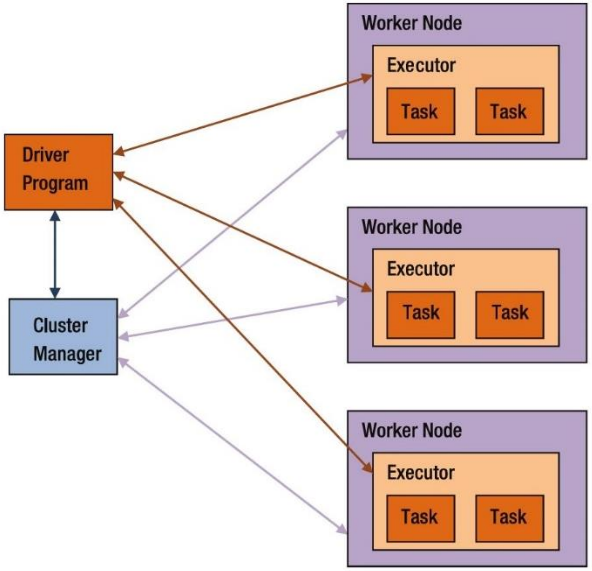
CSC 735 – Data Analytics

Chapter 15

How Spark Runs on a Cluster

The Architecture of a Spark Application

- A Spark application consists of:
  1. driver program
  2. a cluster manager
  3. workers
  4. executors, and
  5. tasks



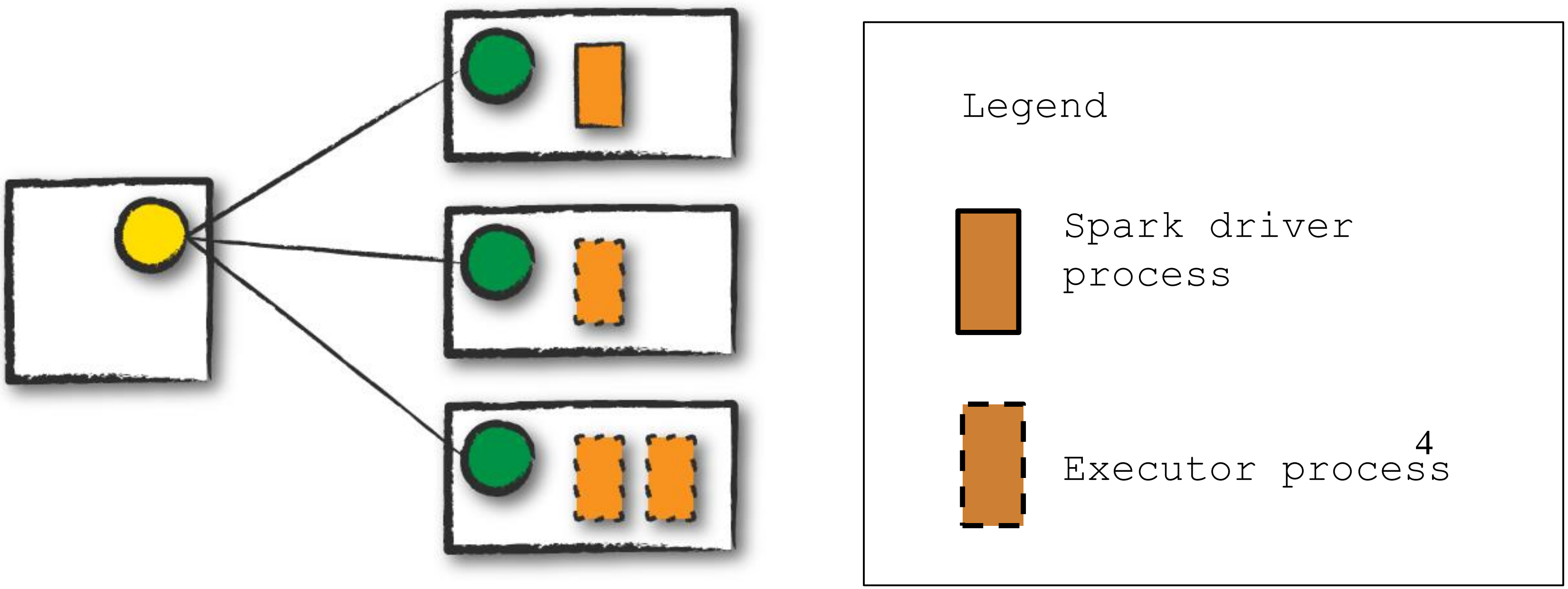
Guller, Mohammed. *Big data analytics with Spark: A practitioner's guide to using Spark for large scale data analysis*. Apress, 2015

