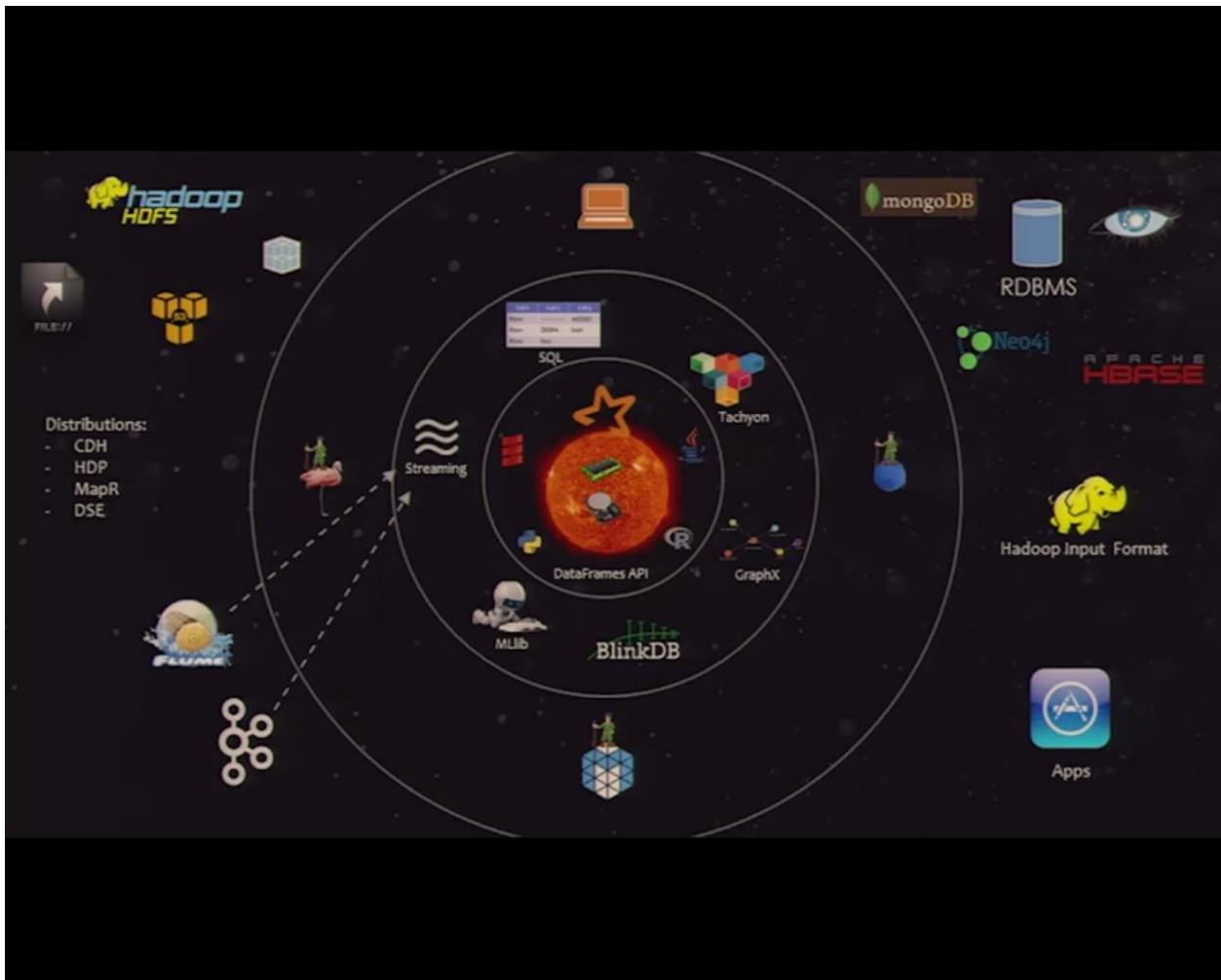
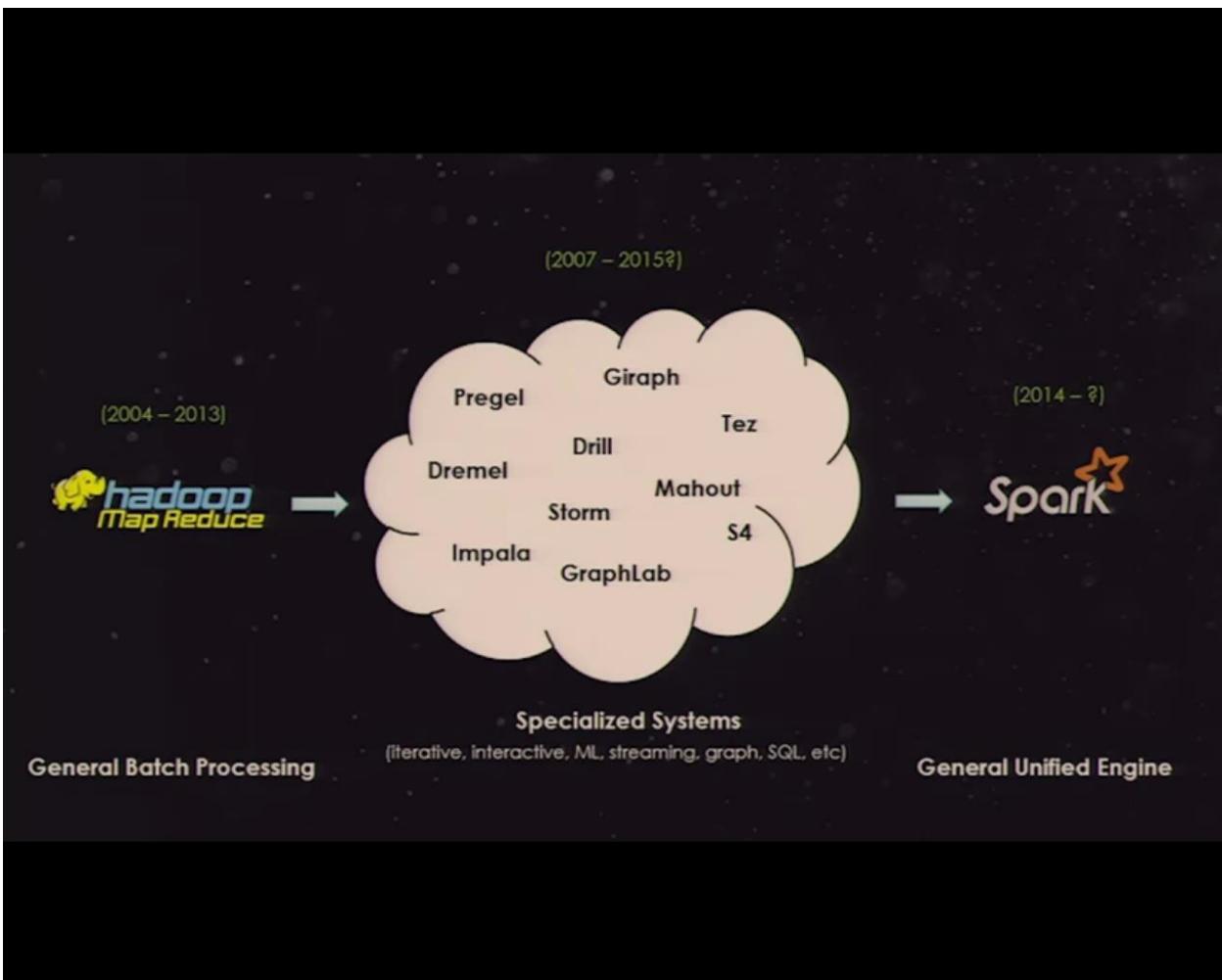
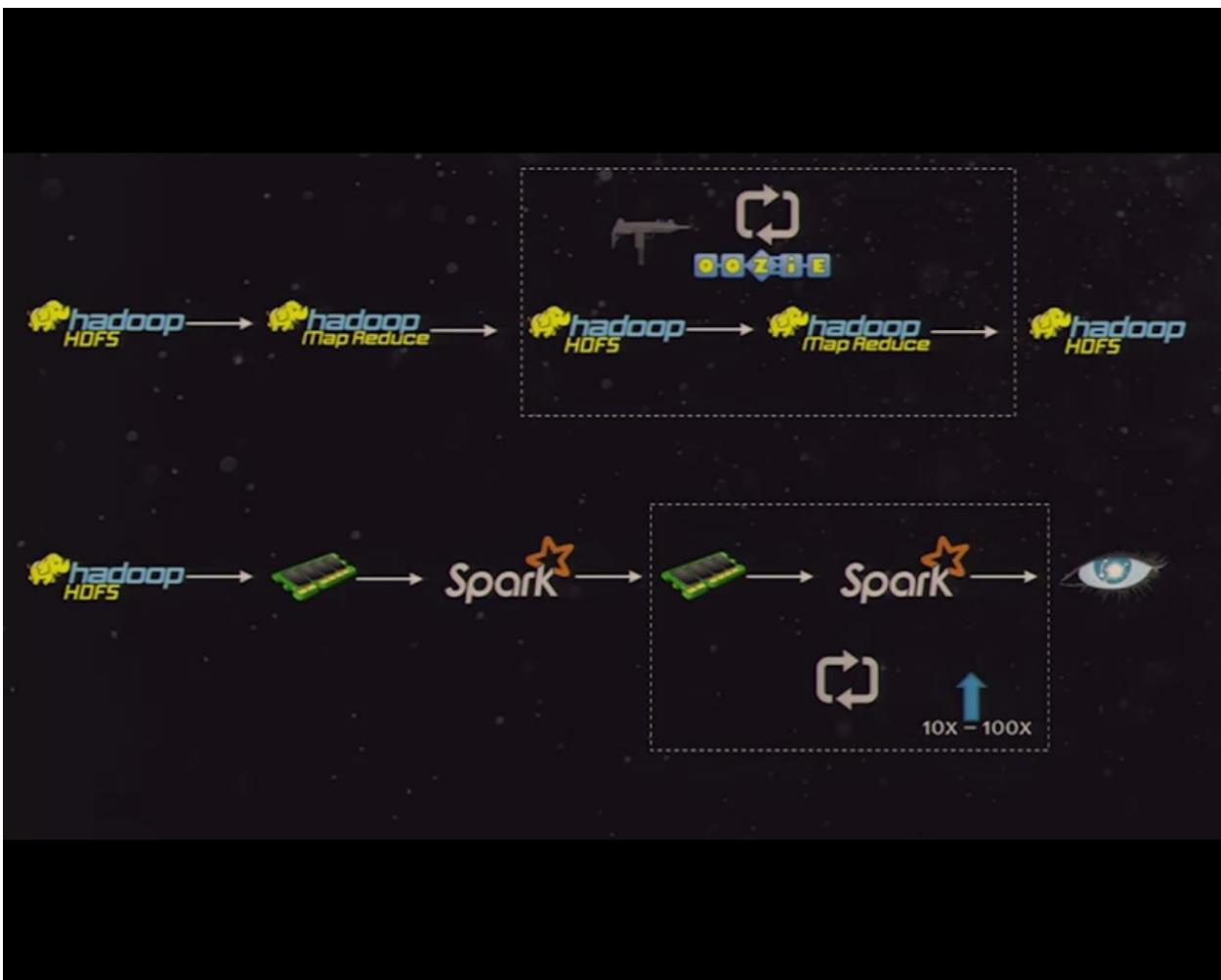
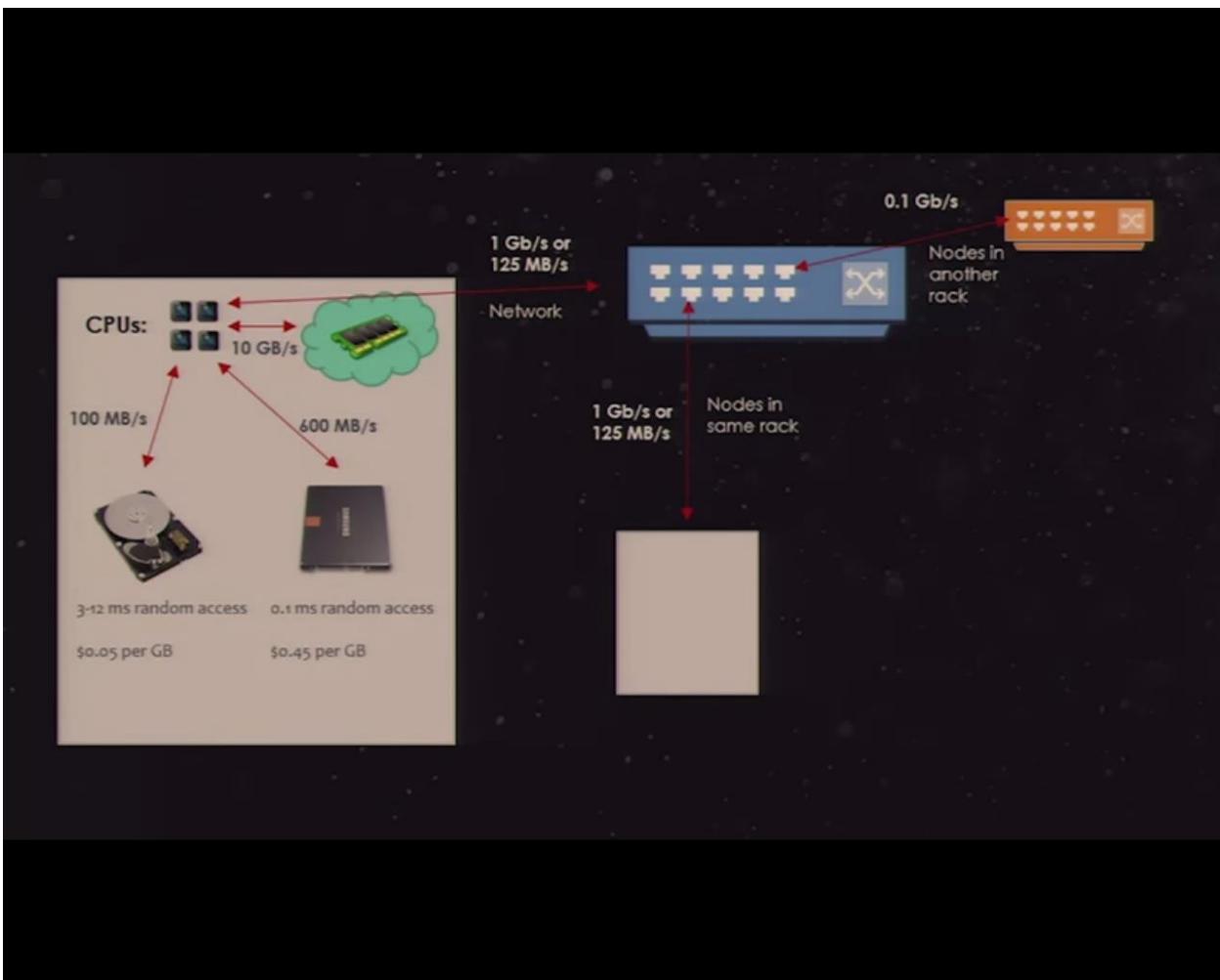


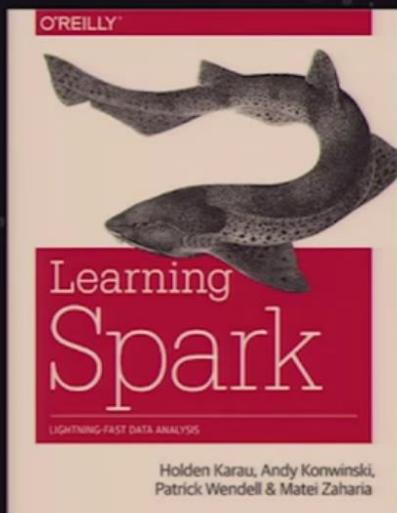
<https://www.youtube.com/watch?v=7ooZ4S7Ay6Y>











<http://shop.oreilly.com/product/0636920028512.do>

eBook: \$33.99 PDF, ePub, Mobi, DAISY

Print: \$39.99 Shipping now!

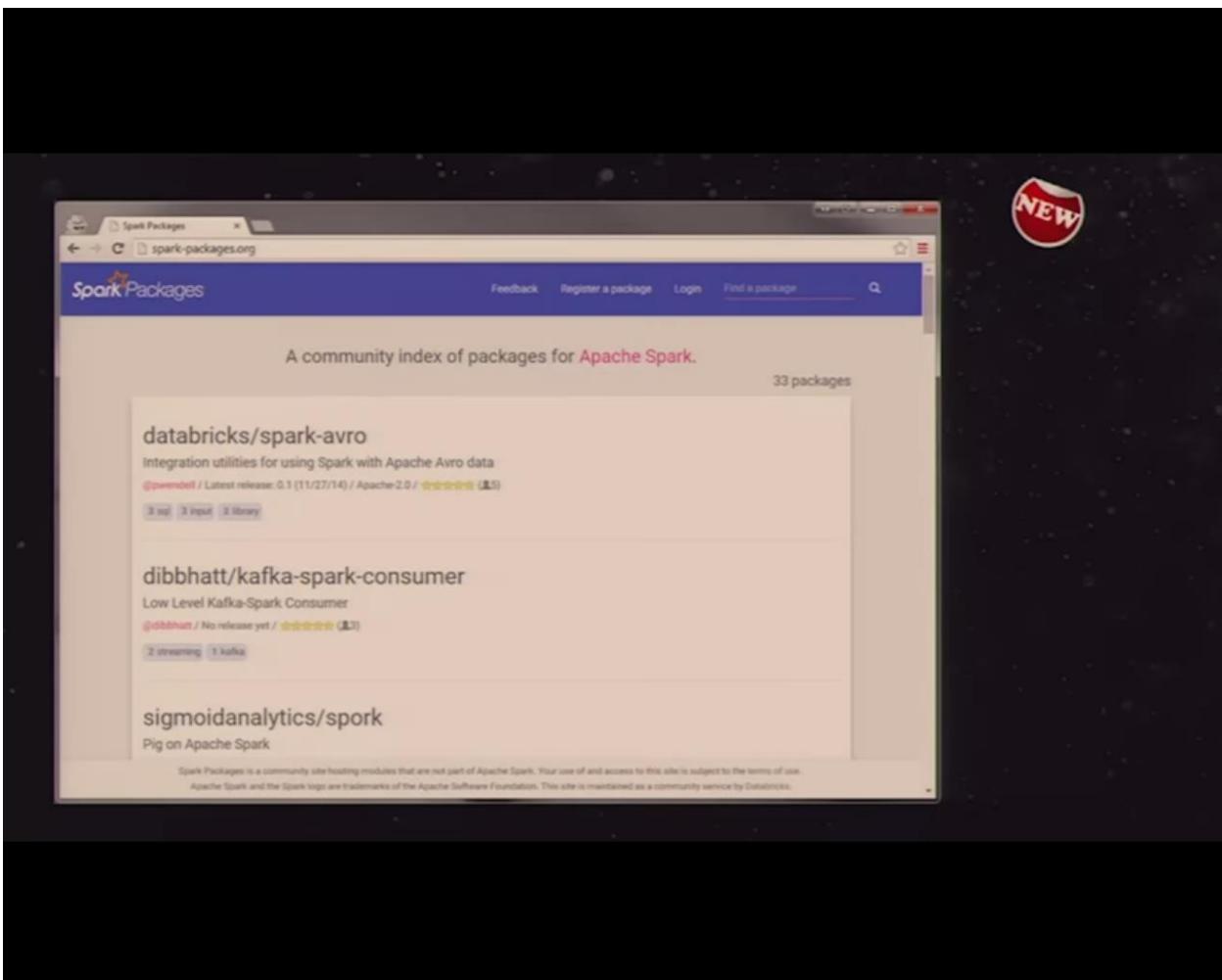
\$30 @ Amazon:

<http://www.amazon.com/Learning-Spark-Lightning-Fast-Data-Analysis/dp/1449358624>



22:25 / 5:58:30

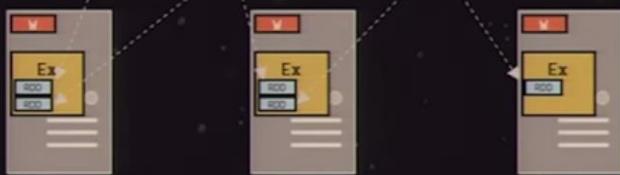




more partitions = more parallelism

RDD

item-1 item-2 item-3 item-4 item-5	item-6 item-7 item-8 item-9 item-10	item-11 item-12 item-13 item-14 item-15	item-16 item-17 item-18 item-19 item-20	item-21 item-22 item-23 item-24 item-25
--	---	---	---	---



RDD w/ 4 partitions

logLinesRDD			
Error, ts, msg1	Info, ts, msg8	Error, ts, msg3	Error, ts, msg4
Warn, ts, msg2	Warn, ts, msg2	Info, ts, msg5	Warn, ts, msg9
Error, ts, msg1	Info, ts, msg8	Info, ts, msg5	Error, ts, msg1

An RDD can be created 2 ways:

- Parallelize a collection
- Read data from an external source (S3, C*, HDFS, etc)

PARALLELIZE



```
# Parallelize in Python
wordsRDD = sc.parallelize(["fish", "cats", "dogs"])
```



```
// Parallelize in Scala
val wordsRDD= sc.parallelize(List("fish", "cats", "dogs"))
```



```
// Parallelize in Java
JavaRDD<String> wordsRDD = sc.parallelize(Arrays.asList("fish", "cats", "dogs"));
```

- Take an existing in-memory collection and pass it to SparkContext's parallelize method

- Not generally used outside of prototyping and testing since it requires entire dataset in memory on one machine

READ FROM TEXT FILE



```
# Read a local txt file in Python
linesRDD = sc.textFile("/path/to/README.md")
```

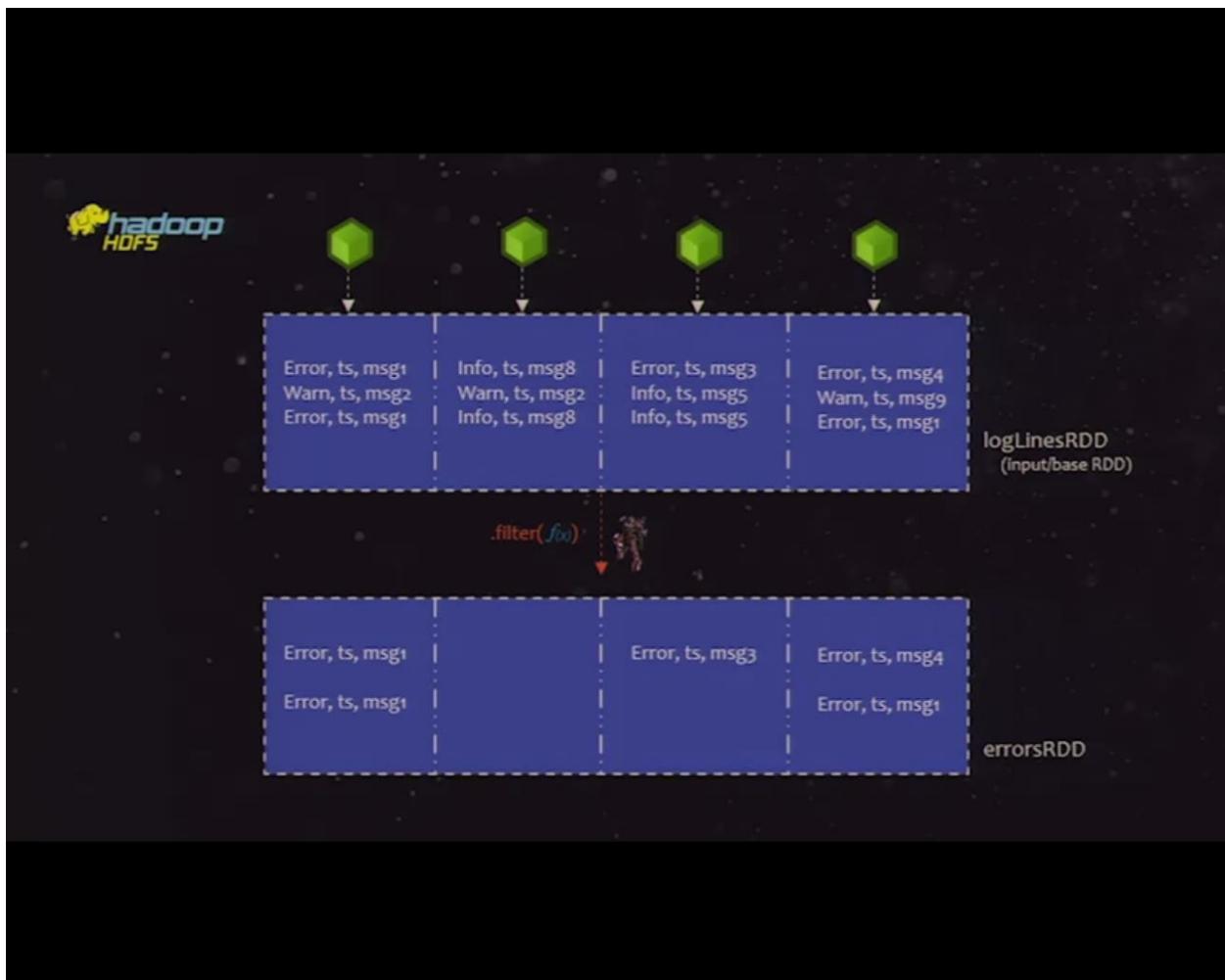
There are other methods
to read data from HDFS,
C*, S3, HBase, etc.



```
// Read a local txt file in Scala
val linesRDD = sc.textFile("/path/to/README.md")
```



```
// Read a local txt file in Java
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

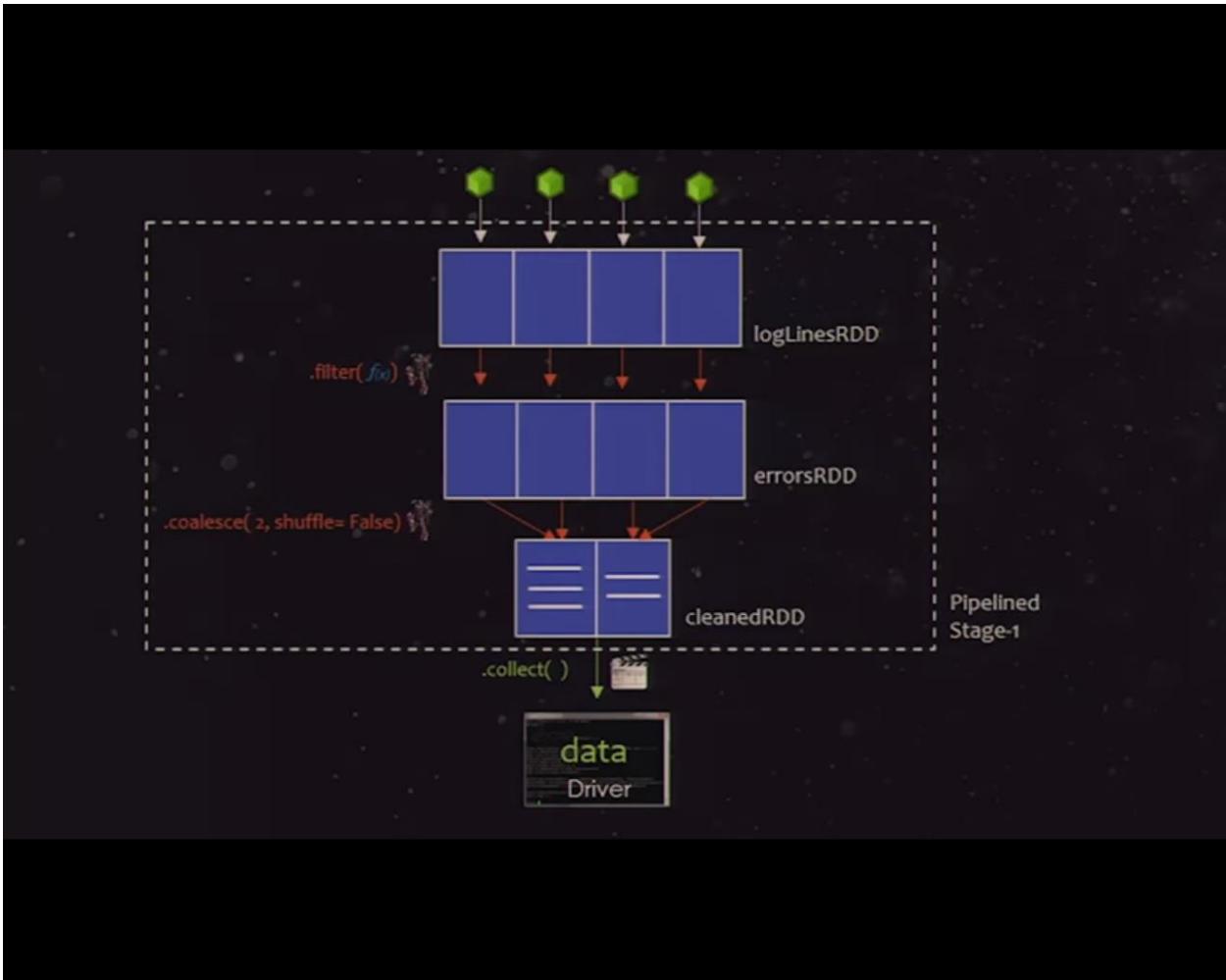








Don't uses collect for terabyte of data RDD, instead save it on hdfs or sample the larger RDD in smaller one



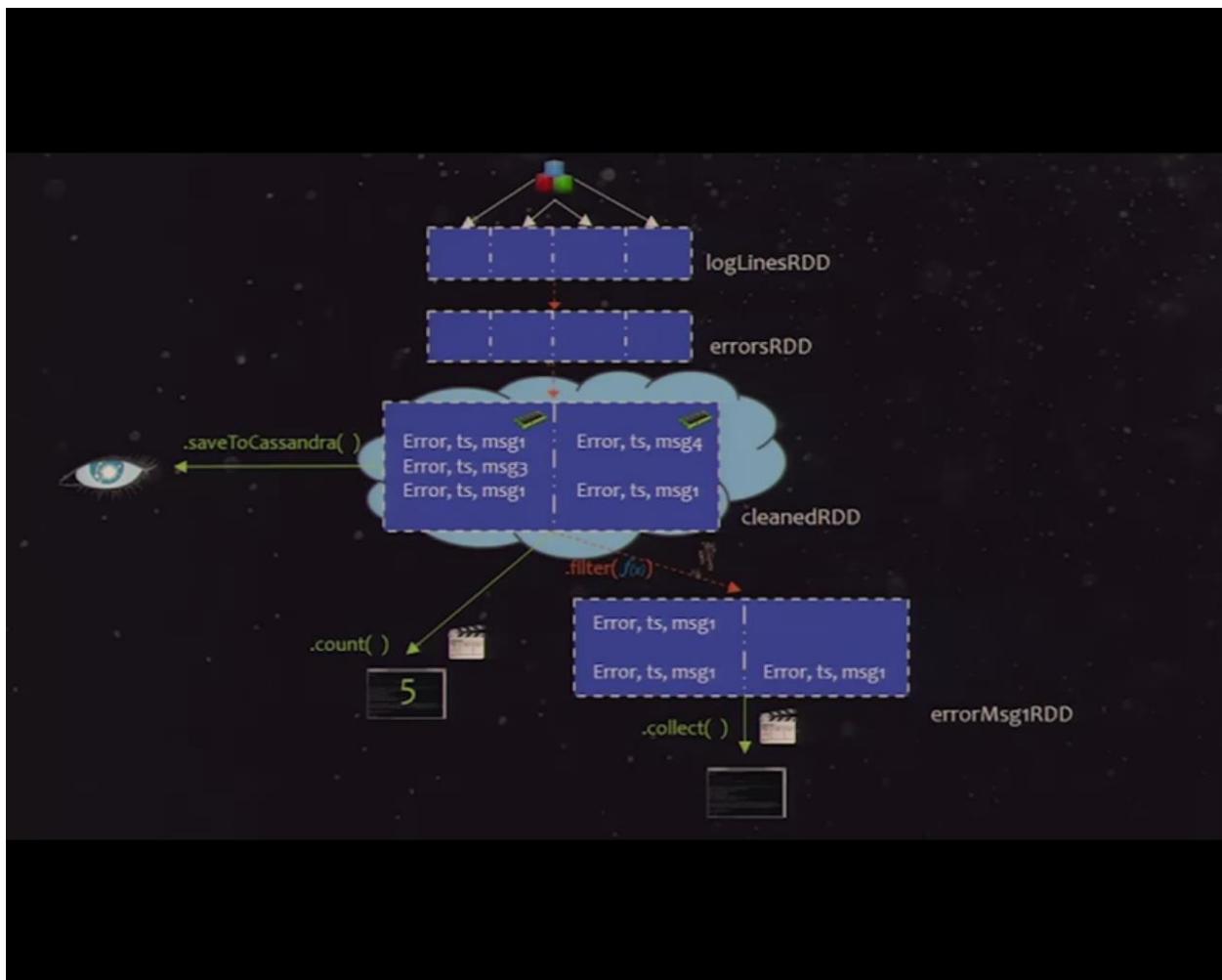
Once we completed execution of DAG we get rid of the RDD , solid line becomes dashed line (just for demonstration purpose)

After action all RDD disappear as we didn't cache the RDD





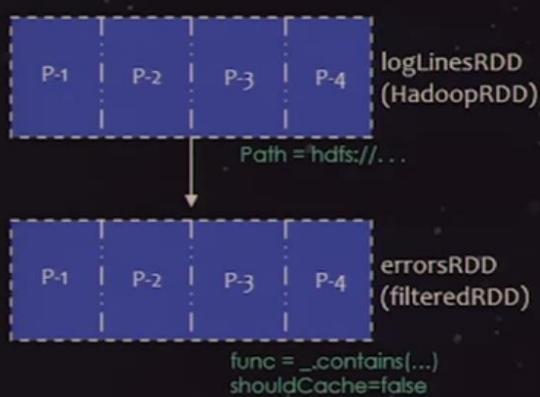
You need to know which RDD should be cached, the RDD which are going to be used multiple times needs to be cached and of course the size of RDD is within the limit, in below example cleaned RDD is a good candidate of cache as we are using it 3 times



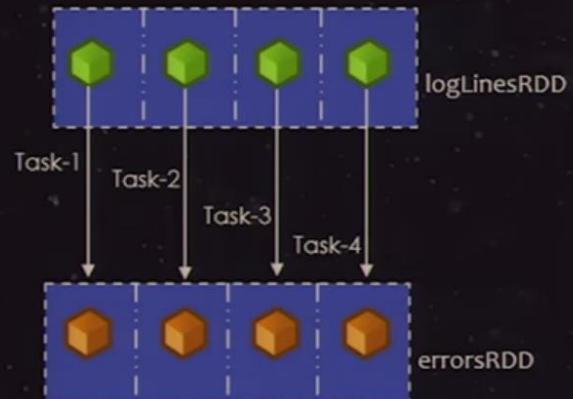
If whole RDD is not able to cached in memory the rest will go to underlined datasource, let say we have partition of 100 and only 30 partition RDD can store in memory during cache so rest 70 will go to underlined data source like hdfs

RDD GRAPH

Dataset-level view:



Partition-level view:



LIFECYCLE OF A SPARK PROGRAM

- 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.

TRANSFORMATIONS (lazy)

map()	intersection()	carterion()
flatMap()	distinct()	pipe()
filter()	groupByKey()	coalesce()
mapPartitions()	reduceByKey()	repartition()
mapPartitionsWithIndex()	sortByKey()	partitionBy()
sample()	join()	...
union()	cogroup()	...

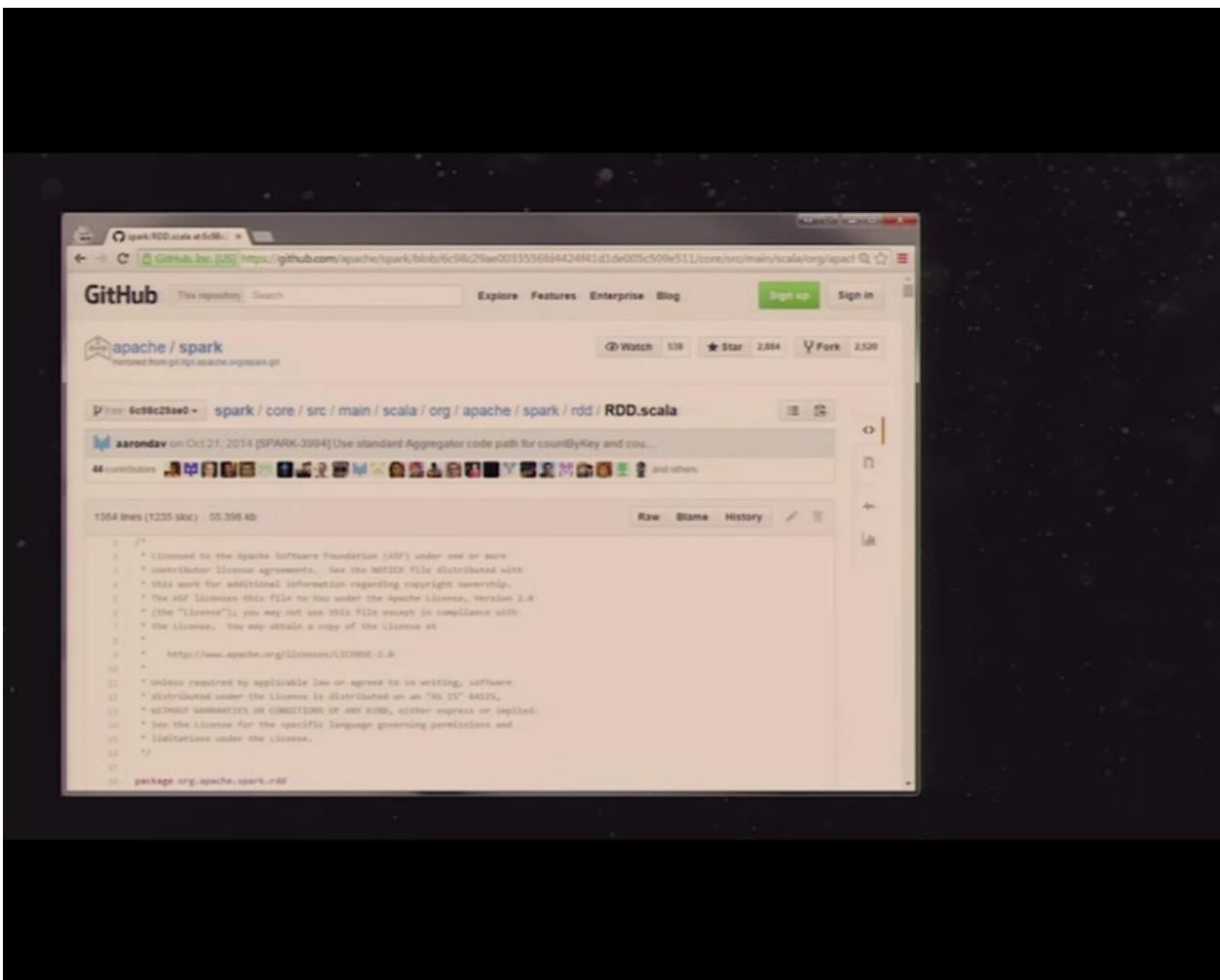
- Most transformations are element-wise (they work on one element at a time), but this is not true for all transformations

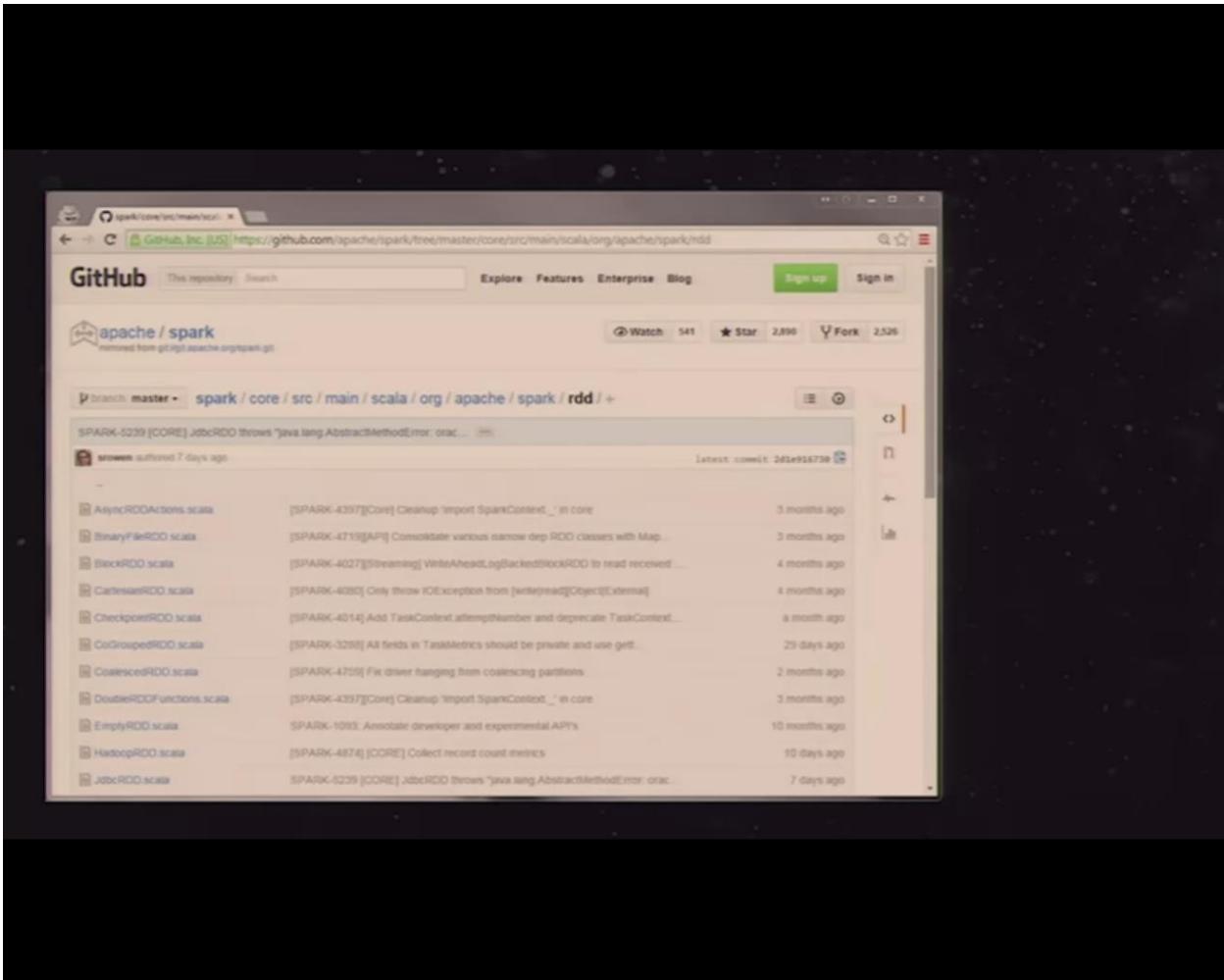
ACTIONS

reduce()	takeOrdered()
collect()	saveAsTextFile()
count()	saveAsSequenceFile()
first()	saveAsObjectFile()
take()	countByKey()
takeSample()	foreach()
saveToCassandra()	...

TYPES OF RDDS

- HadoopRDD
- FilteredRDD
- MappedRDD
- PairRDD
- ShuffledRDD
- UnionRDD
- PythonRDD
- DoubleRDD
- JdbcRDD
- JsonRDD
- SchemaRDD
- VertexRDD
- EdgeRDD
- CassandraRDD (*DataStax*)
- GeoRDD (*ESRI*)
- EsSpark (*ElasticSearch*)





RDD INTERFACE

- * 1) Set of partitions ("splits")
- * 2) List of dependencies on parent RDDs
- * 3) Function to compute a partition given parents
- * 4) Optional preferred locations
- * 5) Optional partitioning info for k/v RDDs (Partitioner)

This captures all current Spark operations!

EXAMPLE: HADOOPRDD

- * Partitions = one per HDFS block
- * Dependencies = none
- * Compute (partition) = read corresponding block

- * preferredLocations (part) = HDFS block location
- * Partitioner = none

EXAMPLE: FILTEREDRDD

- * Partitions = same as parent RDD
- * Dependencies = "one-to-one" on parent
- * Compute (partition) = compute parent and filter it

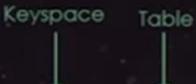
- * preferredLocations (part) = none (ask parent)
- * Partitioner = none

EXAMPLE: JOINEDRDD

- * Partitions = One per reduce task
 - * Dependencies = "shuffle" on each parent
 - * Compute (partition) = read and join shuffled data
-
- * preferredLocations (part) = none
 - * Partitioner = HashPartitioner(numTasks)

READING DATA USING THE C* CONNECTOR

