SPARK ARCHITECTURE & RESOURCE MANAGERS
WAYS TO RUN SPARK

- Local
- Standalone Scheduler
- YARN
- Mesos

Static # of Executors
Dynamic # of Executors
MOST COMMON SPARK DEPLOYMENT ENVIRONMENTS (CLUSTER MANAGERS)

- **48%** Standalone mode
- **40%** YARN
- **11%** Mesos
History: 2 MR APPS RUNNING

JobTracker

NameNode

OS

JT

DN

TT

MM

R

OS

DN

TT

MM

R

OS

DN

TT

MM

R

OS

DN

TT

MM

RR

OS
val conf = new SparkConf()
  .setMaster("local[12]")
  .setAppName("MyFirstApp")
  .set("spark.executor.memory", "3g")
val sc = new SparkContext(conf)
STANDALONE MODE
different spark-env.sh

- SPARK_WORKER_CORES

> /bin/spark-submit --name "SecondApp"
   --master spark://host4:port1
   myApp.jar

spark-env.sh
- SPARK_LOCAL_DIRS
different spark-env.sh

- SPARK_WORKER_CORES

> /bin/spark-submit --name "SecondApp"
--master spark://host1:port1,host2:port2
  myApp.jar
**SPARK_STANDALONE**

*(single app)*

**conf/spark-env.sh**

- **SPARK_WORKER_INSTANCES**: [default: 1] # of worker instances to run on each machine
- **SPARK_WORKER_CORES**: [default: ALL] # of cores to allow Spark applications to use on the machine
- **SPARK_WORKER_MEMORY**: [default: TOTAL RAM – 1 GB] Total memory to allow Spark applications to use on the machine
- **SPARK_DAEMON_MEMORY**: [default: 512 MB] Memory to allocate to the Spark master and worker daemons themselves
Standalone settings

- Apps submitted will run in FIFO mode by default

`spark.cores.max`: maximum amount of CPU cores to request for the application from across the cluster

`spark.executor.memory`: Memory for each executor
YARN MODE
What is YARN?

Resource Management and negotiation so that multiple apps can live and operate together in a Hadoop cluster

- Manage CPU and memory and allocate it to different apps
- Multiple users
- Multiple apps
- Lots of advanced features
YARN Benefits

- Allows for multiple data processing engines against HDFS or HBase
- Dynamically allocates cluster resources and improves utilization over static MR-v
- Better scalability in the future (>5,000 node clusters)
- Can run multiple Executors on each node for a single Spark App with just one NodeManager
- Solid Kerberos integration

- Allow applications to request specific nodes for scheduling tasks
SPARK YARN

NodeManager

Resource Manager

Scheduler

Apps Master

I'm HA via ZooKeeper

Client #1

Client #2

App Master

Container

App Master

Container

Container

Container
SPARK YARN
(cluster mode)

- Does not support Spark Shells
YARN settings

--num-executors: controls how many executors will be allocated

--executor-memory: RAM for each executor

--executor-cores: CPU cores for each executor

Dynamic Allocation:

spark.dynamicAllocation.enabled
spark.dynamicAllocation.minExecutors
spark.dynamicAllocation.maxExecutors
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout (N)
spark.dynamicAllocation.schedulerBacklogTimeout (M)
spark.dynamicAllocation.executorIdleTimeout (K)

YARN resource manager UI: http://<ip.address>:8088

(No apps running)
[ec2-user@ip-10-0-72-36 ~]$ spark-submit --class org.apache.spark.examples.SparkPi --deploy-mode client --master yarn /opt/cloudera/parcels/CDH-5.2.1-1.cdh5.2.1.p0.12/jars/spark-examples-1.1.0-cdh5.2.1-hadoop2.5.0-cdh5.2.1.jar 10
App running in **client** mode

### All Applications

#### Cluster Metrics

<table>
<thead>
<tr>
<th>Apps Submitted</th>
<th>Apps Pending</th>
<th>Apps Running</th>
<th>Apps Completed</th>
<th>Containers Running</th>
<th>Memory Used</th>
<th>Memory Total</th>
<th>Memory Reserved</th>
<th>VCPUs Used</th>
<th>VCPUs Total</th>
<th>VCPUs Reserved</th>
<th>Active Nodes</th>
<th>Decommissioned Nodes</th>
<th>Lost Nodes</th>
<th>Unhealthy Nodes</th>
<th>Rebooted Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### User Metrics for dr.who