Application Execution Modes

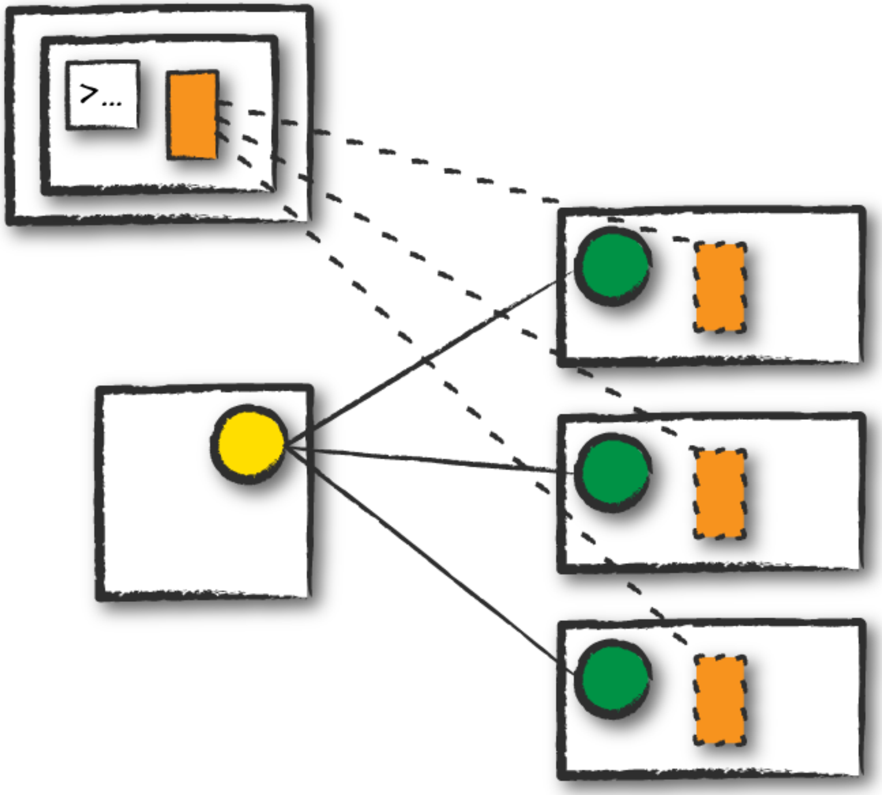
- An application can run in one of three execution modes

