getindata

Apache Spark

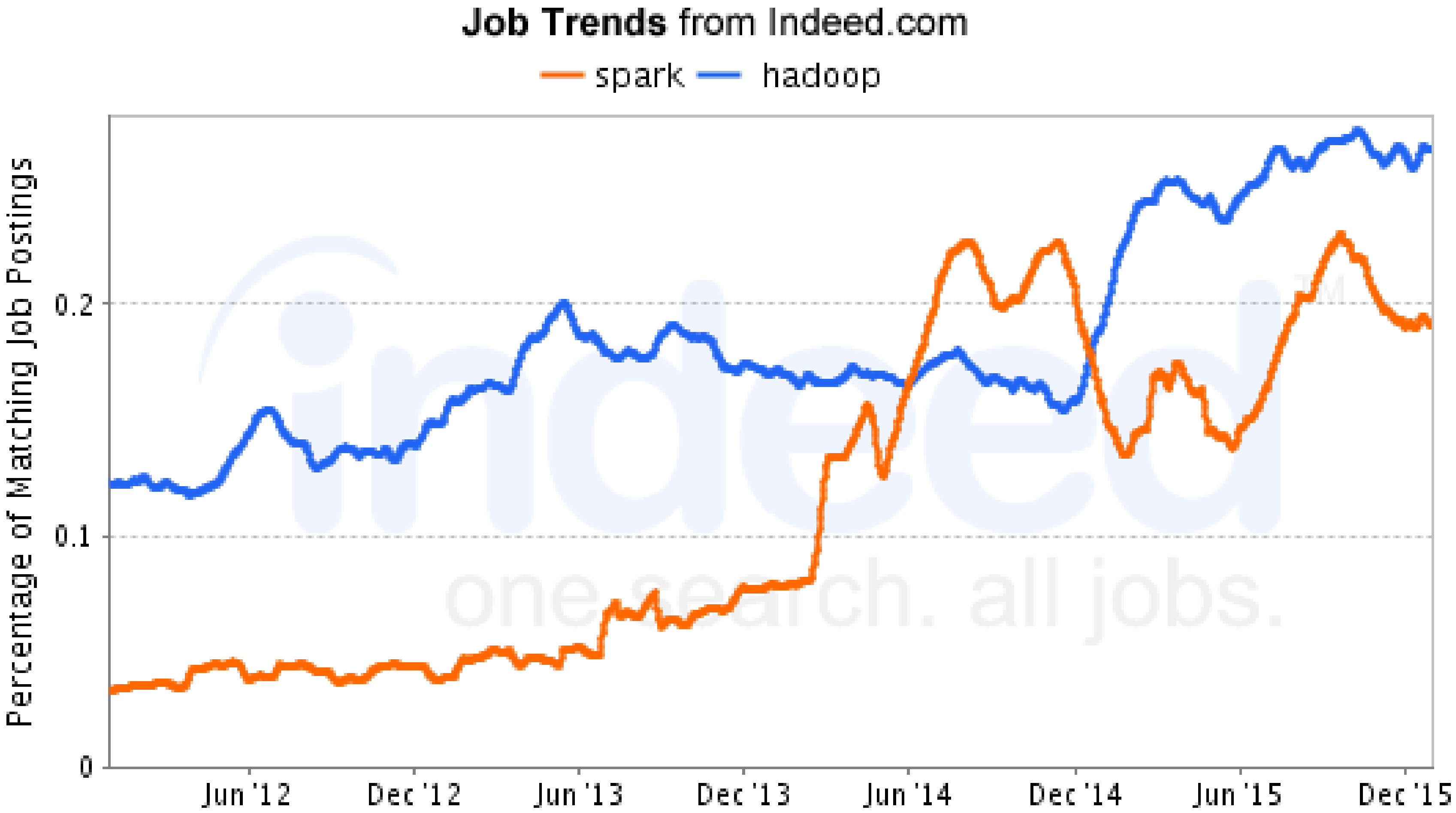
Chapter

Spark Overview

Apache Spark

- **The most popular framework for developing distributed applications on Hadoop**
  - Designed to solve MapReduce inefficiencies

Spark’s Excitement



Benefits Of Spark

- **Excellent performance and scalability**
- **Intuitive and concise API**
  - In several languages
- **Well integrated with Hadoop**
- **Variety of data sources and sinks**
- **A unified solution for various use-cases**

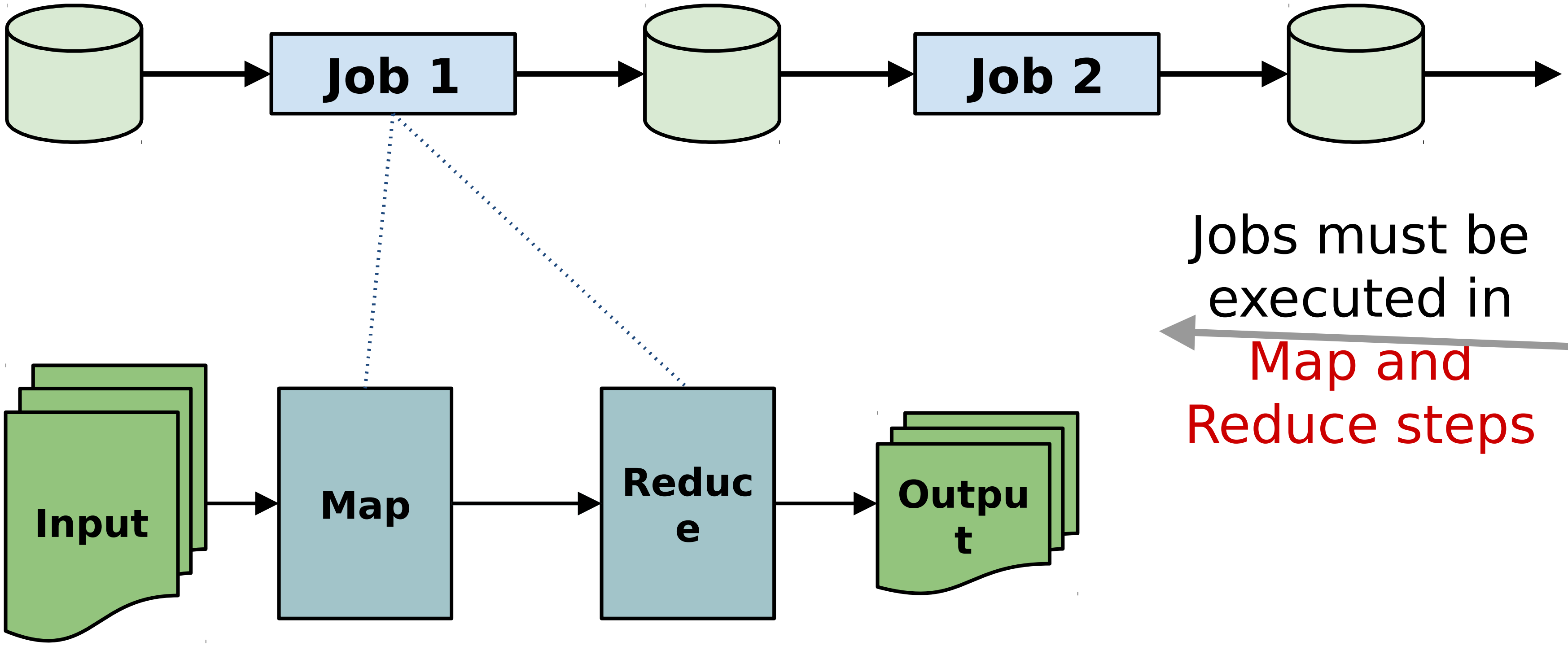
Intuitive And Concise API ...

- **High-level operators**
  - Transforming, grouping, aggregating, joining, filtering
- **UDFs are just plain code**

```
val input = sc.textFile("/training/data/track")

val artist = input.map(line=> getArtist(line))

val artistAndOne = artist.map(artist => (artist, 1))

val artistCount = artistAndOne.reduceByKey(a, b => a + b)

artistCount.saveAsTextFile("/training/output/artistCount")
```

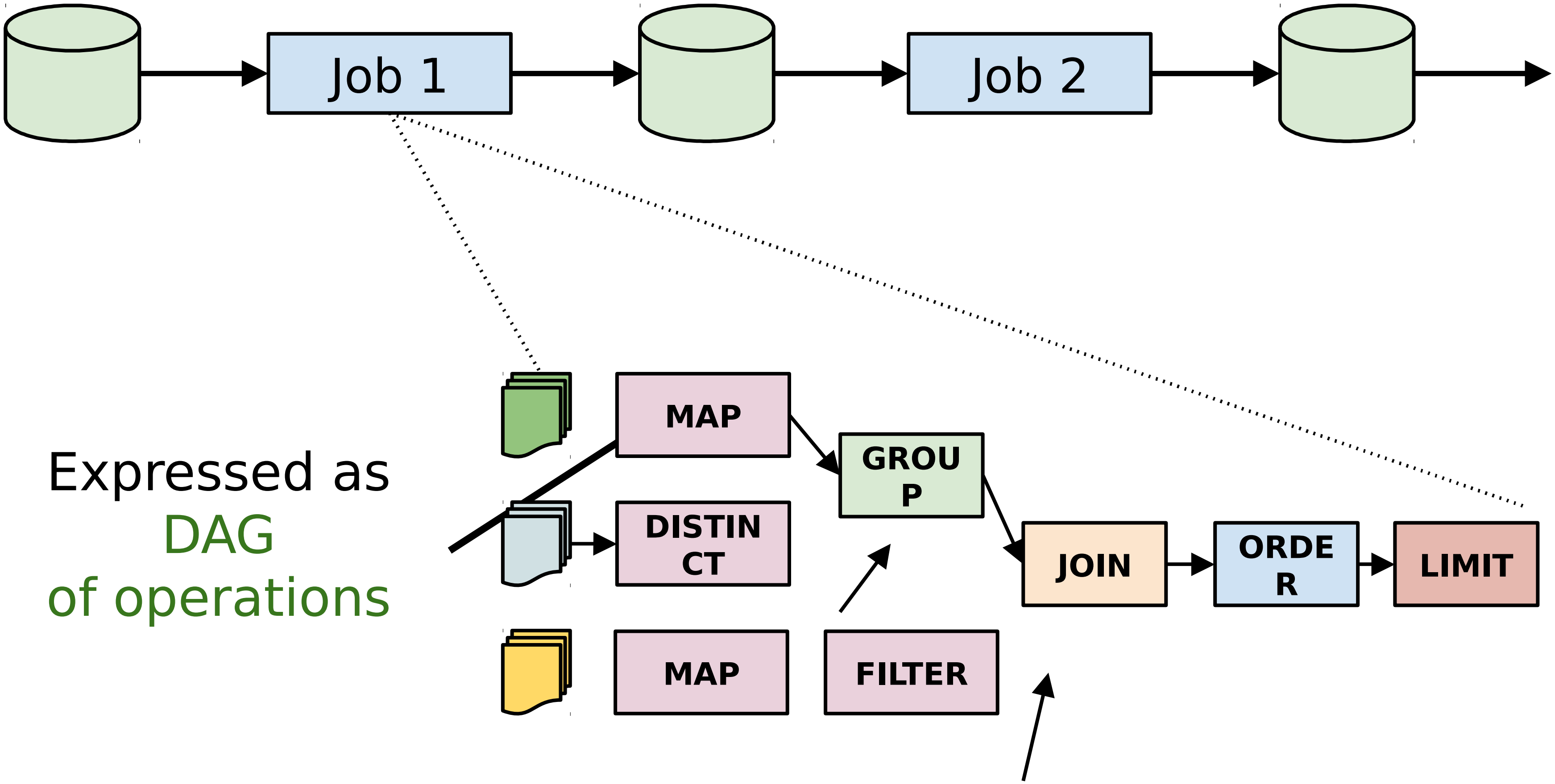
… In Several Languages

Speed (MapReduce)



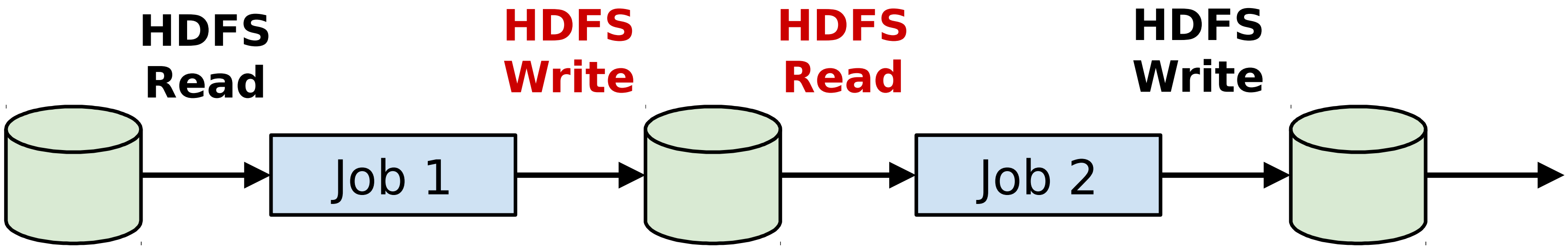
Jobs must be executed in Map and Reduce steps

Speed Spark



Expressed as
DAG
of operations

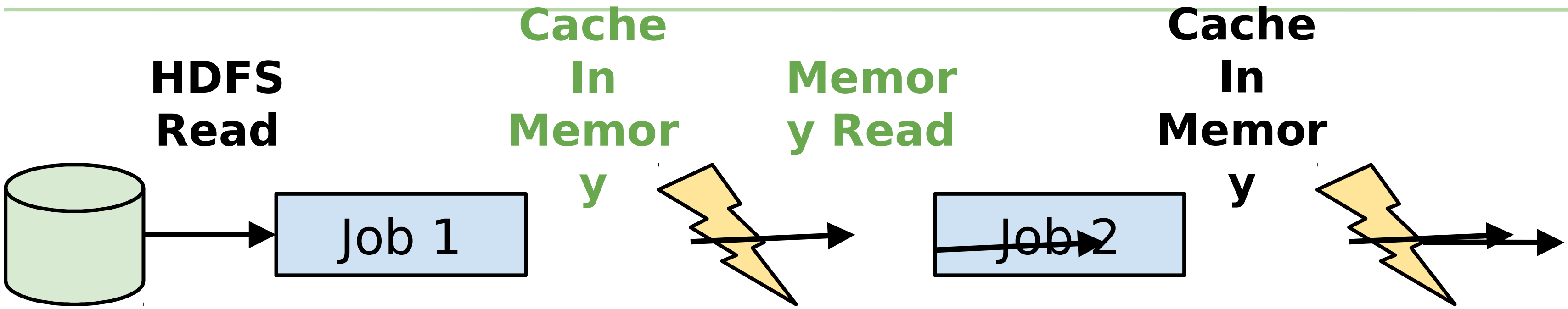
Speed (MapReduce)



Data between jobs is written to HDFS

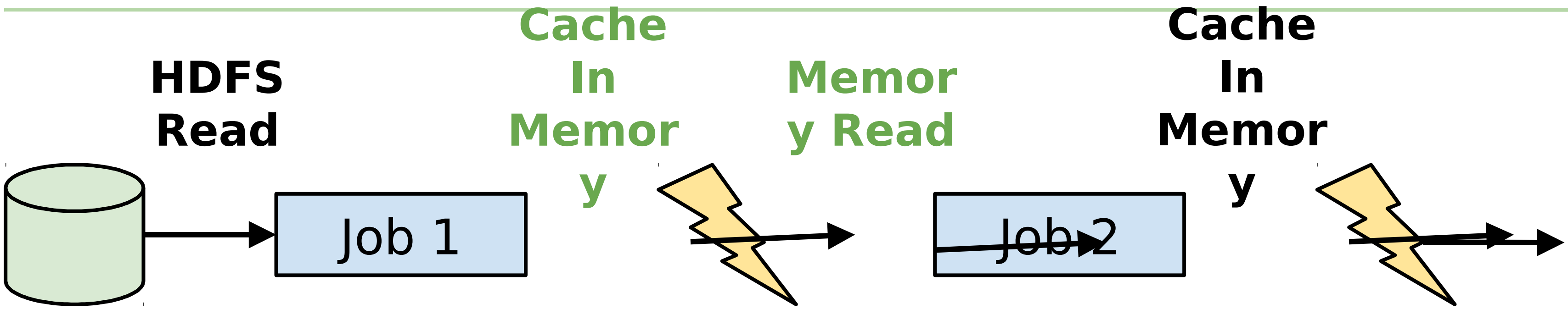
- disk IO overhead
- data serialization and replication