<table>
<thead>
<tr>
<th>ID</th>
<th>User</th>
<th>Name</th>
<th>Application Type</th>
<th>Queue</th>
<th>StartTime</th>
<th>FinishTime</th>
<th>State</th>
<th>FinalStatus</th>
<th>Progress</th>
<th>Tracking UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>application_1417641524005_0001</td>
<td>ec2-user</td>
<td>Spark PI</td>
<td>SPARK</td>
<td>root,ec2-user</td>
<td>Thu, 04 Dec 2014 15:30:43 GMT</td>
<td>Thu, 04 Dec 2014 15:31:14 GMT</td>
<td>FINISHED</td>
<td>SUCCEEDED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>application_1417641524005_0002</td>
<td>ec2-user</td>
<td>Spark PI</td>
<td>SPARK</td>
<td>root,ec2-user</td>
<td>Thu, 04 Dec 2014 15:25:18 GMT</td>
<td>Thu, 04 Dec 2014 15:26:19 GMT</td>
<td>FINISHED</td>
<td>SUCCEEDED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
[ec2-user@ip-10-0-72-36 ~]$ spark-submit --class org.apache.spark.examples.SparkPi --deploy-mode cluster --master yarn/opt/cloudera/parcels/CDH-5.2.1-1.cdh5.2.1.p0.12/jars/spark-examples-1.1.0-cdh5.2.1-hadoop2.5.0-cdh5.2.1.jar 10
App running in **cluster** mode

### All Applications

#### Cluster Metrics

<table>
<thead>
<tr>
<th>Apps Submitted</th>
<th>Apps Pending</th>
<th>Apps Running</th>
<th>Apps Completed</th>
<th>Containers Running</th>
<th>Containers Pending</th>
<th>Containers Running</th>
<th>Containers Pending</th>
<th>Memory Used</th>
<th>Memory Total</th>
<th>VCore Used</th>
<th>VCore Total</th>
<th>Active Nodes</th>
<th>Decommissioned Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3.45 GB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### User Metrics for dr.who

<table>
<thead>
<tr>
<th>Apps Submitted</th>
<th>Apps Pending</th>
<th>Apps Running</th>
<th>Apps Completed</th>
<th>Containers Running</th>
<th>Containers Pending</th>
<th>Containers Running</th>
<th>Containers Pending</th>
<th>Memory Used</th>
<th>Memory Total</th>
<th>VCore Used</th>
<th>VCore Total</th>
<th>Active Nodes</th>
<th>Decommissioned Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Example Application

- **ID**: application_1417641264003_0004
- **User**: ec2-user
- **Name**: org.apache.spark.examples.SparkPi
- **Queue**: SPARK
- **Start Time**: Thu, 04 Dec 2014 15:37:10 GMT
- **Finish Time**: Thu, 04 Dec 2014 15:37:54 GMT
- **State**: FINISHED
- **Final Status**: SUCCESS
App running in **cluster** mode.
App running in **cluster** mode

```
Log Type: stderr
Log Length: 22704

Showing 4096 bytes of 22704 total. Click here for the full log.

14/12/04 10:37:52 INFO yarn.YarnAllocationManager: Completed container container_1417641624005_0004_01_0000002 (state: COMPLETE)
14/12/04 10:37:52 INFO yarn.ExecutorRunnable: Setting up executor with environment: Map(CLASSPATH -> /home/hadoop⎨jars⎬)
14/12/04 10:37:52 INFO yarn.ExecutorRunnable: Setting up executor with commands: list($JAVA_HOME/bin/java, -server, -XX:OnOutOfMemoryError=(ErrorAction=Abort)
14/12/04 10:37:52 INFO yarn.ApplicationMaster: Allocating 1 containers to make up for (potentially) lost containers
14/12/04 10:37:52 INFO yarn.YarnAllocationManager: Will allocate 1 executor containers, each with 1448 memory
14/12/04 10:37:52 INFO yarn.YarnAllocationManager: container request (host: Any, priority: 1, capability: (memory:1448, vcores:1))
14/12/04 10:37:53 INFO spark.MapOutputTrackerMasterActor: MapOutputTrackerActor stopped!
14/12/04 10:37:53 INFO network.ConnectionManager: Selector thread was interrupted!
14/12/04 10:37:53 INFO network.ConnectionManager: ConnectionManager stopped
14/12/04 10:37:53 INFO storage.MemoryStore: MemoryStore cleared
14/12/04 10:37:53 INFO storage.BlockManager: BlockManager stopped
14/12/04 10:37:53 INFO storage.BlockManagerMaster: BlockManagerMaster stopped
14/12/04 10:37:54 INFO spark.SparkContext: Successfully stopped SparkContext
14/12/04 10:37:54 INFO yarn.ApplicationMaster: Unregistering ApplicationMaster with SUCCEEDED
14/12/04 10:37:54 INFO impl.AMMClientImpl: Waiting for application to be successfully unregistered.
14/12/04 10:37:54 INFO yarn.ApplicationMaster: All executors have launched.
14/12/04 10:37:54 INFO yarn.ApplicationMaster: Started progress reporter thread - heartbeat interval : 5000
14/12/04 10:37:54 INFO yarn.ApplicationMaster: AppMaster received a signal.
14/12/04 10:37:54 INFO yarn.AppMaster: Deleting staging directory .sparkStaging/application_1417641624005_0004
14/12/04 10:37:54 INFO yarn.ApplicationMaster: shutdown Invoking sc stop from shutdown hook
14/12/04 10:37:54 INFO spark.SparkContext: SparkContext already stopped

Log Type: stdout
Log Length: 93

H1 is roughly 3.142392
```
### Spark History Server