1. Cluster mode
2. Client mode
3. Local mode

Spark's Cluster Mode

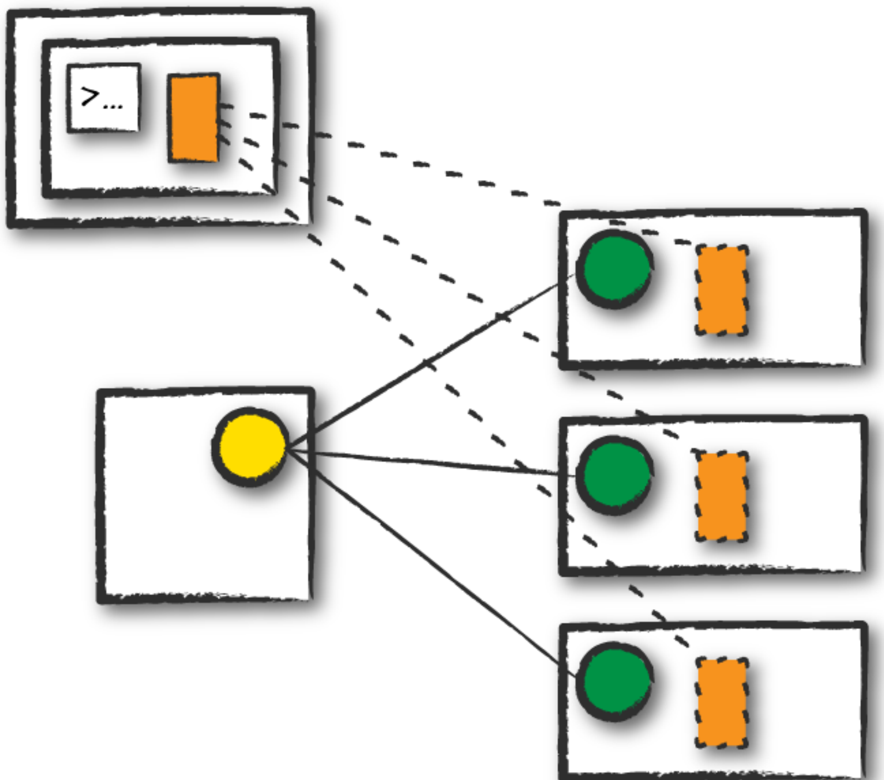


Spark’s Client Mode



Spark's Client Mode

- **gateway machines** (aka **edge nodes**)



Spark's Local Mode

- The entire application runs on a single machine

Spark's Local Mode

- The entire application runs on a single machine
- Threads are used for parallelism

Spark’s Local Mode

- The entire application runs on a single machine
- Threads are used for parallelism
- Good to learn Spark or to test your applications

Spark’s Local Mode

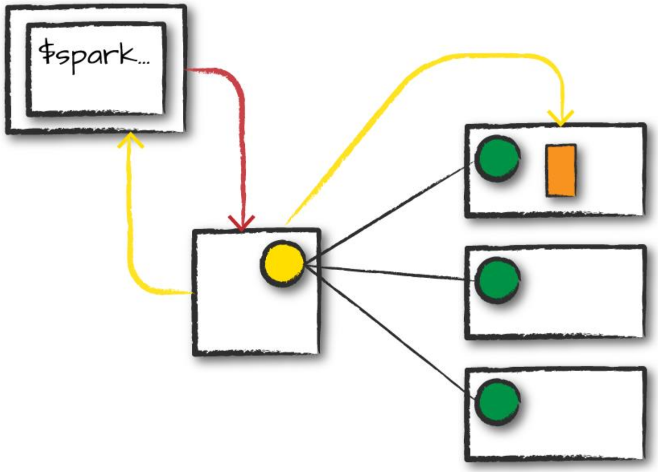
- The entire application runs on a single machine
- Threads are used for parallelism
- Good to learn Spark or to test your applications
- Not recommended for production mode

Spark's Modes

- **Cluster Mode**: the driver runs on one of the worker nodes in the cluster.
  - the typical mode for running Spark applications in production because it ensures fault tolerance and scalability
- **Client Mode**: the driver runs on the machine where the Spark application is submitted.
  - used for debugging because the driver's output is shown in the client console; involve the REPL (e.g. Spark shell).
- **Local Mode**: Spark runs on a single machine as a single JVM (Java Virtual Machine).
  - used for development, debugging, and testing small-scale Spark applications

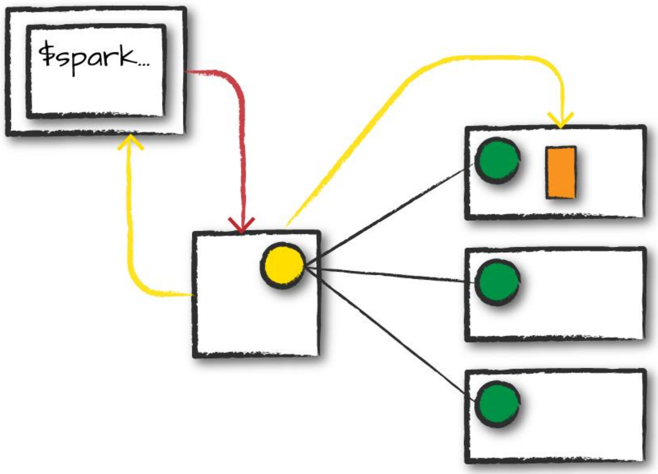
The Life Cycle of a Spark Application (Outside Spark)

- Assume an application is submitted to a cluster consisting of 4 nodes: a cluster manager driver and three worker nodes
- What is the Application's life cycle from initialization until program exit?