```
val cassandraRDD = sc
    .cassandraTable("ks", "mytable")
Server side column   .select("col-1", "col-3")
& row selection     .where("col-5 = ?", "blue")
```



The diagram illustrates the mapping of database concepts to the code. Two arrows point downwards from the words 'Keyspace' and 'Table' to the corresponding parts in the Scala code. The arrow from 'Keyspace' points to the string 'ks' in the line '.cassandraTable("ks", "mytable")'. The arrow from 'Table' points to the string 'mytable' in the same line.

INPUT SPLIT SIZE

(for dealing with wide rows)

Start the Spark shell by passing in a custom cassandra.input.split.size:

```
ubuntu@ip-10-0-53-24:~$ dse spark -Dspark.cassandra.input.split.size=2000
Welcome to
```



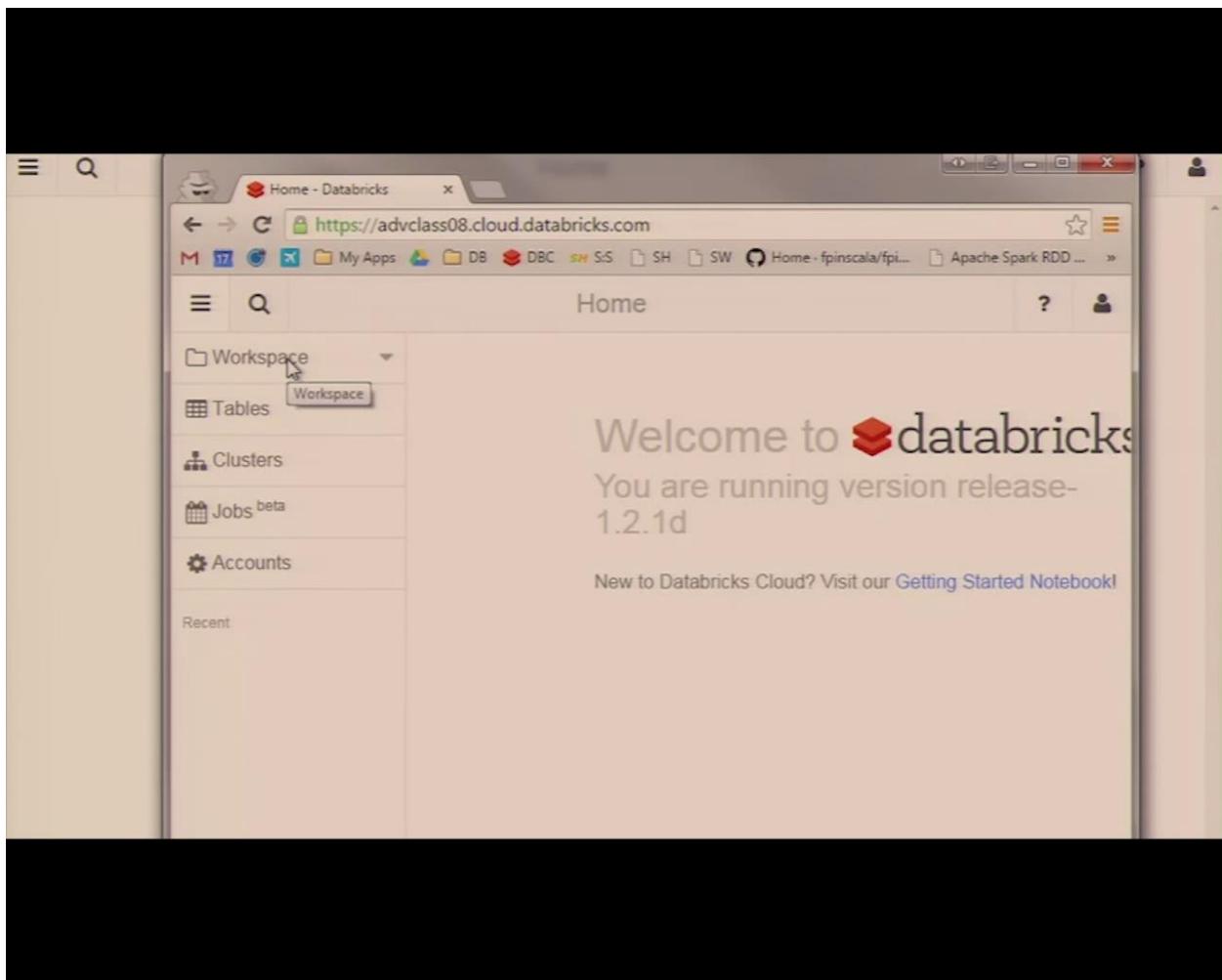
version 0.9.1

```
Using Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java
1.7.0_51)
```

```
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Created spark context..
Spark context available as sc.
Type in expressions to have them evaluated.
Type :help for more information.
```

