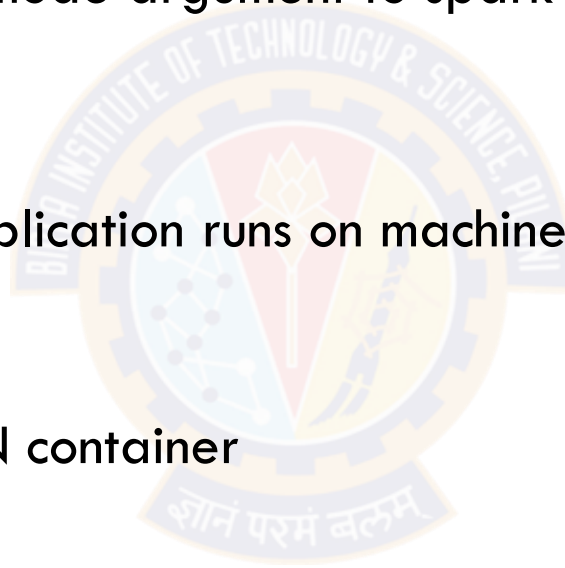
Submitting applications

- Introduced in Hadoop 2.0
- Typically installed on the same nodes as the HDFS
- Lets Spark access HDFS data quickly, on the same nodes where the data is stored
- Using it straightforward:
- Set environment variable pointing to Hadoop configuration directory
 - ✓ HADOOP_CONF_DIR
 - ✓ Directory that contains yarn-site.xml and other config files
 - ✓ Usually HADOOP_HOME/conf
- Submit jobs to a special master URL with spark-submit
 - `spark-submit --master yarn my_app`

Hadoop YARN (2)

Deployment modes

- Set the mode through `-deploy-mode` argument to `spark-submit`
- Two deployment modes
 - ❖ Client mode
 - ✓ Driver program for application runs on machine from which application is submitted
 - ❖ Cluster mode
 - ✓ Driver runs inside YARN container



Hadoop YARN (3)

Resource configurations

- **--num-executors**
 - ✓ Spark applications use a fixed number of executors (default:2)
 - ✓ Set through spark-submit command
- **--executor-memory**
 - ✓ Memory used by each executor (default : 8GB)
- **--executor-cores**
 - ✓ Number of cores it claims from YARN
- Usually, Spark will run better with a smaller number of large executors (with more memory and cores) since it can optimize the communication within each executor

Optimum Parallelism in Spark Framework on Hadoop YARN for Maximum Cluster Resource Utilization

https://link.springer.com/chapter/10.1007/978-981-15-0029-9_28

Apache Mesos

Submitting applications

- General purpose cluster manager – can run analytics workloads and long running services
- Need to pass mesos URI with spark-submit
 - ✓ `spark-submit --master mesos://masternode:5050 my_app`
- Zookeeper can be used to elect a master when running on multimaster node
- Connecting Spark to Mesos
 - ✓ To use Mesos from Spark, you need a Spark binary package available in a place accessible by Mesos, and a Spark driver program configured to connect to Mesos.
 - ✓ Alternatively, you can also install Spark in the same location in all the Mesos slaves, and configure `spark.mesos.executor.home` (defaults to `SPARK_HOME`) to point to that location.

Apache Mesos (2)

Scheduling modes

- Spark can run over Mesos in two modes: “coarse-grained” and “fine-grained” mode
- In “coarse-grained” mode
 - ✓ Each Spark executor runs as a single Mesos task.
 - ✓ Spark executors are sized according to the following configuration variables:
 - Executor memory: `spark.executor.memory`
 - Executor cores: `spark.executor.cores`
 - Number of executors: `spark.cores.max/spark.executor.cores`
- Executors are brought up eagerly when the application starts, until `spark.cores.max` is reached
 - ✓ If you don't set `spark.cores.max`, the Spark application will consume all resources offered to it by Mesos
- The benefit of coarse-grained mode is much lower startup overhead, but at the cost of reserving Mesos resources for the complete duration of the application.

Apache Mesos (3)

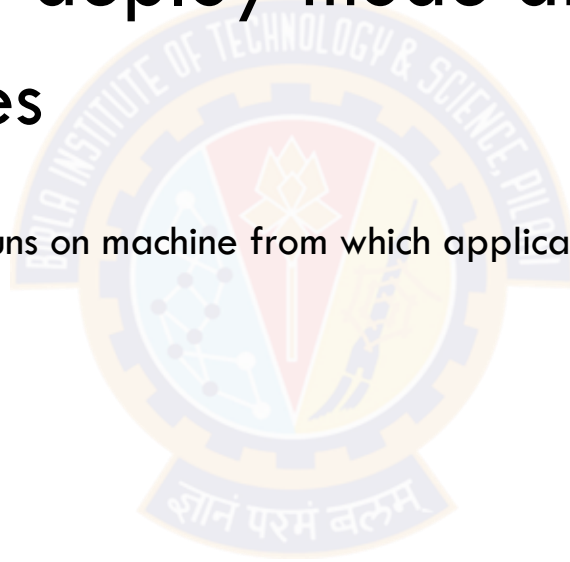
Scheduling modes

- In “fine-grained” mode,
 - ✓ Each Spark task inside the Spark executor runs as a separate Mesos task
 - ✓ This allows multiple instances of Spark (and other frameworks) to share cores at a very fine granularity, where each application gets more or fewer cores as it ramps up and down, but it comes with an additional overhead in launching each task
 - ✓ This mode may be inappropriate for low-latency requirements like interactive queries or serving web requests
 - ✓ It is deprecated now!

Apache Mesos (4)

Deployment modes

- Set the mode through `–deploy-mode` argument to `spark-submit`
- Two deployment modes
 - ❖ Client mode
 - ✓ Driver program for application runs on machine from which application is submitted
 - ❖ Cluster mode
 - ✓ Driver runs inside Mesos Cluster



Which Cluster Manager to use?

Guidelines to choose a cluster manager

- **If this is new deployment**
 - ✓ Start with standalone cluster as its easy to setup
 - ✓ Provides almost all the features as the other cluster managers
- **If needs to run Spark alongside with other applications or to use richer resource scheduling capabilities**
 - ✓ Both YARN and Mesos provide these features
 - ✓ YARN will be preinstalled in Hadoop distributions
 - ✓ Mesos offers more fine grained resource management
- **In all cases,**
 - ✓ Best to run Spark on the same nodes as HDFS for fast access to storage
 - ✓ Hadoop distribution installs YARN and HDFS together
 - ✓ Mesos and Standalone cluster managers can be installed manually on the same nodes

Summary

- RDD operations to help create data analysis programs
- Spark applications can be run standalone or on clusters with no code changes
- Easier to move from small development environments to larger staging / production environments by just scaling number of nodes
- Can run on a variety of clusters, including Hadoop YARN
- Developers focus more on logic than deployment scenarios
- Extensive visibility into application execution and cluster operations



**Next Session:
Spark - Part 3**