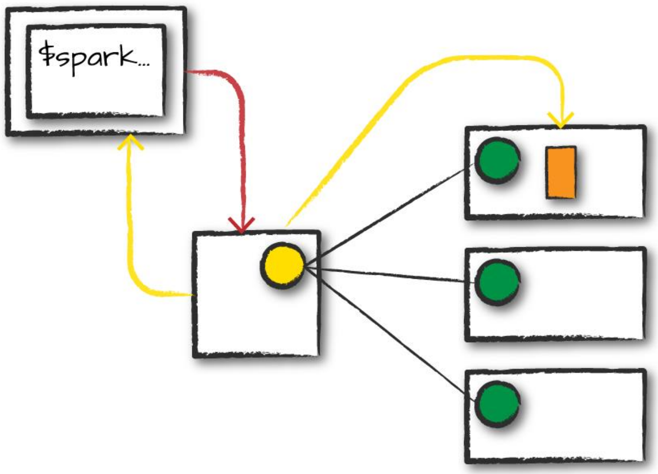
Client Request

- You make a request to the cluster manager driver node to run an application
  - requesting resources for the Spark driver process only



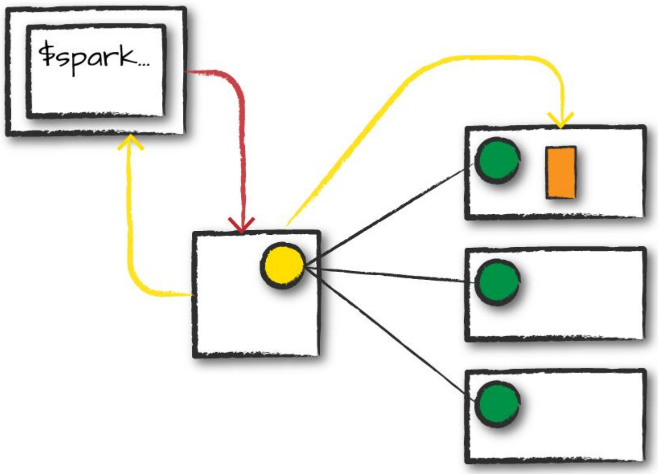
Client Request

- You make a request to the cluster manager driver node to run an application
  - requesting resources for the Spark driver process only
- Cluster manager accepts the request



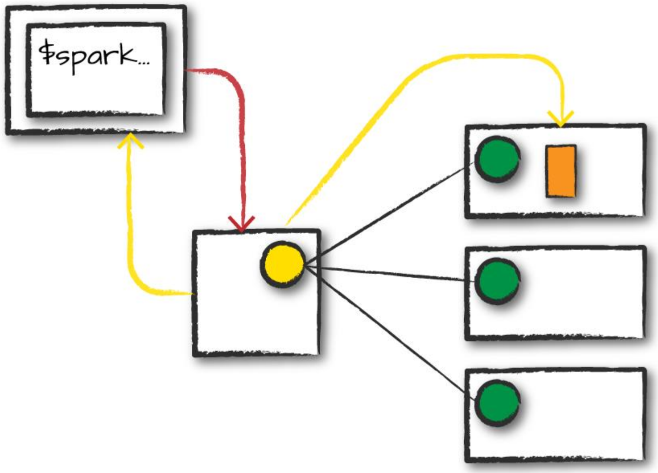
Client Request

- You make a request to the cluster manager driver node to run an application
  - requesting resources for the Spark driver process only
- Cluster manager accepts the request
- It places the Spark driver onto a worker node



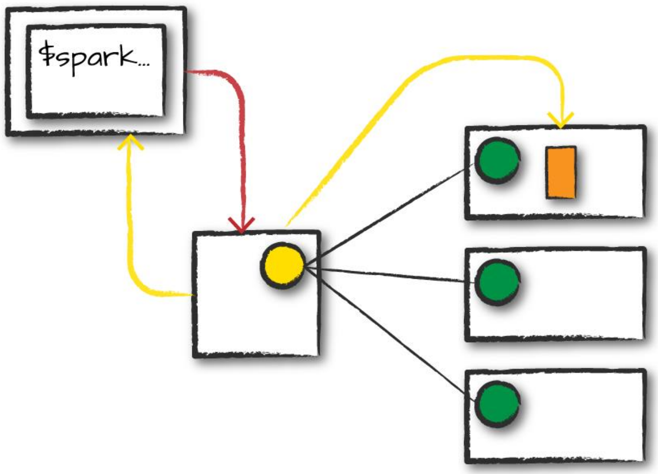
Client Request

- You make a request to the cluster manager driver node to run an application
  - requesting resources for the Spark driver process only
- Cluster manager accepts the request
- It places the Spark driver onto a worker node
- The client process that submitted the original job exits