```
scala>
```

The `cassandra.input.split.size` parameter defaults to 100.000. This is the approximate number of physical rows in a single Spark partition. If you have really wide rows (thousands of columns), you may need to lower this value. The higher the value, the fewer Spark tasks are created. Increasing the value too much may limit the parallelism level."



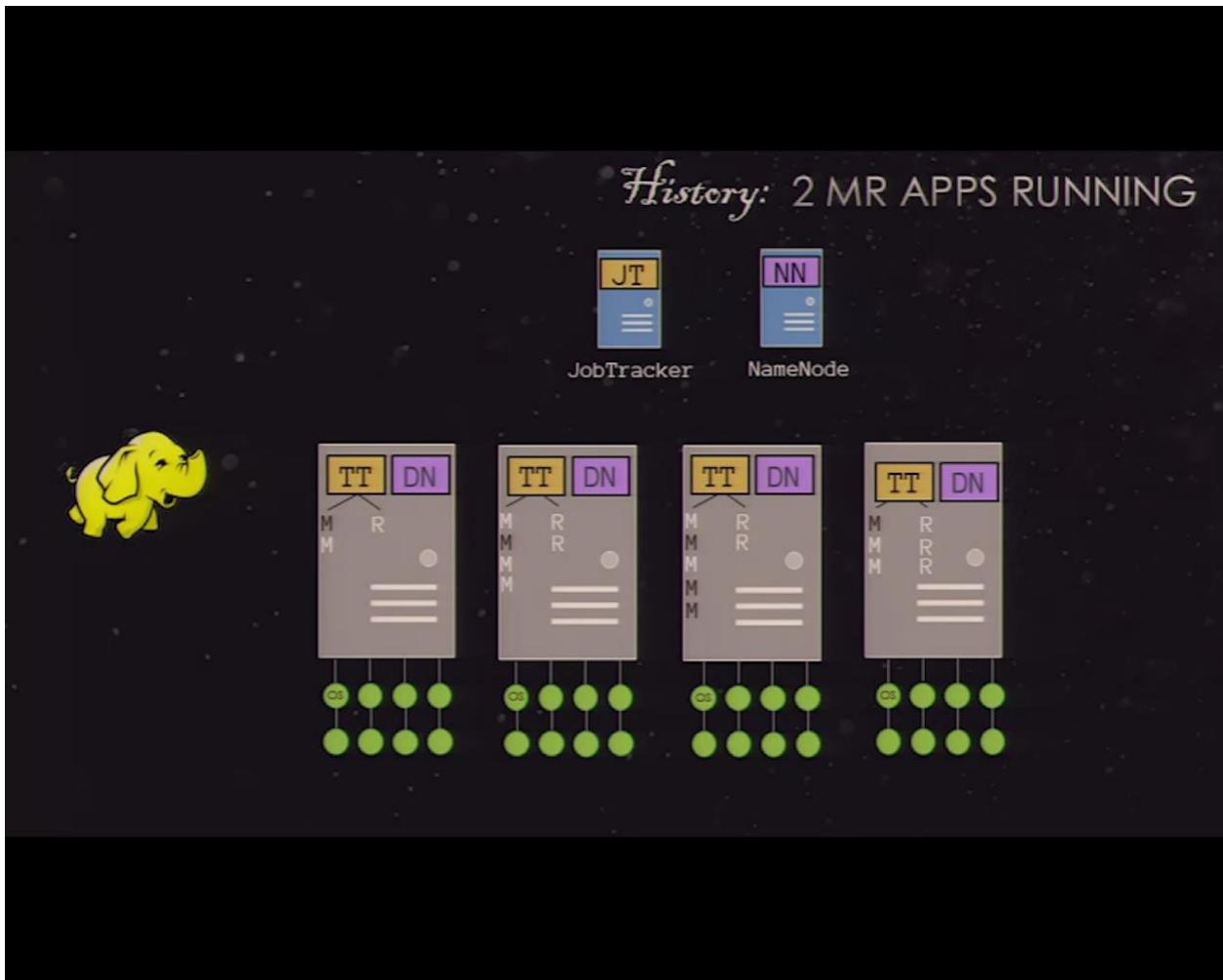
WAYS TO RUN SPARK

-  - Local 
 -   - Standalone Scheduler 
 -   - YARN 
 -  - Mesos 
- Static Partitioning
- Dynamic Partitioning

In map reduce there are different JVM to get the parallelism of the task while in spark we have different threads inside the executors

In Spark you have one executor JVM in each machine, inside the executor JVM you have slots, inside the slots tasks can run.

In Map reduce(Hadoop 1.x) there are different slots for map phase and reduce phase; there are not generic slots which can use either map or reduce. at the time of map phase is running (which takes most of the time), slots for reduce phase are sit idle and no one can use it while in spark the slots are generic slots they can run map , reduce or other tasks



Local Mode

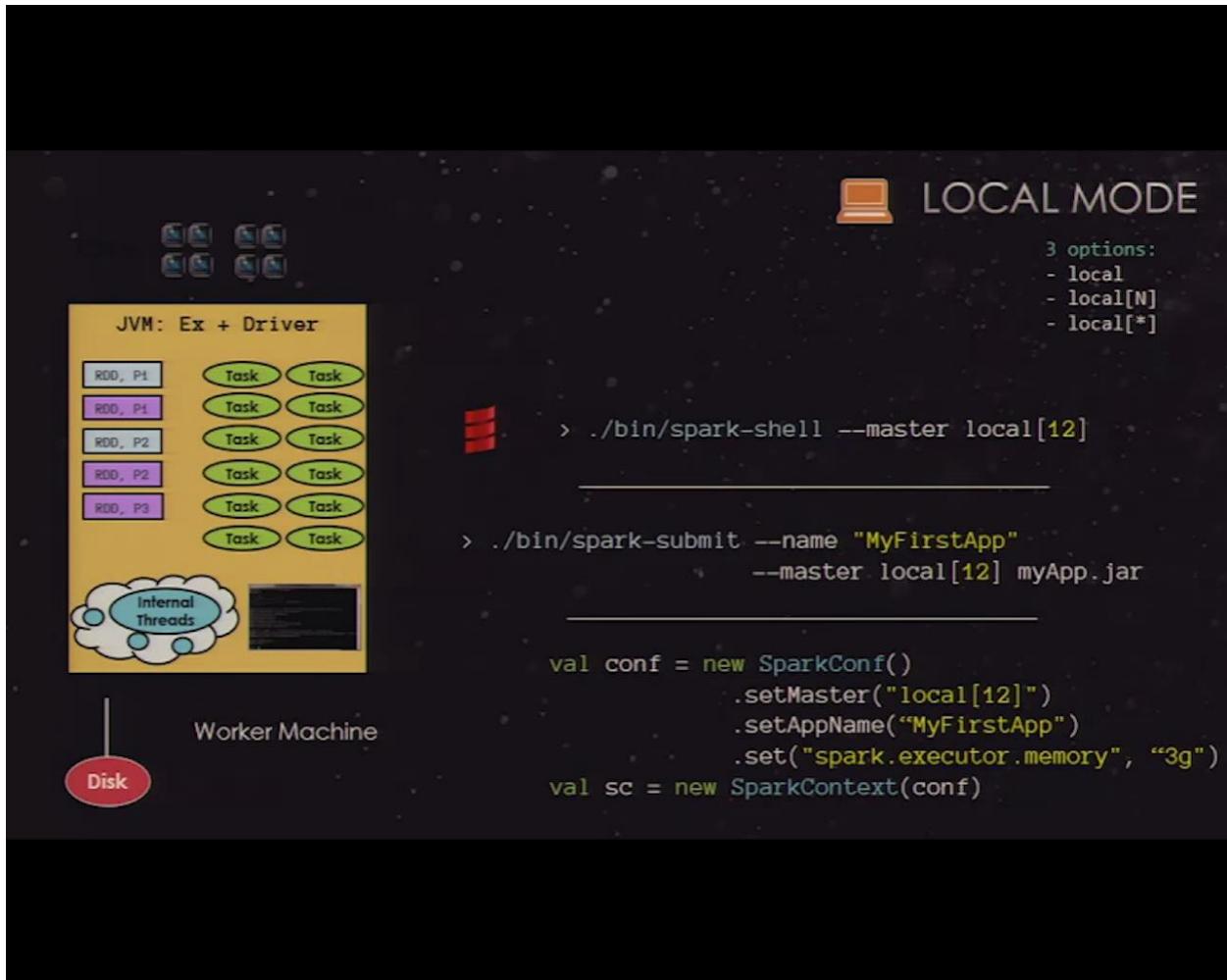
When you start Scala shell, it actually starts the JVM which execute the executor process and the driver process.

Tasks in spark are called cores (threads) that we can run in parallel, its common mistake that we correlate it with CPU core, most of the time we think if we have machine with 8 CPU core, we will use 2 cores for OS and rest 6 cores for spark JVM which is not correct. Generally in spark we define cores in multiple of 2 or 3. And we can define core more than our CPU core if we have 6 CPU cores we can go with 12 or 18 tasks core.

We can pass the threads or tasks when we start spark-shell by defining local [N], N is number of cores

If we pass only local it means only one worker thread which is not enough.

If we pass local [*] , it means run worker threads as many as there are logical cores in the machine, once again it is not a good setting as we want to run more threads than CPU cores.

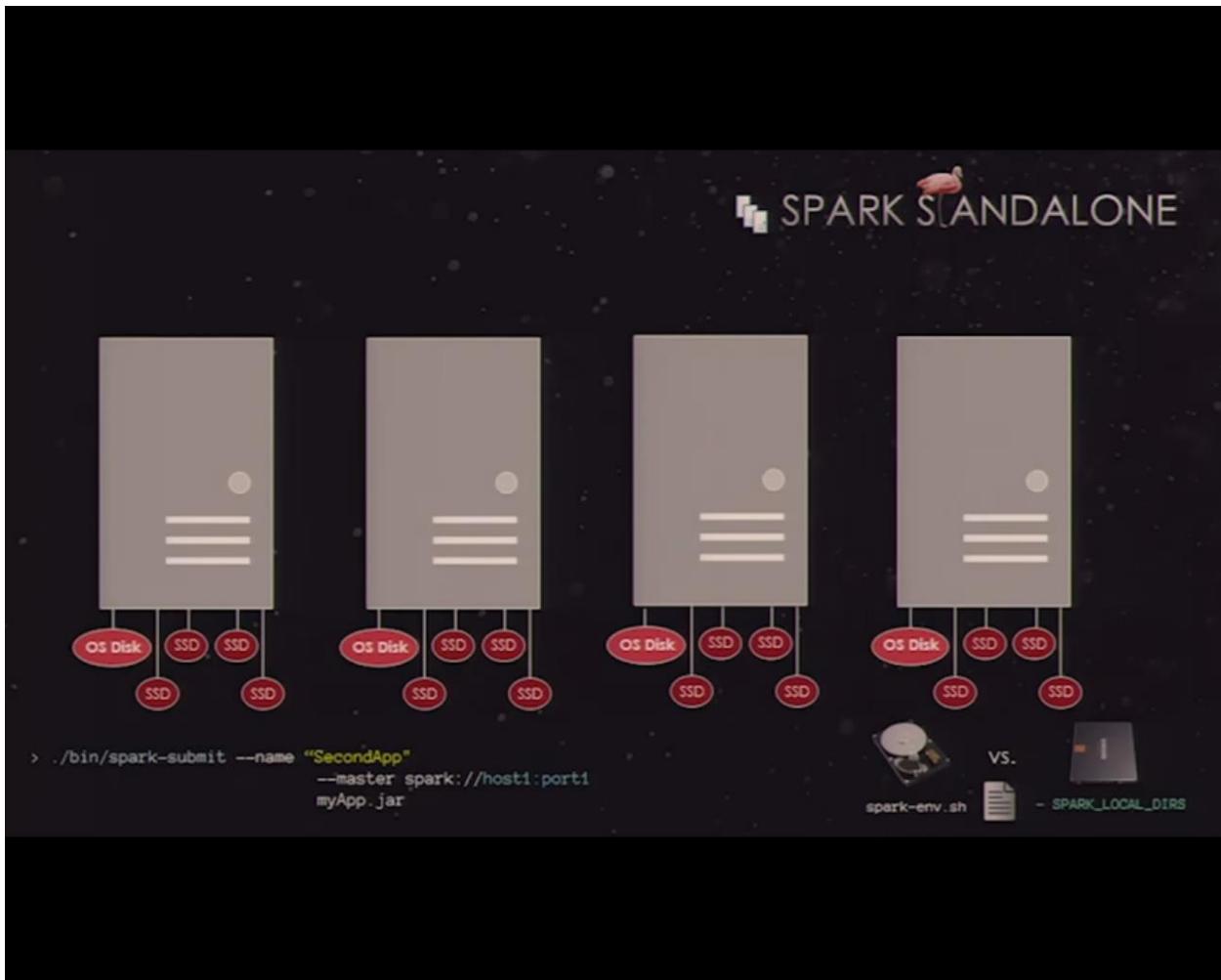


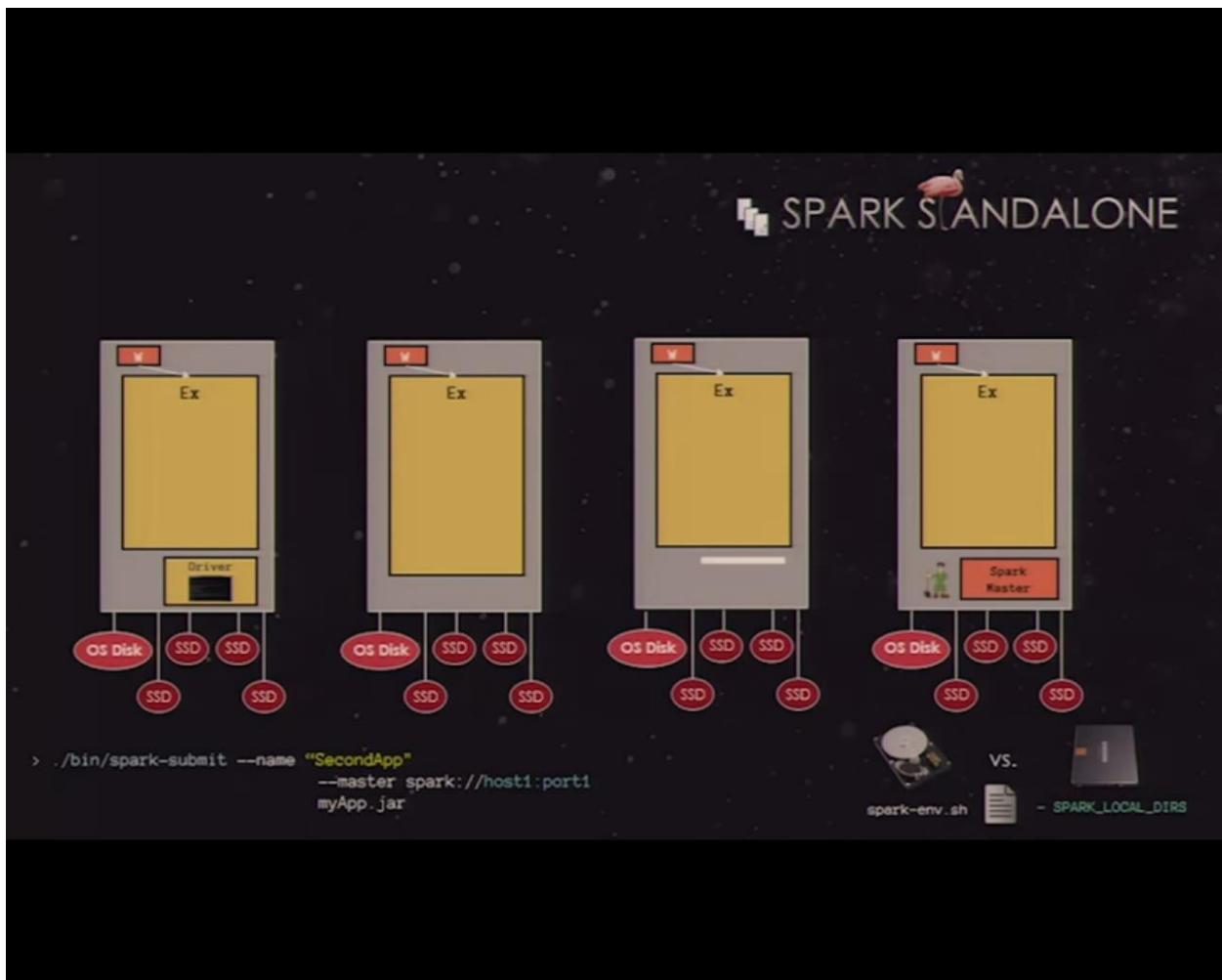
Code overwrites the number of thread submitted

Standalone mode

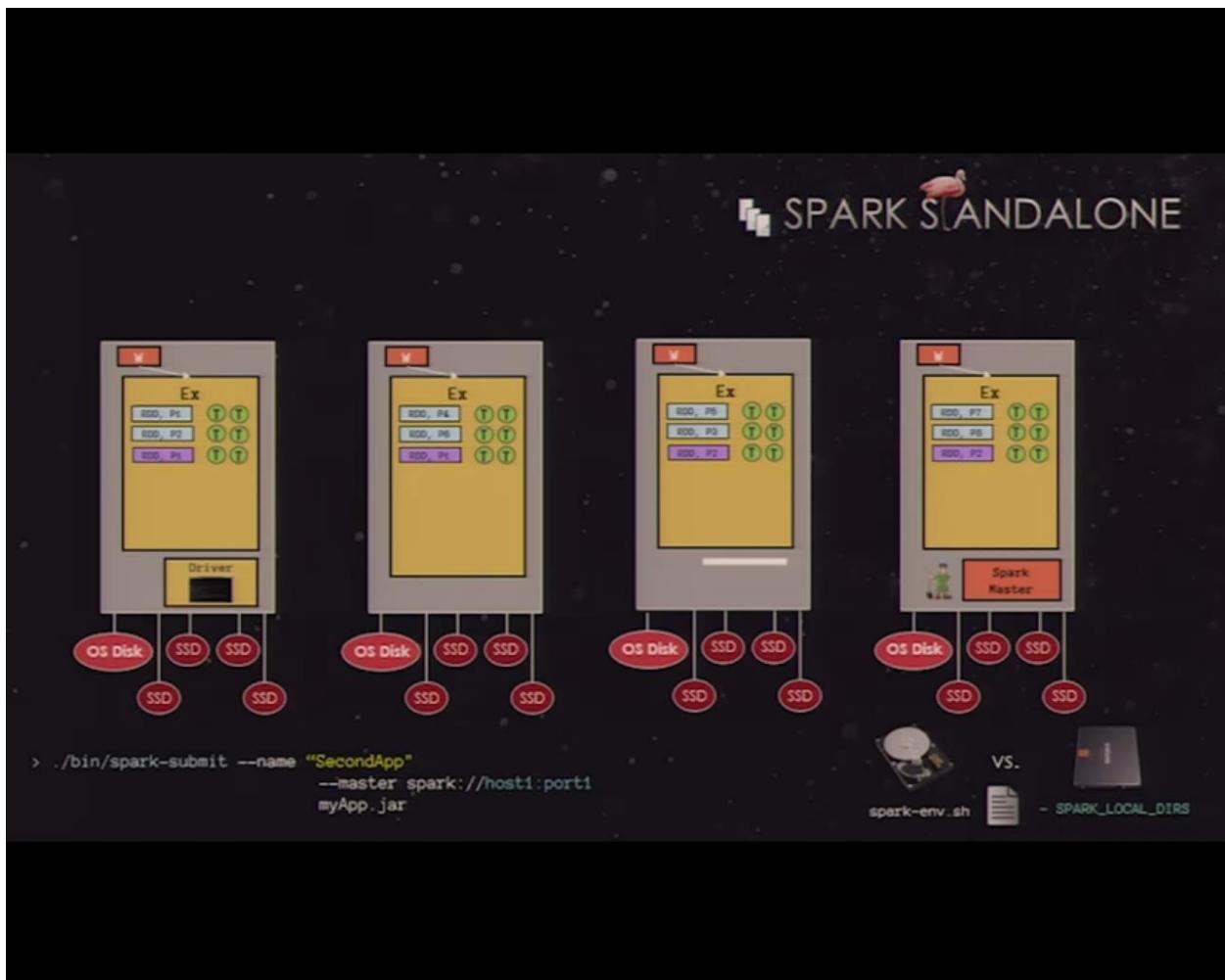
Example 4 node cluster

Spark-env.sh file has a property called SPARK_LOCAL_DIRS, which contains N, <N location of file mount points> what this property does is when RDD is very big and doesn't fit into memory, the rest portion are stored in those location which define in SPARK_LOCAL_DIRS. It is also used for intermediate shuffle data, like when map side operation runs and spills some data down to local disk and reducer come up and pull the data of the map spills over the network. So map side spills data goes down to this DIRS

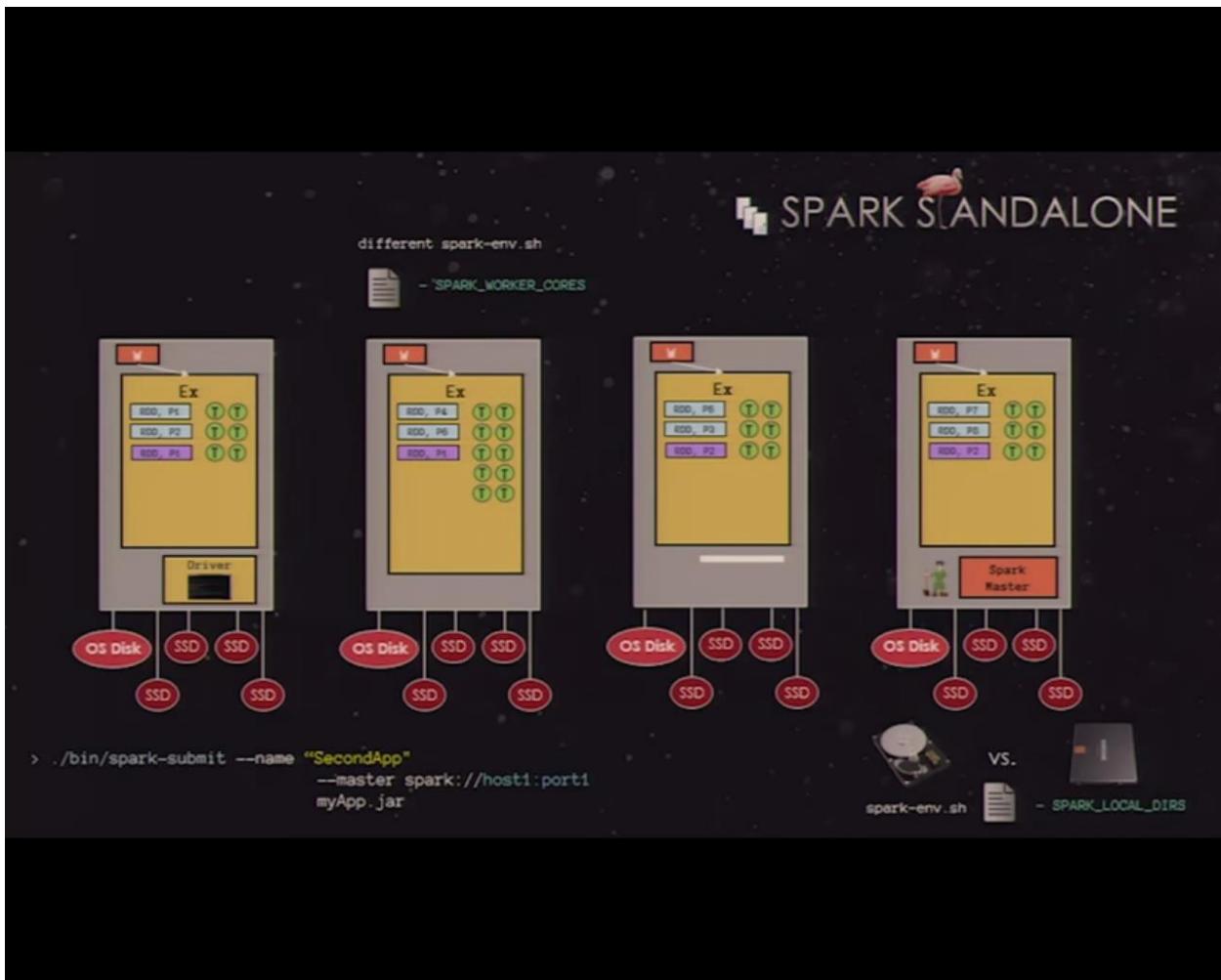


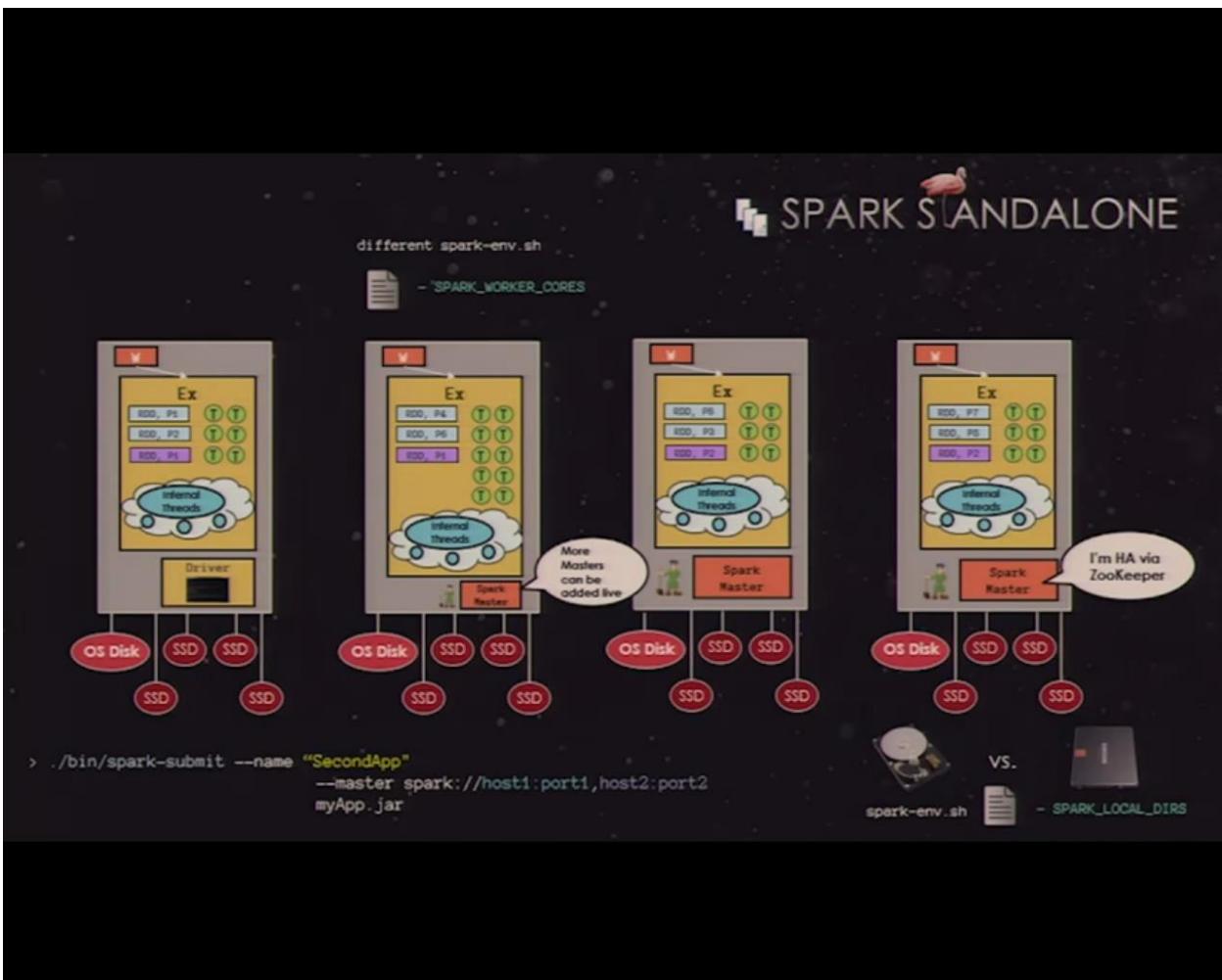


Yellow – spark application Orange – Default resource manager



If we have a machine with different configuration let's say in this example we have machine 2 with large CPU core than rest 3 we can set the more number of cores for second machine by specify property SPARK_WORKER_CORES





Multiple executors in standalone mode

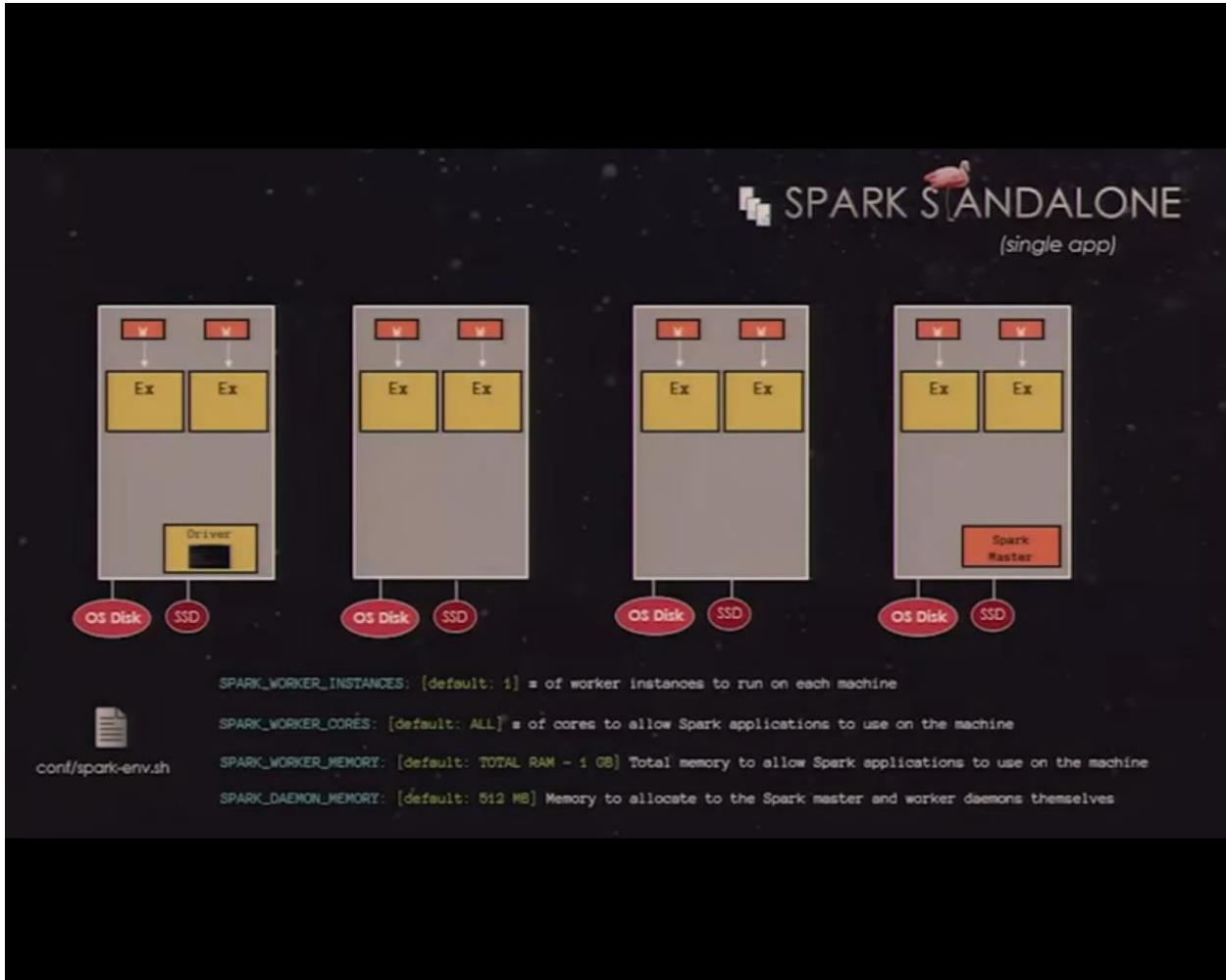
Advanced Apache Spark Training - Sameer Farooqui (Databricks)

SPARK STANDALONE
(multiple apps)

The diagram illustrates four separate nodes, each representing a Spark application running in standalone mode. Each node consists of a grey rectangular box containing a yellow 'Driver' box at the bottom and two green 'Ex' (Executor) boxes above it. In the first three nodes, the 'Driver' box is positioned below the 'Ex' boxes. In the fourth node, the 'Driver' box is positioned to the right of the 'Ex' boxes. Below each node, there are two red circles labeled 'OS Disk' and 'SSD'. A progress bar at the bottom indicates the video is at 1:50:36 / 5:58:30.

One worker in standalone mode cannot start more than one executor for same application. We cannot give more than 45 GB memory to one JVM.

If we only want to run one driver or one application and we have almost 125 GB memory



Here each worker starts two yellow executors but one worker cannot start two yellow executors, so we need to configure two workers in every machine

Set **SPARK_WORKER_INSTANCES=2**

SPARK_WORKERS_CORES - doesn't mean how many threads run in the worker JVM, it means how many cores can that worker give out to its underlying executors, so if value is configured as 12 and if start one executor with 6 cores so worker can still start another executor with 6 more cores.

SPARK_WORKER_MEMORY – means how much memory can worker give out to its underlying executors or JVM

SPARK_DAEMON_MEMORY – is the memory to allocate to the actual spark master JVM and spark worker demons.

In standalone mode we cannot define number of executors and number of tasks/cores for each executor but we can define how many cores we want and spark load balancer distribute the cores across the cluster and divide the cores among executors



The screenshot shows the Spark Master UI at `spark://10.0.64.177:7077`. The page displays cluster statistics and a detailed view of a single worker.

Cluster Statistics:

- URL: `spark://10.0.64.177:7077`
- Workers: 1
- Cores: 2 Total, 0 Used
- Memory: 4.0 GB Total, 0.0 B Used
- Applications: 0 Running, 0 Completed
- Drivers: 0 Running, 0 Completed

Workers:

ID	Address	State	Cores	Memory
worker-20140905191420-10.0.64.177-33571	10.0.64.177:33571	ALIVE	2 (0 Used)	4.0 GB (0.0 B Used)

Annotations:

- A callout bubble points to the "Memory" section of the cluster stats, stating: "Total potential memory this Spark cluster has access to is 4 GB (aka sum of how much memory each Worker, below, has access to)"
- A callout bubble points to the "Memory" column of the worker table, stating: "Amount of potential memory this particular Spark worker has access to"

Running Applications:

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications:

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Misconfiguration that we can see in UI below

Here we can see there is only one worker is running and it can give at max 3 cores to its worker and we also can see there is one running application, this running application is nothing but one of the executor and we can see that all the 3 cores is being used by this executor but it only took 512 MB memory out of 7.7 GB. So if only one executor is running and it is using all three cores it should use all 7.7 GB of memory to boost the performance. So probably we set **spark.executor.memory** to fix this scenario increase it to around 7.6 GB

The screenshot shows the Spark Master UI at `spark://10.0.12.60:7077`. The UI displays the following information:

Spark Master at spark://10.0.12.60:7077

URL: `spark://10.0.12.60:7077`
Workers: 1
Cores: 3 Total, 3 Used
Memory: 7.7 GB Total, 512.0 MB Used
Applications: 1 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

ID	Address	State	Cores	Memory
worker-20141110195851-10.0.12.60-35935	10.0.12.60:35935	ALIVE	3 (3 Used)	7.7 GB (512.0 MB Used)

Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20141110204831-0000	Spark shell	3	512.0 MB	2014/11/10 20:48:31	ec2-user	RUNNING	23 min

Completed Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Spark Worker at 10.0.12.60:35935

ec2-54-187-238-98.us-west-2.compute.amazonaws.com:7081

Google

Spark Worker at 10.0.12.60:35935

ID: worker-20141110195851-10.0.12.60-35935

Master URL: spark://10.0.12.60:7077

Cores: 3 (3 Used)

Memory: 7.7 GB (512.0 MB Used)

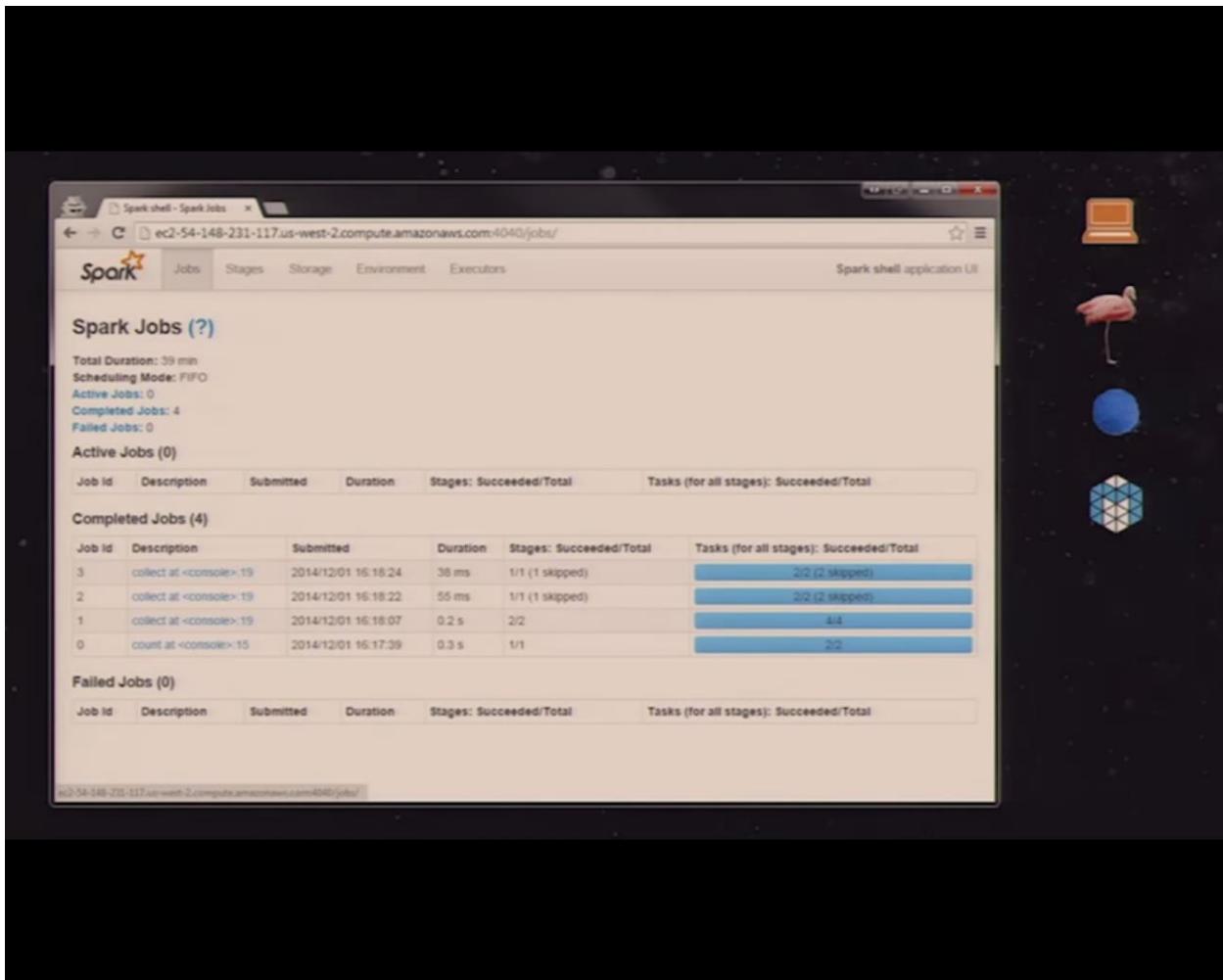
[Back to Master](#)

Running Executors (1)

ExecutorID	Cores	State	Memory	Job Details	Logs
0	3	RUNNING	512.0 MB	ID: app-20141110204831-0000 Name: Spark shell User: cassandra	stdout stderr

Spark Driver UI

Spark application is made of multiple jobs, every time we trigger an action it kickoff a new job



Job decomposes either one or multiple Stages

The screenshot shows the Spark UI interface with the title "Spark Stages (for all jobs)". It displays the following statistics:

- Total Duration: 39 min
- Scheduling Mode: FIFO
- Active Stages: 0
- Completed Stages: 5
- Failed Stages: 0

The "Active Stages (0)" section is empty.

The "Completed Stages (5)" section lists the following stages:

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
6	collect at <console>:19	+details 2014/12/01 16:18:24	28 ms	2/2	552.0 B			
4	collect at <console>:19	+details 2014/12/01 16:18:22	45 ms	2/2				
2	collect at <console>:19	+details 2014/12/01 16:18:07	69 ms	2/2				
1	map at <console>:16	+details 2014/12/01 16:18:07	76 ms	2/2	254.0 B		737.0 B	
0	count at <console>:15	+details 2014/12/01 16:17:40	0.2 s	2/2	254.0 B			

Stage decomposes one or more tasks

Storage Tab

Storage tab let us know the RDD that cached either in memory or in disk

Here 2 partition is there in RDD and 100% is cached means both partitioned are cached and size in memory is 552 bytes.

We can get number of partition in RDD by calling `getNumPartition` method on RDD

The screenshot shows the Spark UI application running in a browser window titled "Spark shell - Storage". The URL is "ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/storage/". The UI has tabs for Jobs, Stages, Storage, Environment, and Executors. The Storage tab is active. Below the tabs is a section titled "Storage". A table displays the following data:

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
5	Memory Deserialized 1x Replicated	2	100%	552.0 B	0.0 B	0.0 B

If we read some java string in hdfs file which is around 100 MB and if we cache that file into memory it will take around 200-300 MB space (That is overhead of java serialization)

We can see more details of cached RDD below, Data is distributed only one executor

The screenshot shows a Java-based application window titled "Spark shell - RDD Storage". The URL in the address bar is "ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/storage/rdd/?id=5". The window has tabs for Jobs, Stages, Storage, Environment, and Executors, with "Storage" selected. On the right side, there are four decorative icons: a laptop, a flamingo, a blue sphere, and a hexagonal grid.

RDD Storage Info for 5

Storage Level: Memory Deserialized 1x Replicated
Cached Partitions: 2
Total Partitions: 2
Memory Size: 552.0 B
Disk Size: 0.0 B

Data Distribution on 1 Executors

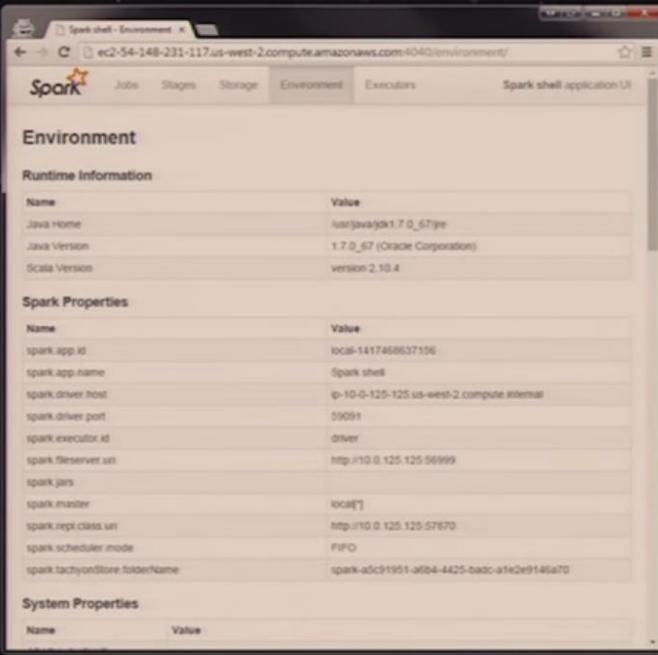
Host	Memory Usage	Disk Usage
localhost:38329	552.0 B (265.4 MB Remaining)	0.0 B

2 Partitions

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_5_0	Memory Deserialized 1x Replicated	424.0 B	0.0 B	localhost:38329
rdd_5_1	Memory Deserialized 1x Replicated	128.0 B	0.0 B	localhost:38329

This is not a good scenario that we have two partition and most of the data contain by only one partition

Environment

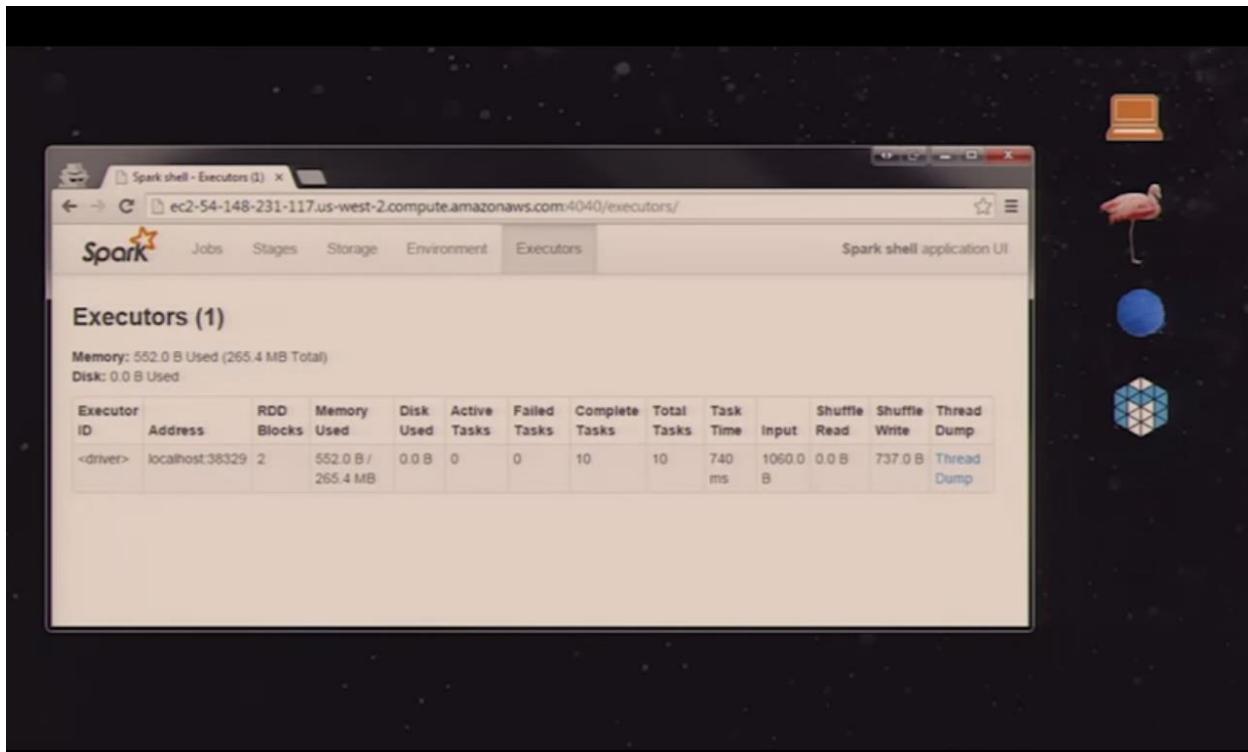


The screenshot shows the 'Environment' tab of the Spark shell application UI. The page title is 'Spark shell - Environment'. The top navigation bar includes links for Jobs, Stages, Storage, Environment (which is selected), Executors, and Spark shell application UI. The main content area is titled 'Environment' and contains three sections: 'Runtime Information', 'Spark Properties', and 'System Properties'. The 'Runtime Information' section lists Java Home (usr/java/jdk1.7.0_67/jre), Java Version (1.7.0_67 (Oracle Corporation)), and Scala Version (version 2.10.4). The 'Spark Properties' section lists various configuration parameters with their values, such as spark.app.id (local-1417469637156), spark.app.name (Spark shell), spark.driver.host (ip-10-0-125-125.us-west-2.compute.internal), spark.driver.port (55091), spark.executor.id (driver), spark.flrserver.uri (http://10.0.125.125:56999), spark.jars, spark.master (local[*]), spark.repl.class.uri (http://10.0.125.125:57670), spark.scheduler.mode (FIFO), and spark.tachyonstore.tokenName (spark-ac5c91951-a6b4-4425-badc-a1e0e99146a70). The 'System Properties' section is currently empty.

Setting – Default >> configuration file >>at the time of submit job >> in the spark source code

Highest precedence will override the settings of lower.

Executor tab



The screenshot shows a Java-based application window titled "Spark shell - Executors (1) X". The URL in the address bar is "ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/executors/". The window has a header with tabs: "Jobs", "Stages", "Storage", "Environment", "Executors" (which is selected), and "Spark shell application UI". Below the header, there is a section titled "Executors (1)". It displays memory usage statistics: "Memory: 552.0 B Used (265.4 MB Total)" and "Disk: 0.0 B Used". A table follows, showing the details of the single executor:

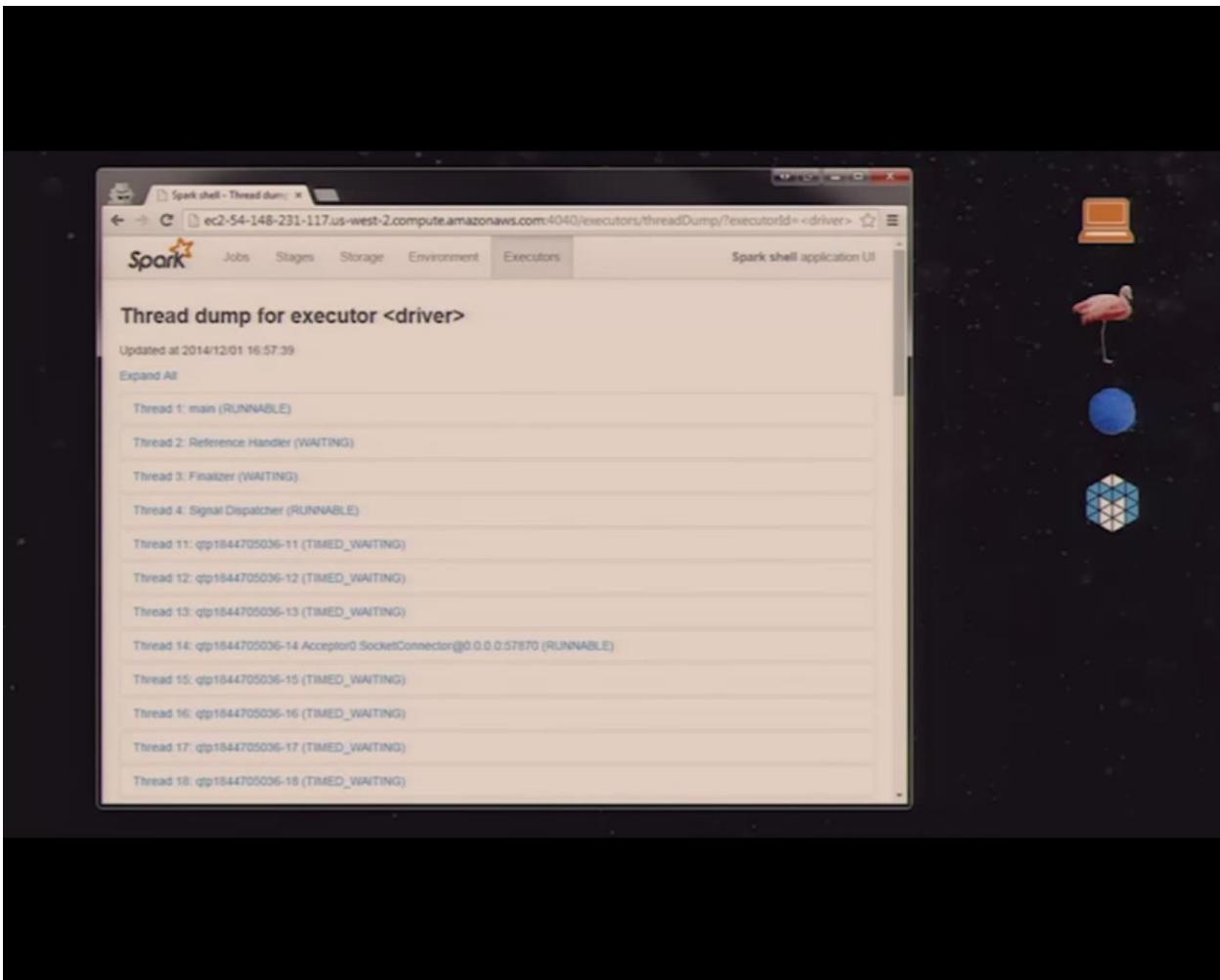
Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Thread Dump
<driver>	localhost:38329	2	552.0 B / 265.4 MB	0.0 B	0	0	10	10	740 ms	1060.0 B	0.0 B	737.0 B	Thread Dump

On the right side of the window, there are four decorative icons: a yellow typewriter, a pink flamingo, a blue circle, and a blue hexagonal pattern.

You can see only one executor and executor id is <driver> that is a hint that we are probably running on local mode.

Thread Dump

We can collect stack thread from this tab



Spark shell - Thread dump

ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/executors/threadDump/<executorId><driver>

```
Thread 56: qtp1837961181-56 (TIMED_WAITING)
Thread 57: qtp1837961181-57 (TIMED_WAITING)
Thread 58: qtp1837961181-58 (TIMED_WAITING)
Thread 59: Timer-0 (WAITING)
Thread 60: Driver Heartbeater (TIMED_WAITING)
Thread 69: shuffle-server-0 (RUNNABLE)

sun.nio.ch.EPollArrayWrapper.epollWait0(Native Method)
sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:108)
sun.nio.ch.EPollSelectorImpl$noteSelect(EPollSelectorImpl.java:79)
sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
sun.nio.ch.SelectorImpl.select(SelectorImpl.java:94)
io.netty.channel.nio.NioEventLoop.select(NioEventLoop.java:422)
io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:338)
io.netty.util.concurrent.SingleThreadEventExecutor$2.run(SingleThreadEventExecutor.java:118)
java.lang.Thread.run(Thread.java:744)

Thread 78: Spark Context Cleaner (TIMED_WAITING)
Thread 79: sparkDriver akka.actor.default-dispatcher-14 (TIMED_WAITING)