Speed



1. Cache dataset in a cluster’s (distributed) memory
2. Reuse it in future jobs

Speed Spark



HDFS
Read

Job 1

Cache
In
Memor
y

Memor
y Read

Job 2

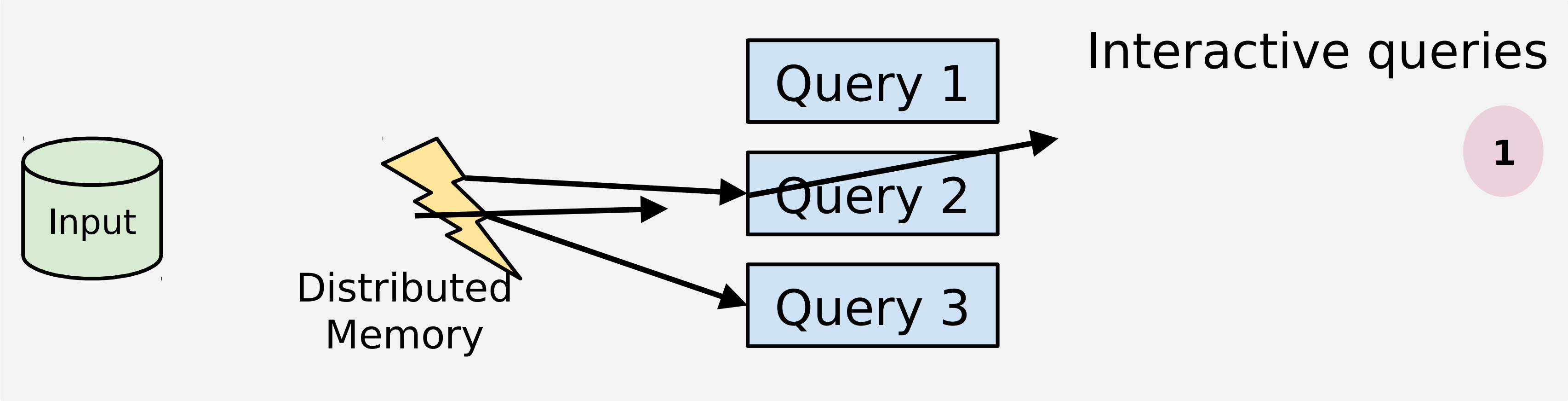
Cache
In
Memor
y

1. Cache dataset in a cluster’s (distributed) memory
2. Reuse it in future jobs

Great fit for iterative algorithms and interactive queries!

Speed

- **Iterative algorithm vs. interactive queries**



Interactive queries

1

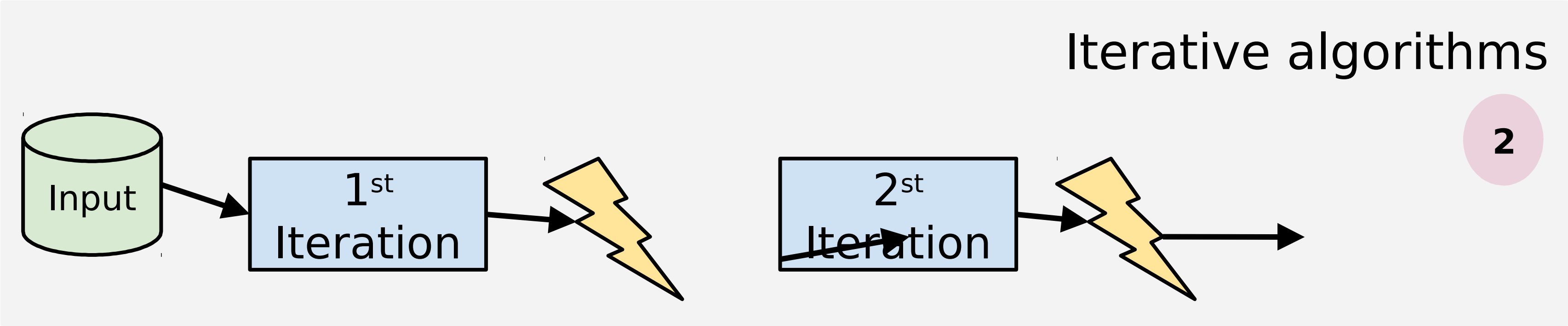
Input

Distributed Memory

Query 1

Query 2

Query 3



Iterative algorithms

2

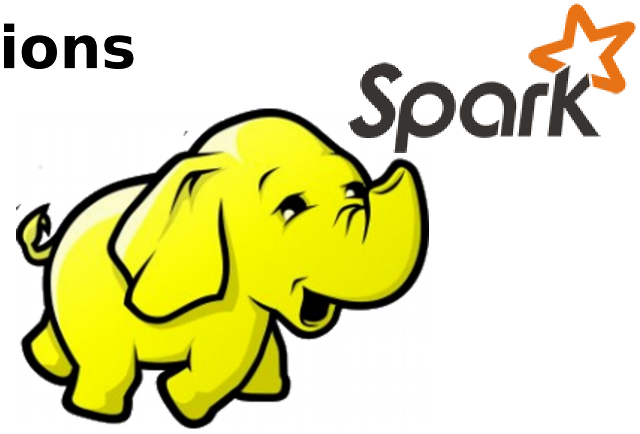
Input

1st Iteration

2st Iteration

Integration With Hadoop

- **Runs on YARN**
- **Reads existing Hadoop data from HDFS and Hive**
  - … and many other data-sources
- **Supports data locality**
- **Integrates with Kerberos**
- **Included in popular distributions**
  - HDP, CDH, MapR

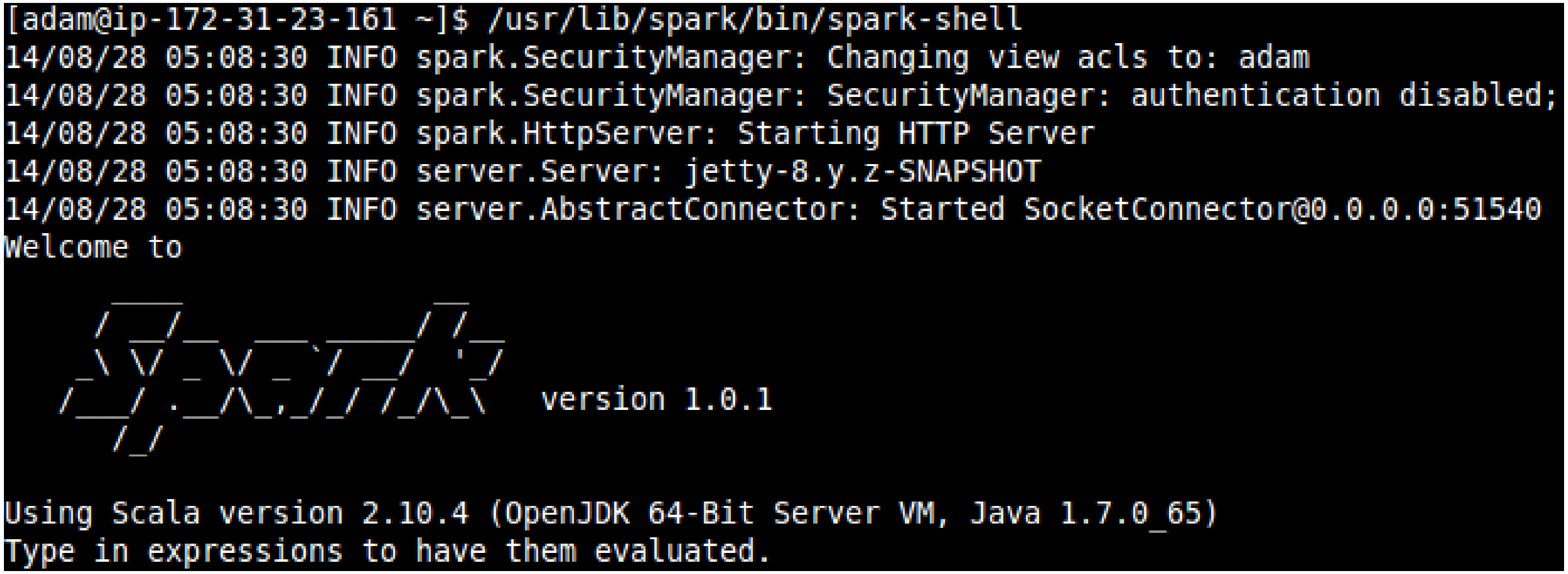


Data Sources

- **Files/directories**
  - Local Fs, HDFS, Amazon S3 etc.
- **Different file formats**
  - Avro, Parquet, ORC, JSON etc.
- **Hive tables**
- **NoSQL databases**
  - Cassandra, HBase, Elasticsearch, MongoDB etc.
- **MySQL tables**
- **Additional connectors are being developed**

Interactive Shells

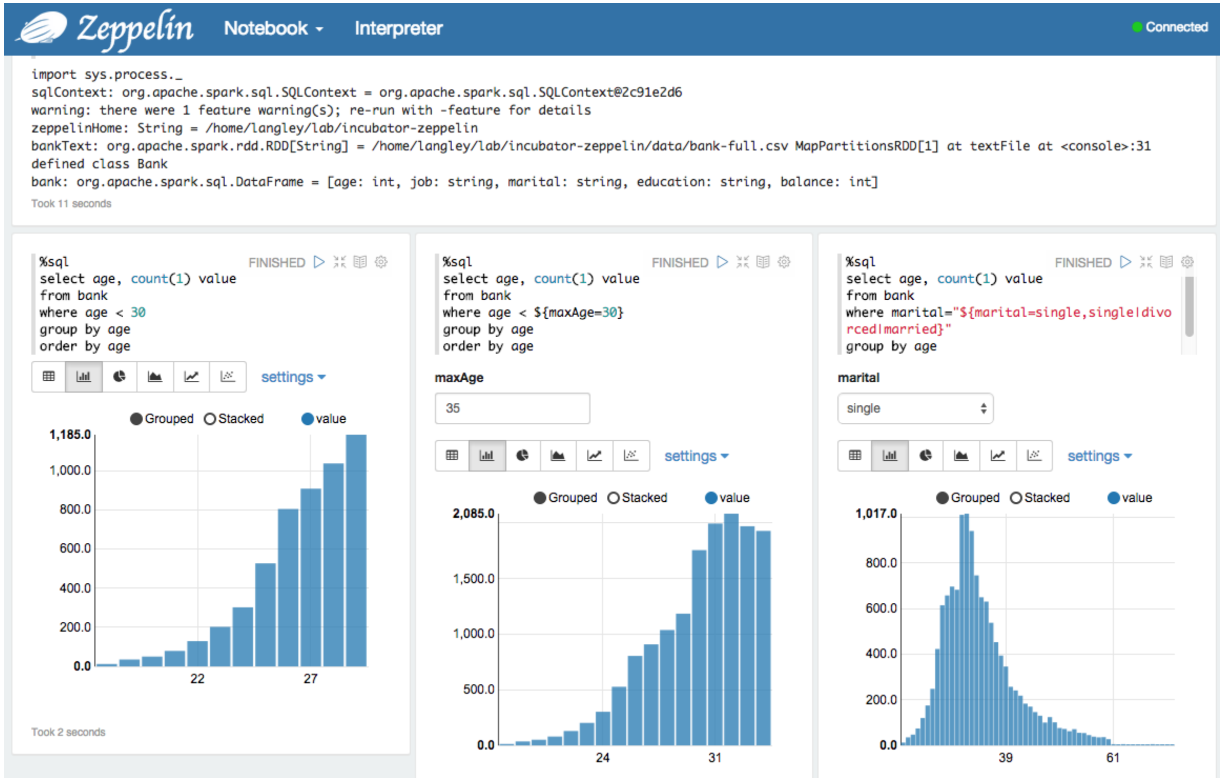
- **Scala (spark-shell)**
- **Python (pyspark)**
- **R (sparkR)**



```
[adam@ip-172-31-23-161 ~]$ /usr/lib/spark/bin/spark-shell
14/08/28 05:08:30 INFO spark.SecurityManager: Changing view acls to: adam
14/08/28 05:08:30 INFO spark.SecurityManager: SecurityManager: authentication disabled;
14/08/28 05:08:30 INFO spark.HttpServer: Starting HTTP Server
14/08/28 05:08:30 INFO server.Server: jetty-8.y.z-SNAPSHOT
14/08/28 05:08:30 INFO server.AbstractConnector: Started SocketConnector@0.0.0.0:51540
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.0.1
      /_/

Using Scala version 2.10.4 (OpenJDK 64-Bit Server VM, Java 1.7.0_65)
Type in expressions to have them evaluated.
```

Interactive Notebooks

- **Zeppelin, Jupyter and more**



```
import sys.process._
sqlContext: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@2c91e2d6
warning: there were 1 feature warning(s); re-run with -feature for details
zeppelinHome: String = /home/langley/lab/incubator-zeppelin
bankText: org.apache.spark.rdd.RDD[String] = /home/langley/lab/incubator-zeppelin/data/bank-full.csv MapPartitionsRDD[1] at textFile at <console>:31
defined class Bank
bank: org.apache.spark.sql.DataFrame = [age: int, job: string, marital: string, education: string, balance: int]
```

Took 11 seconds

```
%sql
select age, count(1) value
from bank
where age < 30
group by age
order by age
```

Took 2 seconds

```
%sql
select age, count(1) value
from bank
where age < ${maxAge=30}
group by age
order by age
```

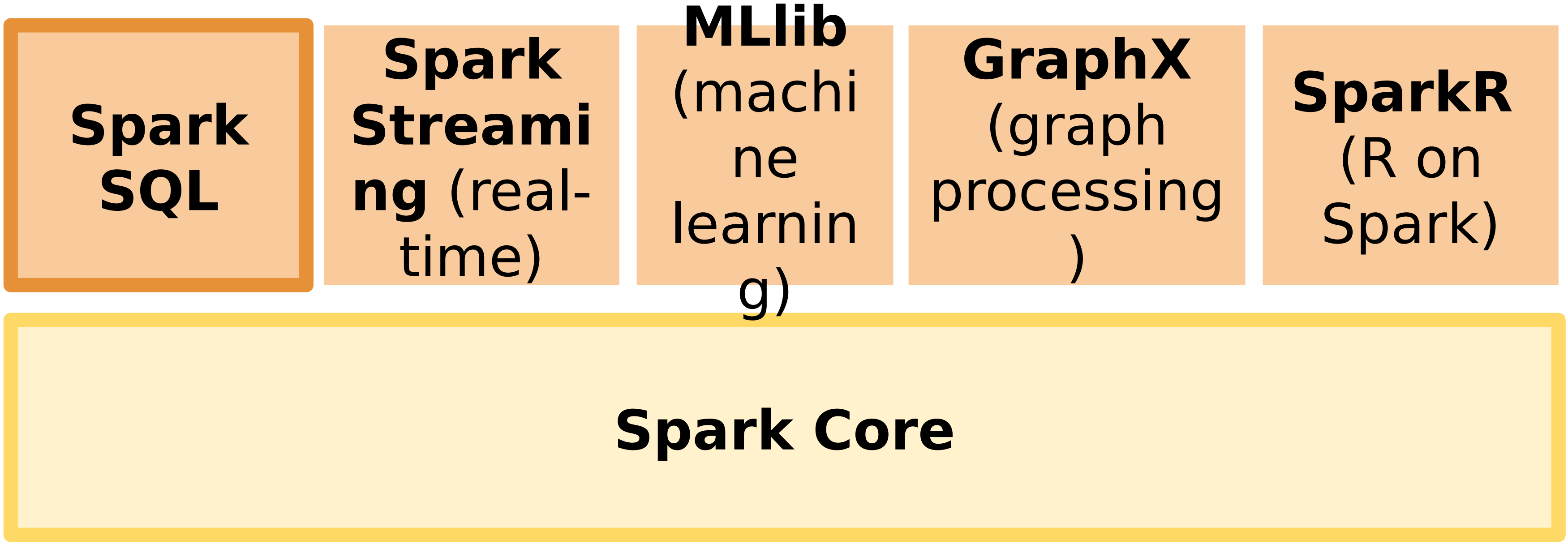
maxAge: 35

```
%sql
select age, count(1) value
from bank
where marital="${marital=single,single|divo
rced|married}"
group by age
```

marital: single

Spark Stack

- **Growing ecosystem of high-level and nicely-integrated tools for various use-cases**