Client Request

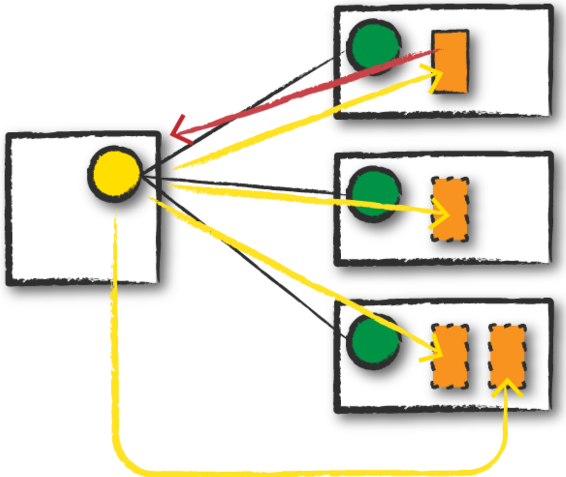
- You make a request to the cluster manager driver node to run an application
  - requesting resources for the Spark driver process only



- Cluster manager accepts the request
- It places the Spark driver onto a worker node
- The client process that submitted the original job exits
- Your application is running on the cluster

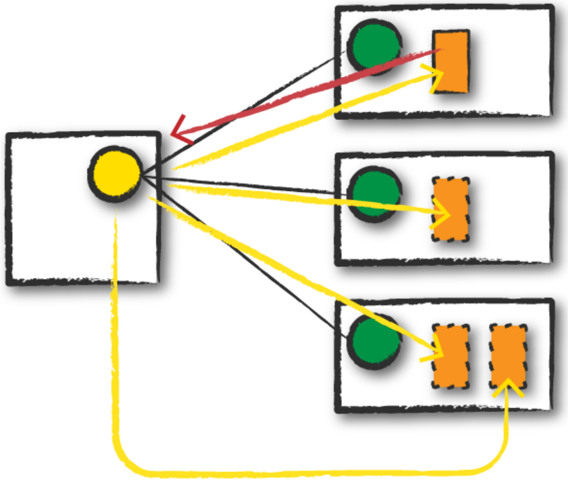
Launch

- driver begins running user code



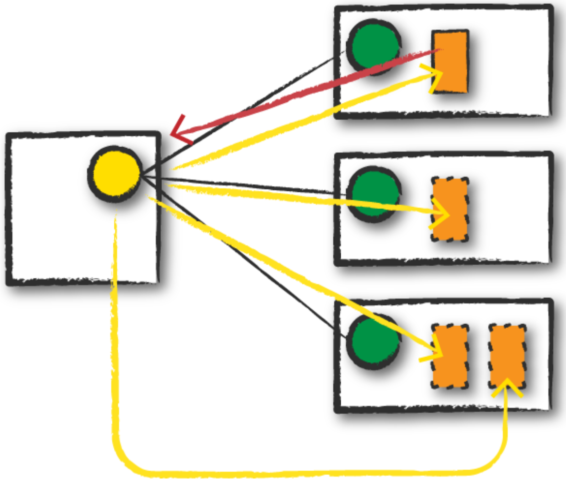
Launch

- driver begins running user code
- code must include a SparkSession



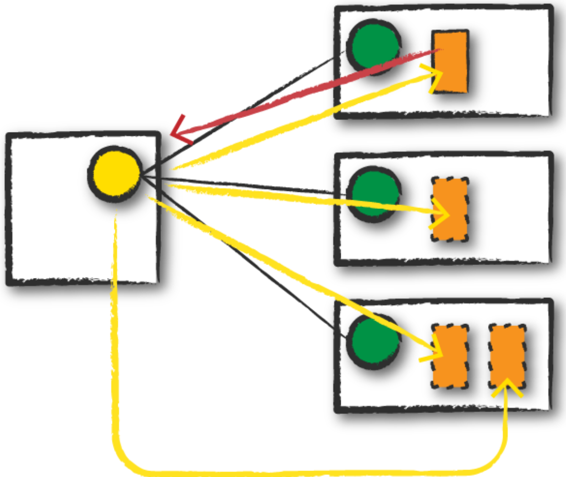
Launch

- driver begins running user code
- code must include a SparkSession
- SparkSession communicates with the cluster manager, asking it to launch executors



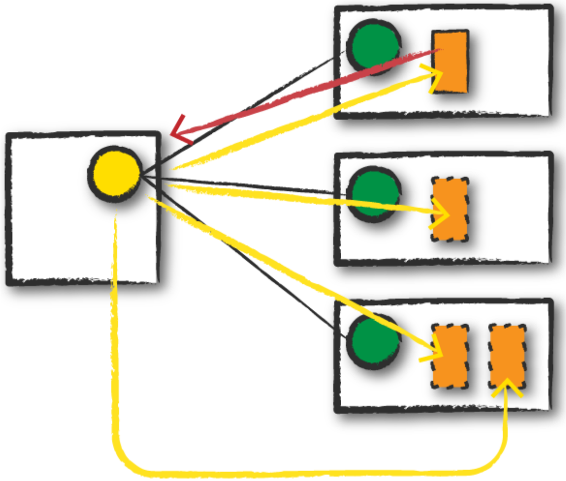
Launch

- driver begins running user code