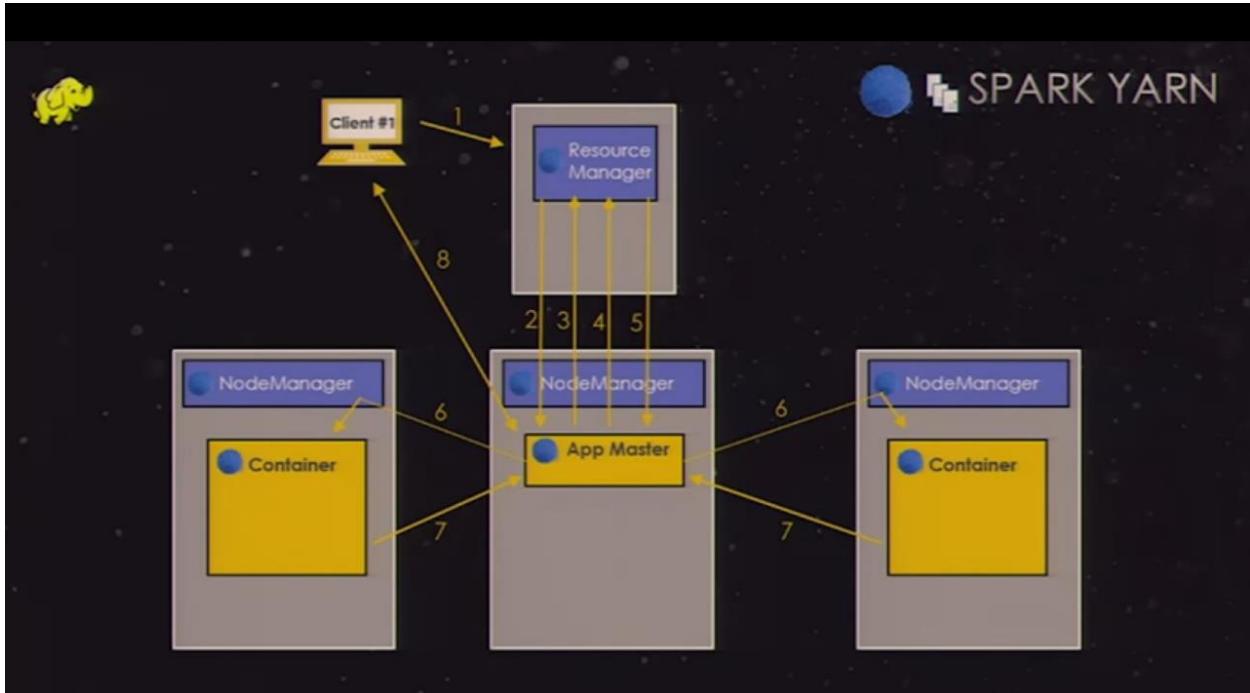
Thread 83: task-result-getter-0 (WAITING)
Thread 84: task-result-getter-1 (WAITING)
Thread 85: ForkJoinPool-3-worker-7 (WAITING)
```

YARN Mode



App Master is not going to do any of the work for your application it is like a job scheduler, App master communicate to the resource manager and ask for Container to schedule application, resource manager than hand over application master some keys and tokens to start the container

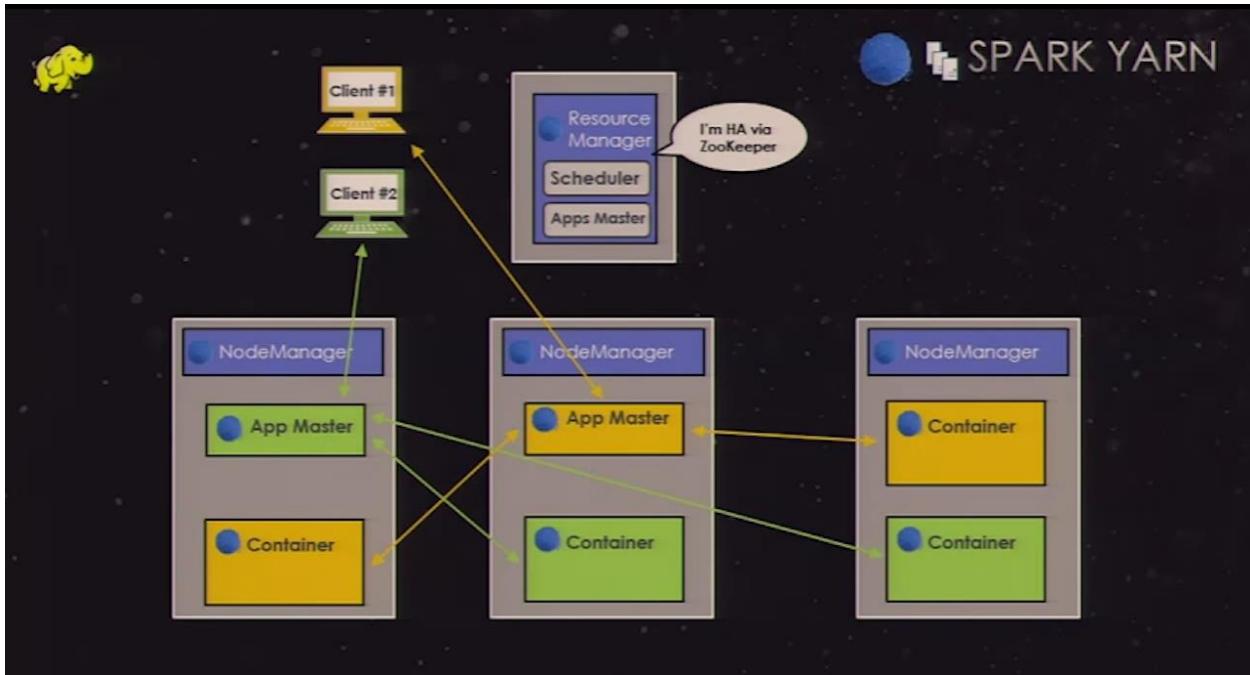
Example – one application running



Once App master get the control Resource manager work is over and App master directly communicate with client

Example – two different applications are running

Green and yellow and each App master has two containers doing the work for him



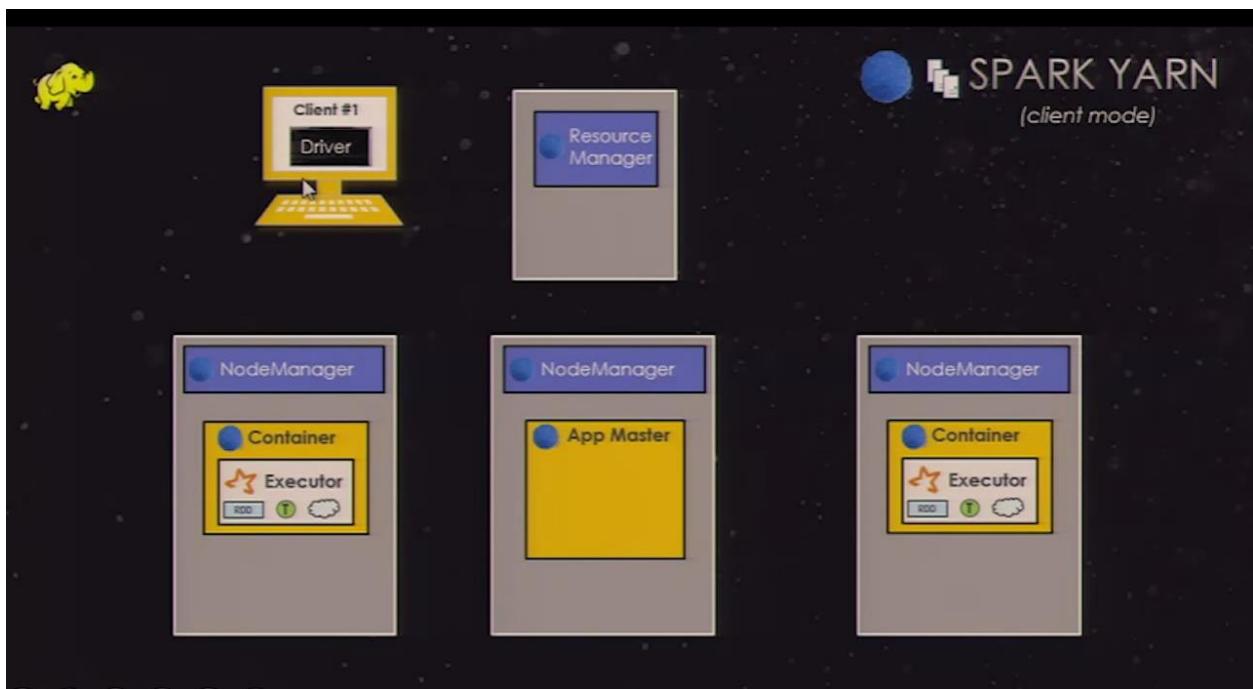
Resource manager has scheduler that decides where the App master will run and where these containers will schedule and has one Application master manager

YARN -Client Mode

In client mode driver runs on the client machine, it's like you start scala and python shell on your laptop

So if you use collect action on an RDD, the output of action will be shipped to the driver jvm and you can see on your machine (client machine)

In Yarn the two executors running under two containers can direct contact with driver (same as standalone mode there we have workers and in yarn we have node manager which launch container and start the executors inside it), Resource manager is doing same what the spark-master is doing in standalone mode, RM is scheduling where the executor will run, however there is one more scheduler inside Driver that decide where the tasks will be scheduled within the executors

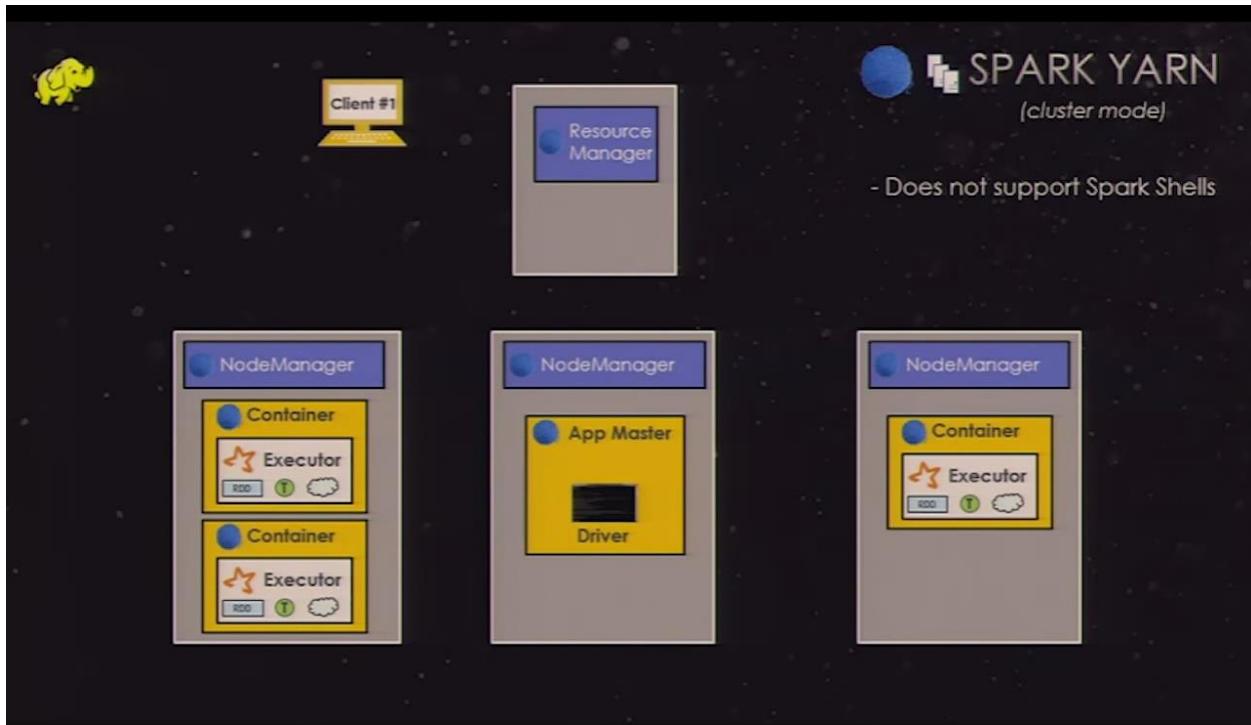


Scenario- if we have 100 node clusters and we ask to launch 20 Executors which are reading hdfs file, the executors should be launched by data locality of blocks , one way to do that launch 100 executors (need to check if yarn itself launch executors as per data locality)

The problem is with this mode what if we remove our laptop from the cluster and left that would terminate the entire application because driver is disconnect with executors, remember Driver is available on our laptop in Client mode

YARN -Cluster Mode

In this mode client submit the application including the driver which might be like jar file or python script and then driver will run within the yarn application master itself. In this case we can't run spark shell so we will not probably use method like collect but we can save the result to HDFS



```
spark.dynamicAllocation.enabled – true / false  
spark.dynamicAllocation.minExecutors - Lower bound on the number of executors (10)  
spark.dynamicAllocation.maxExecutors - Upper bound on the number of executors (100)  
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout = 60  
spark.dynamicAllocation.schedulerBacklogTimeout (M)  
spark.dynamicAllocation.executorIdleTimeout (K)  
spark.dynamicAllocation.initialExecutors - Number of executors to start with
```

If dynamic allocation is enabled so it start with 10 executors and after 60 sec (**sustainedSchedulerBacklogTimeout**) ,check if there is backlog for whole bunch of task built up in the driver which are not being scheduled yet than increase the number of executors little bit like 10 to 20.If **schedulerBacklogTimeout** hit may be 30 sec , again check backlogged tasks, if tasks are still pending increase number of executors exponentially and keep checking every 30 sec. after hit max we might need to release resources so check if any executor idle for **executorIdleTimeout** remove it and keep terminating until it come to **minExecutors**

spark.dynamicAllocation.sustainedSchedulerBacklogTimeout (N) - If the backlog is sustained for this duration, add more executors, this is used only after the initial backlog timeout is exceeded

spark.dynamicAllocation.schedulerBacklogTimeout (M) - If there are backlogged tasks for this duration, add new executors

spark.dynamicAllocation.executorIdleTimeout (K) - If an executor has been idle for this duration, remove this executor



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)
```

<https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/ExecutorAllocationManager.scala>

YARN Resource Manager UI

YARN resource manager UI: http://<ip_address>:8088

(No apps running)

The screenshot shows a web browser window titled "Cluster 1 - YARN (MR2) - All Applications". The URL is "104.130.159.101:8088/cluster". The page has a "hadoop" logo and the title "All Applications". On the left, there's a sidebar with "Cluster Metrics" and "User Metrics for dr.who". The main area shows tables for "Cluster Metrics" and "User Metrics". Both tables have zero entries. A message at the bottom says "No data available in table". Below the table, it says "Showing 0 to 0 of 0 entries".

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	De
0	0	0	0	0	0 B	2.71 GB	0 B	0	4	0	1	0

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Mem Rese
0	0	0	0	0	0	0	0 B	0 B	0 B

|| ▶ 🔍 2:31:22 / 5:58:30



```
[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

ec2-54-149-62-154.us-west-2.compute.amazonaws.com:8088/cluster/apps

Logged in as: dr.who

All Applications

hadoop

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Vcores Used	Vcores Total	Vcores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
3	0	0	3	0	0 B	3.46 GB	0 B	0	4	0	1	0	0	0	0

User Metrics for dr.who

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved	Vcores Used	Vcores Pending	Vcores Reserved
0	0	0	3	0	0	0	0 B	0 B	0 B	0	0	0

Show: 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1417541524005_0003	ec2-user	Spark Pi	SPARK	root,ec2-user	Thu, 04 Dec 2014 15:30:43 GMT	Thu, 04 Dec 2014 15:31:14 GMT	FINISHED	SUCCEEDED	<div style="width: 100%;">100%</div>	History
application_1417541524005_0002	ec2-user	Spark Pi	SPARK	root,ec2-user	Thu, 04 Dec 2014 15:25:48 GMT	Thu, 04 Dec 2014 15:26:19 GMT	FINISHED	SUCCEEDED	<div style="width: 100%;">100%</div>	History
application_1417541524005_0001	ec2-user	Spark Pi	SPARK	root,ec2-user	Thu, 04 Dec 2014 15:25:18 GMT	Thu, 04 Dec 2014 15:25:35 GMT	FINISHED	SUCCEEDED	<div style="width: 100%;">100%</div>	History

Showing 1 to 3 of 3 entries

|| ▶ 🔍 2:31:55 / 5:58:30



The screenshot shows a web browser window displaying the Hadoop Cluster Overview page. The URL is `ec2-54-149-62-154.us-west-2.compute.amazonaws.com:8088/cluster/app/application_1417641624005_0003`. The page is titled "hadoop".

Application Overview

User:	ec2-user
Name:	Spark PI
Application Type:	SPARK
Application Tags:	
State:	FINISHED
Final Status:	SUCCEEDED
Started:	4-Dec-2014 10:30:43
Elapsed:	31sec
Tracking URL:	HUE002
Diagnostics:	

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	57388 MB-seconds, 45 vcore-seconds

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	4-Dec-2014 10:30:43	ip-10-0-72-36.us-west-2.compute.internal:8042	logs

```
[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

The screenshot shows the Hadoop cluster metrics and applications interface. The top navigation bar includes links for About, Nodes, Applications, Scheduler, and Tools. The left sidebar displays cluster metrics and a list of application states: NEW, NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, and KILLED. The main content area is titled "All Applications". It features a table of cluster metrics and a detailed list of applications.

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes
4	0	0	4	0	0 B	3.46 GB	0 B	0	4	0	1	0

User Metrics for dr.who

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved	VCores Used
0	0	0	4	0	0	0	0 B	0 B	0 B	0

All Applications

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus
application_1417641624005_0004	ec2-user	org.apache.spark.examples.SparkPi	SPARK	root.ec2-user	Thu, 04 Dec 2014 15:37:10 GMT	Thu, 04 Dec 2014 15:37:54 GMT	FINISHED	SUCCESS
application_1417541624005_0003	ec2-user	Spark Pi	SPARK	root.ec2-user	Thu, 04 Dec 2014 15:39:13	Thu, 04 Dec 2014 15:39:14	FINISHED	SUCCESS

App running in **cluster** mode

Logged in as: dr.who

hadoop

Application Overview

User:	ec2-user
Name:	org.apache.spark.examples.SparkPI
Application Type:	SPARK
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	4-Dec-2014 10:37:10
Elapsed:	43sec
Tracking URL:	History
Diagnostics:	

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	83705 MB-seconds, 66 vcore-seconds

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	4-Dec-2014 10:37:10	ip-10-0-72-36.us-west-2.compute.internal:8042	logs



App running in **cluster** mode

Log Type: stderr
Log Length: 22704
Showing 4096 bytes of 22704 total. Click [here](#) for the full log.
./sparkStaging/application_1417641624005_0004/spark-assembly-1.1.0-cdh5.2.1-hadoop2.5.0-cdh5.2.1.jar" } size: 95571683 timestamp: 14/12/04 10:37:52 INFO yarn.YarnAllocationHandler: Completed container container_1417641624005_0004_01_000002 (state: COMPLETE)
14/12/04 10:37:52 INFO yarn.ExecutorRunnable: Setting up executor with environment: java:CLASSPATH -> \$PWD:\$PWD/_spark_.jar:\$PWD/_spark_jar\$PWD/_spark_jar\$PWD/_spark_jar\$PWD/_spark_jar
14/12/04 10:37:52 INFO yarn.ExecutorRunnable: Starting execution of task 0.0 in listernerContainer
14/12/04 10:37:52 INFO org.ContainerManagementProtocol: Opening proxy : ip-10-0-72-36.us-west-2.compute.internal:8041
14/12/04 10:37:52 INFO org.ContainerManagementProtocol: RemoteContainerManager: RemoteContainerManager
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.YarnAllocationHandler: Will allocate 1 executor containers, each with 1408 memory
14/12/04 10:37:52 INFO yarn.YarnAllocationHandler: Container request (host: Any, priority: 1, capability: <memory:1408, vCores:
14/12/04 10:37:53 INFO spark.MapOutputTrackerMasterActor: MapOutputTrackerActor stopped!
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.DistributedCacheManager: DistributedCacheManager stopped
14/12/04 10:37:53 INFO remote.RemoteActorRefProvider\$RemotingTerminator: Shutting down remote daemon.
14/12/04 10:37:53 INFO remote.RemoteActorRefProvider\$RemotingTerminator: Remote daemon shut down, proceeding with flushing remotes.
14/12/04 10:37:53 INFO Remoting: Remoting shut down.
14/12/04 10:37:53 INFO spark.RemoteActorRefProvider\$RemotingTerminator: Remoting shut down.
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.AWSClientImpl: 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 spark.SparkContext: Stopped spark context at http://ip-10-0-72-36.us-west-2.compute.internal:41825
14/12/04 10:37:54 INFO yarn.ApplicationMaster\$Main\$1: Invoking sc stop from shutdown hook
14/12/04 10:37:54 INFO ui.SparkUI: Stopped Spark web UI at <http://ip-10-0-72-36.us-west-2.compute.internal:41825>
14/12/04 10:37:54 INFO spark.SparkContext: SparkContext already stopped.

Log Type: stdout
Log Length: 23
Std is roughly 3.142392

History server to keep track of your terminated jobs

The screenshot shows a web browser window titled "History Server" with the URL "ec2-54-149-62-154.us-west-2.compute.amazonaws.com:18088". The page displays the "Spark History Server" interface. At the top, it says "Event Log Location: hdfs://ip-10-0-72-36.us-west-2.compute.internal:8020/user/spark/applicationHistory". Below that, it says "Showing 1-2 of 2". A red arrow points to the "App Name" column header of a table. The table has columns: App Name, Started, Completed, Duration, Spark User, and Last Updated. It contains two rows:

App Name	Started	Completed	Duration	Spark User	Last Updated
Spark shell	2014/12/04 09:14:01	2014/12/04 09:21:19	7.3 min	ec2-user	2014/12/04 09:21:20
Spark shell	2014/12/04 09:07:36	2014/12/04 09:13:47	6.2 min	ec2-user	2014/12/04 09:13:48

Conclusion

PLUGGABLE RESOURCE MANAGEMENT			
	Spark Central Master	Who starts Executors?	Tasks run in
Local	[none]	Human being	Executor
Standalone	Standalone Master	Worker JVM	Executor
YARN	YARN App Master	Node Manager	Executor
Mesos	Mesos Master	Mesos Slave	Executor