**Event Log Location:** hdfs://ip-10-0-72-36.us-west-2.compute.internal:8020/user/spark/applicationHistory

**Showing 1-2 of 2**

<table>
<thead>
<tr>
<th>App Name</th>
<th>Started</th>
<th>Completed</th>
<th>Duration</th>
<th>Spark User</th>
<th>Last Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark shell</td>
<td>2014/12/04 09:14.01</td>
<td>2014/12/04 09:21.19</td>
<td>7.3 min</td>
<td>ec2-user</td>
<td>2014/12/04 09:21.20</td>
</tr>
<tr>
<td>Spark shell</td>
<td>2014/12/04 09:07.36</td>
<td>2014/12/04 09:13.47</td>
<td>6.2 min</td>
<td>ec2-user</td>
<td>2014/12/04 09:13.48</td>
</tr>
</tbody>
</table>
PLUGGABLE RESOURCE MANAGEMENT

<table>
<thead>
<tr>
<th></th>
<th>Spark Central Master</th>
<th>Who starts Executors?</th>
<th>Tasks run in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>[none]</td>
<td>Human being</td>
<td>Executor</td>
</tr>
<tr>
<td>Standalone</td>
<td>Standalone Master</td>
<td>Worker JVM</td>
<td>Executor</td>
</tr>
<tr>
<td>YARN</td>
<td>YARN App Master</td>
<td>Node Manager</td>
<td>Executor</td>
</tr>
<tr>
<td>Mesos</td>
<td>Mesos Master</td>
<td>Mesos Slave</td>
<td>Executor</td>
</tr>
</tbody>
</table>
spark-submit provides a uniform interface for submitting jobs across all cluster managers

```
bin/spark-submit --master spark://host:7077
   --executor-memory 10g
   my_script.py
```

### Table 7-2. Possible values for the --master flag in spark-submit

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>spark://host:port</td>
<td>Connect to a Spark Standalone master at the specified port. By default Spark Standalone master's listen on port 7077 for submitted jobs.</td>
</tr>
<tr>
<td>mesos://host:port</td>
<td>Connect to a Mesos cluster master at the specified port. By default Mesos masters listen on port 5050 for submitted jobs.</td>
</tr>
<tr>
<td>yarn</td>
<td>Indicates submission to YARN cluster. When running on YARN you'll need to export HADOOP_CONF_DIR to point the location of your Hadoop configuration directory.</td>
</tr>
<tr>
<td>local</td>
<td>Run in local mode with a single core.</td>
</tr>
<tr>
<td>local[N]</td>
<td>Run in local mode with N cores.</td>
</tr>
<tr>
<td>local[*]</td>
<td>Run in local mode and use as many cores as the machine has.</td>
</tr>
</tbody>
</table>

Source: Learning Spark
Summary

To summarize the concepts in this section, let’s walk through the exact steps that occur when you run a Spark application on a cluster.

1. The user submits an application using spark-submit.
2. spark-submit launches the driver program and invokes the main method specified by the user.
3. The driver program contacts the cluster manager to ask for resources to launch executors.
4. The cluster manager launches executors on behalf of the driver program.
5. The driver process runs through the user application. Based on the RDD actions and transformations in the program, it sends work to executors in the form of tasks.
6. Tasks are run on executor processes to compute and save results.
YARN settings:
http://spark.apache.org/docs/latest/running-on-yarn.html
<table>
<thead>
<tr>
<th>YARN Executor Container</th>
<th>spark.yarn.executor.memoryOverhead (10%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark Executor Memory</td>
<td>spark.executor.memory</td>
</tr>
<tr>
<td>RDD partition caching</td>
<td>(60%)</td>
</tr>
<tr>
<td>Shuffle memory</td>
<td>(20%)</td>
</tr>
<tr>
<td>User Tasks memory</td>
<td>(20%)</td>
</tr>
</tbody>
</table>
Data locality in YARN?

Coming in Spark 1.5

https://issues.apache.org/jira/browse/SPARK-4352
Each Spark application scales the number of executors up and down based on workload.