- code must include a SparkSession
- SparkSession communicates with the cluster manager, asking it to launch executors
- cluster manager launches the executors
- it sends their locations to the Spark driver



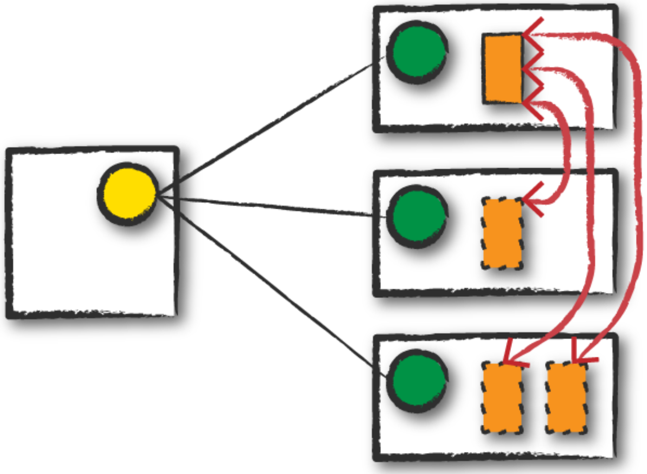
Launch

- driver begins running user code
- code must include a SparkSession
- SparkSession communicates with the cluster manager, asking it to launch executors
- cluster manager launches the executors
- it sends their locations to the Spark driver
- After everything is hooked up correctly, you have a Spark Cluster



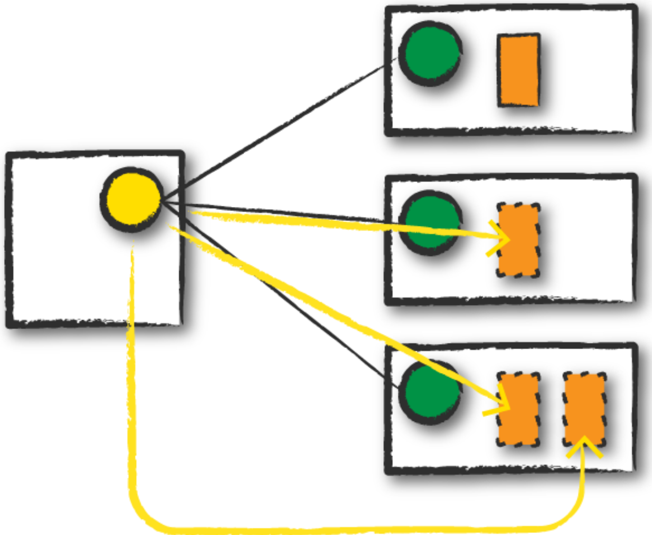
Execution

- Spark executes the code
- The driver and the workers communicate among themselves, executing code and moving data around
  - driver schedules tasks onto the executors
  - each executor responds with the status of those tasks and success or failure