DEPLOYING AN APP TO THE CLUSTER

`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
```

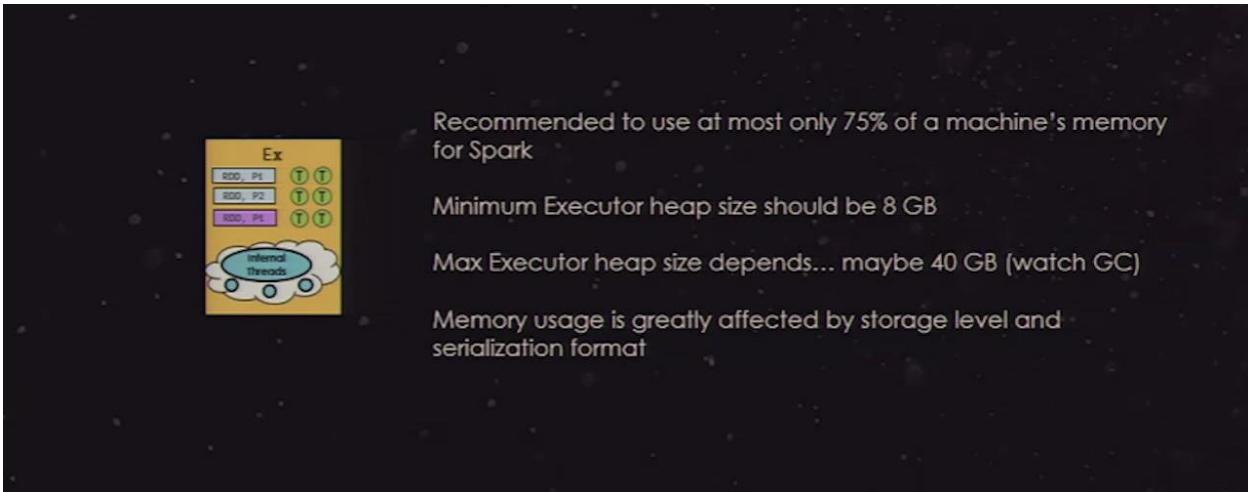
Table 7-2. Possible values for the `--master` flag in `spark-submit`

Value	Explanation
 spark://host:port	Connect to a Spark Standalone master at the specified port. By default Spark Standalone master's listen on port 7077 for submitted jobs.
 mesos://host:port	Connect to a Mesos cluster master at the specified port. By default Mesos masters listen on port 5050 for submitted jobs.
 yarn	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.
 local	Run in local mode with a single core.
local[N]	Run in local mode with N cores.
local[*]	Run in local mode and use as many cores as the machine has.

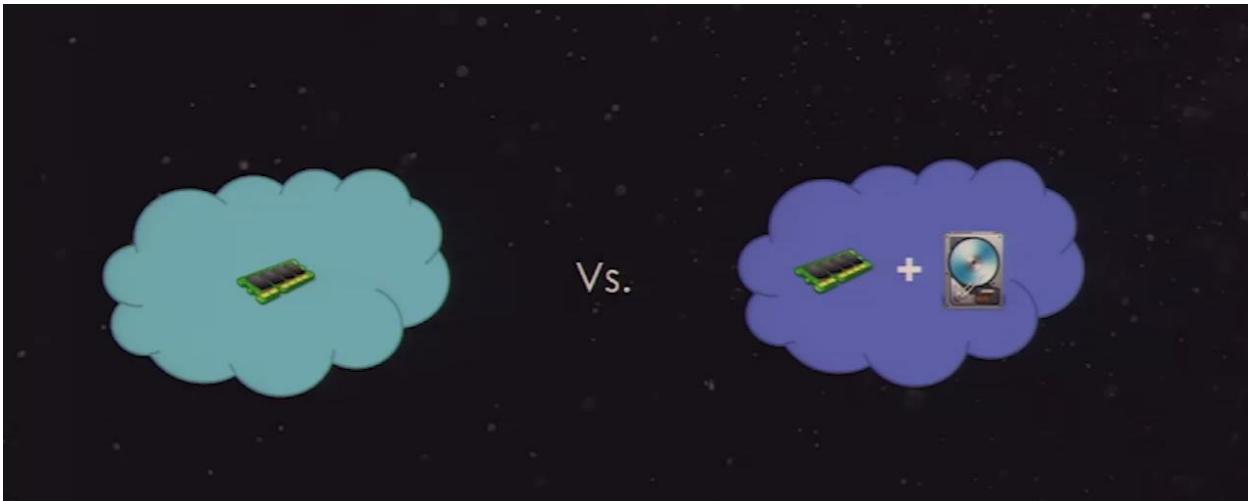
Source: Learning Spark

<http://tinyurl.com/sparknycsurveyresults>

Memory and persistence



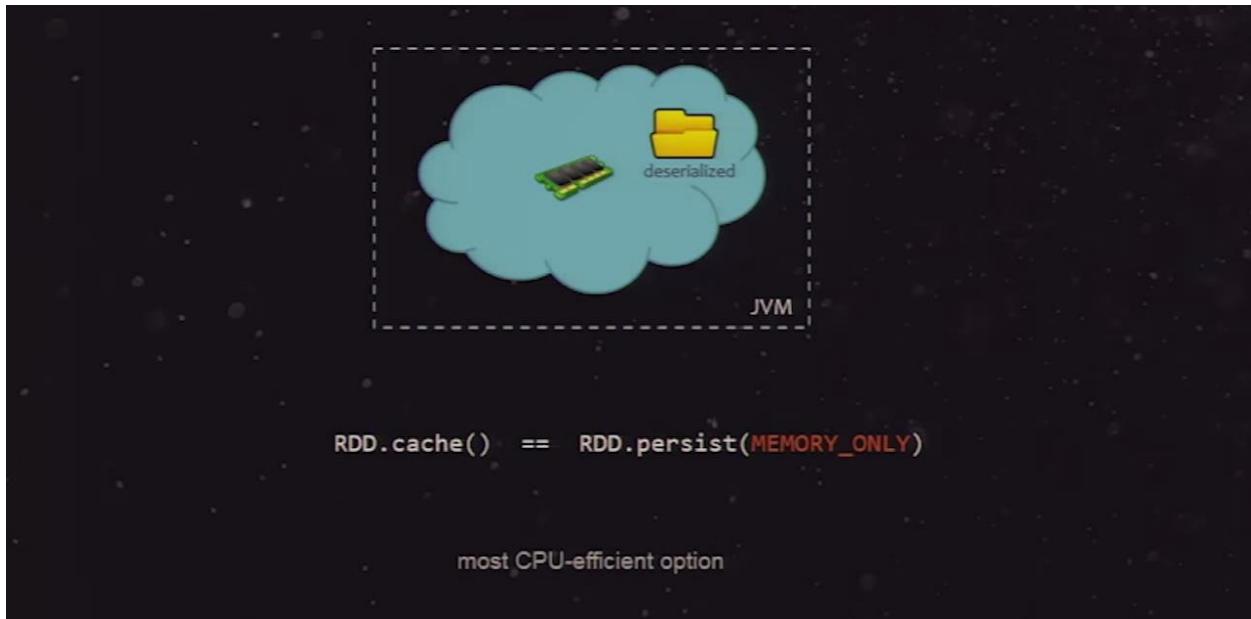
When we persist RDD in memory we can either persist to memory or memory+disk



cache() is just a shortcut of persist but when we do persist () we can pass lot more thing as argument

Memory Only

when we store RDD using cache or memory only it stores the RDD deserialized as java object inside the JVM, if certain partition of the RDD do not fit in memory then those partition will not be cached at all they are dropped not store on disk either and that RDD will be recomputed on the fly



Memory Only Serialized

We can also persist RDD with memory only serialized , it creates serialized java object , one byte array per partition, this is generally more space efficient than memory only (de-serialized) RDD by default it uses java serialization process to store RDD in memory. But it is little bit slow because cost of serializing object. It reduced the cost of GC because many individual records can be stored as a single serialized buffer



We can use kryo serializer instead which is faster than java serializer

Memory and Disk

In this case it's going to try store RDD in de-serialized form in the JVM memory and if the RDD partition don't fit in memory it stores it in disk in serialized form instead

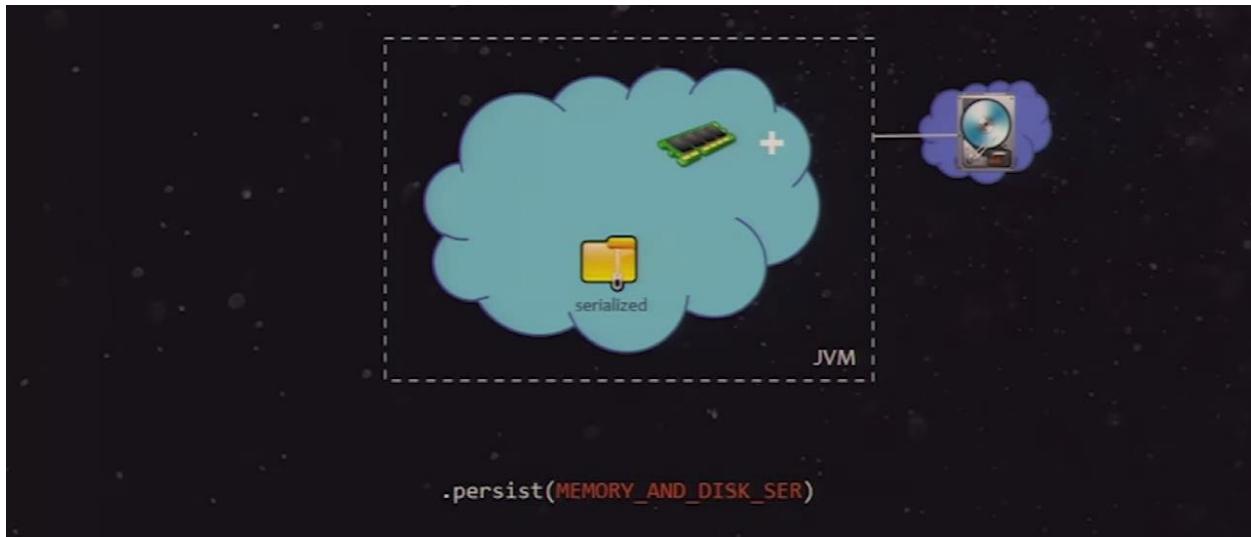


When it goes down to disk it will always be serialized



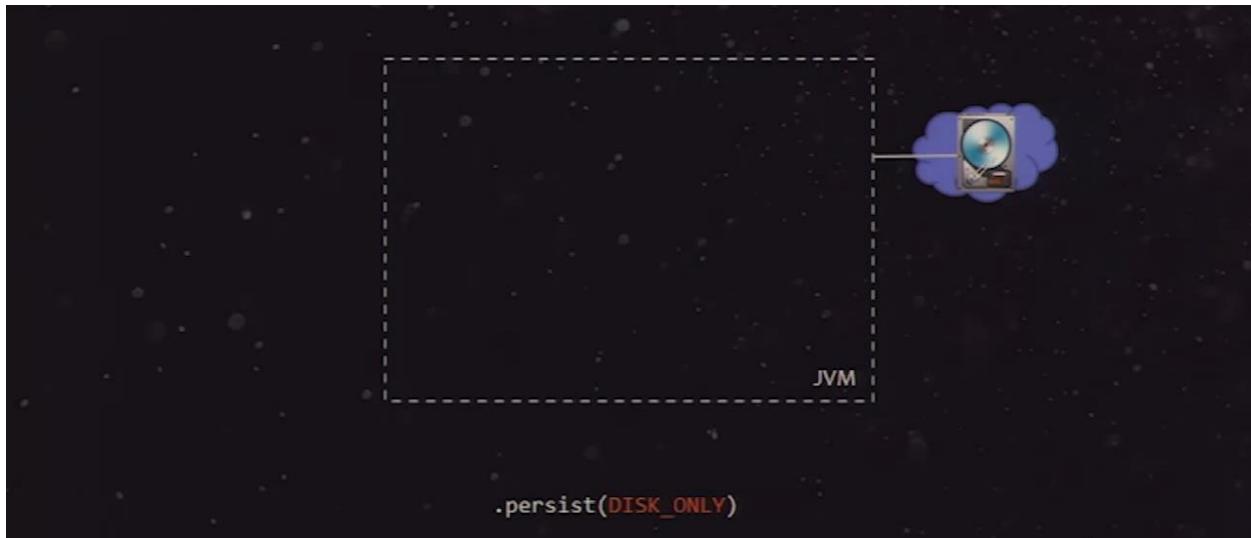
Memory and Disk Serialized

In this case RDD partition stores in serialization form on both memory and disk



Disk Only

RDD persist only on disk



Memory only 2

This stores RDD partitions as de-serialized java objects in two different JVMS, you can use this if your RDD is extremely costly to recreate and if you lose one partition of RDD. In general don't use this. In general spark work with replication factor 1. No need to create different replicas of RDD



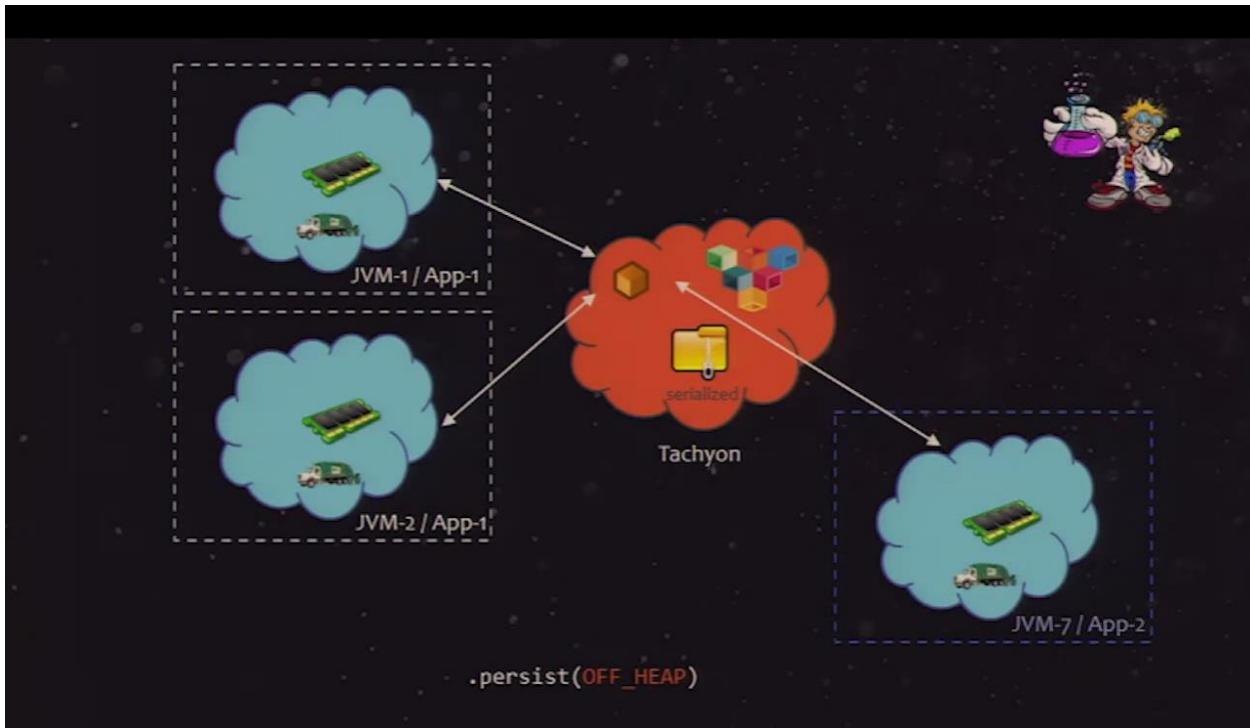
Memory and disk 2

In this Case it stored partition in both locations



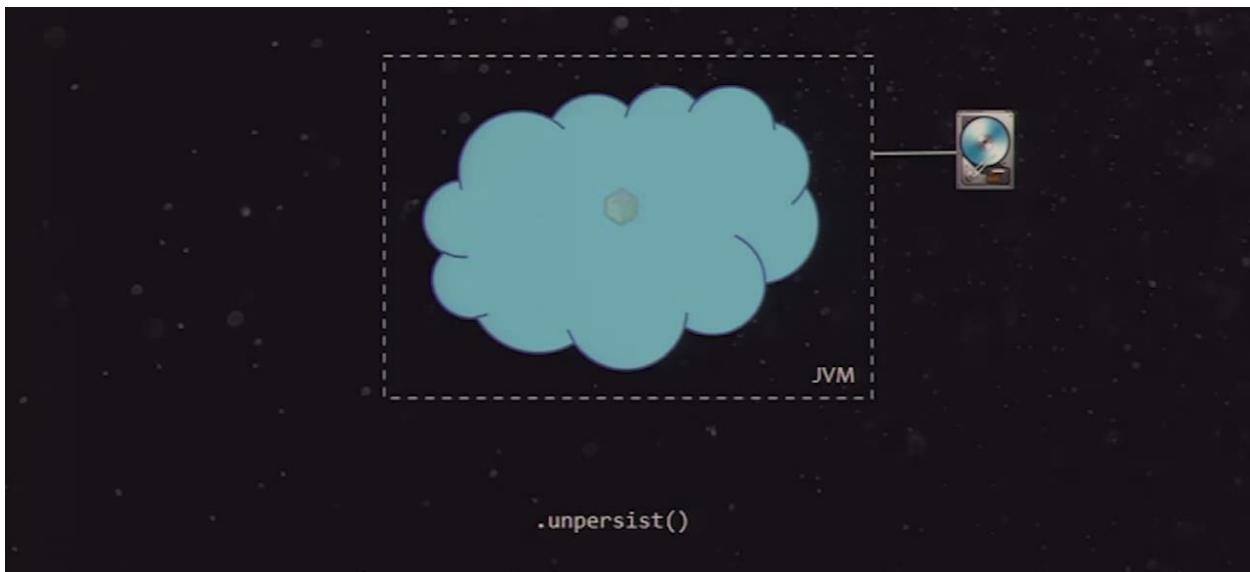
Off Heap

If you use OFF_HEAP, RDD will store in tachyon in serialized form, in this case RDD continue living in Tachyon even if executor JVM crashes

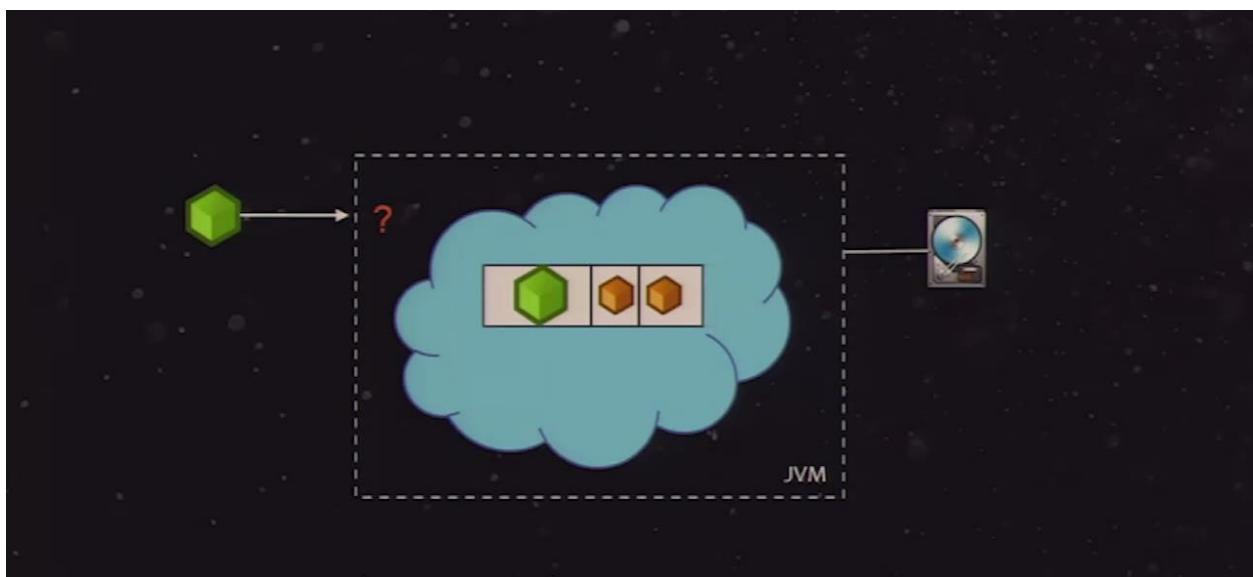
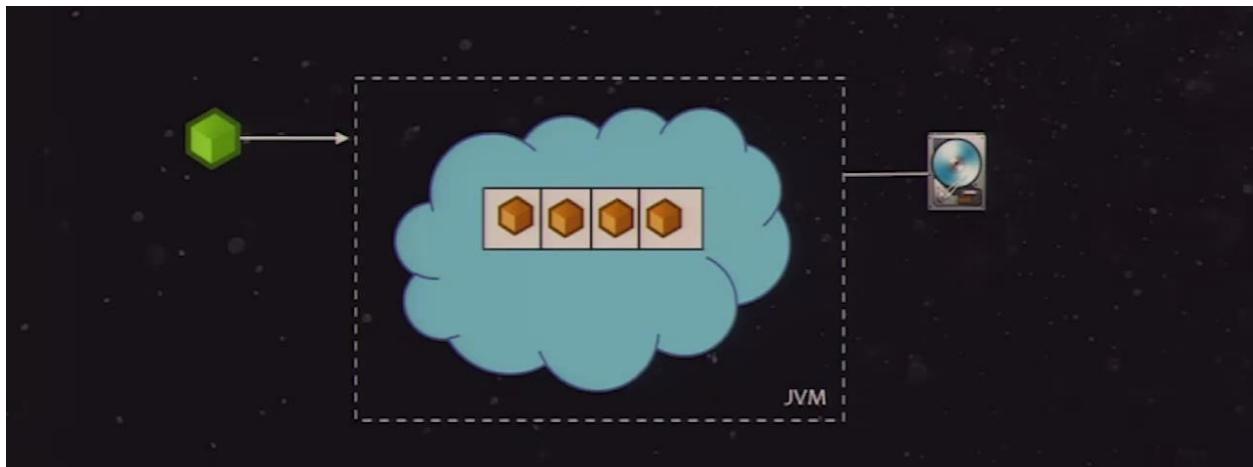


Unpersist

Remove forcefully out of memory

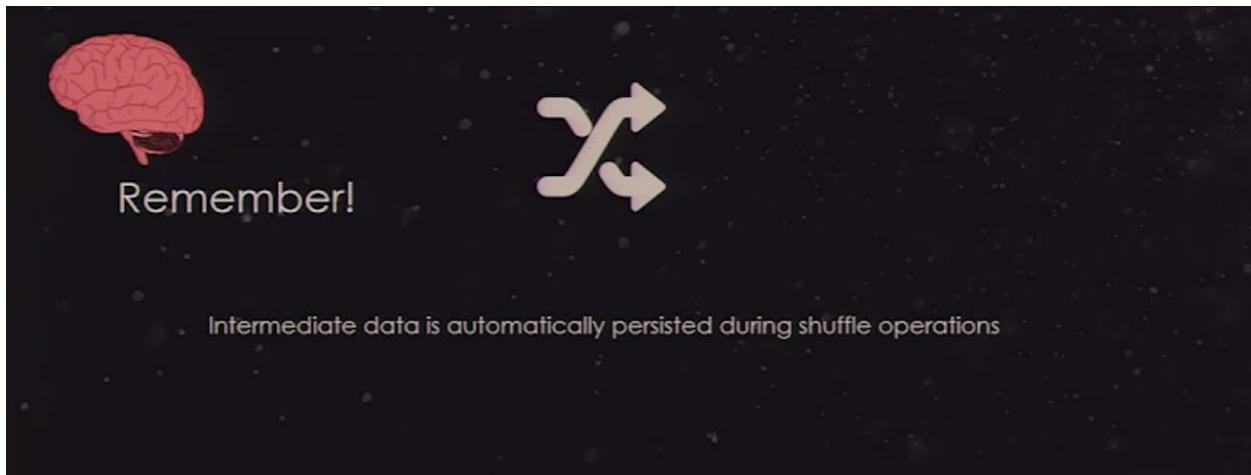


When executor JVM is full and you need to make room for a new partition then using least recently used (LRU) algo it removes some old partition and make room for new partition





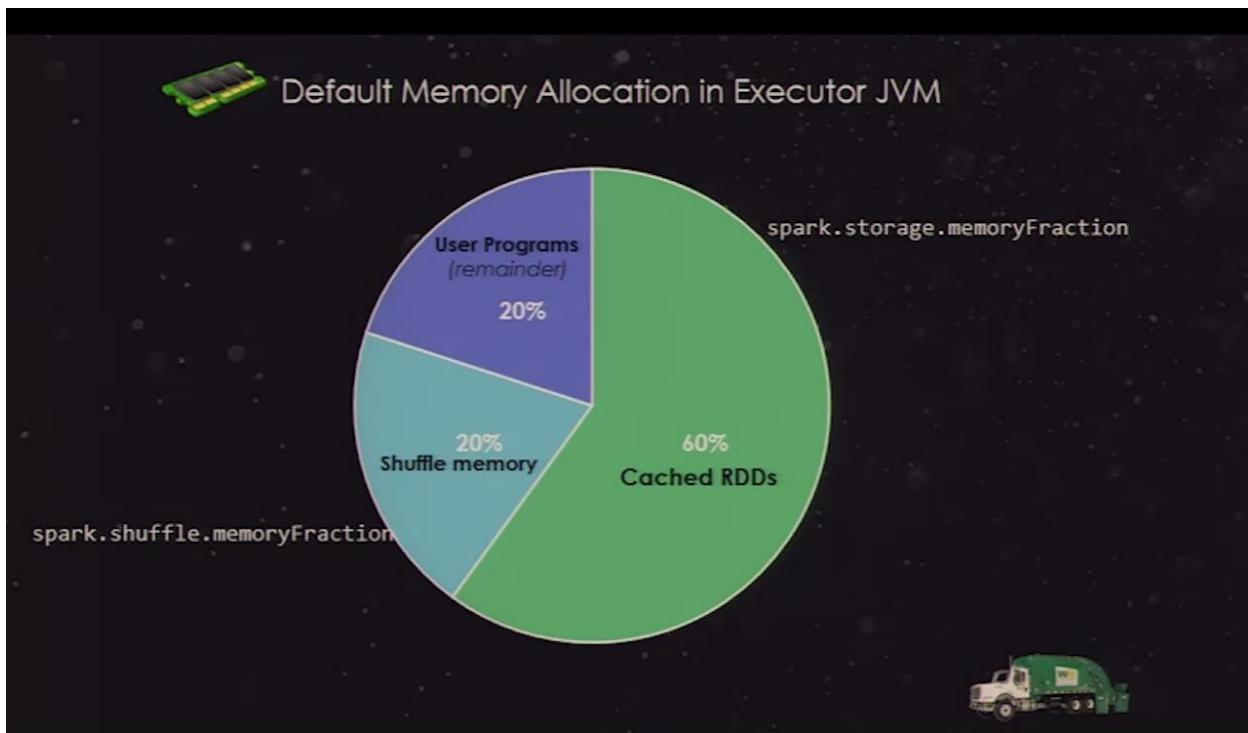
- If RDD fits in memory, choose `MEMORY_ONLY`
- If not, use `MEMORY_ONLY_SER` w/ fast serialization library
- Don't spill to disk unless functions that computed the datasets are very expensive or they filter a large amount of data.
(recomputing may be as fast as reading from disk)
- Use replicated storage levels sparingly and only if you want fast fault recovery (maybe to serve requests from a web app)



Imagine that you have long DAG of 200 RDDs and at the very bottom you call an action to materialized all 200 RDDS and get the action results and you never cached 200 RDDS at all and then you call exact same action again, the second time you call the action you might think that it's going to do all 200 transformation and read the data all over again from the original data source. **That is not true**, when we cache anything what happened is, from where you call the action spark will call up the DAG to see where the closest shuffle happen and when shuffle happen intermediate data spill to the local disk so it get the result much faster second time



PySpark: stored objects will always be serialized with Pickle library, so it does not matter whether you choose a serialized level.



MEMORY

Spark uses memory for:

RDD Storage: when you call `.persist()` or `.cache()`. Spark will limit the amount of memory used when caching to a certain fraction of the JVM's overall heap, set by `spark.storage.memoryFraction`

Shuffle and aggregation buffers: When performing shuffle operations, Spark will create intermediate buffers for storing shuffle output data. These buffers are used to store intermediate results of aggregations in addition to buffering data that is going to be directly output as part of the shuffle.

User code: Spark executes arbitrary user code, so user functions can themselves require substantial memory. For instance, if a user application allocates large arrays or other objects, these will contribute to overall memory usage. User code has access to everything "left" in the JVM heap after the space for RDD storage and shuffle storage are allocated.

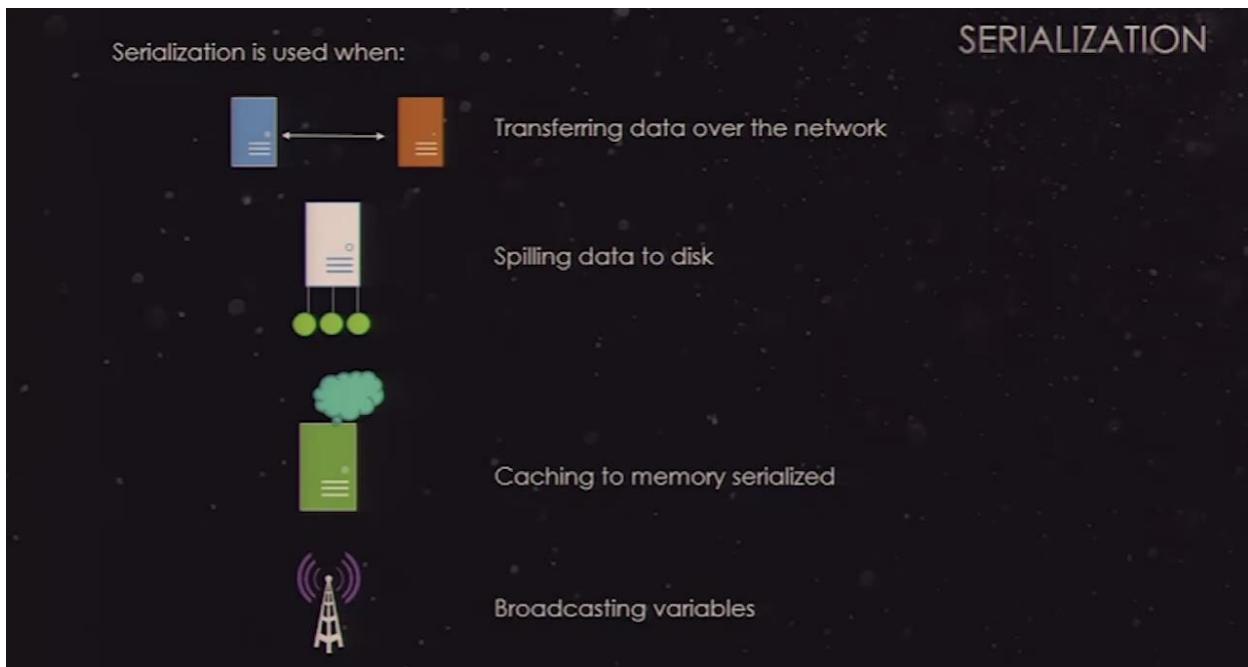
This log says partition 1 of RDD 0 consumed 717 KB memory

DETERMINING MEMORY CONSUMPTION

1. Create an RDD
2. Put it into cache
3. Look at `SparkContext` logs on the driver program or Spark UI

logs will tell you how much memory each partition is consuming, which you can aggregate to get the total size of the RDD

```
INFO BlockManagerMasterActor: Added rdd_0_1 in memory on mbk.local:50311 (size: 717.5 KB, free: 332.3 MB)
```



By default spark uses java serialization which is slow

The slide compares Java serialization and Kryo serialization. It features two logos: a steaming coffee cup for Java serialization and the Kryo logo for Kryo serialization. Below the logos, the text "Java serialization" and "vs." are on the left, and "Kryo serialization" is on the right. To the left of "vs." is a small blue square containing the letter "v".

Java serialization	vs.	Kryo serialization
<ul style="list-style-type: none">• Uses Java's <code>ObjectOutputStream</code> framework• Works with any class you create that implements <code>java.io.Serializable</code>• You can control the performance of serialization more closely by extending <code>java.io.Externalizable</code>• Flexible, but quite slow• Leads to large serialized formats for many classes		<ul style="list-style-type: none">• Recommended serialization for production apps• Use Kryo version 2 for speedy serialization (10x) and more compactness• Does not support all <code>Serializable</code> types• Requires you to register the classes you'll use in advance• If set, will be used for serializing shuffle data between nodes and also serializing RDDs to disk
<pre>conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")</pre>		

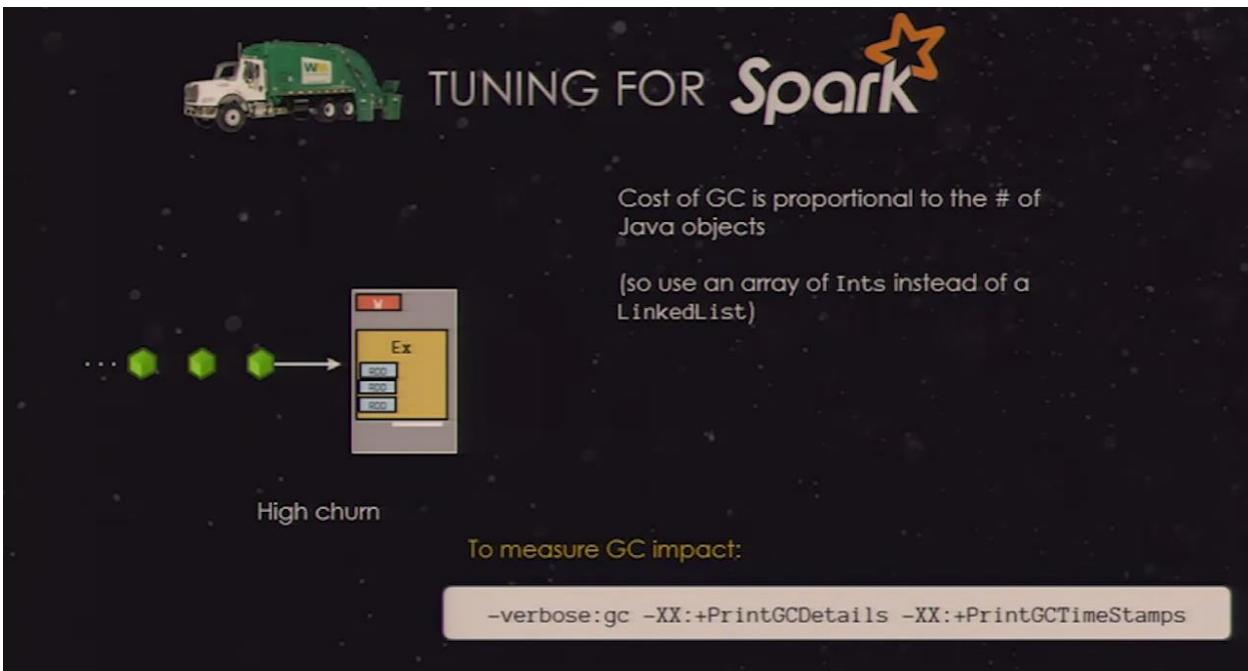
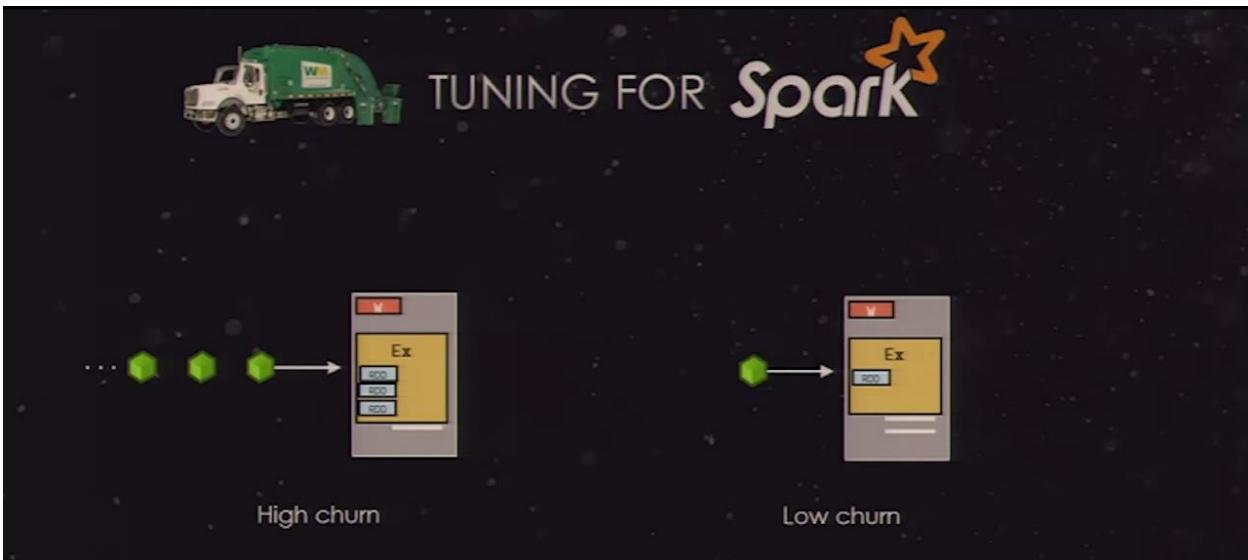
The slide shows the Kryo logo at the top. Below it, a bullet point states: "To register your own custom classes with Kryo, use the `registerKryoClasses` method:"

```
val conf = new SparkConf().setMaster(...).setAppName(...)  
conf.registerKryoClasses(Seq(classOf[MyClass1], classOf[MyClass2]))  
val sc = new SparkContext(conf)
```

Below the code block, another bullet point provides a note about memory usage:

- If your objects are large, you may need to increase `spark.kryoserializer.buffer.mb` config property
 - The default is 2, but this value needs to be large enough to hold the *largest* object you will serialize.

Tuning for Spark





TUNING

Parallel GC

-XX:+UseParallelGC
-XX:ParallelGCThreads=<#>

- Uses multiple threads to do young gen GC
- Will default to Serial on single core machines
- Aka "throughput collector"
- Good for when a lot of work is needed and long pauses are acceptable
- Use cases: batch processing

Parallel Old GC

-XX:+UseParallelOldGC

- Uses multiple threads to do both young gen and old gen GC
- Also a multithreading compacting collector
- HotSpot does compaction only in old gen

CMS GC

-XX:+UseConcMarkSweepGC
-XX:ParallelCMSThreads=<#>

- Concurrent Mark Sweep aka "Concurrent low pause collector"
- Tries to minimize pauses due to GC by doing most of the work concurrently with application threads
- Uses same algorithm on young gen as parallel collector
- Use cases:

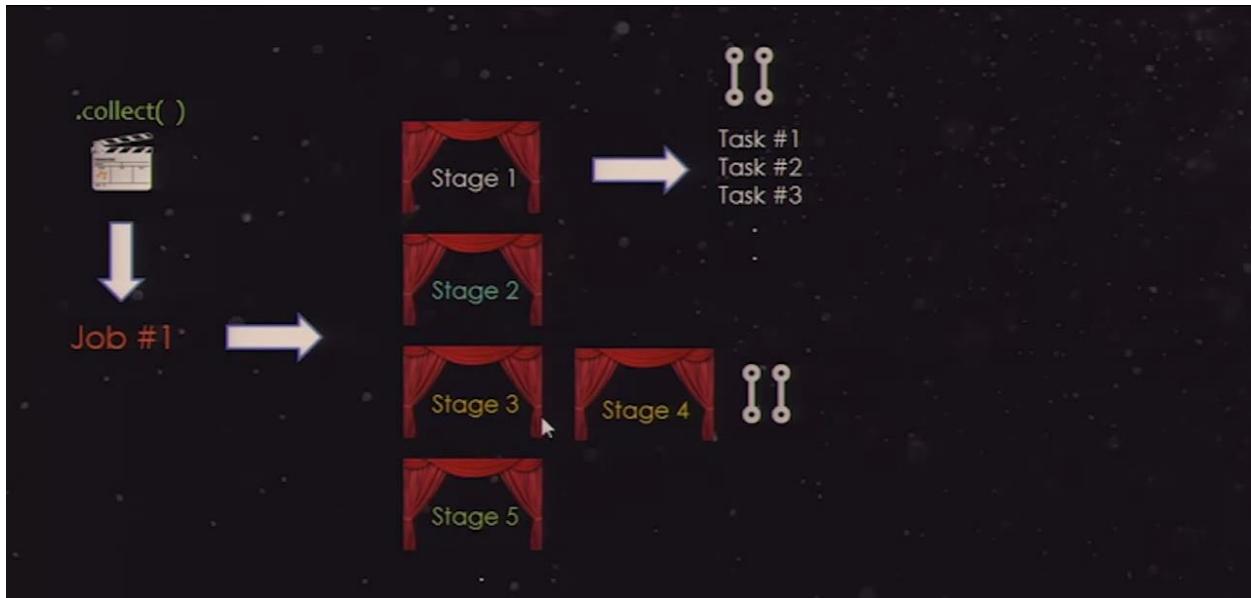
G1 GC

-XX:+UseG1GC

- Garbage First is available starting Java 7
- Designed to be long term replacement for CMS
- Is a parallel, concurrent and incrementally compacting low-pause GC

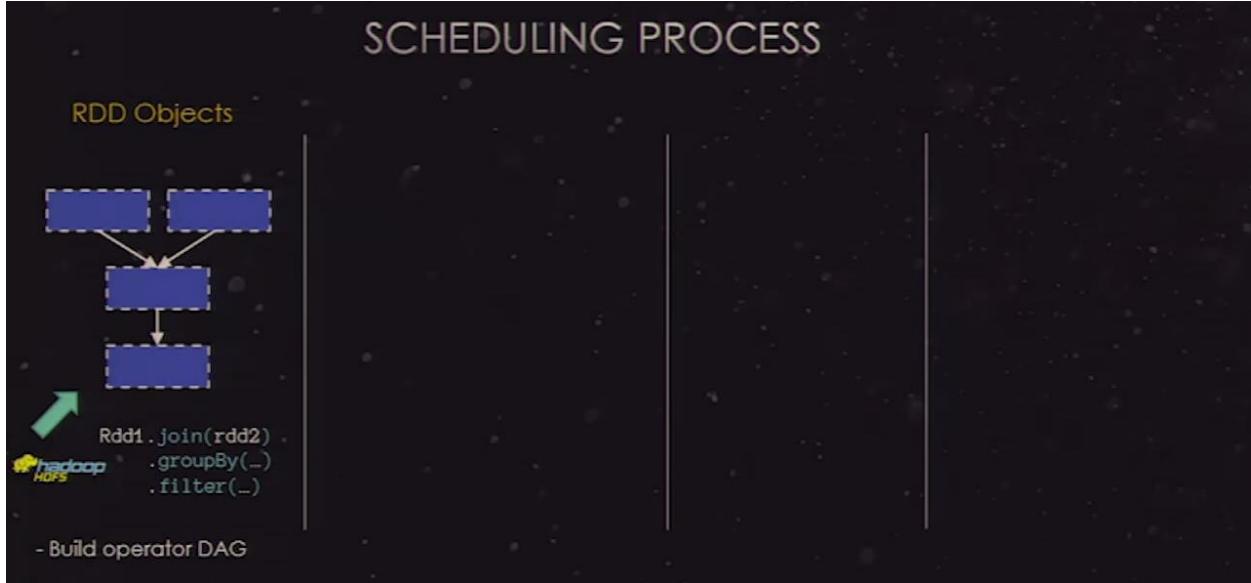
Jobs - Stack- Tasks

If you call an action like collect it trigger a Job to run, if you click on a job id in spark UI, you can see job is made of multiple stages, some stages runs parallel (stage3, 4), inside the stage there are multiple tasks which run in parallel, each task operating on one partition do some processing on parent RDD and emits a new child partition out.

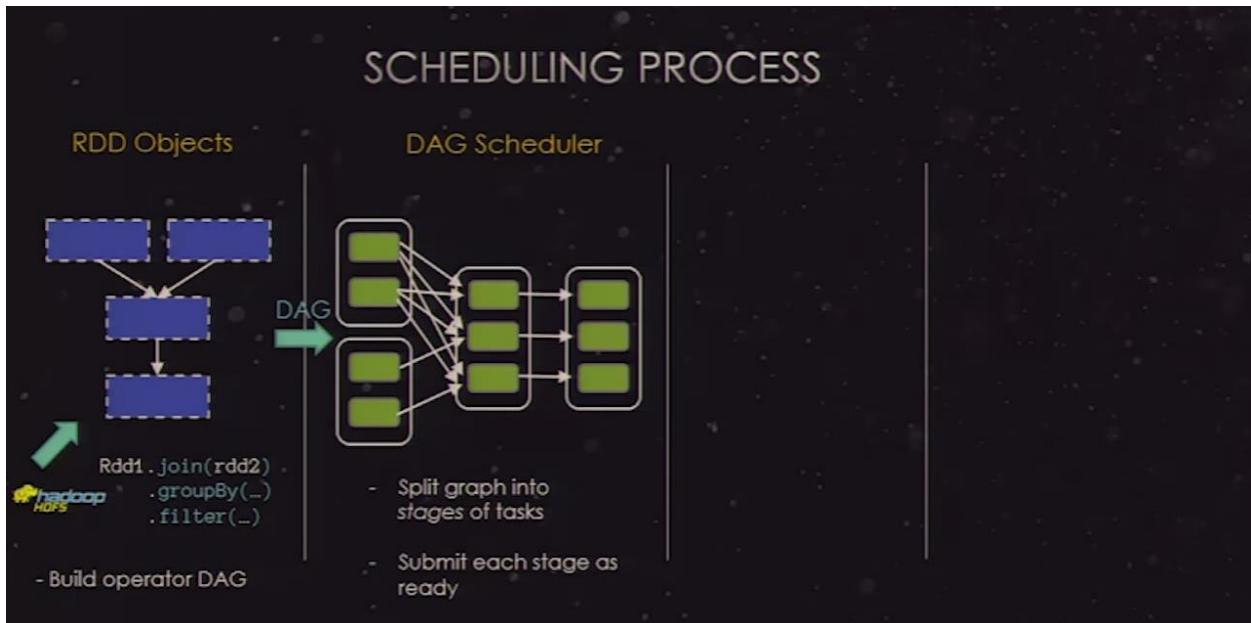


Scheduling process

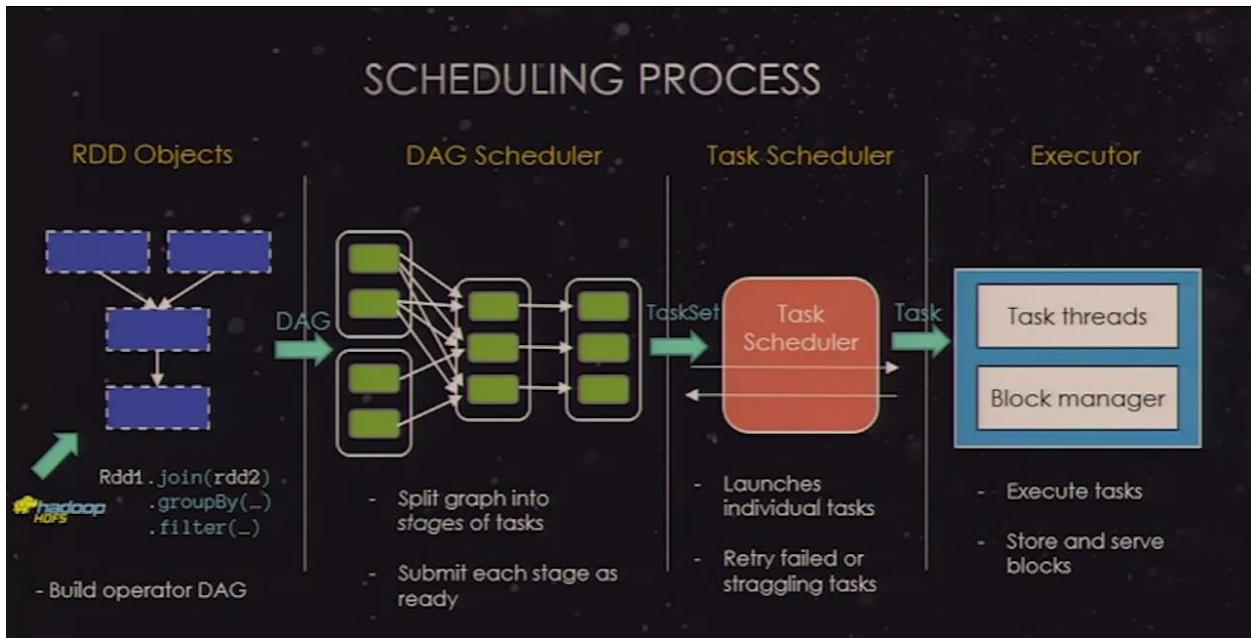
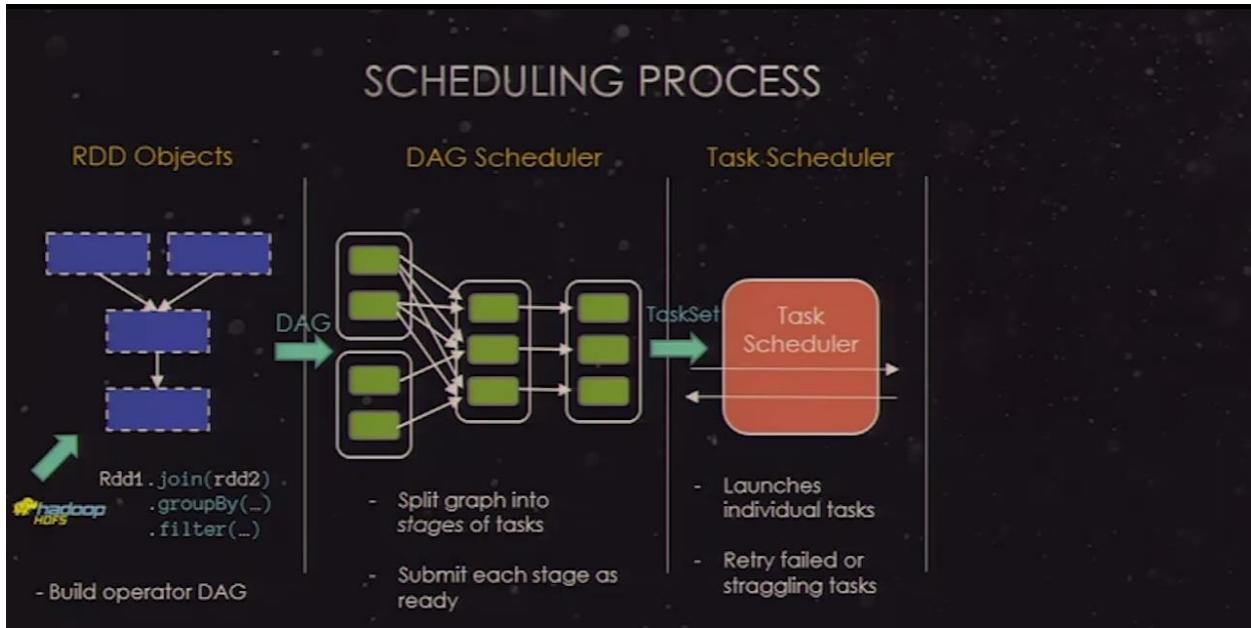
When you do lazy operation on RDD it will create DAG (Directed Acyclic Graph)



When we call action on RDD, spark driver JVM will figure out how to carve up that DAG into stage boundaries, basically creates different stages. That done by the DAG scheduler



Each stage will be submitted to task scheduler, task scheduler looks out of stage and schedule individual task to run on individual executor



LINEAGE

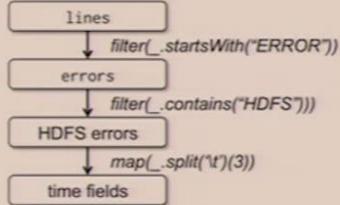


Figure 1: Lineage graph for the third query in our example.
Boxes represent RDDs and arrows represent transformations.

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(_.startsWith("ERROR"))  
errors.persist()
```