Spark SQL

Spark Streaming (real-time)

MLlib (machine learning)

GraphX (graph processing)

SparkR (R on Spark)

Spark Core

Unified Solution For Various Use-Cases

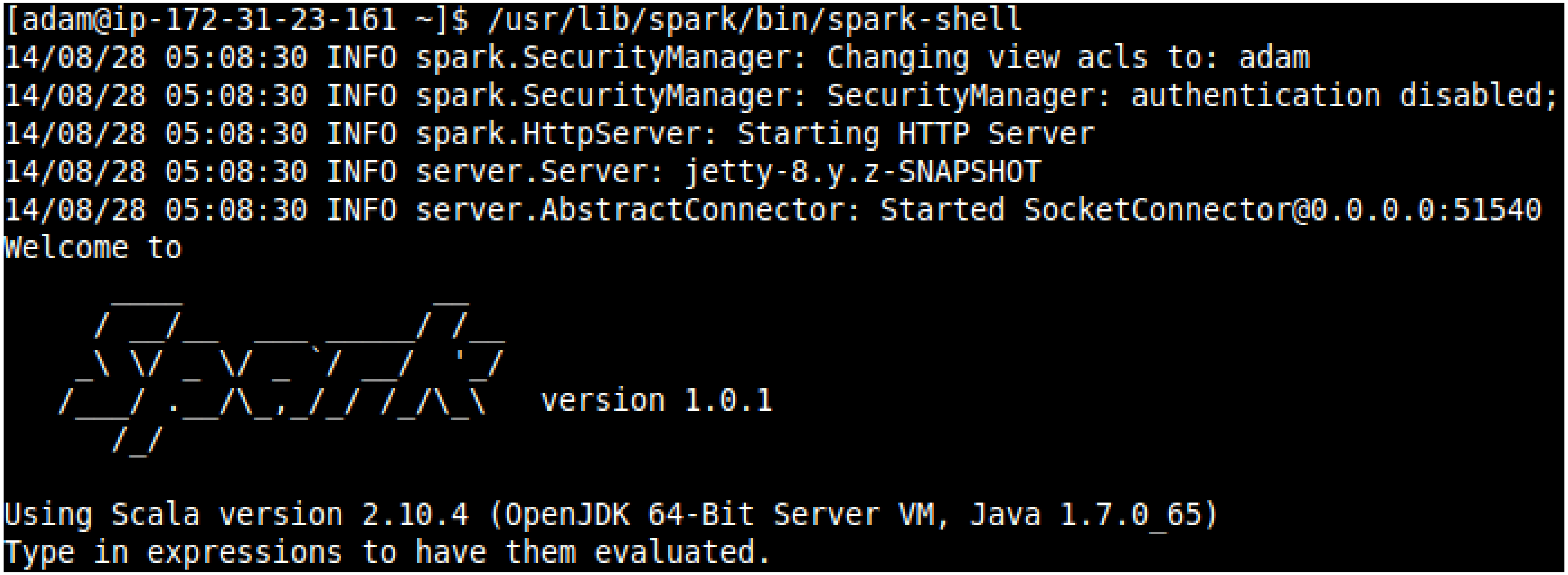
- **Spark aims to be a single platform for**
  - Batch processing
  - Real-time stream processing (micro-batching)
  - Complex iterative analysis (ML and graph processing)
  - Interactive ad-hoc queries (including SQL)

Chapter

Basic Concepts

Launching Spark Shell

```
$ spark-shell --master yarn-client --driver-memory 256m --executor-memory 128m --num-executors 3
```



```
[adam@ip-172-31-23-161 ~]$ /usr/lib/spark/bin/spark-shell
14/08/28 05:08:30 INFO spark.SecurityManager: Changing view acls to: adam
14/08/28 05:08:30 INFO spark.SecurityManager: SecurityManager: authentication disabled;
14/08/28 05:08:30 INFO spark.HttpServer: Starting HTTP Server
14/08/28 05:08:30 INFO server.Server: jetty-8.y.z-SNAPSHOT
14/08/28 05:08:30 INFO server.AbstractConnector: Started SocketConnector@0.0.0.0:51540
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.0.1
      /_/

Using Scala version 2.10.4 (OpenJDK 64-Bit Server VM, Java 1.7.0_65)
Type in expressions to have them evaluated.
```

Reading Data

```
scala> input = sc.textFile("/training/data/user")


scala> input.take(3)
[u'1\tEDWIN\tBATES\tM\t1965-03-14\tColorado\t2013-12-25',
u'2\tISABELLE\tHEATH\tF\t1999-09-12\tOregon\t2013-05-15',
u'3\tANNIKA\tFREEMAN\tF\t1967-09-23\tMichigan\t2013-09-22']
```

Splitting Line By Delimiter

```
scala> val splitted = input.map(line => line.split("\t"))

scala> splitted.take(3)
[[u'1', u'EDWIN', u'BATES', u'M', u'1965-03-14', u'Colorado',
u'2013-12-25'], [u'2', u'ISABELLE', u'HEATH', u'F', u'1999-09-12',
u'Oregon', u'2013-05-15'], [u'3', u'ANNIKA', u'FREEMAN', u'F',
u'1967-09-23', u'Michigan', u'2013-09-22']]
```

Projecting Records

```
scala> val state = splitted.map(line => line[5])


scala> state.take(3)
[u'Colorado', u'Oregon', u'Michigan', u'Wisconsin', u'Texas']
```

Mapping Records

```
scala> val stateAndOne = state.map(state=> (state, 1))


scala> stateAndOne.take(5)
[(u'Colorado', 1), (u'Oregon', 1), (u'Michigan', 1), (u'Wisconsin', 1), (u'Texas', 1)]
```

Aggregating Numbers

```
scala> stateCount= stateAndOne.reduceByKey(a,b=> a
+ b)

scala> stateCount.take(5)
[(u'Mississippi',11),(u'Oklahoma',14),(u'Arkansas',8),
(u'Maryland',21),(u'Louisiana',11)]
```

Dumping Output To The Screen

```
scala> stateCount.collect()
[(u'Mississippi',11),(u'Oklahoma',14),(u'Arkansas',8),
(u'Maryland',21),(u'Louisiana',11),(u'Idaho',6),(u'Iowa',5),
(u'Michigan',28),(u'Utah',12),(u'Oregon',13),(u'Connecticut',
11),(u'California',113),(u'Texas',84),(u'South Carolina',19),
(u'New Hampshire',3)…]
```

Resilient Distributed Dataset (RDD)

- **Core abstraction in Spark**
- **Represents a collection of data**
  - Divided into partitions
  - Partitions are distributed across cluster machines
  - Can be cached in memory and/or on disk
  - Replicated
  - Fault-tolerant
  - Immutable
- **RDDs are operated on in parallel**

What Is RDD?

- **Examples of RDDs in Spark**
  - Lines from text files stored in HDFS
  - Rows from a MySQL table
  - Key-value pairs from a Cassandra table
  - Results of Spark transformation using the filter function

Creating RDDs

- **Load data from external sources**

```
sc.textFile("streamrock/data/played.tsv")
```

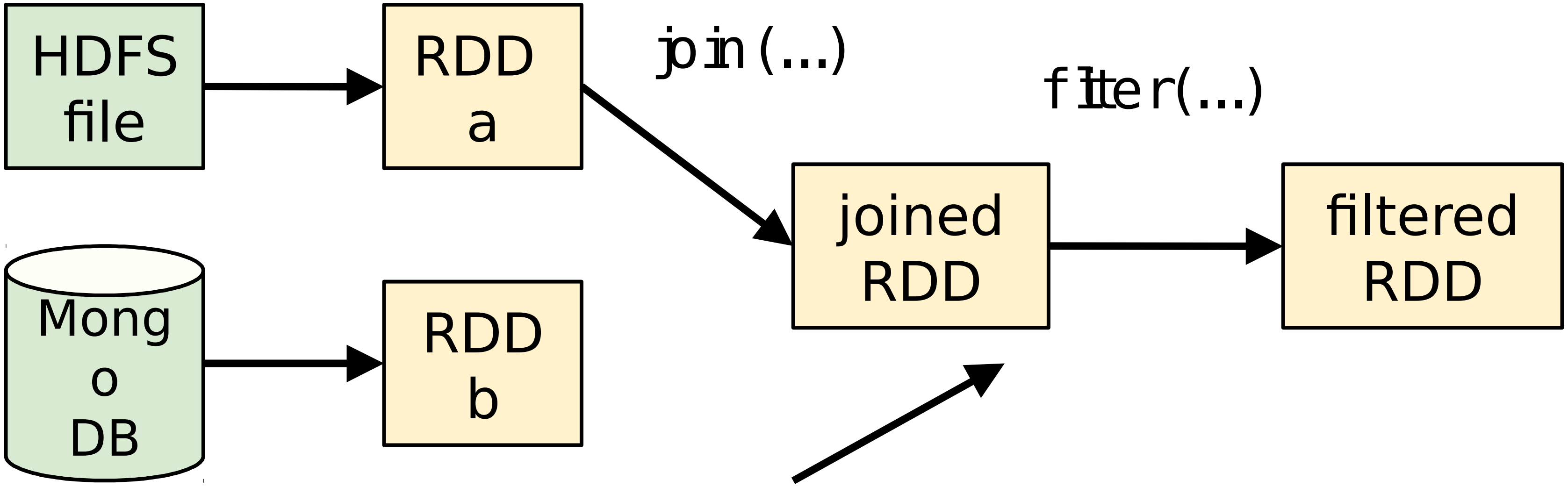
- **Use parallelize method**

```
sc.parallelize(Array("how","do","you","do"))
```

Good for testing, prototyping and learning, but not for production applications

RDD Transformations

- **Create new RDDs from existing ones**
- **Examples:**
  - map, flatMap, filter, sample, sort



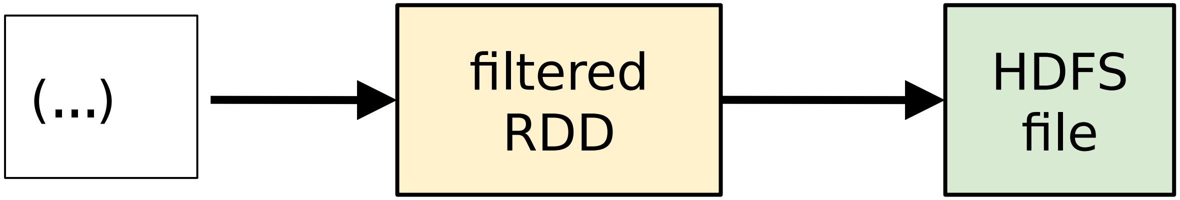
Example RDD Transformations

- **map**(func)
- **filter**(func)
- **sample**(withReplacement, fraction, seed)
- **distinct**([numTasks]))
- **groupByKey**([numTasks])
- **reduceByKey**(func, [numTasks])
- **sortByKey**([ascending], [numTasks])
- **join**(otherDataset, [numTasks])
- **pipe**(command, [envVars])

RDD Actions

- **Return a result or write it to the storage**
- **Trigger RDDs evaluation**
- **Examples:**
  - `count, first, take, saveAs, collect`

`saveAsTextFile("/some/hdfs/path")`



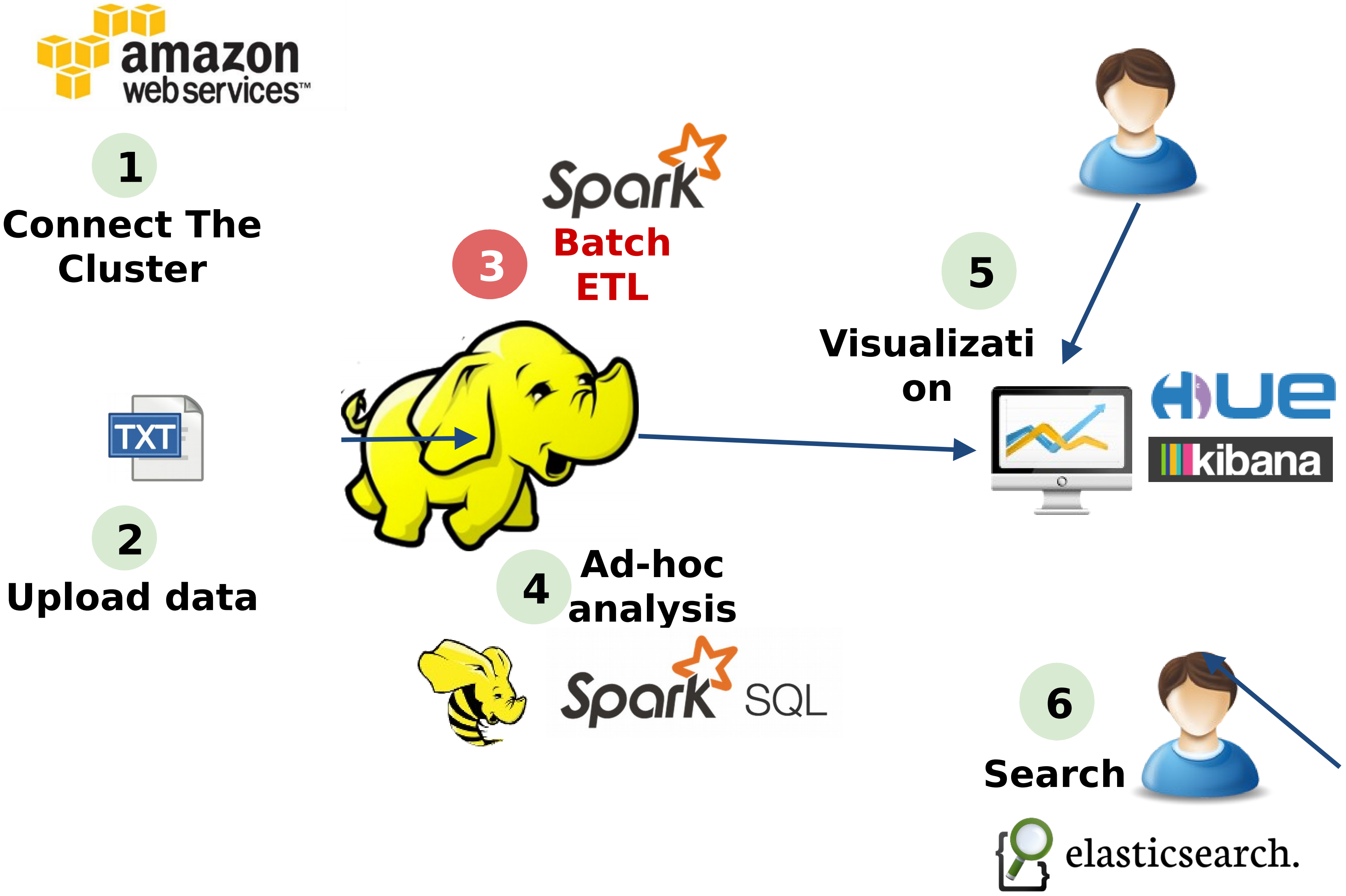
Example RDD Actions

- **count**()
- **countByKey**()
- **foreach**(func)
- **reduce**(func)
- **first**()
- **take**(n)
- **takeOrdered**(n, [ordering])
- **collect**()
- **saveAsTextFile**(path)