- If executors are idle, remove them
- If we need more executors, request them
In Spark, each executor is long-running and runs many small tasks.
In MapReduce, each container is short-lived and runs one large task.
More resources allocated than used...
More resources allocated than used
More efficient utilization of cluster resources
DYNAMIC ALLOCATION

spark.dynamicAllocation.enabled
spark.dynamicAllocation.minExecutors
spark.dynamicAllocation.maxExecutors
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout (N)
spark.dynamicAllocation.schedulerBacklogTimeout (M)
spark.dynamicAllocation.executorIdleTimeout (K)

DYNAMIC ALLOCATION USE CASES

Long-running ETL jobs
E.g. Parsing JSON into Parquet in S3

Interactive applications / server
E.g. Spark shell, Ooyala Job server

Any application with large shuffles
CASE STUDY

- 100 TB cluster with 1500+ nodes
- 15+ PB S3 warehouse (7 PB Parquet)
- Dynamic allocation with up to 10,000 executors
- Enabled dynamic allocation for all Spark applications
- Run Spark alongside Hive, Pig, MapReduce
- Used for ad-hoc query and experimentation

* this team only

* this team only
CASE STUDY

• 400+ TB cluster with 8,000+ nodes
• 150 PB data warehouse
• Use dynamic allocation with up to 1,500 executors
• Primary use cases are ETL and SQL
• Run Spark alongside Storm, MapReduce, Pig
DYNAMIC ALLOCATION: FUTURE

- Support for Mesos mode  SPARK-4922 (PR ready)
- Support for Standalone mode  SPARK-4751 (PR soon)
- Better support for caching  SPARK-7955 (PR ready)
- Pluggable scaling heuristics
WHO IS USING DYNAMIC ALLOCATION?

NETFLIX  cloudera

NTT DATA  mi  PayPal

Alpine Data Labs  MESOSPHERE  intel

Spotify  Tencent 腾讯
When Dynamic Allocation kills executors, Spark loses their cached RDDs.

**Short term solution in Spark 1.5:**
- Keep the Executors w/ cached partitions around longer
- different timeout for executors with cached data

**Long term solutions:**
- Cache rebalancing
- Container resizing (free v-cores)
- Off-heap caching via HDFS DDMs (HDFS-5851)
EXTERNAL SHUFFLE SERVICE

- Worker JVM serves files
- Must turn this on for dynamic allocation in YARN
- Node Manager serves files
Where does Spark need encryption?

• Driver <-> Executor communication (Current: Akka-over SSL) (upcoming: RPC-over-SASL)
• File Distribution (sc.addjar or sc.addfile)
• Block Manager
• User UI / REST API (upcoming SPARK-2750)
• Data at rest (shuffle files) (upcoming: SPARK-5682)

In Spark 1.4:
• SASL is now available for shuffle data and block manager communication
• SSL is available for Akka (driver to executor communication) / file distribution.
PySpark at a Glance

Write Spark jobs in Python

Run interactive jobs in the shell

Supports C extensions
Spark Core Engine (Scala)
Standalone Scheduler
YARN
Mesos
Local
Java API
Spark Core Engine (Scala)

PySpark

41 files
8,100 loc
6,300 comments
Choose Your Python Implementation

Driver Machine

Spark Context

CPython
(default python)

Worker Machine

pypy

- JIT, so faster
- less memory
- CFFI support

$ PYSPARK_DRIVER_PYTHON=pypy PYSPARK_PYTHON=pypy
./bin/pyspark

OR

$ PYSPARK_DRIVER_PYTHON=pypy PYSPARK_PYTHON=pypy ./bin/spark-submit wordcount.py
The performance speed up will depend on work load (from 20% to 3000%).

Here are some benchmarks:

<table>
<thead>
<tr>
<th>Job</th>
<th>CPython 2.7</th>
<th>PyPy 2.3.1</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Count</td>
<td>41 s</td>
<td>15 s</td>
<td>2.7 x</td>
</tr>
<tr>
<td>Sort</td>
<td>46 s</td>
<td>44 s</td>
<td>1.05 x</td>
</tr>
<tr>
<td>Stats</td>
<td>174 s</td>
<td>3.6 s</td>
<td>48 x</td>
</tr>
</tbody>
</table>

Here is the code used for benchmark:

```python
rdd = sc.textFile("text")
def wordcount():
    rdd.flatMap(lambda x:x.split('/'))
    .map(lambda x:(x,1)).reduceByKey(lambda x,y:x+y).collectAsMap()
def sort():
    rdd.sortBy(lambda x:x, 1).count()
def stats():
    sc.parallelize(range(1024), 20).flatMap(lambda x: xrange(5024)).stats()

https://github.com/apache/spark/pull/2144
```
<table>
<thead>
<tr>
<th>spark.python.worker.memory</th>
<th>512m</th>
</tr>
</thead>
</table>

Amount of memory to use per python worker process during aggregation, in the same format as JVM memory strings (e.g. 512m, 2g). If the memory used during aggregation goes above this amount, it will spill the data into disks.