LINEAGE

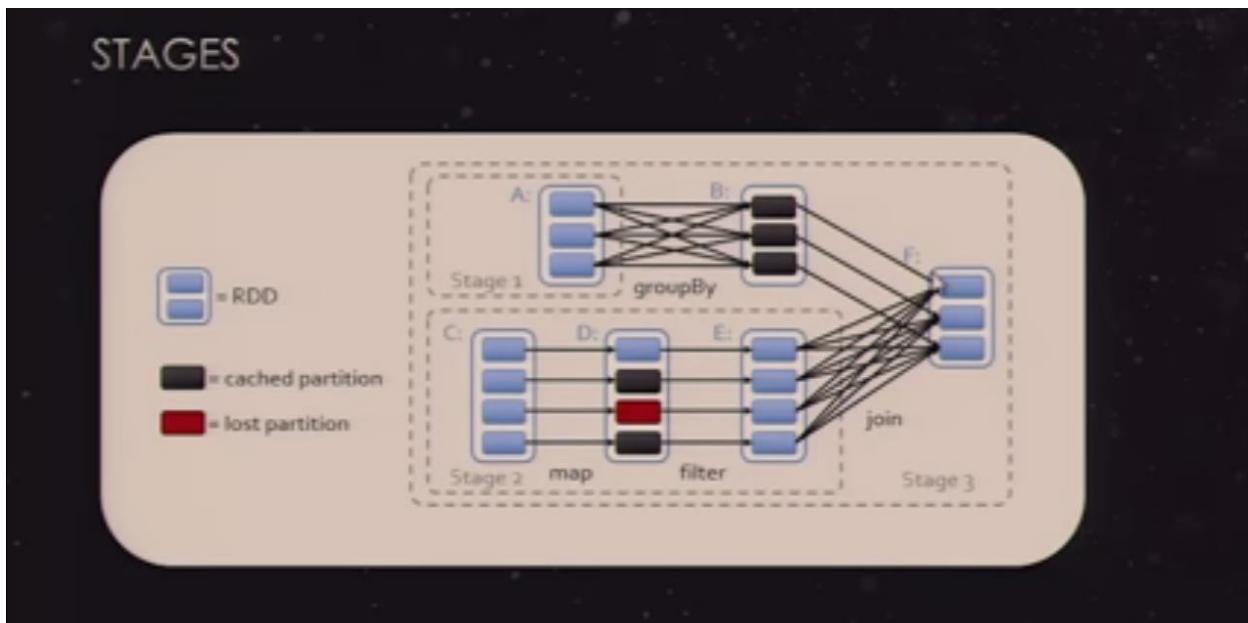
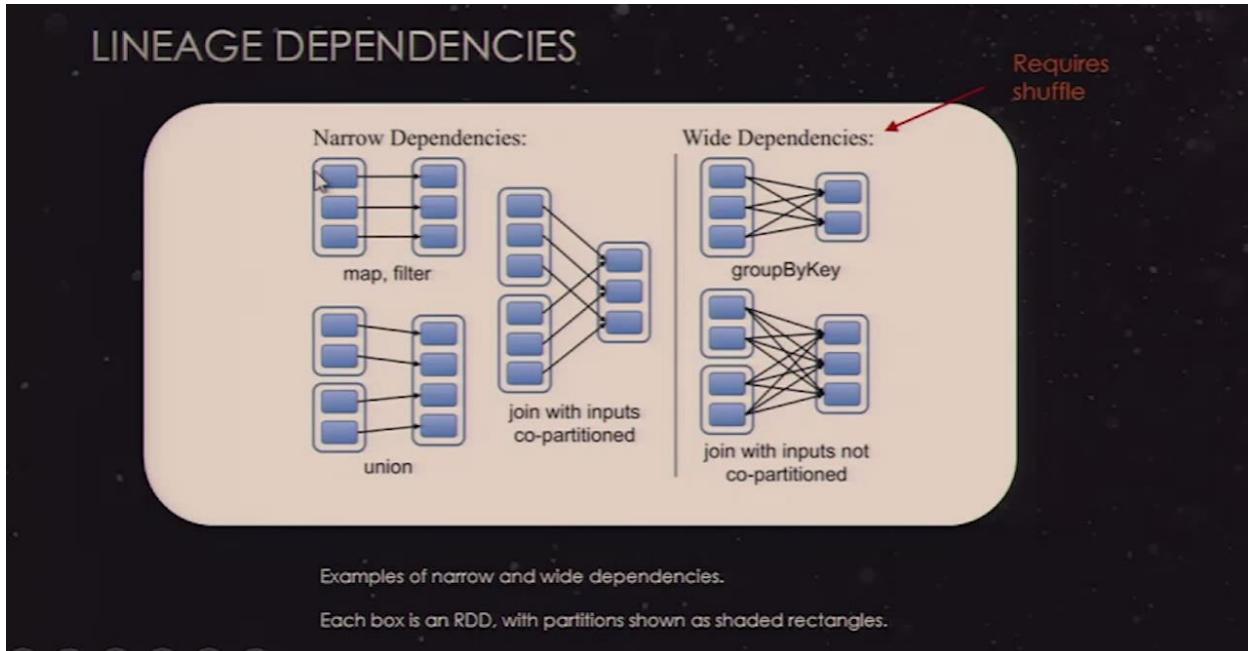
"One of the challenges in providing RDDs as an abstraction is choosing a representation for them that can track lineage across a wide range of transformations."

"The most interesting question in designing this interface is how to represent dependencies between RDDs."

"We found it both sufficient and useful to classify dependencies into two types:

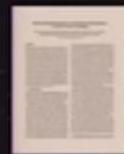
- narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD
- wide dependencies, where multiple child partitions may depend on it."

In narrow dependencies parent RDD is made up of just one child RDD while in wide parent made up of multiple Children



LINEAGE

Dependencies: Narrow vs Wide



"This distinction is useful for two reasons:

1] Narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a map followed by a filter on an element-by-element basis.

In contrast, wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce-like operation.

2] Recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution."

To display the lineage of an RDD, Spark provides a `toDebugString` method:

```
scala> input.toDebugString
res85: String =
(2) data.text MappedRDD[292] at textFile at <console>:13
 | data.text HadoopRDD[291] at textFile at <console>:13

scala> counts.toDebugString
res84: String =
(2) ShuffledRDD[296] at reduceByKey at <console>:17
 +- (2) MappedRDD[295] at map at <console>:17
   | FilteredRDD[294] at filter at <console>:15
   | MappedRDD[293] at map at <console>:15
   | data.text MappedRDD[292] at textFile at <console>:13
   | data.text HadoopRDD[291] at textFile at <console>:13
```



How do you know if a shuffle will be called on a Transformation?

- repartition, join, cogroup, and any of the *By or *ByKey transformations can result in shuffles
- If you declare a numPartitions parameter, it'll probably shuffle
- If a transformation constructs a shuffledRDD, it'll probably shuffle
- combineByKey calls a shuffle (so do other transformations like groupByKey, which actually end up calling combineByKey)

Note that repartition just calls coalesce w/ True:

```
RDD.scala    def repartition(numPartitions: Int)(implicit  
                      ord: Ordering[T] = null): RDD[T] = {  
    coalesce(numPartitions, shuffle = true)  
}
```



How do you know if a shuffle will be called on a Transformation?

Transformations that use "numPartitions" like distinct will probably shuffle:

```
def distinct(numPartitions: Int)(implicit ord: Ordering[T] =  
null): RDD[T] =  
    map(x => (x, null)).reduceByKey((x, y) => x,  
    numPartitions).map(_._1)
```

PERSERVES PARTITIONING

- An extra parameter you can pass a k/v transformation to let Spark know that you will not be messing with the keys at all
- All operations that shuffle data over network will benefit from partitioning
- Operations that benefit from partitioning:
cogroup, groupWith, join, leftOuterJoin, rightOuterJoin, groupByKey,
reduceByKey, combineByKey, lookup,

<https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala#L302>

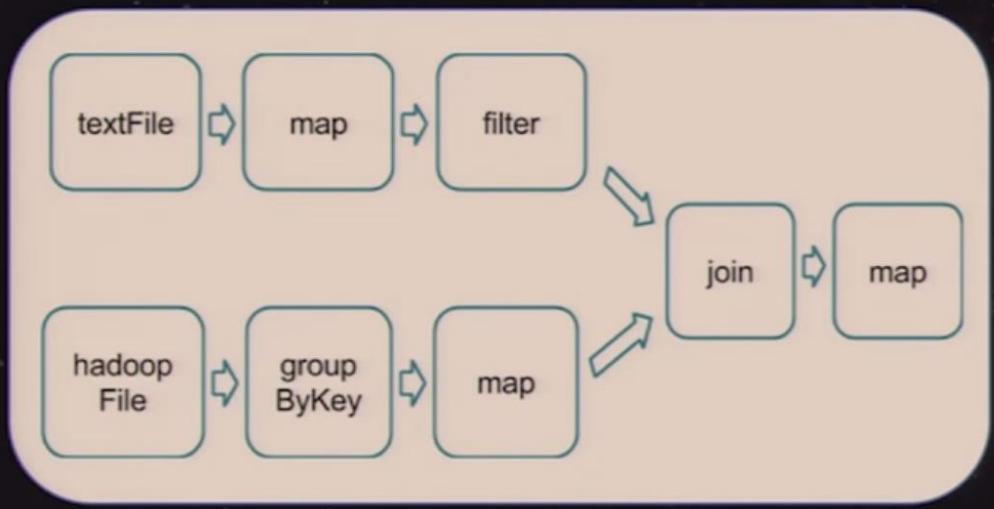
```
179  /**
180  * Return a new RDD containing only the elements that satisfy a predicate.
181  */
182  def filter(f: T => Boolean): RDD[T] = {
183    val cleanF = sc.clean(f)
184    new MapPartitionsRDD[T, T]{
185      this,
186      (context, pid, iter) => iter.filter(cleanF),
187      preservesPartitioning = true
188    }
189  }
```

How many Stages will this code require?

```
sc.textFile("someFile.txt").
  map(mapFunc).
  flatMap(flatMapFunc).
  filter(filterFunc).
  count()
```

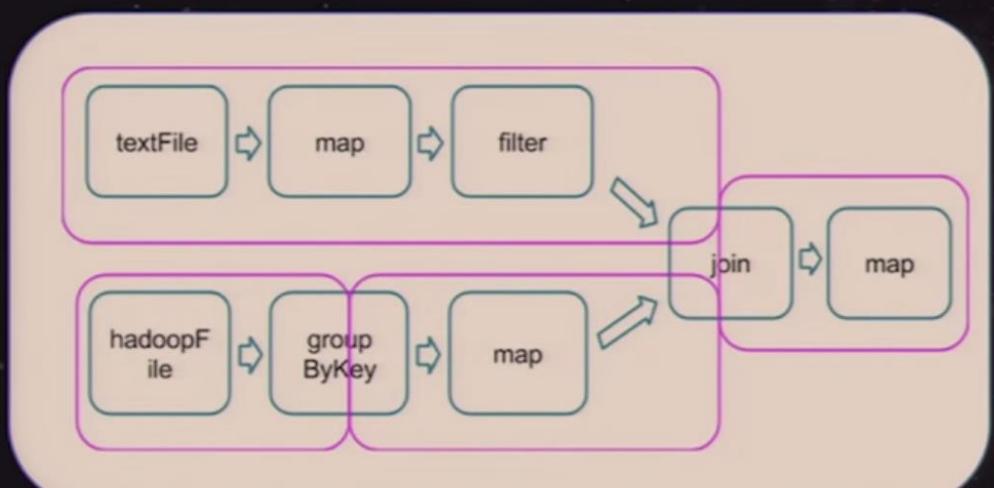
Map, Flatmap and filter all require narrow transformation so this will require only one stage

How many Stages will this DAG require?



3 stages, groupByKey causes a shuffle so group By key one of the stage boundaries, join also cause a shuffle as well

How many Stages will this DAG require?



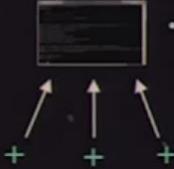
Source: Cloudera

Broadcast and Accumulators

USE CASES:



- Broadcast variables – Send a large read-only lookup table to all the nodes, or send a large feature vector in a ML algorithm to all nodes



- Accumulators – count events that occur during job execution for debugging purposes. Example: How many lines of the input file were blank? Or how many corrupt records were in the input dataset?

If one of the two datasets is small enough to fit in memory inside of a single executor then it can be loaded up in a hash table of a driver and broadcast to every executors and map transformation can look up the broadcast variables to do the lookups

BROADCAST VARIABLES



Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

For example, to give every node a copy of a large input dataset efficiently

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost



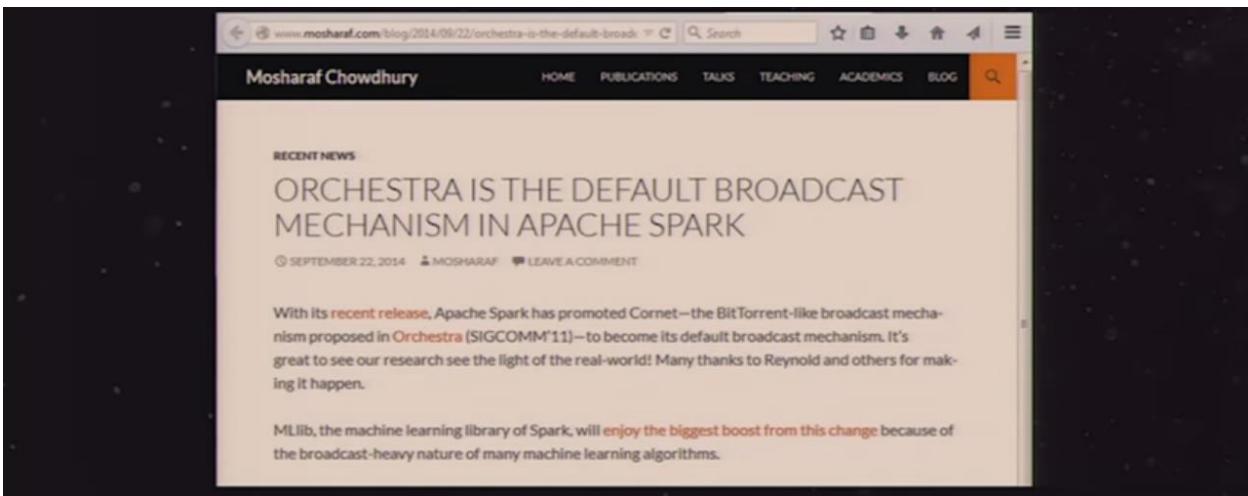
BROADCAST VARIABLES

Scala:

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Python:

```
broadcastVar = sc.broadcast(list(range(1, 4)))
broadcastVar.value
```



Mosharaf Chowdhury

HOME PUBLICATIONS TALKS TEACHING ACADEMICS BLOG

RECENT NEWS

ORCHESTRA IS THE DEFAULT BROADCAST MECHANISM IN APACHE SPARK

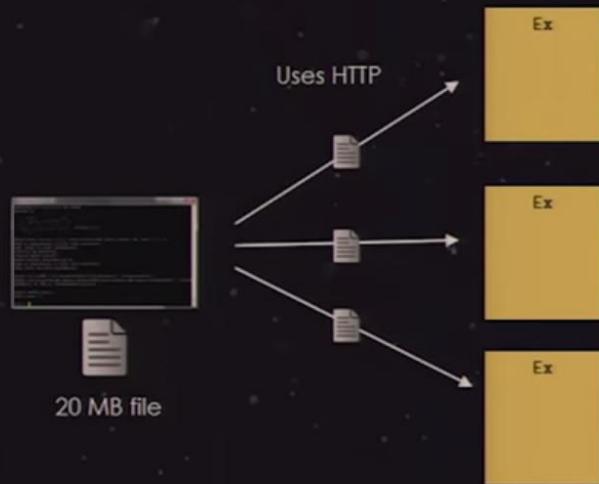
SEPTEMBER 22, 2014 MOSHARAF LEAVE A COMMENT

With its [recent release](#), Apache Spark has promoted `Cornet`—the BitTorrent-like broadcast mechanism proposed in `Orchestra` ([SIGCOMM’11](#))—to become its default broadcast mechanism. It’s great to see our research see the light of the real-world! Many thanks to Reynold and others for making it happen.

MLlib, the machine learning library of Spark, will enjoy the biggest boost from this [change](#) because of the broadcast-heavy nature of many machine learning algorithms.



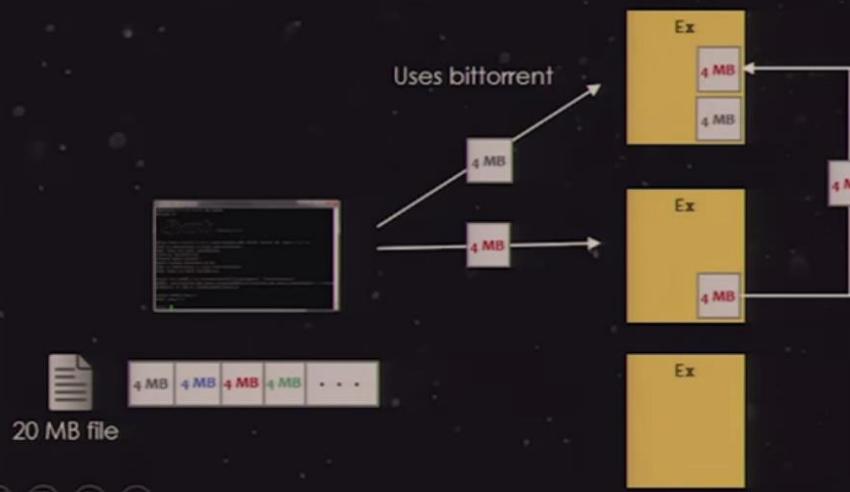
History: OLD TECHNIQUE FOR BROADCAST



All executors coordinate with each other to get the full 20 MB files

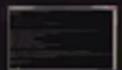
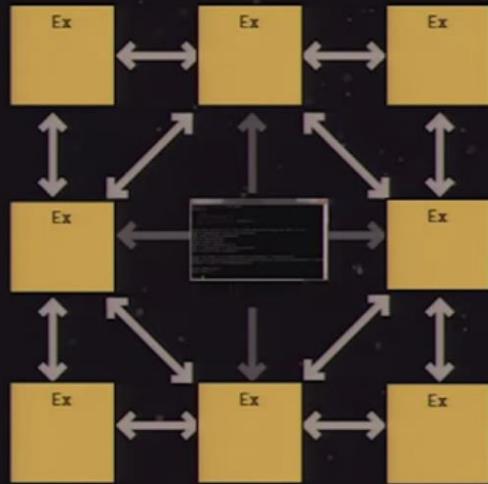


BITTORRENT TECHNIQUE FOR BROADCAST





BITTORENT TECHNIQUE FOR BROADCAST



ACCUMULATORS

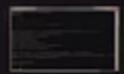


Accumulators are variables that can only be “added” to through an associative operation

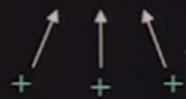
Used to implement counters and sums, efficiently in parallel

Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types

Only the driver program can read an accumulator’s value, not the tasks



ACCUMULATORS



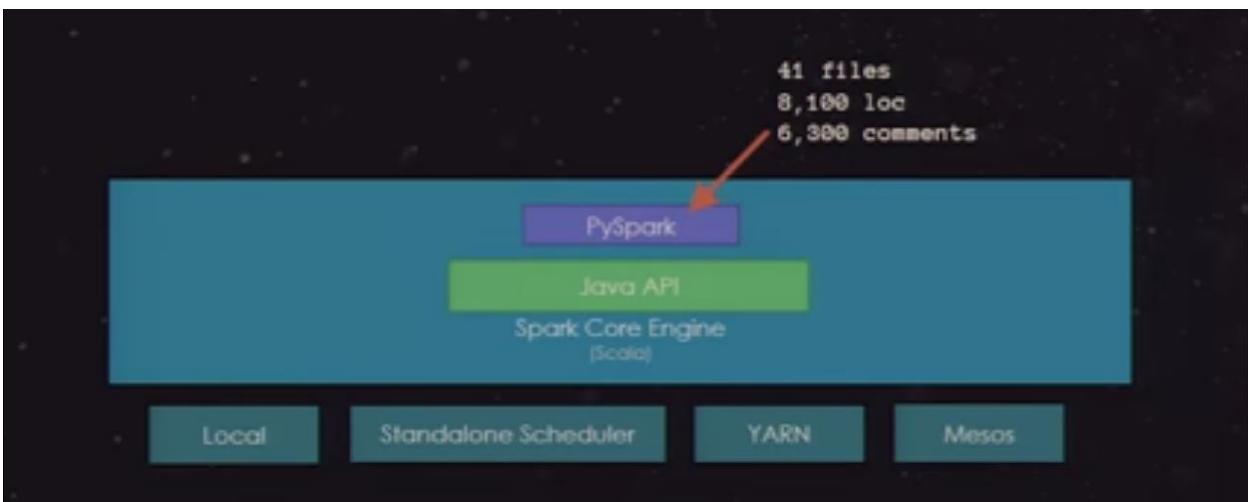
Scala:

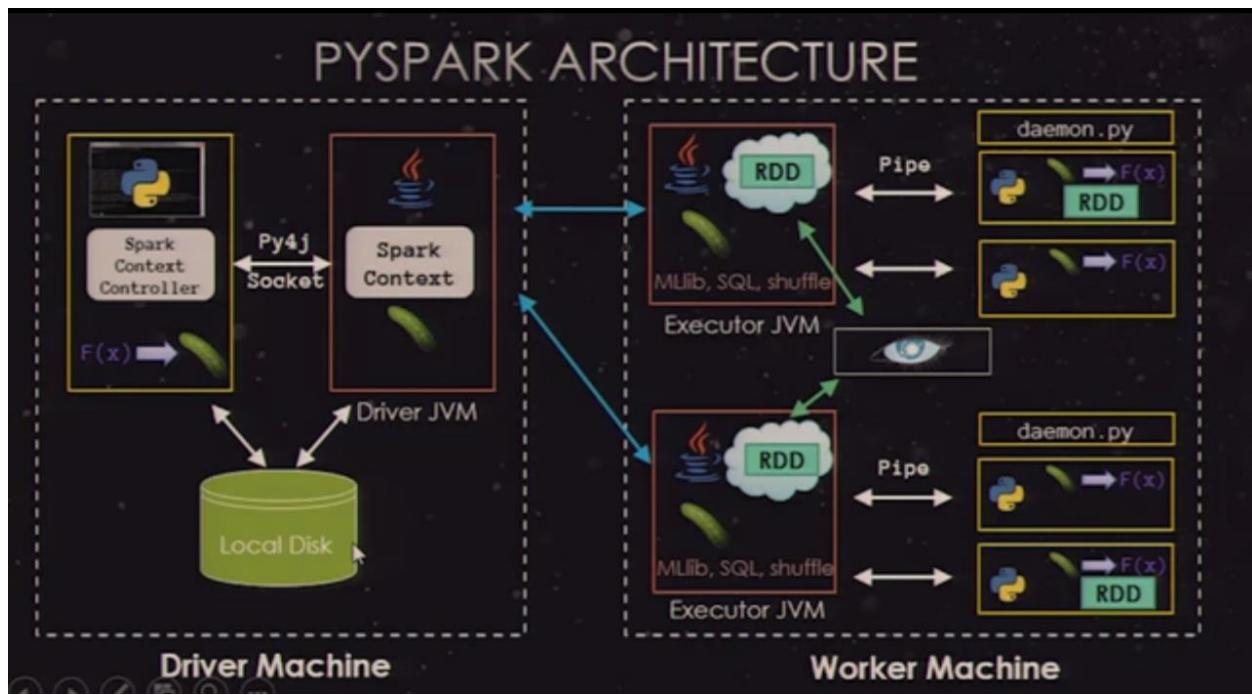
```
val accum = sc.accumulator(0)  
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)  
accum.value
```

Python:

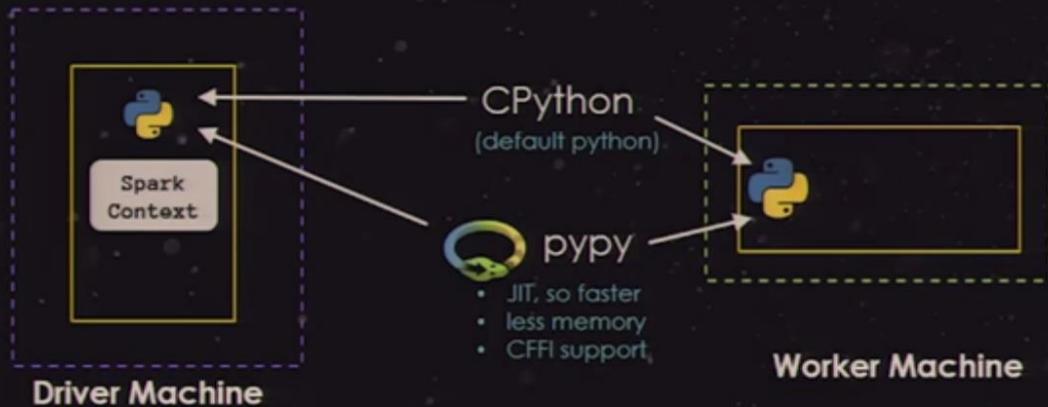
```
accum = sc.accumulator(0)  
rdd = sc.parallelize([1, 2, 3, 4])  
def f(x):  
    global accum  
    accum += x  
  
rdd.foreach(f)  
accum.value
```

Python API





Choose Your Python Implementation



```
$ 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:

Job	CPython 2.7	PyPy 2.3.1	Speed up
Word Count	41 s	15 s	2.7 x
Sort	46 s	44 s	1.05 x
Stats	174 s	3.6 s	48 x

Here is the code used for benchmark:

```
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>

spark.python.worker.memory 512m 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.

New Shuffle Implementation

100TB Daytona Sort Competition 2014

databricks

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	10Gbps network
Sort rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Sort rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min

Spark sorted the same data **3X faster** using **10X fewer machines** than Hadoop MR in 2013.

All the sorting took place on disk (HDFS) without using Spark's in-memory cache!

More info:

<http://sortbenchmark.org>

<http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html>

Work by Databricks engineers: Reynold Xin, Parviz Deyhim, Xiangrui Meng, Ali Ghodsi, Matei Zaharia

WHY SORTING?