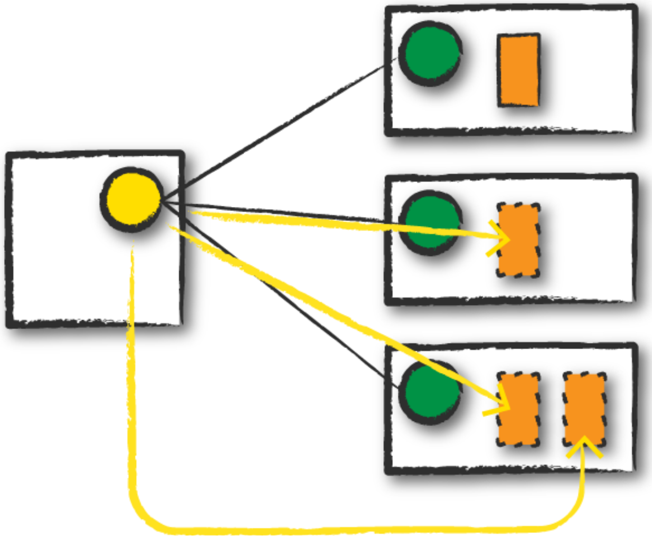
Completion

- After application completes, the driver process exits with either success or failure



Completion

- After application completes, the driver process exits with either success or failure
- The cluster manager shuts down the executors allocated for the driver
- You can see the success or failure of the Spark Application



The Life Cycle of a Spark Application (Inside Spark)

- This is the life cycle of your Spark Application **code**
- It describes what happens inside Spark when we run an application
- Remember, an application consists of one or more Spark jobs

The SparkSession

- SparkSession is the object that represents your entrance point to writing Spark code
- It manages your Spark application

The SparkSession

- SparkSession is the object that represents your entrance point to writing Spark code
- It manages your Spark application
- In interactive mode, a SparkSession object, named **spark**, is created automatically for you

The SparkSession

- SparkSession is the object that represents your entrance point to writing Spark code
- It manages your Spark application
- In interactive mode, a SparkSession object, named **spark**, is created automatically for you
- Old version code might use the SparkContext pattern. This should be avoided in favor of the builder method on the SparkSession

The SparkSession

- SparkSession is the object that represents your entrance point to writing Spark code
- It manages your Spark application
- In interactive mode, a SparkSession object, named **spark**, is created automatically for you
- Old version code might use the SparkContext pattern. This should be avoided in favor of the builder method on the SparkSession

```
spark.stop() //stopping a SparkSession

// Creating a SparkSession
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().getOrCreate()
```