SparkContext

- **Entry point for interactions with Spark framework**
- **Contains a whole configuration for your application**
- **Starts a YARN application**

```
scala> sc.textFile("/training/data/user")
        .map(line => line.split("\t"))
        …
```



1
Connect The Cluster

2
Upload data

3
Batch ETL

4
Ad-hoc analysis

5
Visualizati on

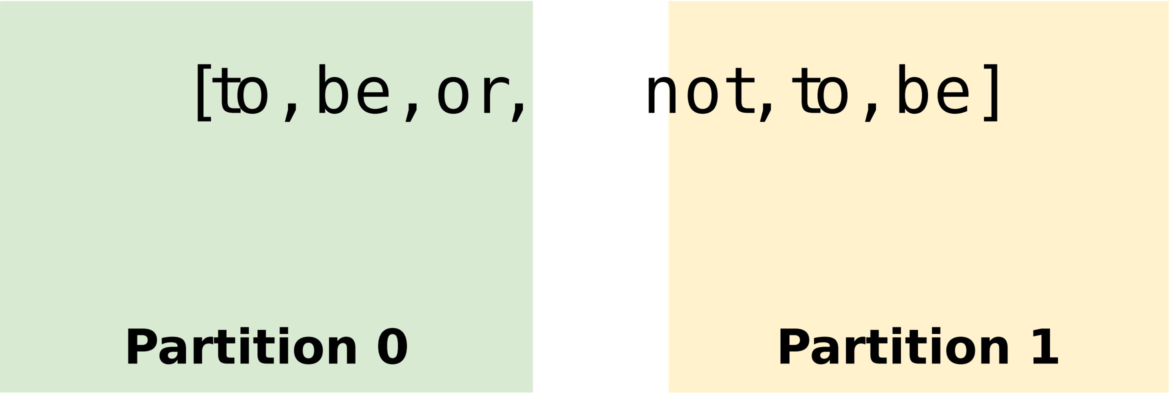
6
Search

Chapter

Detailed RDD Concepts

Distributed

- RDD are partitioned across nodes in a cluster
- RDD operations executed in parallel
- More partitions can increase parallelism



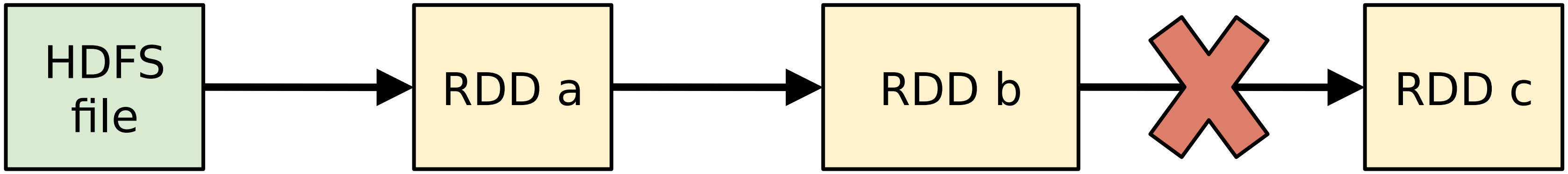
[to,be,or, not,to,be]

Partition 0

Partition 1

Resilient

- **Spark RDD is fault-tolerant**
  - If node/memory is lost, Spark will re-compute lost parts of dataset(s)
- **RDD tracks its “lineage”**
  - A series of transformations used to build it to recompute lost data



If some partition of RDD b is lost, it will be recomputed from the source HDFS file

Lazy Evaluated

- **By default, each transformed RDD is recomputed each time you run an action on it**
  - … unless, you cache it on memory and/or in disk
- **It gives Spark a possibility to apply more optimizations**

RDD Persistence

- **A dataset can be cached in memory across operations**
  - Makes iterative algorithms and interactive queries fast!
- **Caching a dataset is easy**

```
val file = sc.textFile("hdfs://...")
val errors = file.filter(line => line.contains("ERROR"))
errors.cache()
```

RDD Persistence Levels

- **You can control the way how RDD is persisted**
  - Memory only (default)
  - Memory and disk
  - Disk only
  - Offheap (experimental)
- **You can store data in serialized or deserialized form**
  - Serialized form is more compact, but it consumes more time and CPU to serialize elements
- **You can control the replication factor**
  - 1 or 2 replicas

Cache Management

- **Spark automatically evicts old partitions using a Least Recently Used (LRU) cache policy**
  - The “memory-only” level

It recomputes evicted partitions the next time they are accessed

  - The “memory-and-disk” level

It writes evicted partitions to disk

- **Your application won’t break if you cache too much data**
  - Caching unnecessary data, however, can lead to eviction of useful data and longer re-computation time

Persistence Levels Tips

- **Cache wisely!**
  - If you reuse RDD multiple times, then might cache it
- **Use `MEMORY_ONLY` for small RDDs**
- **For larger RDDs consider `MEMORY_ONLY_SER` or `MEMORY_AND_DISK_SER`**
- **If the same RDD is cached by multiple users, consider using**
  - Tachyon (experimental)
  - Spark Thrift Server

It will be explained later

Changing The Number Of Partitions

- **Spark automatically sets the number of partitions**
- **Often, you can change it using the 2nd parameter**
  - When reading data

```
sc.textFile(path, [num Tasks])
```

  - When grouping/shuffling operations e.g.

```
reduceByKey(func, [num Tasks])
sortByKey([ascending], [num Tasks])
join(otherDataset, [num Tasks])
```

- **You can also use `repartition()` and `coalesce()`**
  - `coalesce()` avoids data movement if you are decreasing the number of RDD partitions

Tweaking The Number Of Partitions

- **As a rule of thumb, a task should take at least 100 ms to execute**
  - You can find the task execution time in the Spark Web UI
- **The most straightforward way to tune the number of partitions is experimentation**
  - Look at the number of partitions and then keep multiplying that by 1.5 until performance stops improving

Exercise

RDD Caching and Partitions

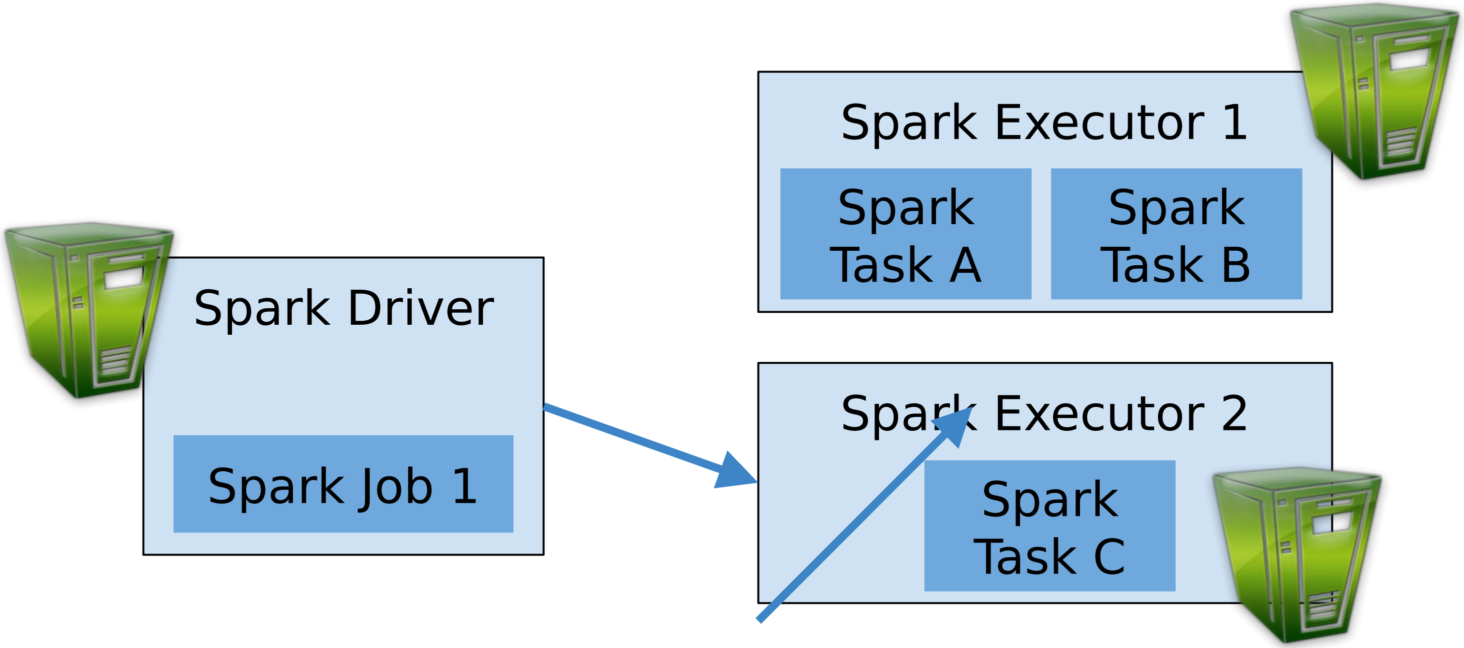
http://bit.ly/1Sh34jS

Pages 9 - 16

Chapter

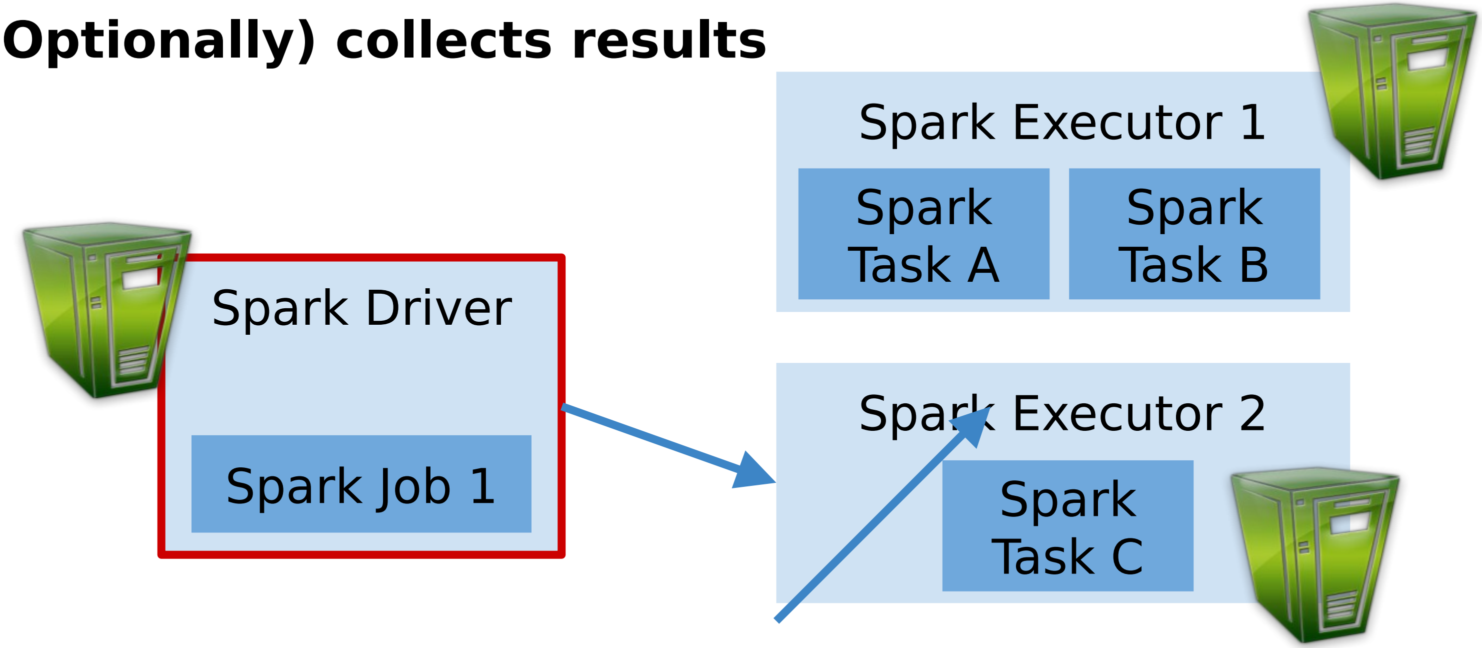
Spark Architecture

Processes In a Spark Application



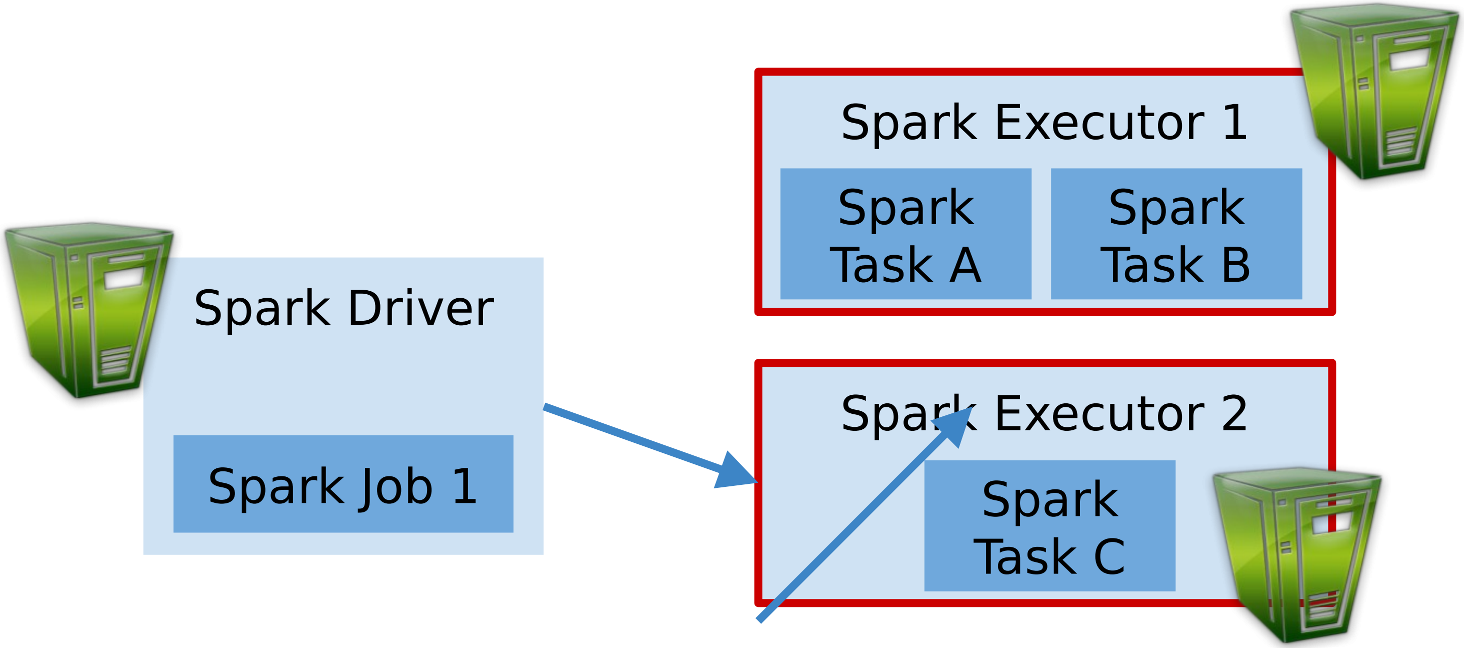
Spark Driver

- **One per Spark application**
- **Defines SparkContext, RDDs, set of RDD transformations and actions**
- **Optimizes and converts DAG into tasks**
- **Keeps track of all of its executors**
- **Schedules tasks on executors**
- **(Optionally) collects results**