- Stresses "shuffle" which underpins everything from SQL to MLlib
- Sorting is challenging b/c there is no reduction in data
- Sort 100 TB = 500 TB disk I/O and 200 TB network

Engineering Investment in Spark:

- Sort-based shuffle (SPARK-2045)
- Netty native network transport (SPARK-2468)
- External shuffle service (SPARK-3796)

Clever Application level Techniques:

- GC and cache friendly memory layout
- Pipelining

TECHNIQUE USED FOR 100 TB SORT



EC2: i2.8xlarge

(206 workers)

- 32 slots per machine
- 6,592 slots total

- Intel Xeon CPU E5 2670 @ 2.5 GHz w/ 32 cores
- 244 GB of RAM
- 8 x 800 GB SSD and RAID 0 setup formatted with /ext4
- ~9.5 Gbps (1.1 GBps) bandwidth between 2 random nodes

- Each record: 100 bytes (10 byte key & 90 byte value)
- OpenJDK 1.7
- HDFS 2.4.1 w/ short circuit local reads enabled
- Apache Spark 1.2.0
- Speculative Execution off
- Increased Locality Wait to infinite
- Compression turned off for input, output & network
- Used Unsafe to put all the data off-heap and managed it manually (i.e. never triggered the GC)



=

groupByKey

sortByKey

reduceByKey

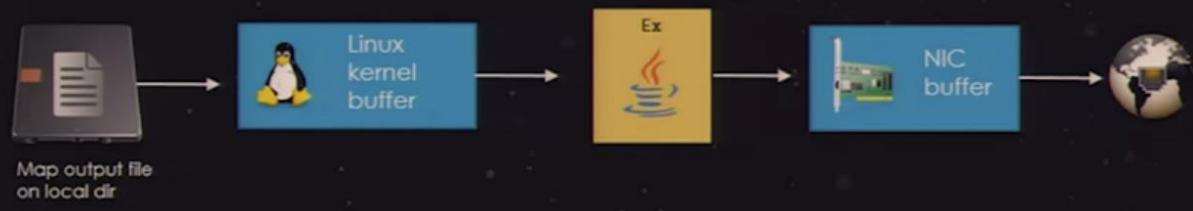
```
spark.shuffle.spill=false
```

(Affects reducer side and keeps all the data in memory)



OLD TECHNIQUE FOR SERVING MAP OUTPUT FILES

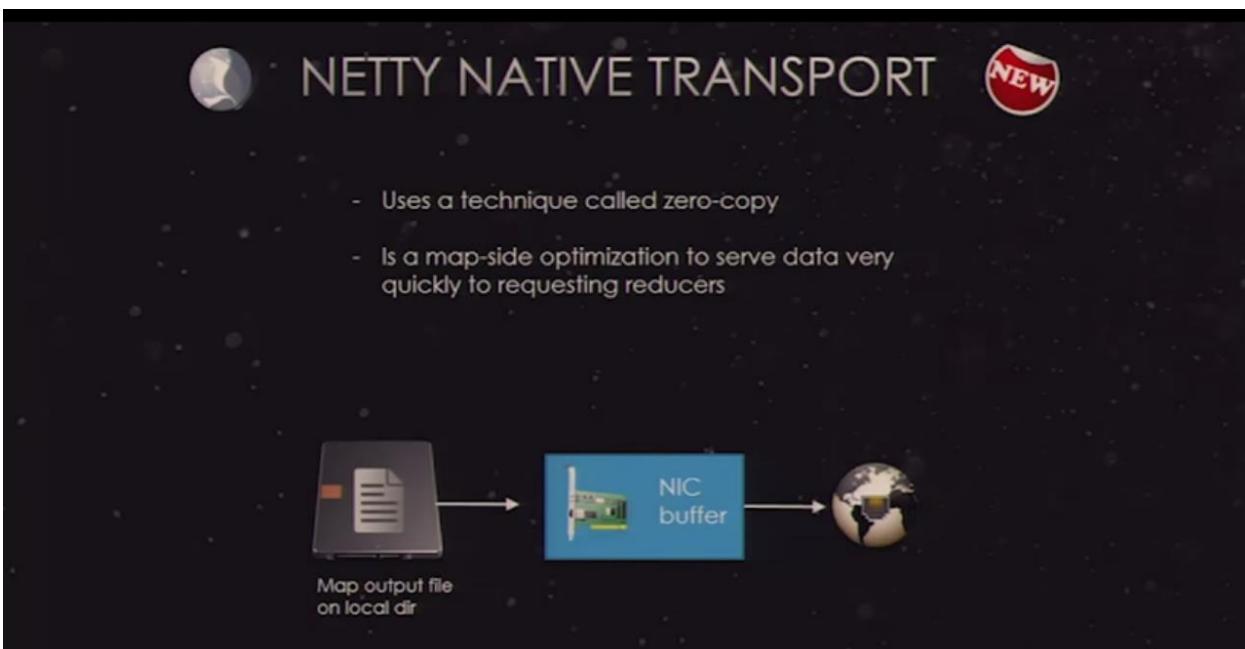
- Was slow because it had to copy the data 3 times



NETTY NATIVE TRANSPORT



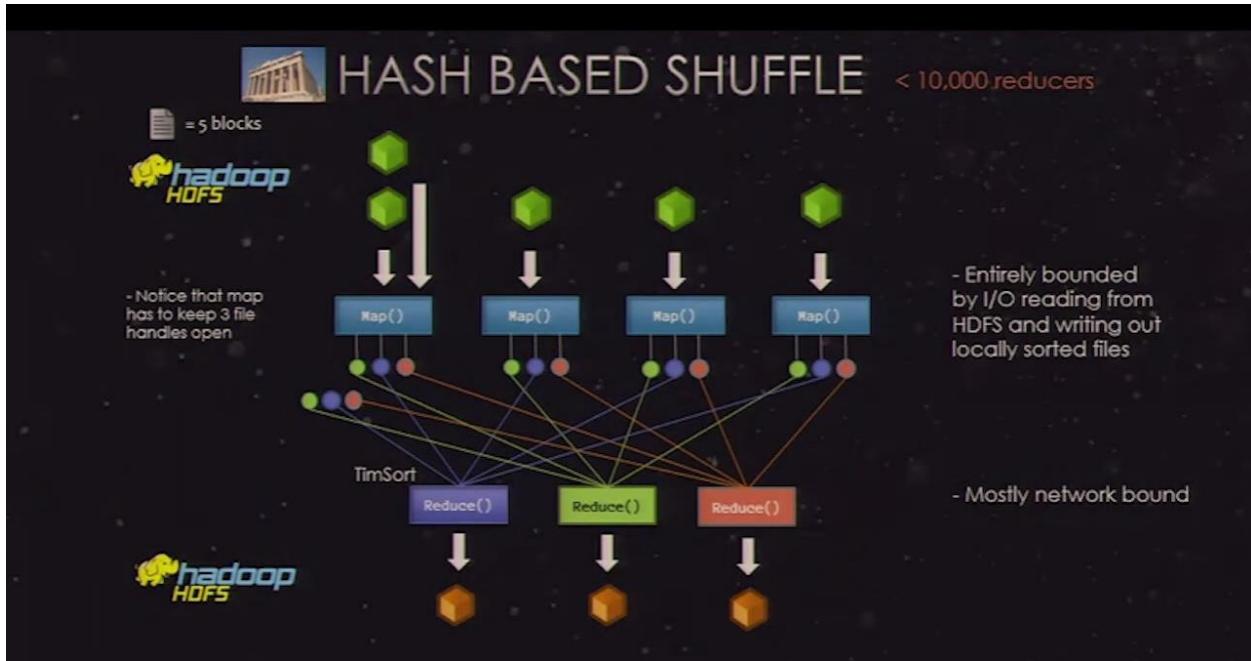
- Uses a technique called zero-copy
- Is a map-side optimization to serve data very quickly to requesting reducers



Hash Based Shuffle

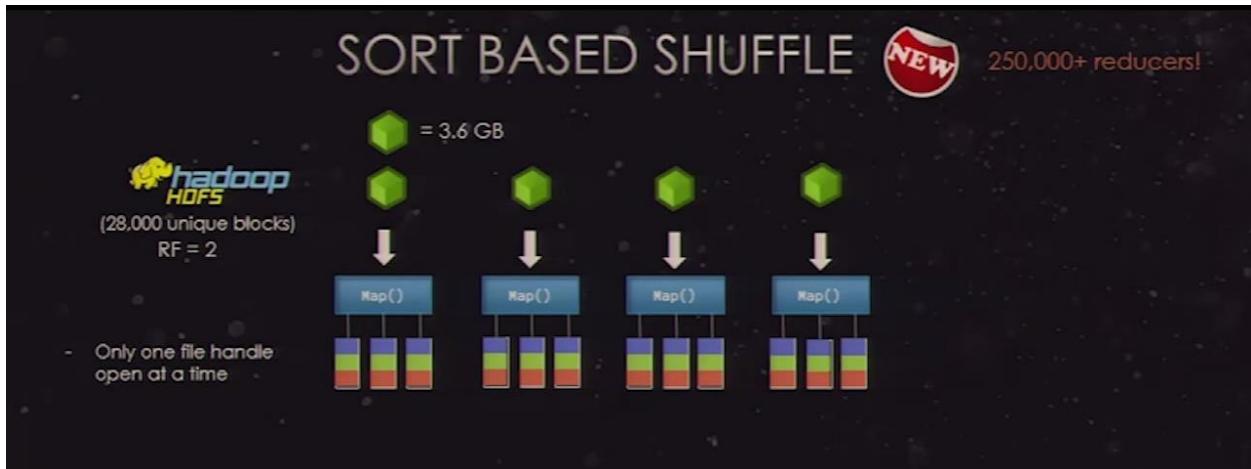
Main issue in below example is map side has to open three file handles open, more number of file handles open, buffer management become costly

Reduce phase is totally network bound as it need to pull the data over network

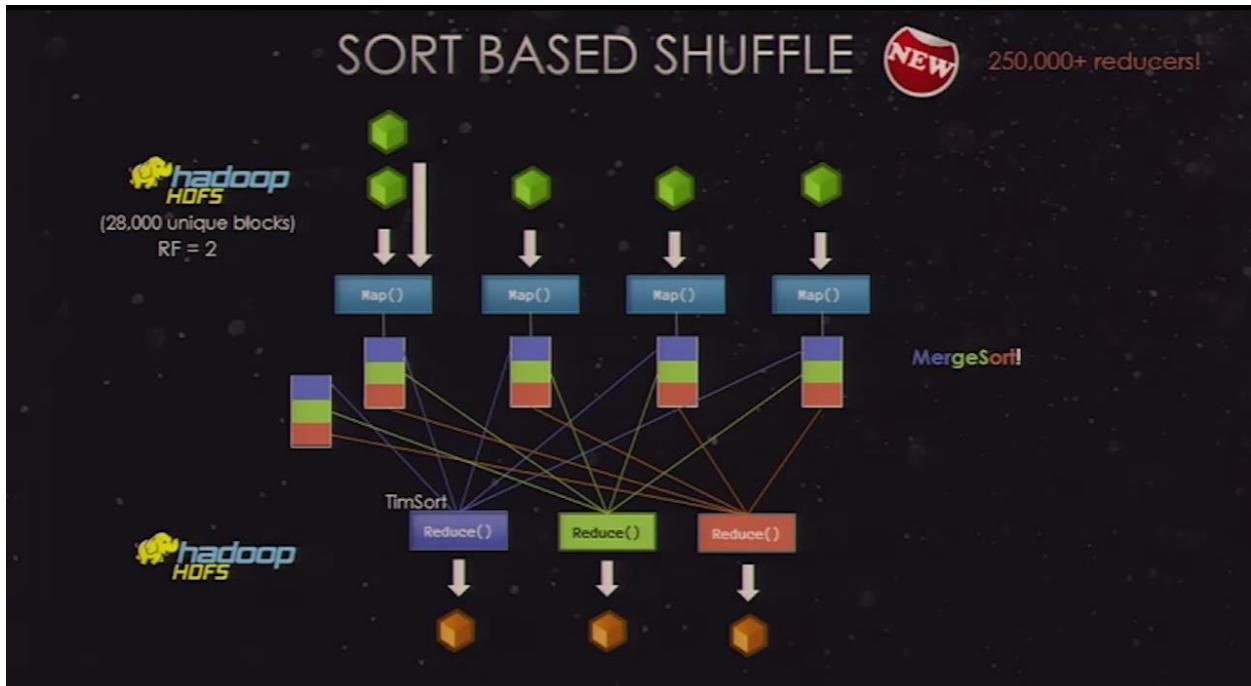


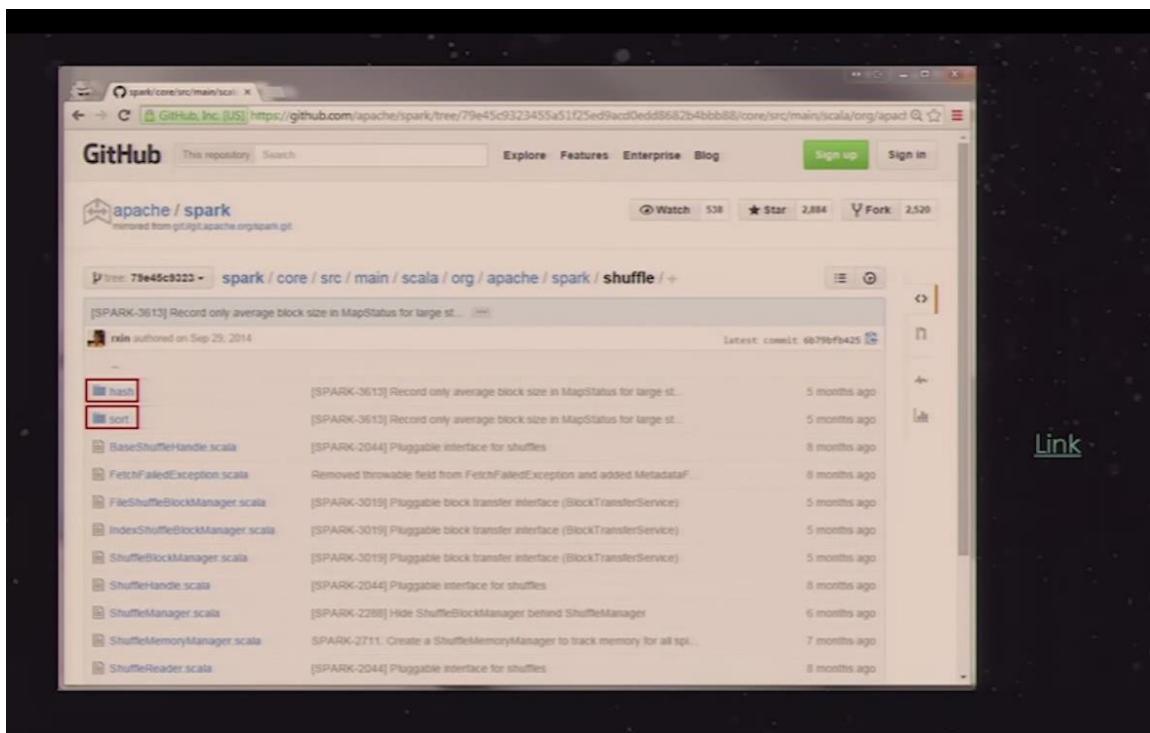
Sort Based Shuffle

Only open one file handle at a time and keep sorted all the data

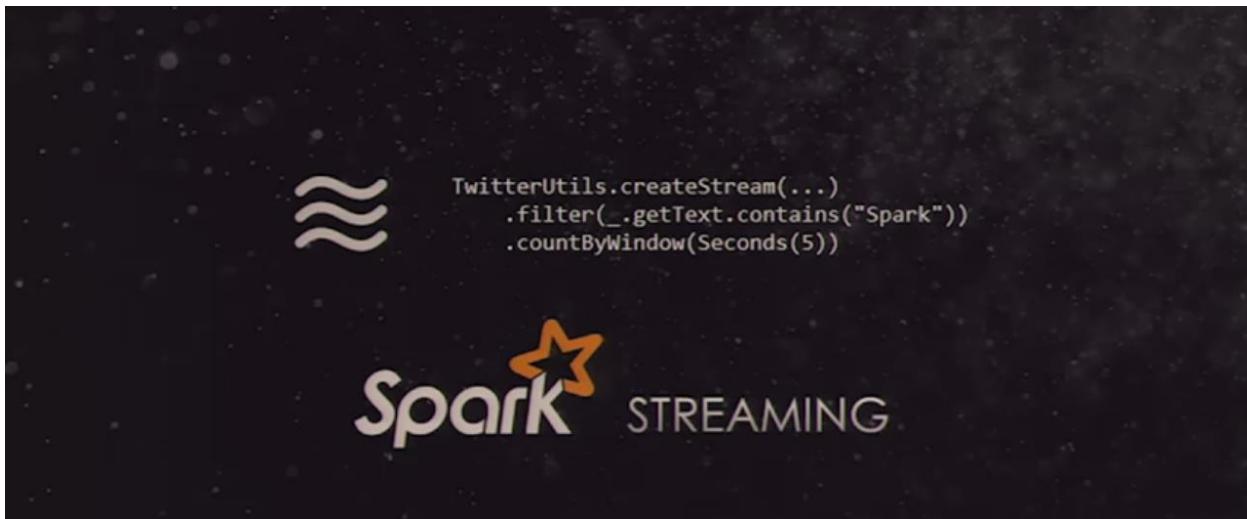


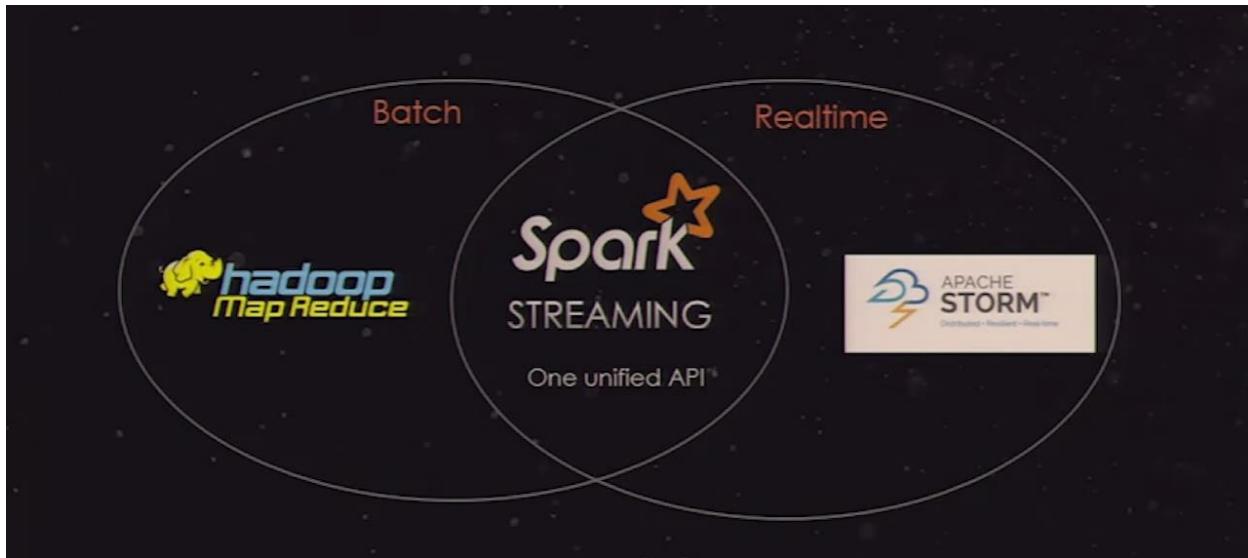
Later do the merge sort to combine all the map files in one large file





Spark Streaming



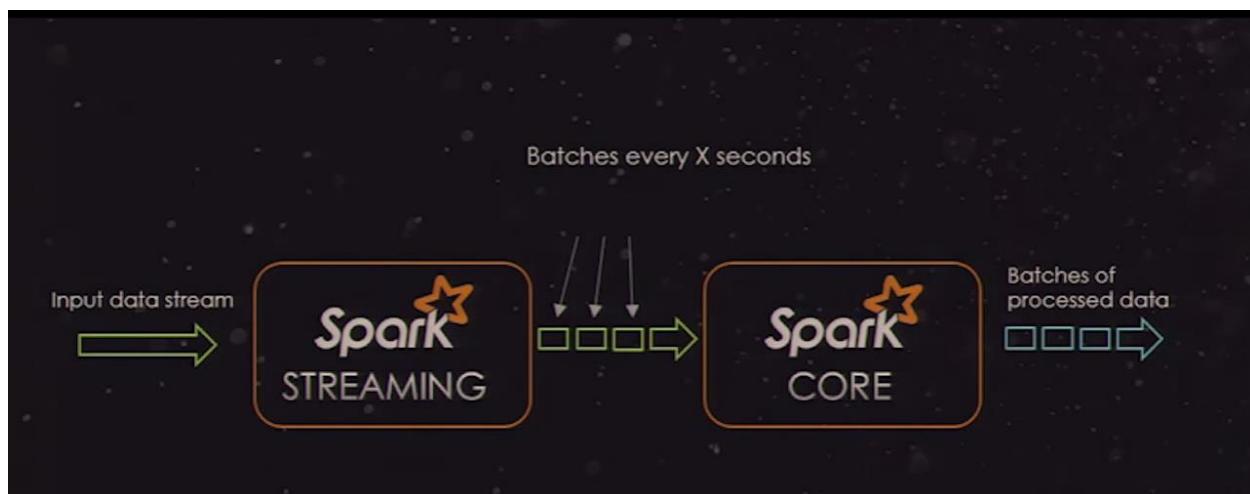
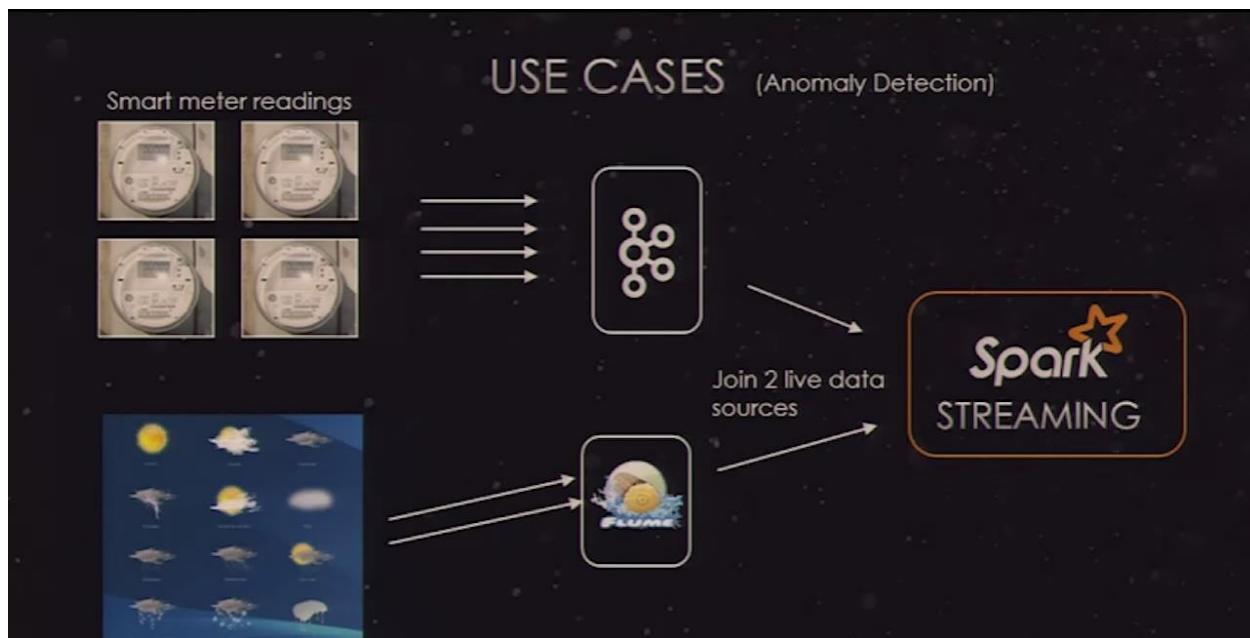
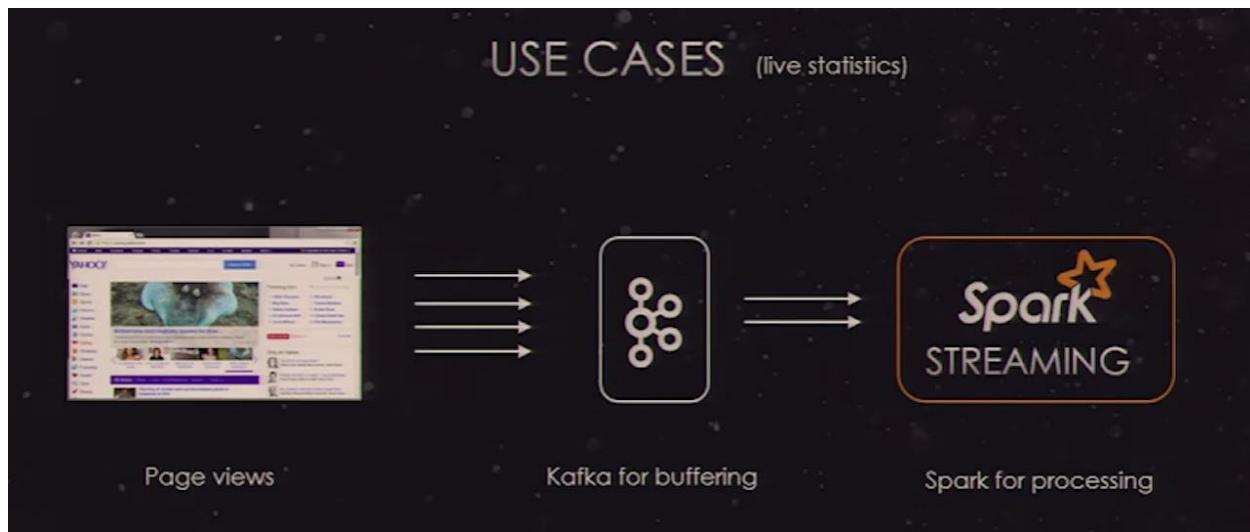


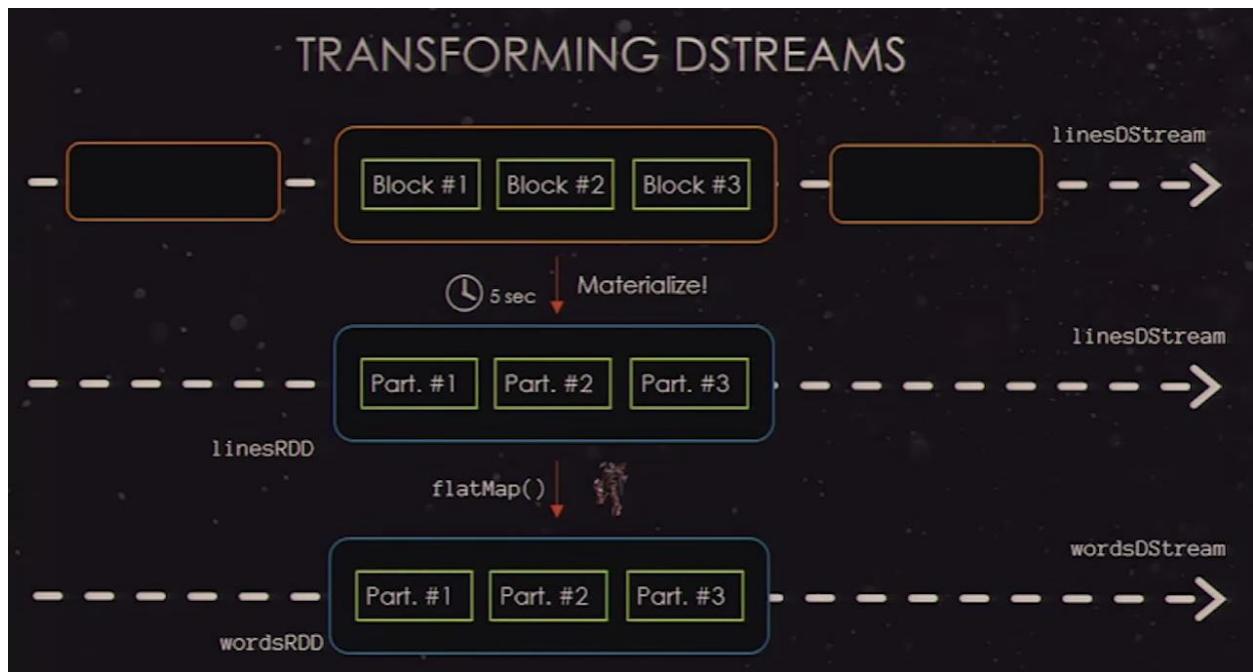
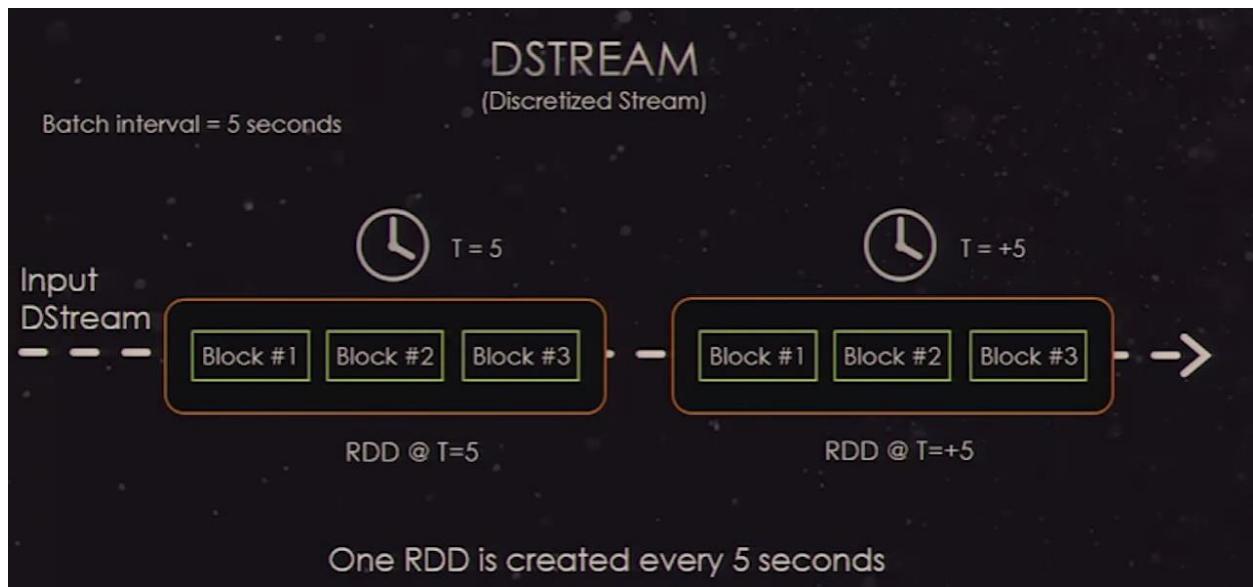


Tathagata Das (TD)

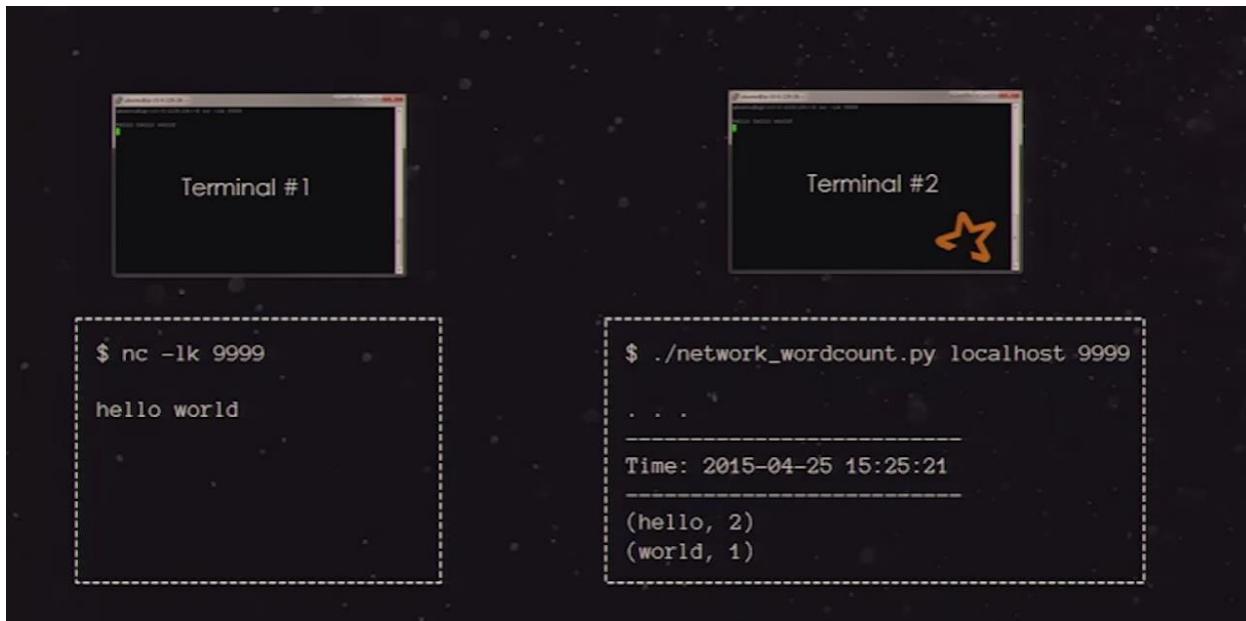
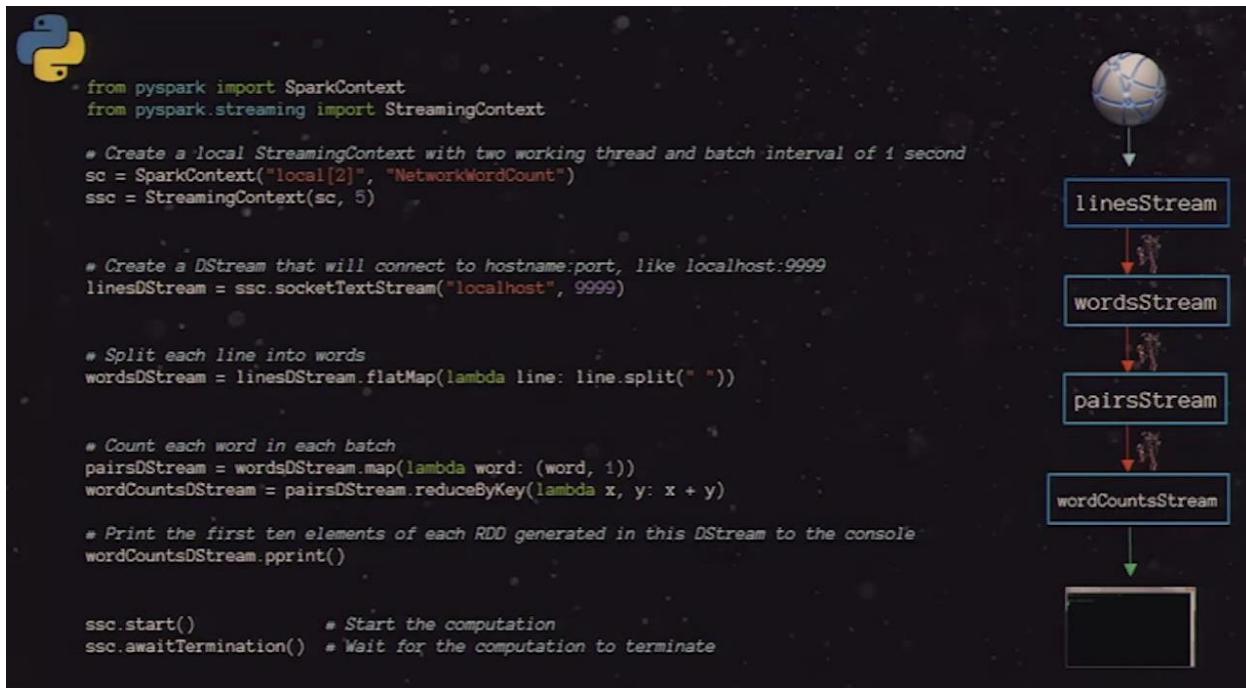
- Lead developer of Spark Streaming + Committer on Apache Spark core
- Helped re-write Spark Core internals in 2012 to make it 10x faster to support Streaming use cases
- On leave from UC Berkeley PhD program
- Ex: Intern @ Amazon, Intern @ Conviva, Research Assistant @ Microsoft Research India
- 1 guy; does not scale

- Scales to 100s of nodes
- Batch sizes as small at half a second
- Processing latency as low as 1 second
- Exactly-once semantics no matter what fails