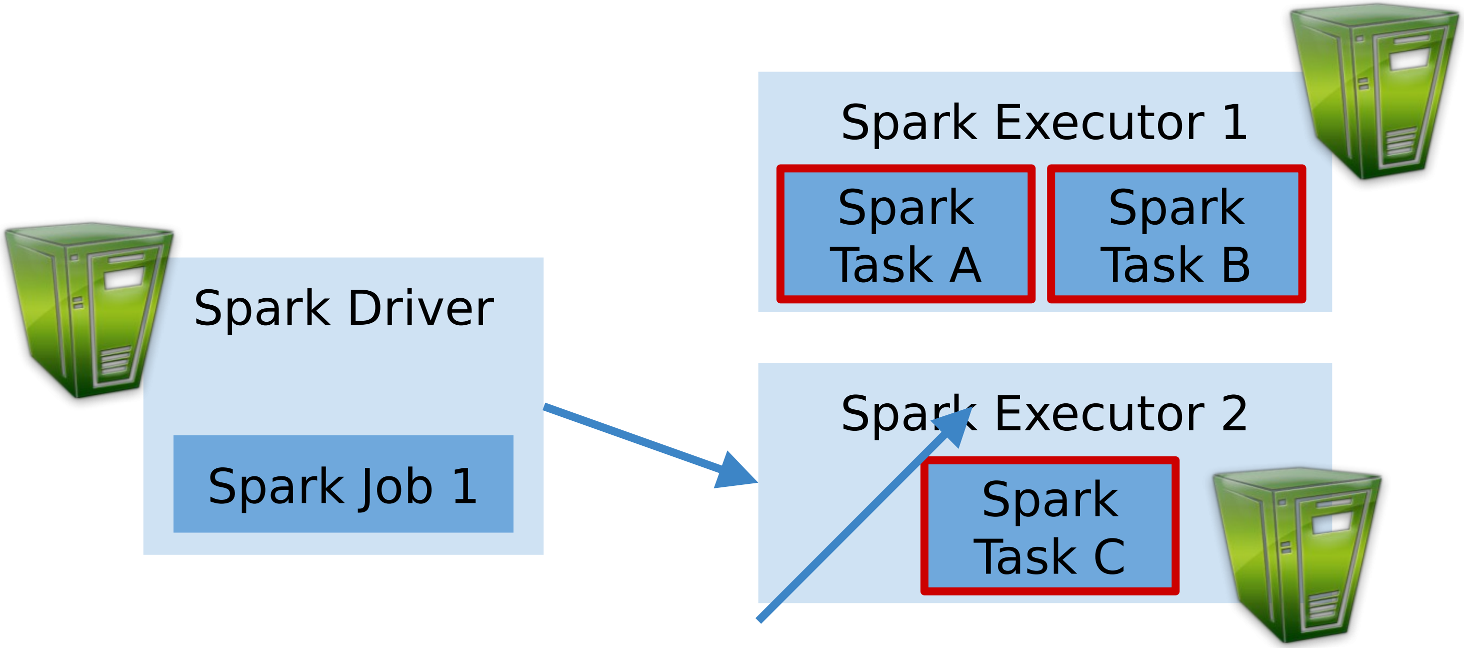
Spark Executor

- Runs assigned tasks
- One executor can run multiple tasks concurrently
- Caches data on disk or in-memory



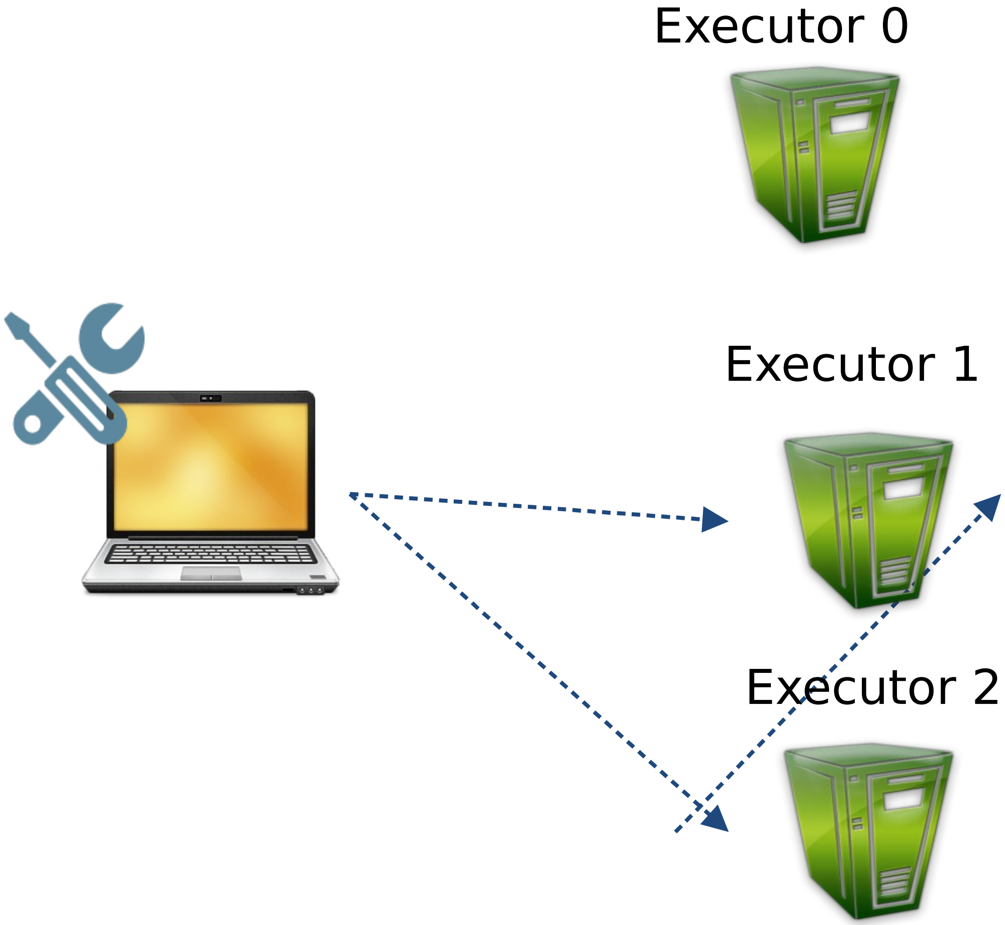
Spark Task

- **Smallest unit of work**
- **Reads input from storage/other RDD/shuffle output**
- **Performs operation e.g. filter, count**
- **Writes output to driver/storage/shuffle**



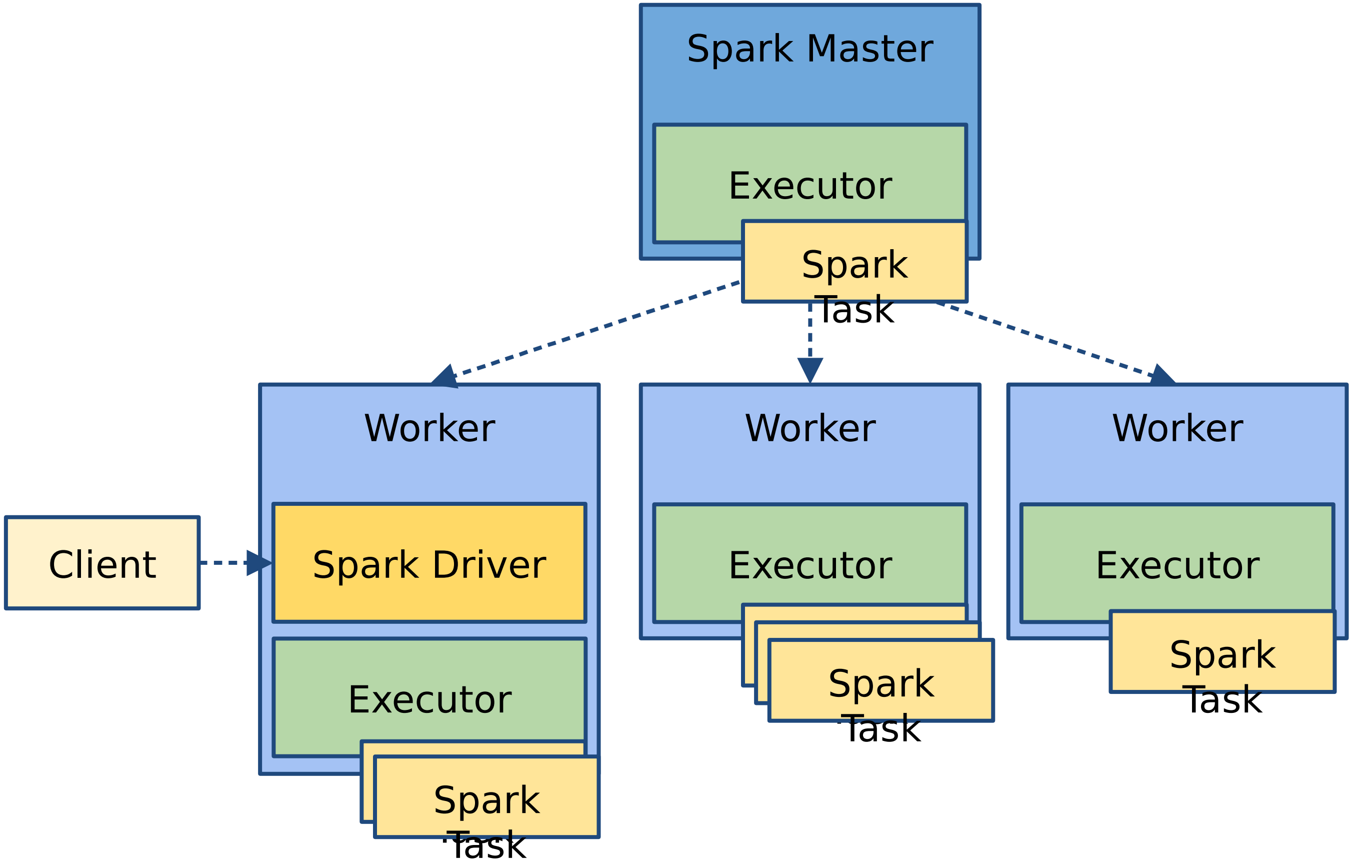
Spark Execution Modes

- **Local**
- **Standalone**
- **Mesos**
- **YARN**

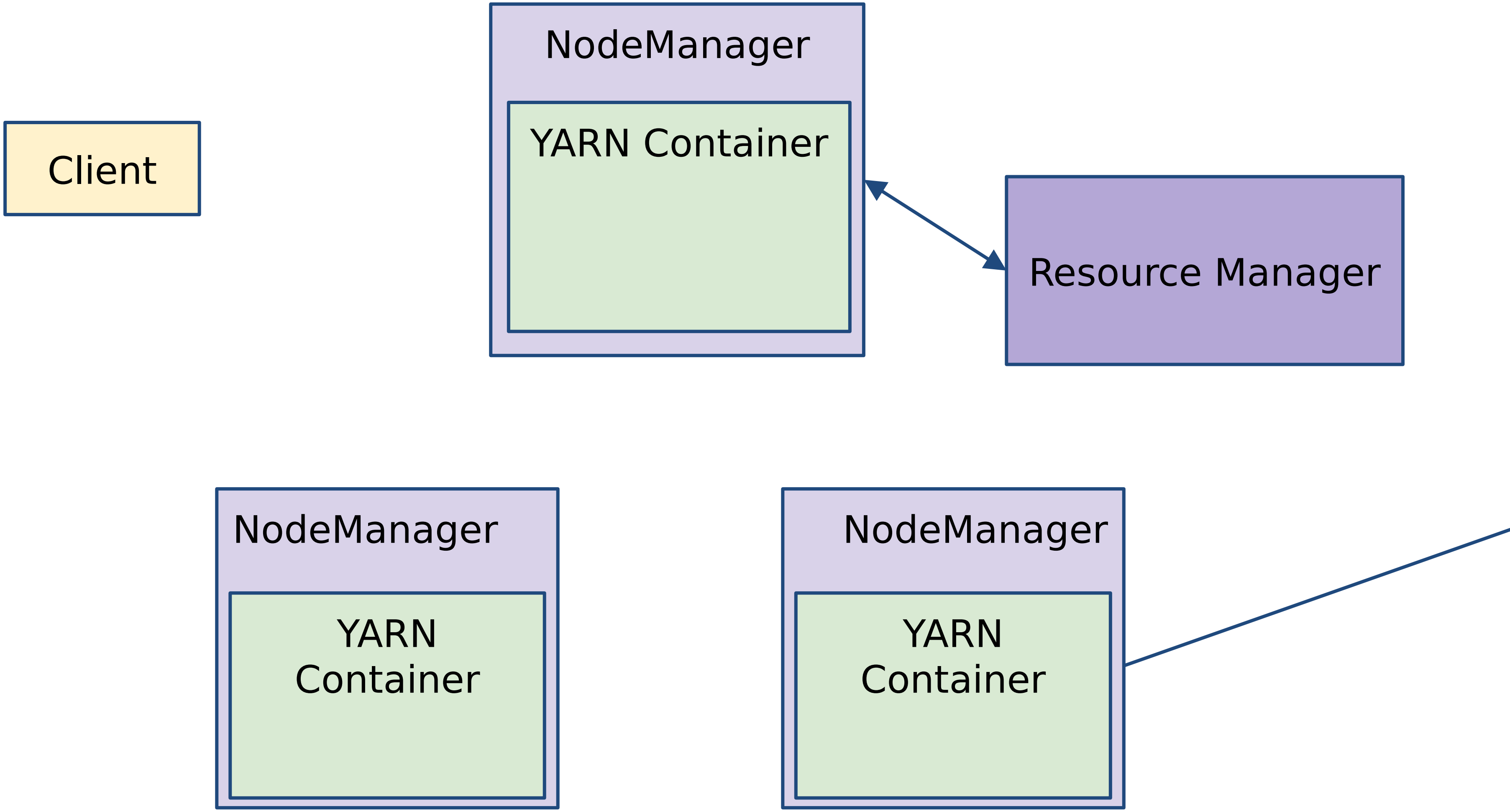


Spark Standalone

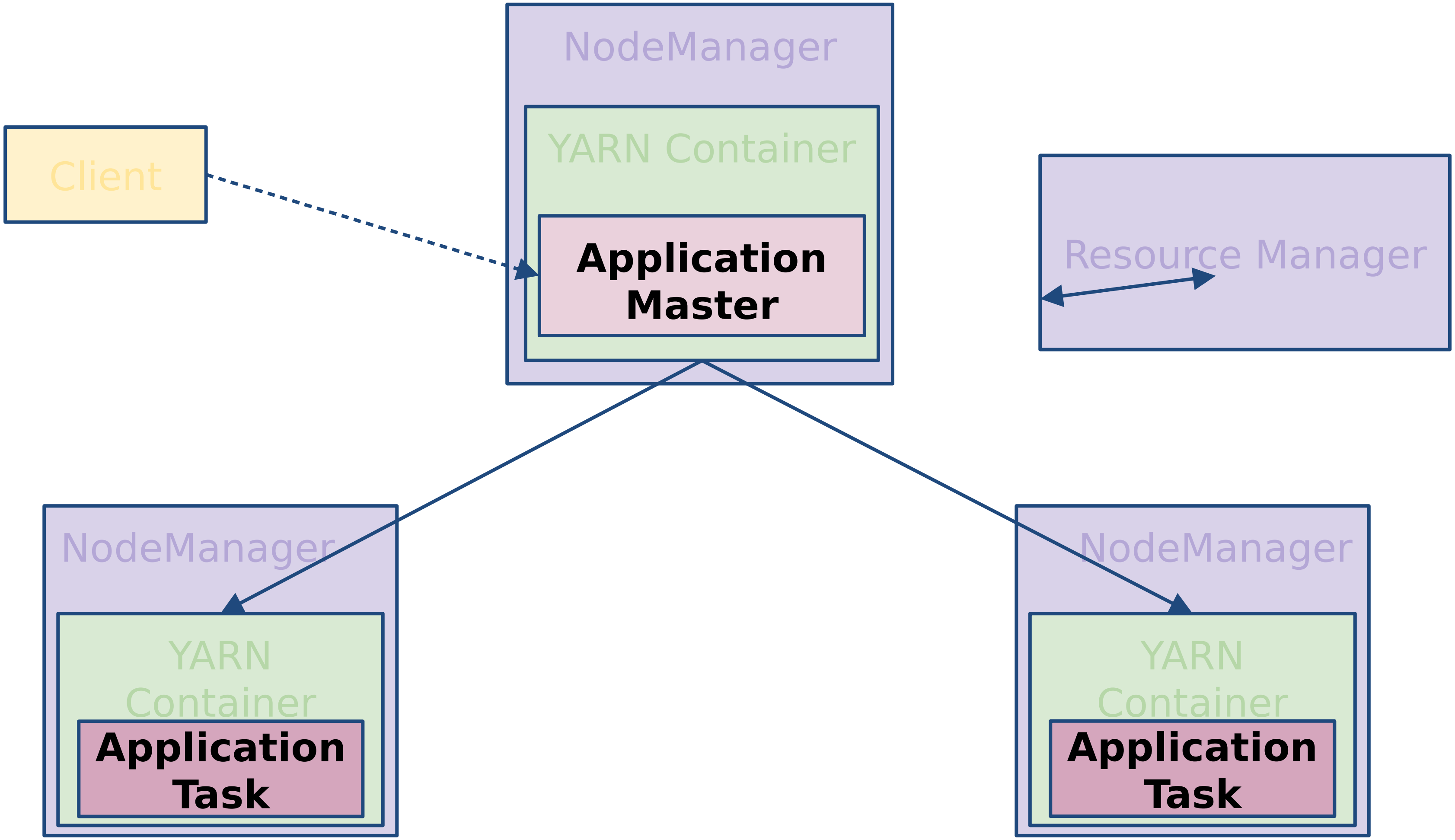
- **Spark Master and Spark Workers**
- **FIFO scheduling**



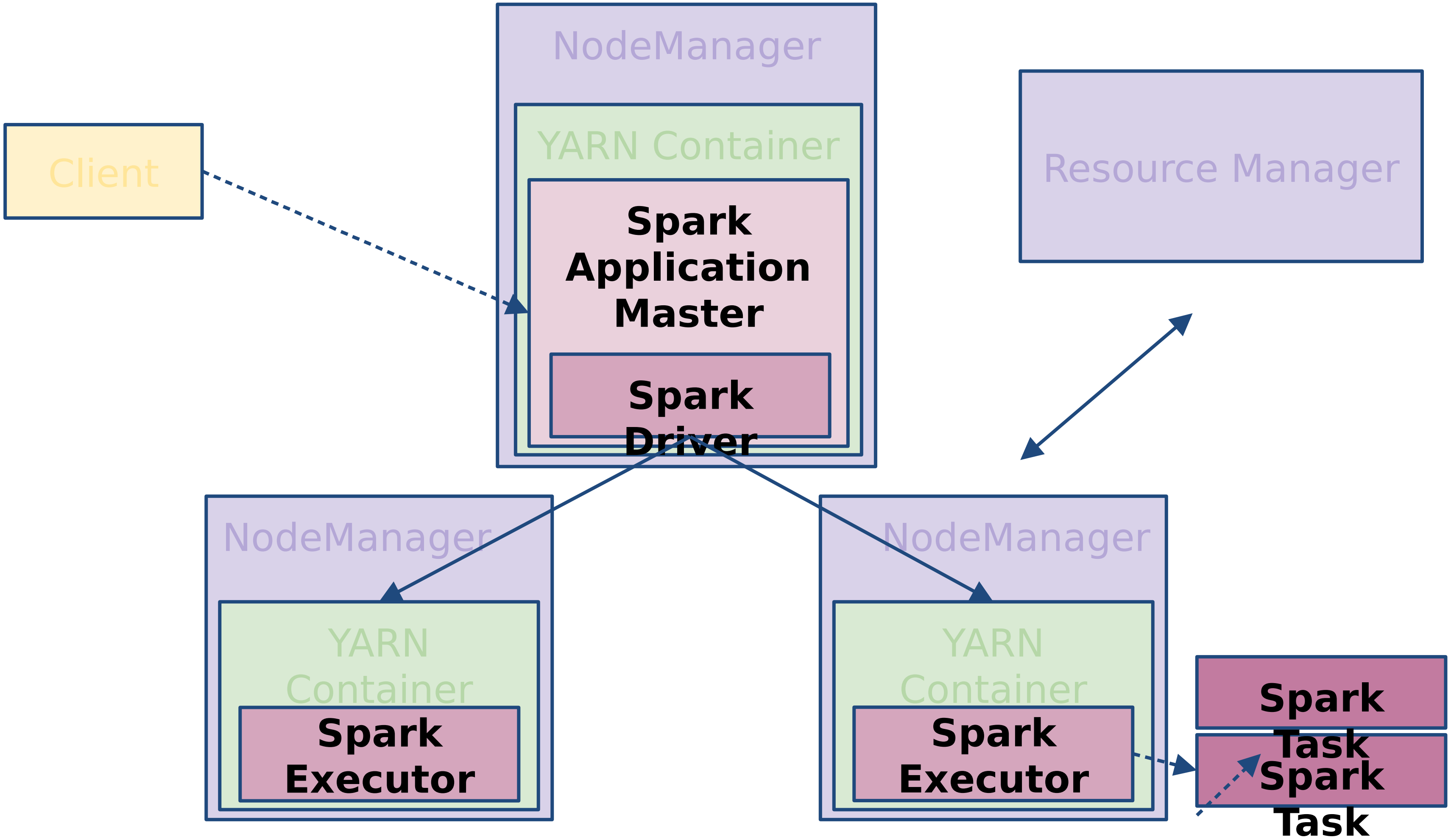
(Simplified) YARN Architecture



YARN Application



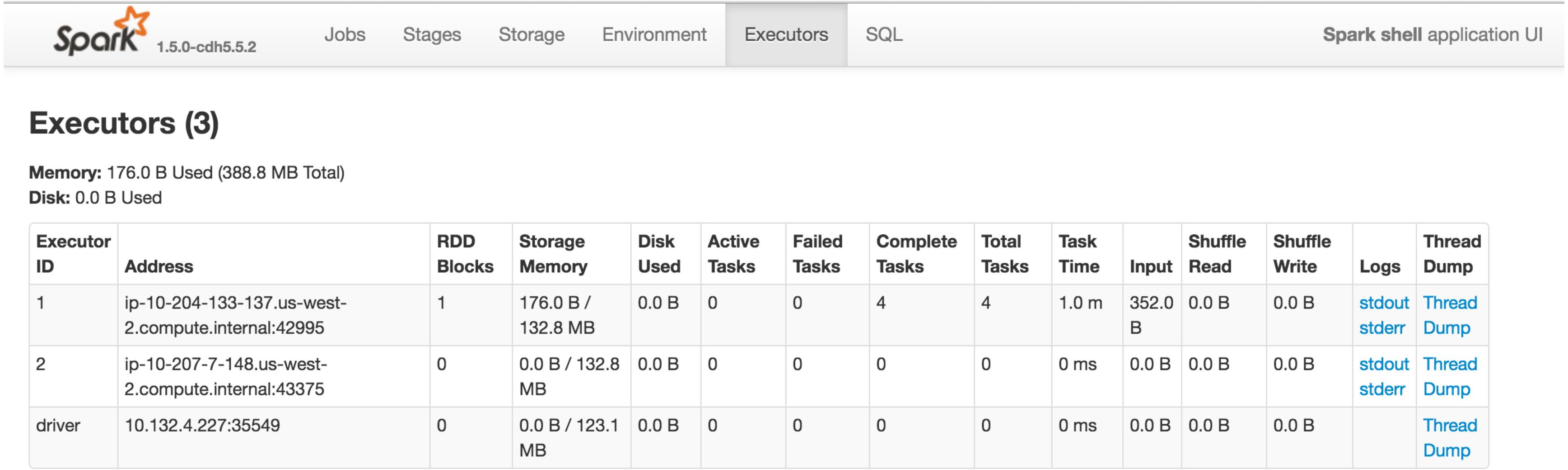
Spark On YARN



Spark On YARN

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master yarn-cluster \
    --num-executors 3 \
    --driver-memory 4g \
    --executor-memory 2g \
    --executor-cores 1 \
    --queue testing \
    lib/spark-examples*.jar \

    10
```

Spark Web UI - Executors

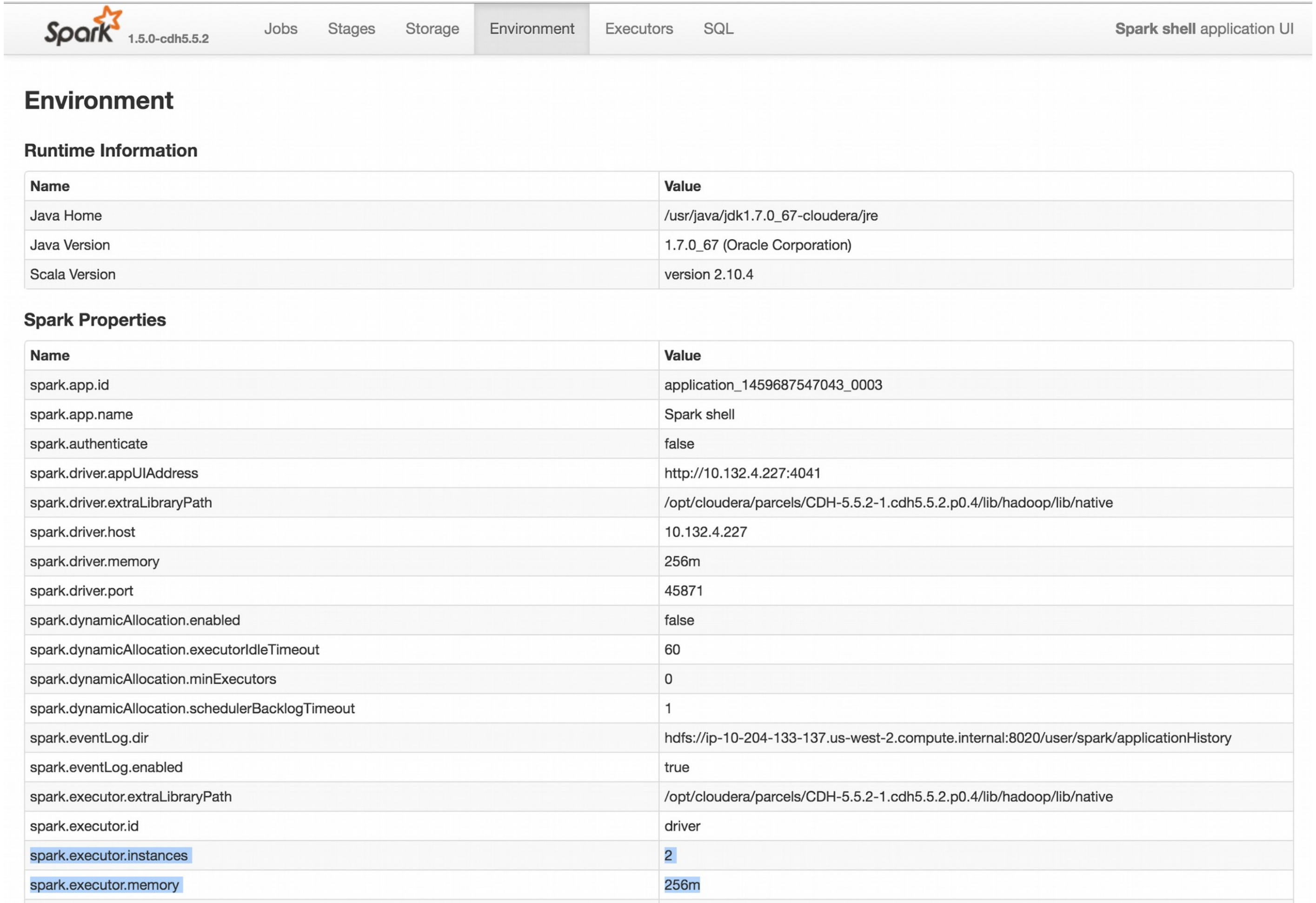


Executors (3)

Memory: 176.0 B Used (388.8 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
1	ip-10-204-133-137.us-west-2.compute.internal:42995	1	176.0 B / 132.8 MB	0.0 B	0	0	4	4	1.0 m	352.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
2	ip-10-207-7-148.us-west-2.compute.internal:43375	0	0.0 B / 132.8 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
driver	10.132.4.227:35549	0	0.0 B / 123.1 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

Spark Web UI - Configuration



Environment

Runtime Information

Name	Value
Java Home	/usr/java/jdk1.7.0_67-cloudera/jre
Java Version	1.7.0_67 (Oracle Corporation)
Scala Version	version 2.10.4

Spark Properties

Name	Value
spark.app.id	application_1459687547043_0003
spark.app.name	Spark shell
spark.authenticate	false
spark.driver.appUIAddress	http://10.132.4.227:4041
spark.driver.extraLibraryPath	/opt/cloudera/parcels/CDH-5.5.2-1.cdh5.5.2.p0.4/lib/hadoop/lib/native
spark.driver.host	10.132.4.227
spark.driver.memory	256m
spark.driver.port	45871
spark.dynamicAllocation.enabled	false
spark.dynamicAllocation.executorIdleTimeout	60
spark.dynamicAllocation.minExecutors	0
spark.dynamicAllocation.schedulerBacklogTimeout	1
spark.eventLog.dir	hdfs://ip-10-204-133-137.us-west-2.compute.internal:8020/user/spark/applicationHistory
spark.eventLog.enabled	true
spark.executor.extraLibraryPath	/opt/cloudera/parcels/CDH-5.5.2-1.cdh5.5.2.p0.4/lib/hadoop/lib/native
spark.executor.id	driver
spark.executor.instances	2
spark.executor.memory	256m

Deploy Modes

- **The cluster mode**
  - The driver runs inside the AppMaster on the cluster
  - The client can go away after initiating the application
- **The client mode**
  - The driver runs in the client process on the client machine
  - The AppMaster is only used for requesting resources from YARN
- **The cluster mode is more scalable**

Fault Tolerance - Driver

- **What happens when the Driver is killed in YARN?**

Fault Tolerance - Driver (Con’t)

- **What happens when the Driver is killed in YARN?**
  - In the cluster mode, it will be re-executed once by default

Because the Driver lives in AM, which can be restarted by YARN

  - In the client mode, the Driver is not re-executed

Fault Tolerance - Executor

- **What happens when a Spark executor is killed?**

Fault Tolerance - Executor (Con’t)

- **What happens when a Spark executor is killed?**
  - Executor lives in a YARN container
  - It will be restarted at most 4 times (by default)

Fault Tolerance - Task

- **What happens when a Spark task is killed?**
  - It will be restarted in some other executor

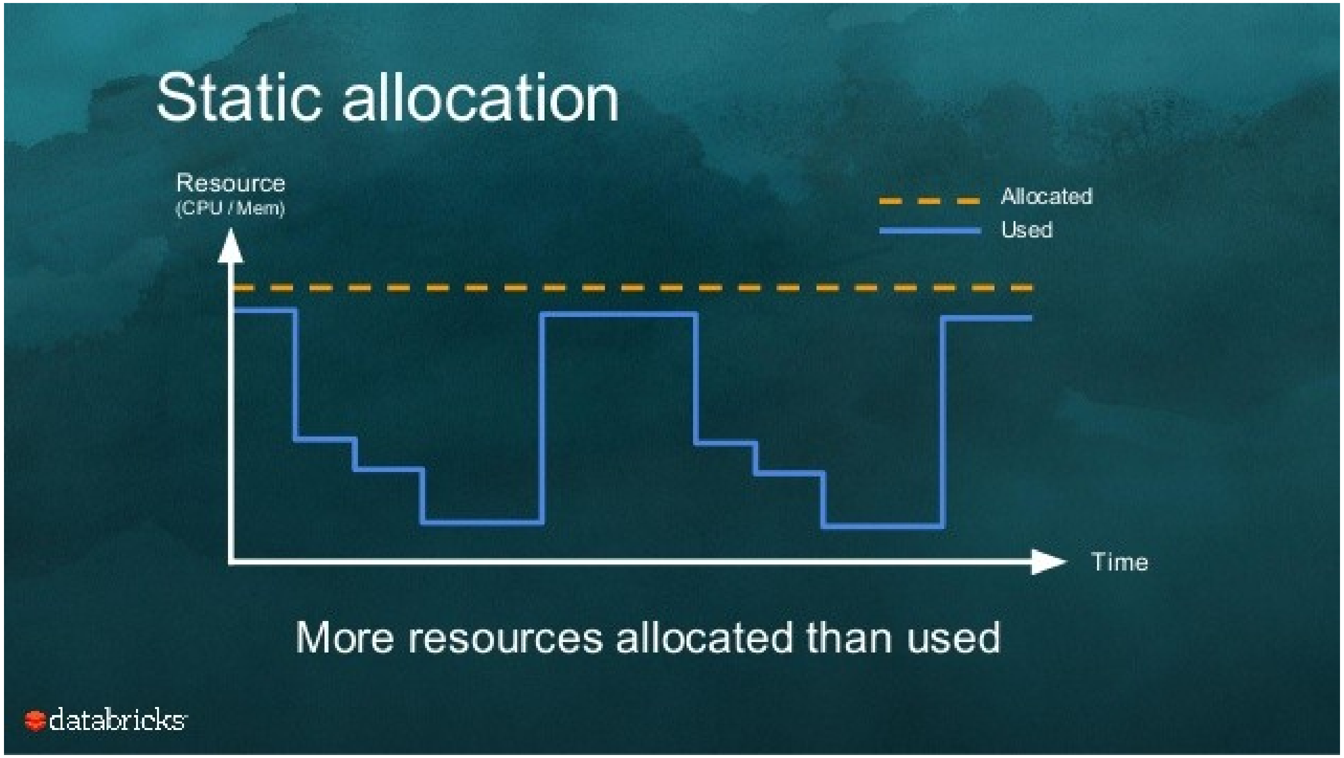
Resource Allocation