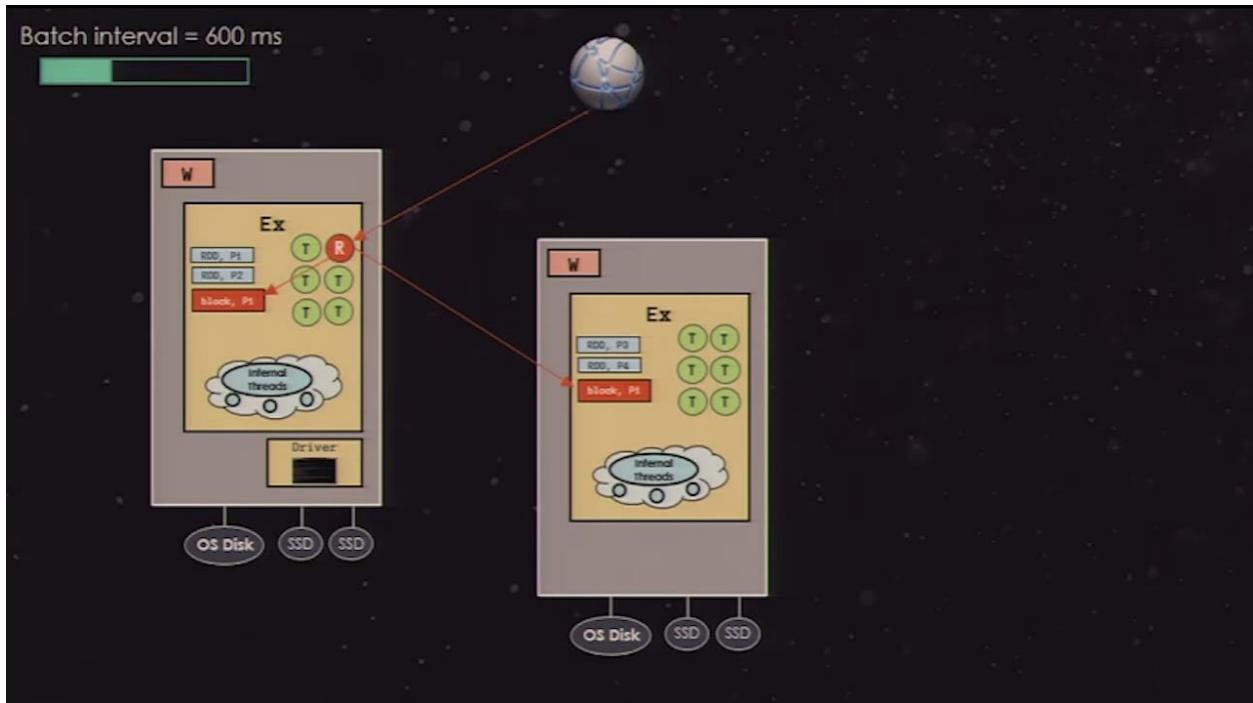


Batch interval of 5 seconds

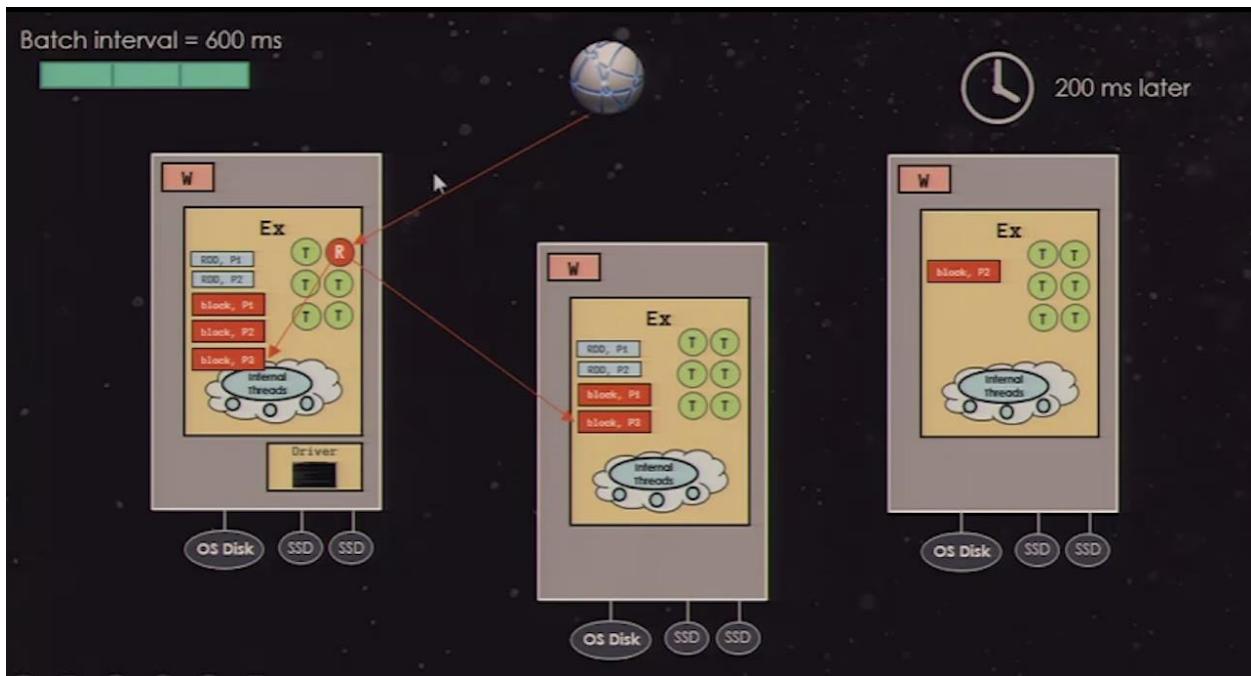
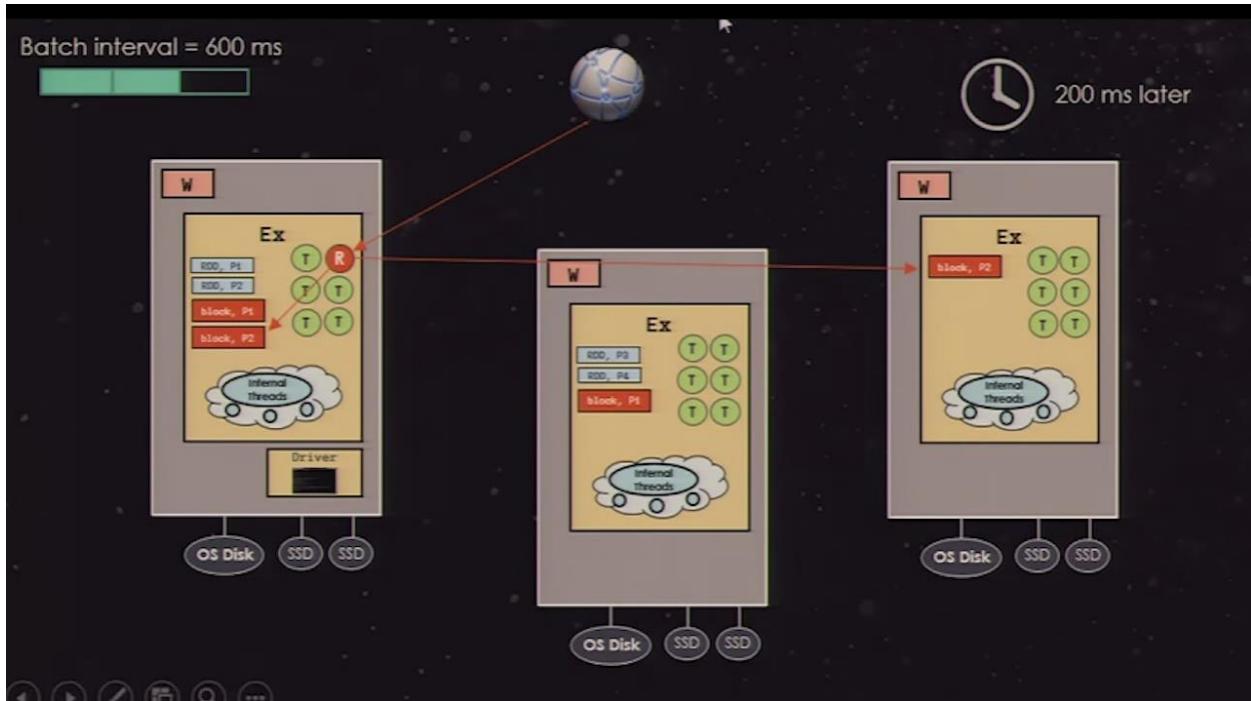


Spark streaming use a receiver to listening on specific IP and port (e.g. 9999) or pulling data from kafka ,flume or twitter , Receiver thread will write out a block (like partition) and copy of the block in a different executor JVM

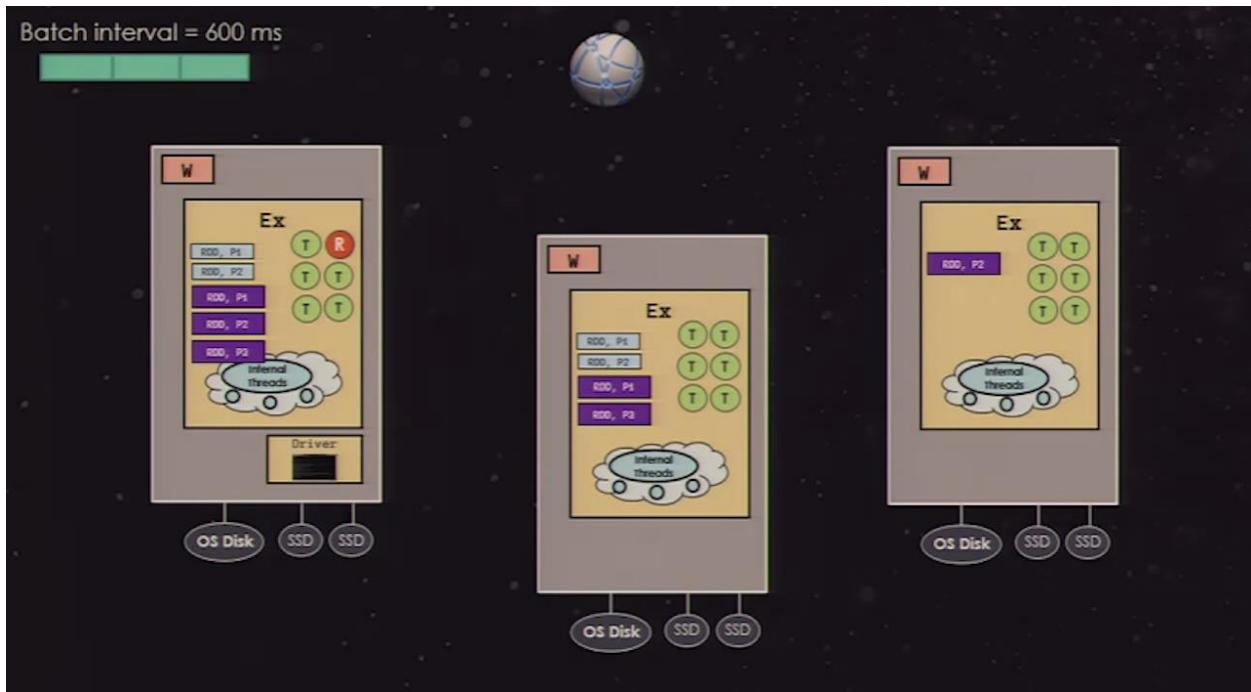
Here our batch interval is 600ms, whatever data is come in first 200ms will create the first block



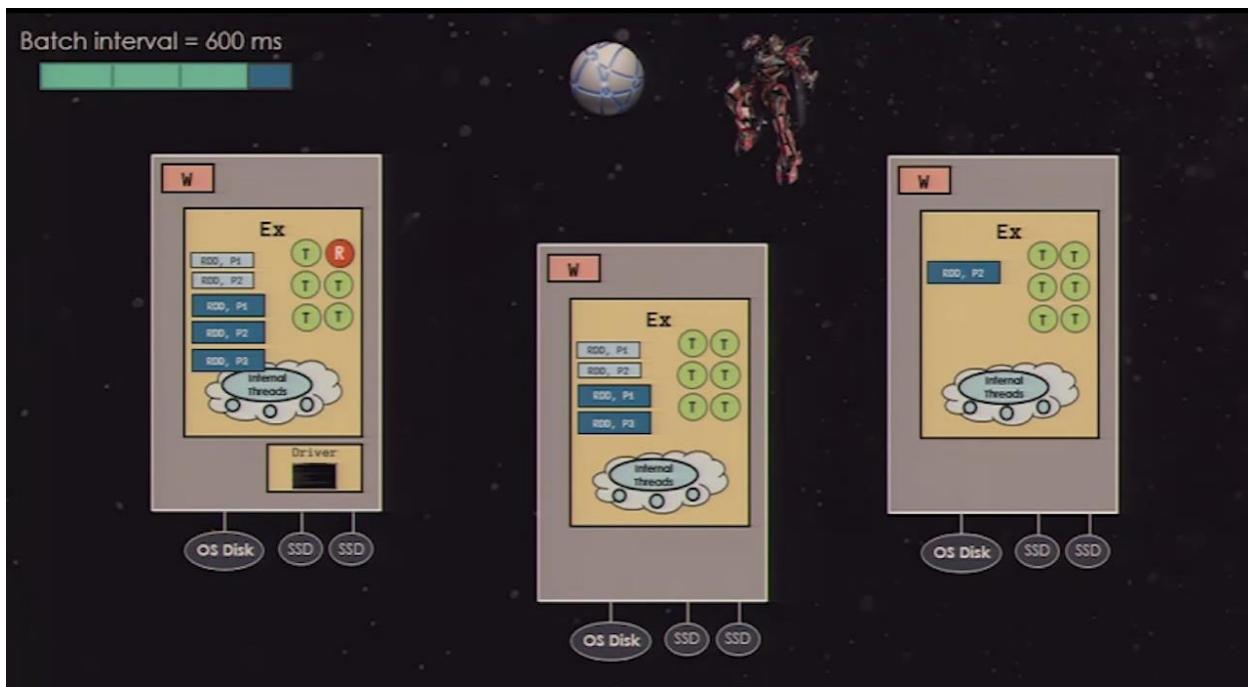
The same receiver continues listening for the next 200ms and creates one new block and continues the same till 600ms.



Now we hit our batch interval period, so now it's time to materialized and make the RDD



Now on these RDDs we are applying transformation function, those transformation we applied is processing latency, this happens after the batch interval over. By that time receiver continue to listen new data, so ideally before creating new RDD, processing for transformation for previous RDD get completed otherwise we will fall behind, so if our processing latency is too high like 6 sec and batch interval is 5 sec then we need to re architecture that, need to decrease batch interval so that processing latency go down. Because if batch interval is go down we will have less data so processing latency also go down.



Spark stream UI

Streaming

Started at: Wed Oct 22 06:11:53 PDT 2014
 Time since start: 27 minutes 20 seconds
 Network receivers: 1
 Batch interval: 1 second
 Processed batches: 1643
 Waiting batches: 0

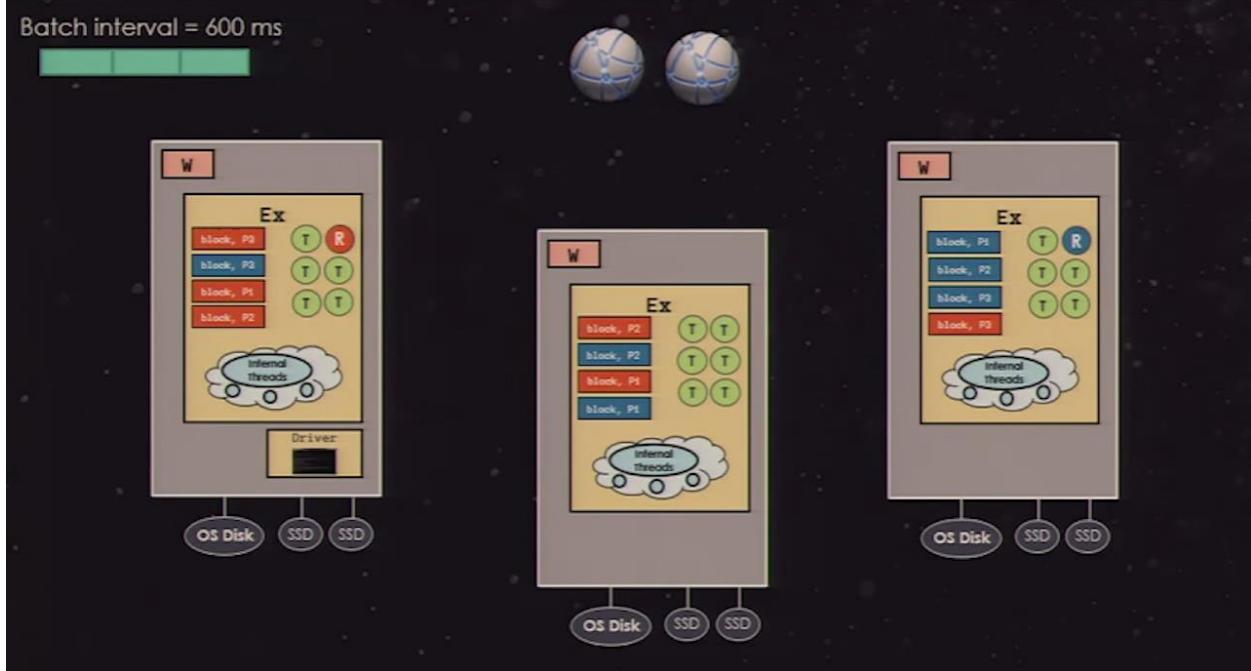
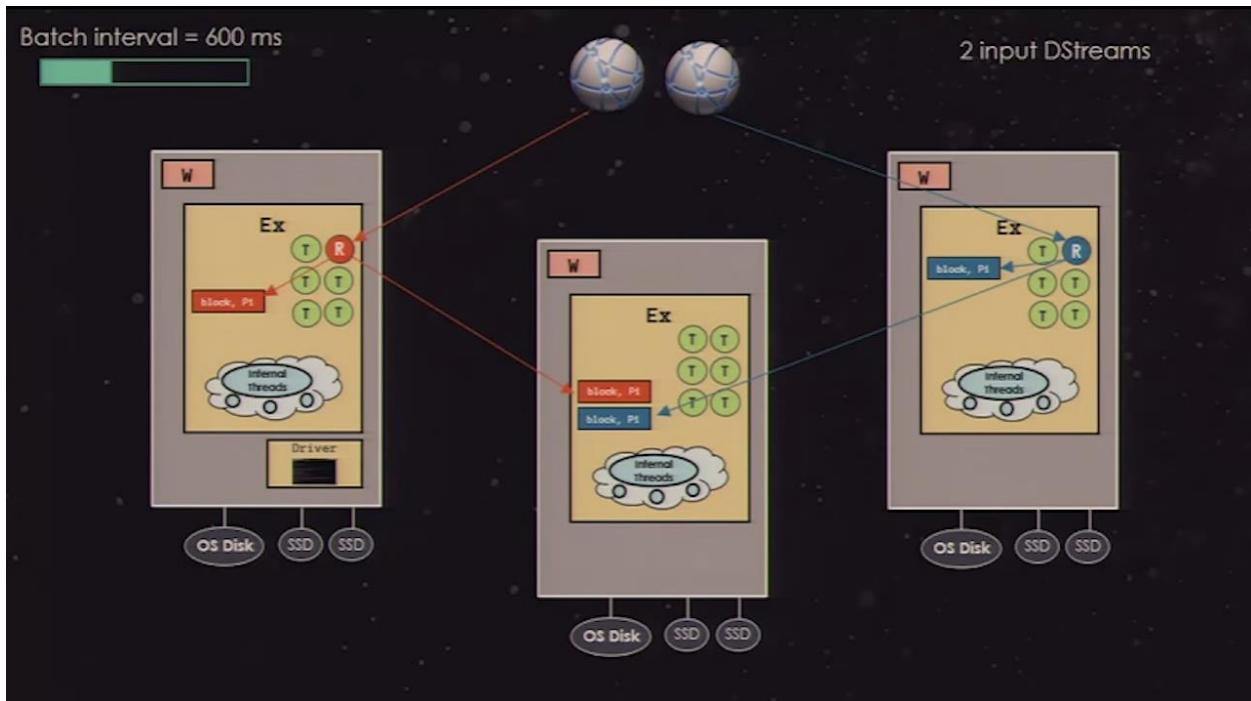
Statistics over last 100 processed batches

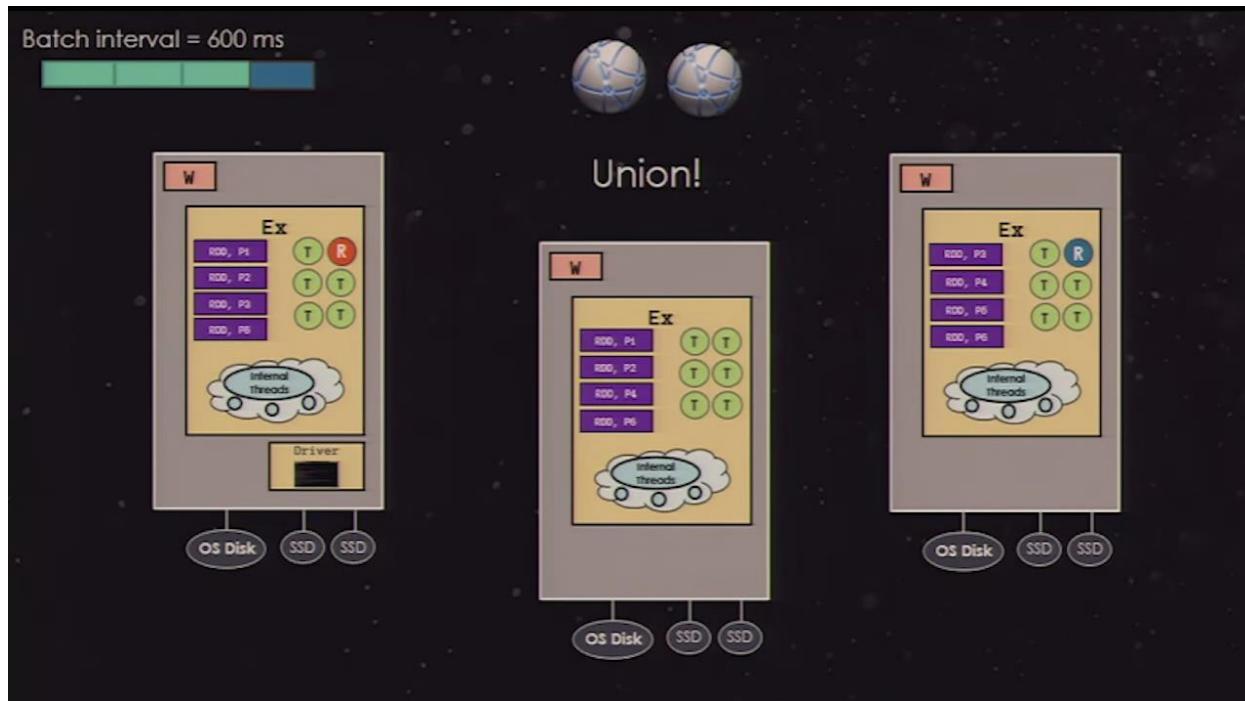
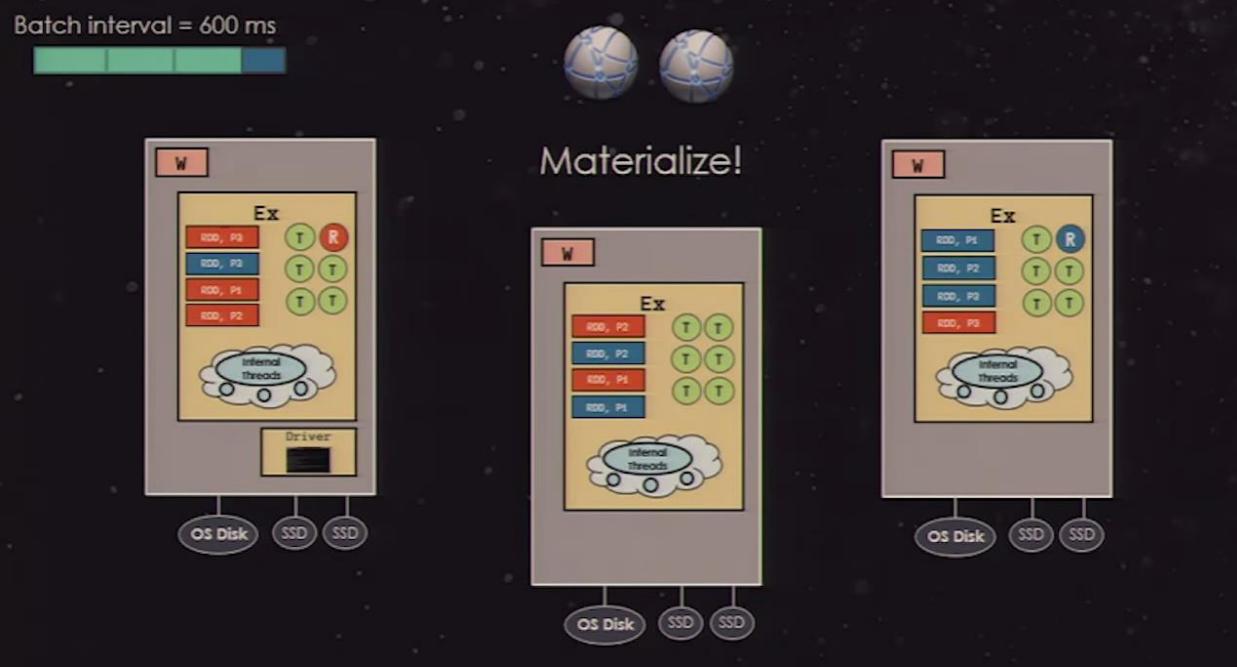
Receiver	Status	Location	Records in last batch	Minimum rate [records/sec]	Median rate [records/sec]	Maximum rate [records/sec]	Last Error
TwitterReceiver-0	ACTIVE	localhost	39	0	61	151	-

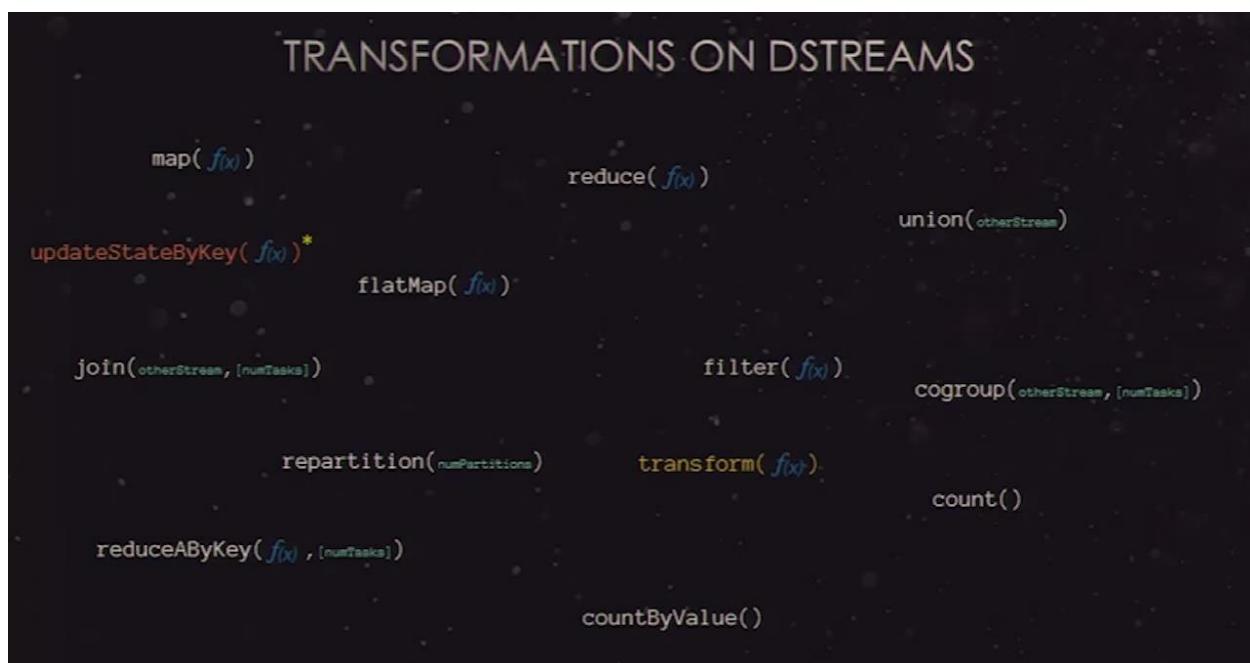
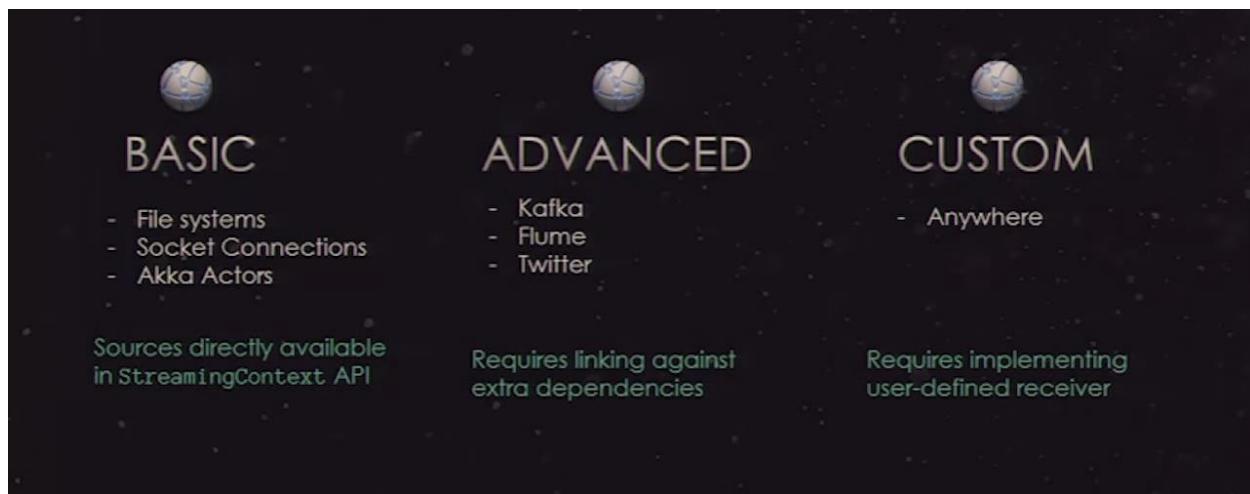
Batch Processing Statistics

Metric	Last batch	Minimum	25th percentile	Median	75th percentile	Maximum
Processing Time	31 ms	5 ms	39 ms	56 ms	457 ms	2 seconds 289 ms
Scheduling Delay	0 ms	0 ms	0 ms	0 ms	1 ms	803 ms
Total Delay	31 ms	31 ms	40 ms	57 ms	499 ms	2 seconds 289 ms

We can set multiple receivers (may be two different kafka topics)







TRANSFORMATIONS ON DSTREAMS

`updateStateByKey(f(x))`* : allows you to maintain arbitrary state while continuously updating it with new information.

To use:

1) Define the state

(an arbitrary data type)

2) Define the state update function

(specify with a function how to update the state using the previous state and new values from the input stream)

To maintain a running count of each word seen in a text data stream (here running count is an integer type of state):

```
def updateFunction(newValues, runningCount):
    if runningCount is None:
        runningCount = 0
    return sum(newValues, runningCount) # add the
                                         # new values with the previous running count
                                         # to get the new count

runningCounts = pairs.updateStateByKey(updateFunction)
```

* Requires a checkpoint directory to be configured

TRANSFORMATIONS ON DSTREAMS

`RDD`
↓
`RDD`
`transform(f(x))` : can be used to apply any RDD operation that is not exposed in the DStream API.

For example:

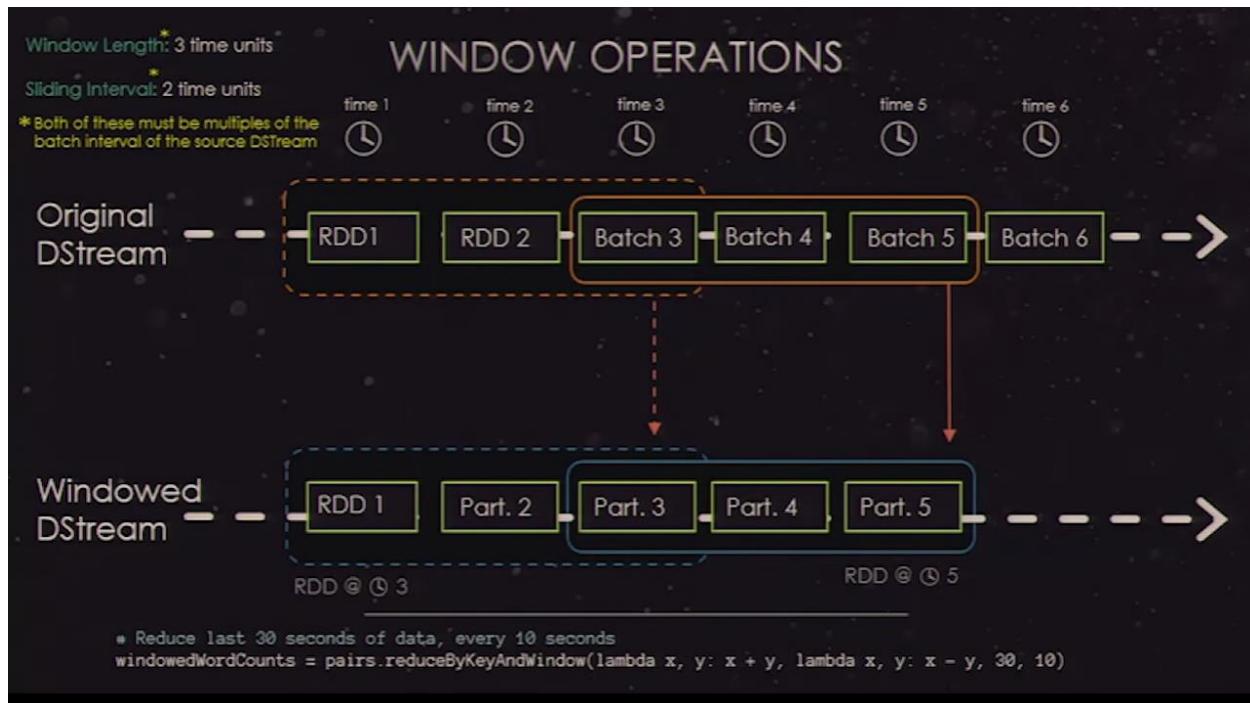
- Functionality to join every batch in a data stream with another dataset is not directly exposed in the DStream API.
- If you want to do real-time data cleaning by joining the input data stream with pre-computed spam information and then filtering based on it.

```
spamInfoRDD = sc.pickleFile(...) # RDD containing spam information
# join data stream with spam information to do data cleaning
cleanedDStream = wordCounts.transform(lambda rdd:
                                         rdd.join(spamInfoRDD).filter(...))

MLlib or GraphX
```

We can set window length and sliding interval

If you want to do analysis on all the data came in last 30 sec and you want to do that analysis every 10 sec



OUTPUT OPERATIONS ON DSTREAMS

```
print()  
  
foreachRDD(f(x))  
saveAsTextFile(prefix, [suffix])  
  
saveAsObjectFiles(prefix, [suffix])  
  
saveAsHadoopFiles(prefix, [suffix])
```