```
$ ./bin/spark-submit --class
org.apache.spark.examples.SparkPi \
   --master yarn-cluster \

   --driver-memory 4g \

   --num-executors 3 \

   --executor-memory 2g \

   --executor-cores 1 \

   --queue testing \

   lib/spark-examples*.jar \

   10
```

Is the number of executors good enough?

Cons of Static Allocation

- **Underutilization of cluster resources**
- **Starvation of the other applications**

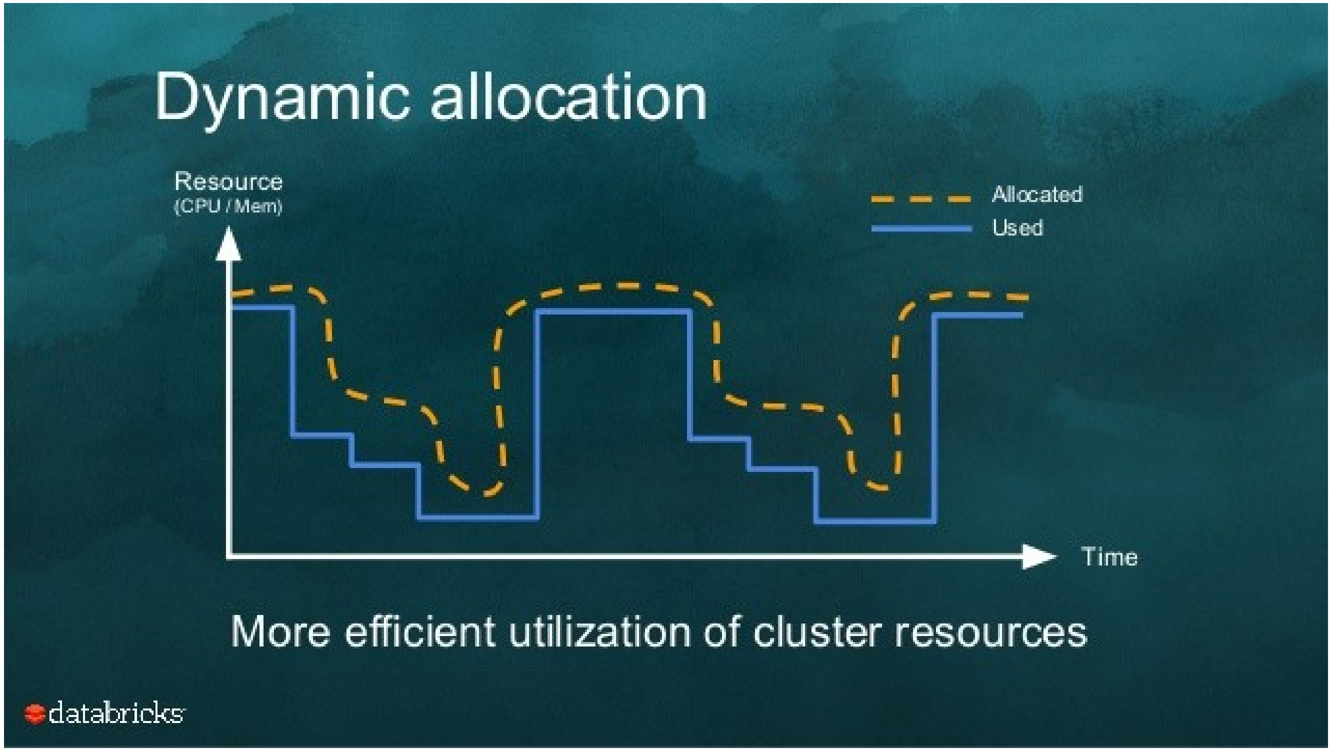


Static allocation

More resources allocated than used

Dynamic Allocation

- **Dynamically adjusting resources consumed by an application based on the current workload**



Dynamic Allocation In Spark

- **Introduced in Spark 1.2**
- **Simple idea**
  - If executors are idle, remove them
  - If we need more executors, request them
  - A set of heuristics to scale up and scale down
- **Configurable settings**
  - Initial number of executors
  - Minimum and maximum number of executors

Request Policy

- **Requests additional executors when it has pending tasks**
- **Spark requests executors in rounds**
  - In each round, the number of requested executors increases exponentially

One executor in the 1st round

Two, four and so on executors in the subsequent rounds

- **Design goals**
  - “Slow start" in case we only need a few more executors
  - An exponential growth in case we need many executors

Remove Policy

- **Removes an executor when it has been idle for a while**
  - By default, for more than a minute (configurable)
- **State associated with the executor might be still needed!**
  - Shuffle state

Written to external shuffle service, so it still can be accessed even if the executor is removed

  - Cached data either on disk or in memory

Unfortunately, all cached data will no be longer accessible when its executor is removed

In future releases, it may be preserved through an off-heap storage - similar to external shuffle service

Resource Allocation

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi
\
   --master yarn-cluster \
   --driver-memory 4g \
   --num-executors 3 \
   --executor-memory 2g \
   --executor-cores 1 \
   --queue testing \
   lib/spark-examples*.jar \
   10
```

Is the size of executors good enough?

Executor’s Cores & Memory

- **Too much memory per executor?**
  - Wasteful memory usage
- **Too little memory per executor?**
  - Not using benefits of running multiple tasks in same JVM

e.g. caching

- **Too many cores per executor?**

Recommendation For Cores & Memory

- **Best is to keep under 4-5 cores per executor**
  - An executor will run up to 4-5 tasks in parallel
- **Enough memory for multiple tasks in executor to run**

Chapter

Good Practices
& Advanced Features

Broadcast Variables

- **Optimization of sharing data between driver and tasks**
- **Read-only variable cached on each executor**
- **Distributed with an efficient p2p protocol**

```
val dict: Map[String, Int] = some huge map
val broadcasted = sc.broadcast(dict)


rdd.map(x => broadcasted.value.get(x))
```

Accumulator

- **Variable that are only added to through an associative operation**
  - Tasks can’t read its value
- **Can be used to implement counters, metrics, sums etc.**
- **Can be tracked in Spark UI (name is required)**

```
valaccum = sc.accumulator(0,“Parse Errors”)

sc.parallelize(Arrays.asList(1,2,3)).foreach(x => accum.add(x))

accum.value()

10
```

GroupBy vs. ReduceByKey

```
val streamCountsWithReduce =
streamPairsRDD
  .reduceByKey(_ + _)
  .collect()
```

```
val streamCountsWithGroup =
streamPairsRDD
  .groupByKey()
  .map(t => (t._1, t._2.sum))
  .collect()
```

- **Two different ways, but both generate correct results**
- **Which one works better on larger scale?**

Avoid GroupBy

- **reduceByKey**
  - Data is combined (pre-aggregated) locally
  - Each partition outputs at most one value for each key to send over the network
- **groupByKey**
  - All the data is (wastefully) sent over the network
  - All data is collected and aggregated by the reduce tasks
  - reduceByKey, aggregateByKey, foldByKey and combineByKey are preferred over groupByKey

Copying Elements To The Driver

- **`collect()` attempts to copy every single element in the RDD onto the driver program**
  - If RDD is too large, it might run out of memory and crash!
- **Similarly, OOM-error can occur when results of other actions are too large**
  - e.g. `countByKey`, `countByValue`, `collectAsMap`

Don’t Copy Large RDD To The Driver

- **Consider**
  - Using take(n)
  - Filtering RDD
  - Sampling your RDD
  - Writing out the RDD to files in HDFS
  - Exporting the RDD to a database that is large enough to hold all the data

Spark Serializer

- **Defines class to use for serializing objects into a binary format when …**
  - … transferring data over the network
  - … caching in serialized form
  - … spilling data to disk
- **The default is Java serialization**
  - Works with any Serializable Java object
  - But it’s quite slow

Kryo Spark Serializer

- **When speed is necessary, it's advised to use KryoSerializer**
  - Almost all applications will benefit from shifting to Kryo

Roughly ~10 times faster than Java serializer

- **Kryo might not serialize all types of objects "out of the box"**
- **Configured with `spark.serializer`**

Error: Task not serializable

- **If you see a following error:**

```
org.apache.spark.SparkException: Job aborted due to stage
failure: Task not serializable:
java.io.NotSerializableException: ...
```

- **use it on one of the Workers**
  - … for example in `map()` function

```
rdd.map(x -> nonSerializable.doIt(x)).collect()
```

Solutions: Task Not Serializable

- **Make your class *Serializable* (obviously!)**
  - Your object will be re-sent to every task
- **… or declare the instance locally inside the lambda function passed to `map()`**
  - It’s fine unless object creation is time-consuming as `nonSerializable` will be created for every input record

```
rdd.map(x -> {
val nonSerializable = new NonSerializable()
nonSerializable.doIt(x) }
).collect()
```

Solutions: Task Not Serializable (Cont’d)

- **… or call `rdd.forEachPartition()` or `mapPartitions()` and create the `NotSerializable` object in there**
  - A bit better, because `mapPartitions` is invoked on a whole partition of data (lambda takes an `Iterator`):

```
rdd.mapPartitions{cpartitionOfElements ->
valnonSerializable = new NonSerializable()
   for(elem in partitionsOfElements){
     elem.doSomethingWith(nonSerializable)
}}
```

Solutions: Task Not Serializable (Cont’d)

- **… or wrap your `NotSerializable` class instance in a object with lazy field**
  - Lazy field will be instantiated when first used (hopefully in the executor)

```
object MyWrapper {
  lazy val notSerializable = new NotSerializable()
}
```

Bonus Exercise

Spark Aggregations and Sorting

http://bit.ly/1Sh34jS

Pages 17 - 21

Quiz

Spark Core

http://bit.ly/1IpKQLE

Q&A

Chapter

Backup Slides

100TB Sort Contest

	Hadoop MapReduce	Spark
Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400 physical	6592 virtualized
Cluster Disk Throughput	3150 GB/s (est.)	618 GB/s
Sort rate	1.42 TB/min	4.27 TB/min
Sort rate/node	0.67 GB/min	20.7 GB/min

http://databricks.com/blog

OLD Exercise

Spark API - First Look And Feel

http://bit.ly/1MGdl1s

OLD Exercise

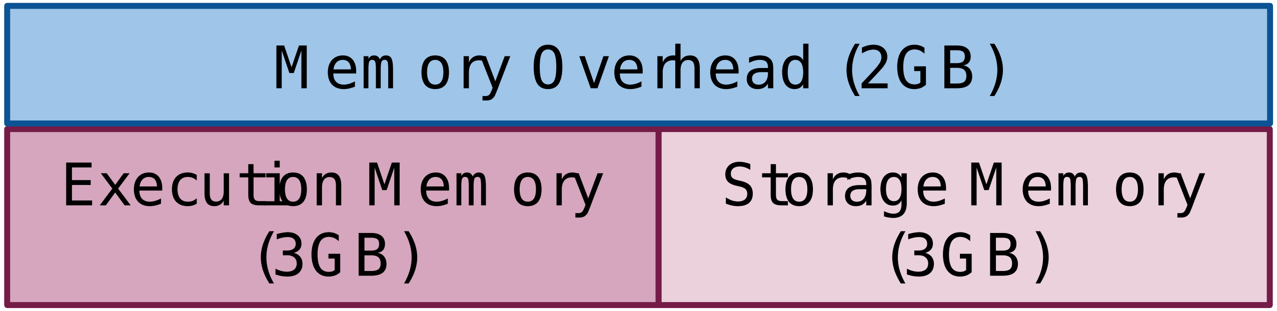
Parallelization And Caching

http://bit.ly/1MGdl1s

Spark Container Memory (Since 1.6)

- **Our example:**
  - 8 GB for a Spark container
  - Default configuration settings for memory management

```
spark.memory.fraction = 0.75
spark.memory.storageFraction = 0.5
```



Memory Overhead (2GB)

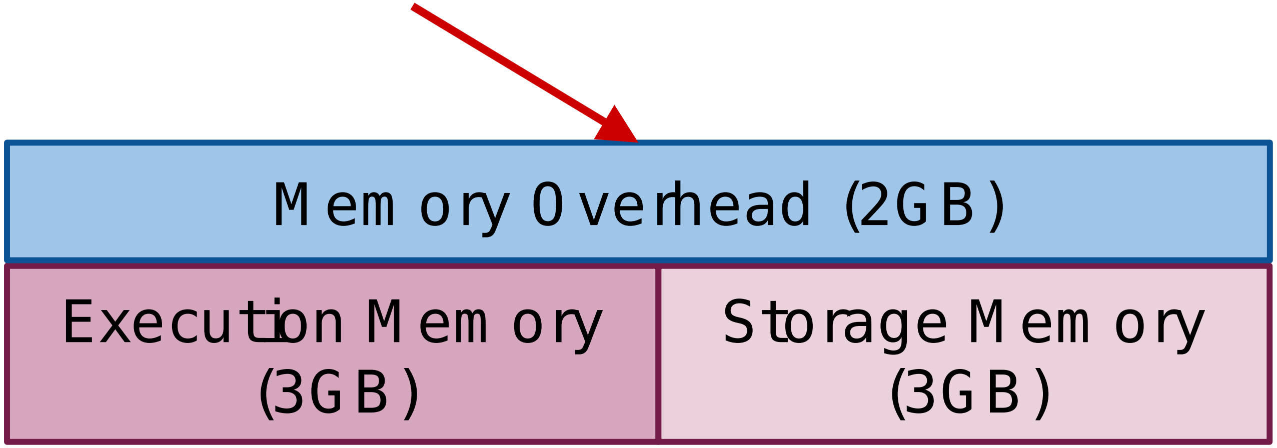
Execution Memory (3GB)

Storage Memory (3GB)

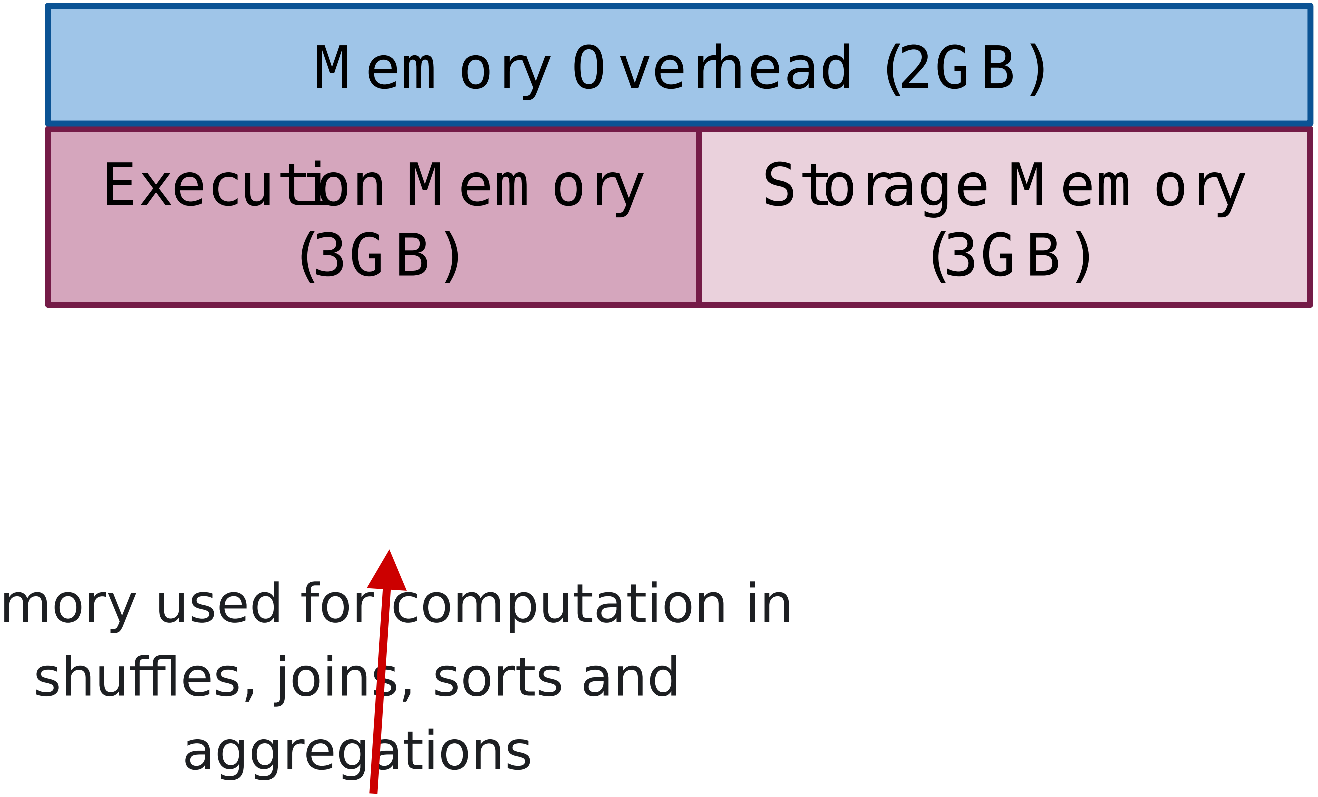
Memory Overhead (Since 1.6)

Reserved for:

- user data structures
- internal metadata in Spark
- safeguarding against OOM errors in the case of sparse and unusually large records



Execution Memory (Since 1.6)



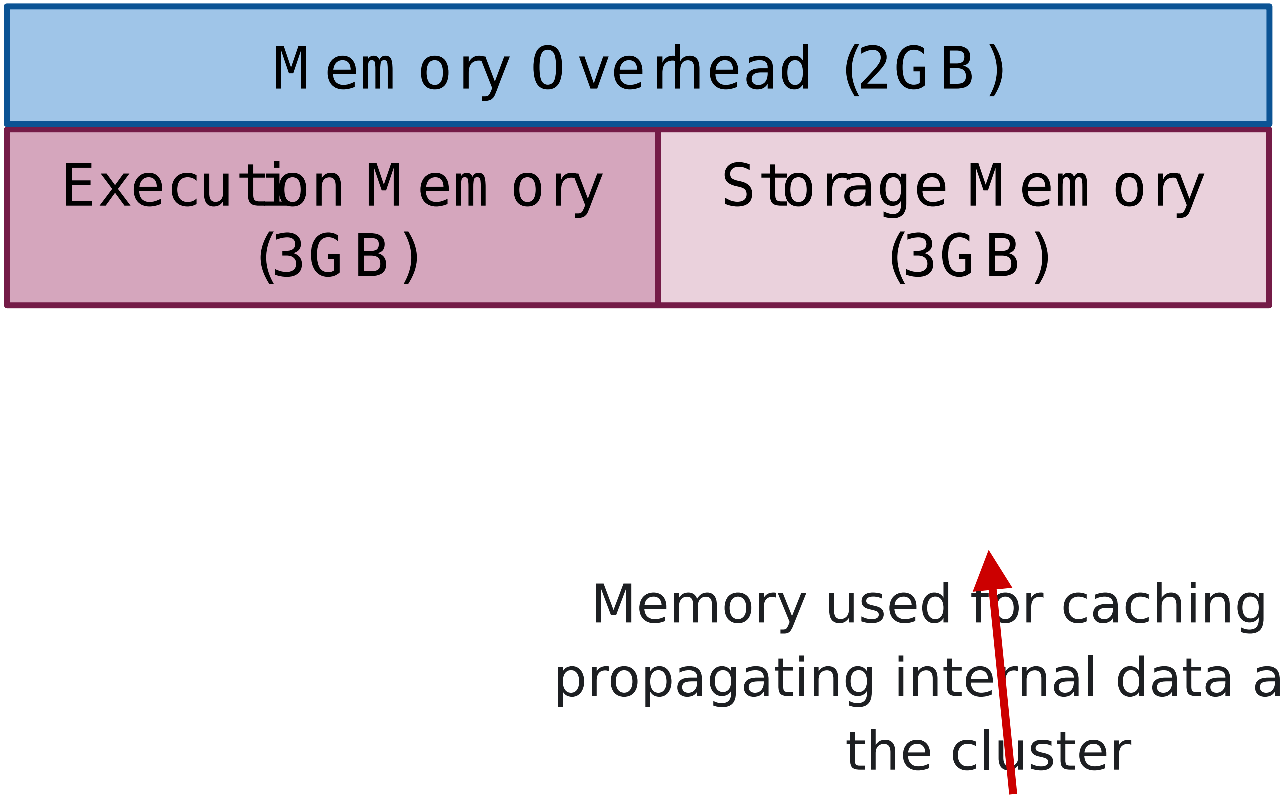
Memory Overhead (2GB)

Execution Memory (3GB)

Storage Memory (3GB)

Memory used for computation in shuffles, joins, sorts and aggregations

Execution Memory (Since 1.6)



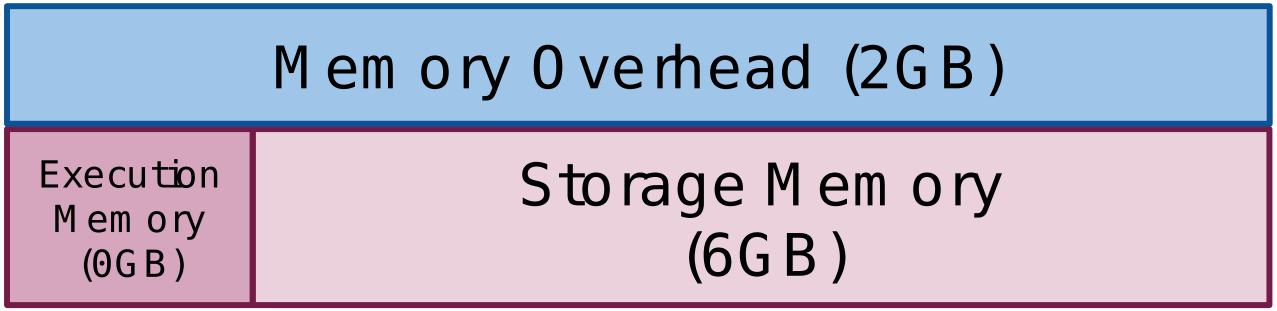
Memory Overhead (2GB)

Execution Memory (3GB)

Storage Memory (3GB)

Memory used for caching and propagating internal data across the cluster

Expanding Storage Memory (Since 1.6)



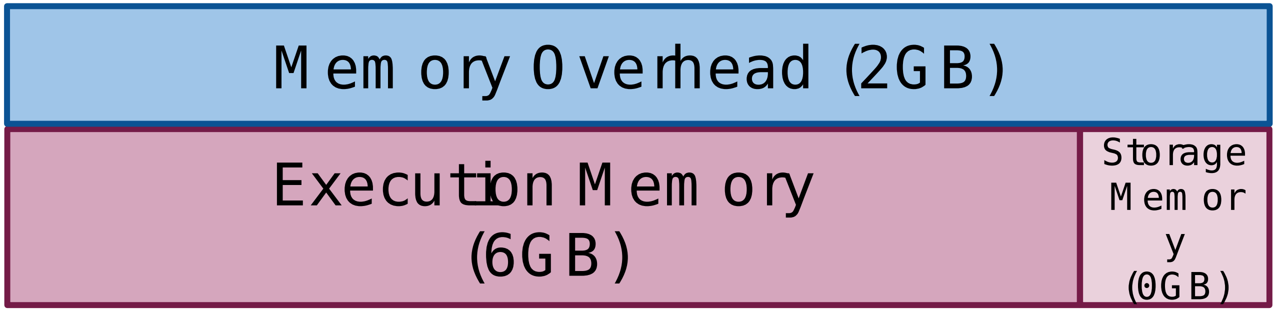
Memory Overhead (2GB)

Execution Memory (0GB)

Storage Memory (6GB)

When no execution memory is used, storage can acquire all the available memory (e.g. for caching).

Expanding Execution Mem. (Since 1.6)

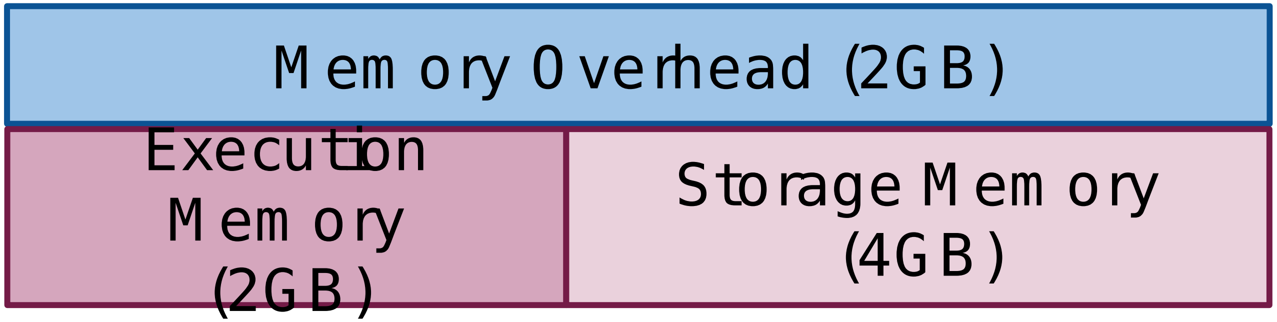

Memory Overhead (2GB)

Execution Memory (6GB)

Storage Memory (0GB)

Applications that don't use caching can use the entire space for execution (avoiding unnecessary disk spills)

Evicting From Execution (Since 1.6)



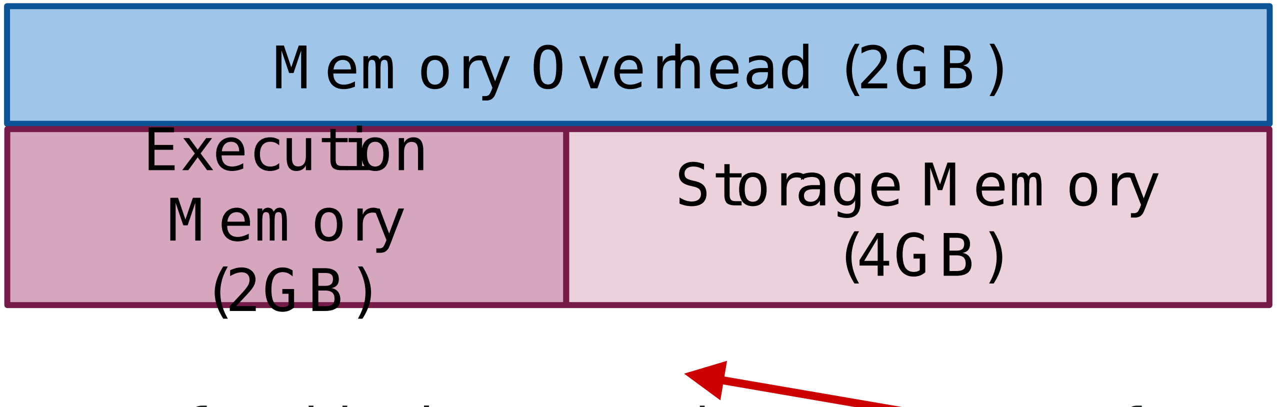
Memory Overhead (2GB)

Execution Memory (2GB)

Storage Memory (4GB)

If suddenly we need more memory for storage and everything else is allocated to eviction,
storage may not evict execution due to complexities in implementation!

Evicting From Storage Mem (Since 1.6)



Memory Overhead (2GB)

Execution Memory (2GB)

Storage Memory (4GB)

If suddenly we need more memory for execution and everything else is allocated to storage,
execution may evict execution only until total storage memory usage falls under a certain threshold (default 50%)

Missing Dependencies In JAR Files

- **By default, Maven does NOT include dependency JARs when it builds a target**
  - The easiest workaround is to create a ***shaded*** or ***uber*** jar to package all dependencies in the jar as well
  - Spark dependencies should be marked as provided since they are already on the Spark cluster

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.10</artifactId>
    <version>1.5.1</version>
    <scope>provided</scope>
</dependency>
```

Missing Dependencies In JAR Files (cont’d)

- **To create uber jar you can use *maven-shade-plugin***

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>2.3</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        …
    </configuration>
</plugin>
```

Shading

- **If you see this error (and you included all deps in your app):**

```
Exception in thread "main": java.lang.NoSuchMethodError:
com.google.common ...
```

- ***Sometimes* it means that your Guava version does not match with Spark's Guava version :(**
  - maven-shade-plugin to the rescue again!

```
<relocation>
    <pattern>com.google.common</pattern>

<shadedPattern>com.mycompany.google.common</shadedPattern
>
</relocation>
```

Easy Processing Of Big Data

- **How to easily implement a distributed application**
  - ... that processes terabytes of data?
  - ... that runs on tens or thousands of machines?

Popular Solutions

- **Pig, Hive**
- **Scalding, Crunch**
- **Spark, Flink**