The SparkContext

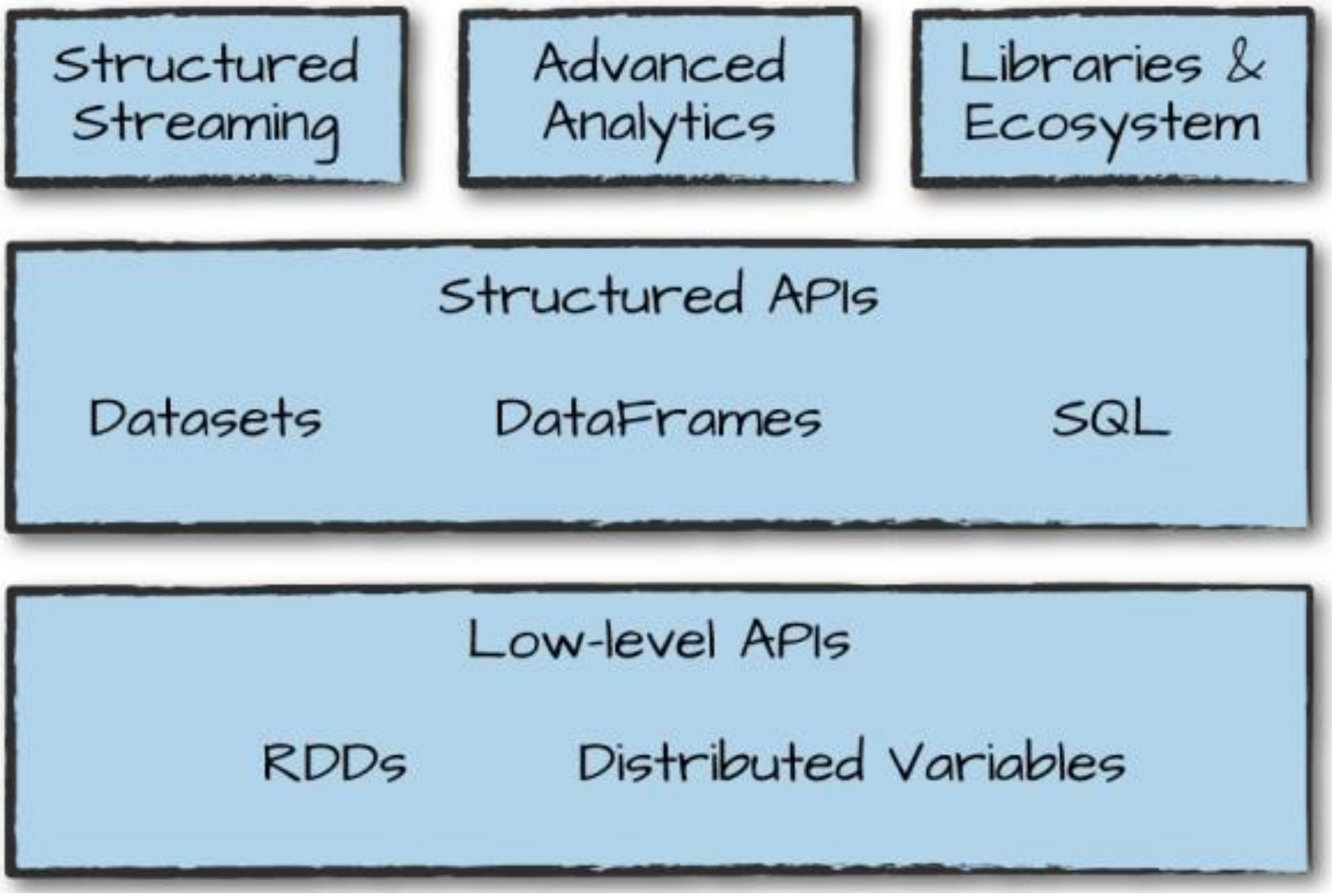
- A SparkContext object within the SparkSession represents a connection to the Spark cluster
- This class allows you to communicate with Spark's lower-level APIs, RDDs
- Through a SparkContext, you can create RDDs

The SparkContext

- A SparkContext object within the SparkSession represents a connection to the Spark cluster
- This class allows you to communicate with Spark’s lower-level APIs, RDDs
- Through a SparkContext, you can create RDDs
- If you have to, you should create a SparkContext in the most general way as follows:

```
import org.apache.spark.SparkContext
val sc = SparkContext.getOrCreate()
```

Spark Tools and Libraries



Logical Instructions to Physical Execution

- Spark code consists mainly of transformations and actions
- to understanding how Spark runs on a cluster:
  - Understand how a physical execution plans is generated

The Code

```
%python

df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")

step4.collect() # 2500000000000
step4.explain()
```

Output of step4.explain()

```
== Physical Plan ==
*HashAggregate(keys=[], functions=[sum(id#15L)])
+- Exchange SinglePartition
   +- *HashAggregate(keys=[], functions=[partial_sum(id#15L)])
      +- *Project [id#15L]
         +- *SortMergeJoin [id#15L], [id#10L], Inner
            :- *Sort [id#15L ASC NULLS FIRST], false, 0
            :  +- Exchange hashpartitioning(id#15L, 200)
            :     +- *Project [(id#7L * 5) AS id#15L]
            :        +- Exchange RoundRobinPartitioning(5)
            :           +- *Range (2, 10000000, step=2, splits=8)
            +- *Sort [id#10L ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(id#10L, 200)
                  +- Exchange RoundRobinPartitioning(6)
                     +- *Range (2, 10000000, step=4, splits=8)
```

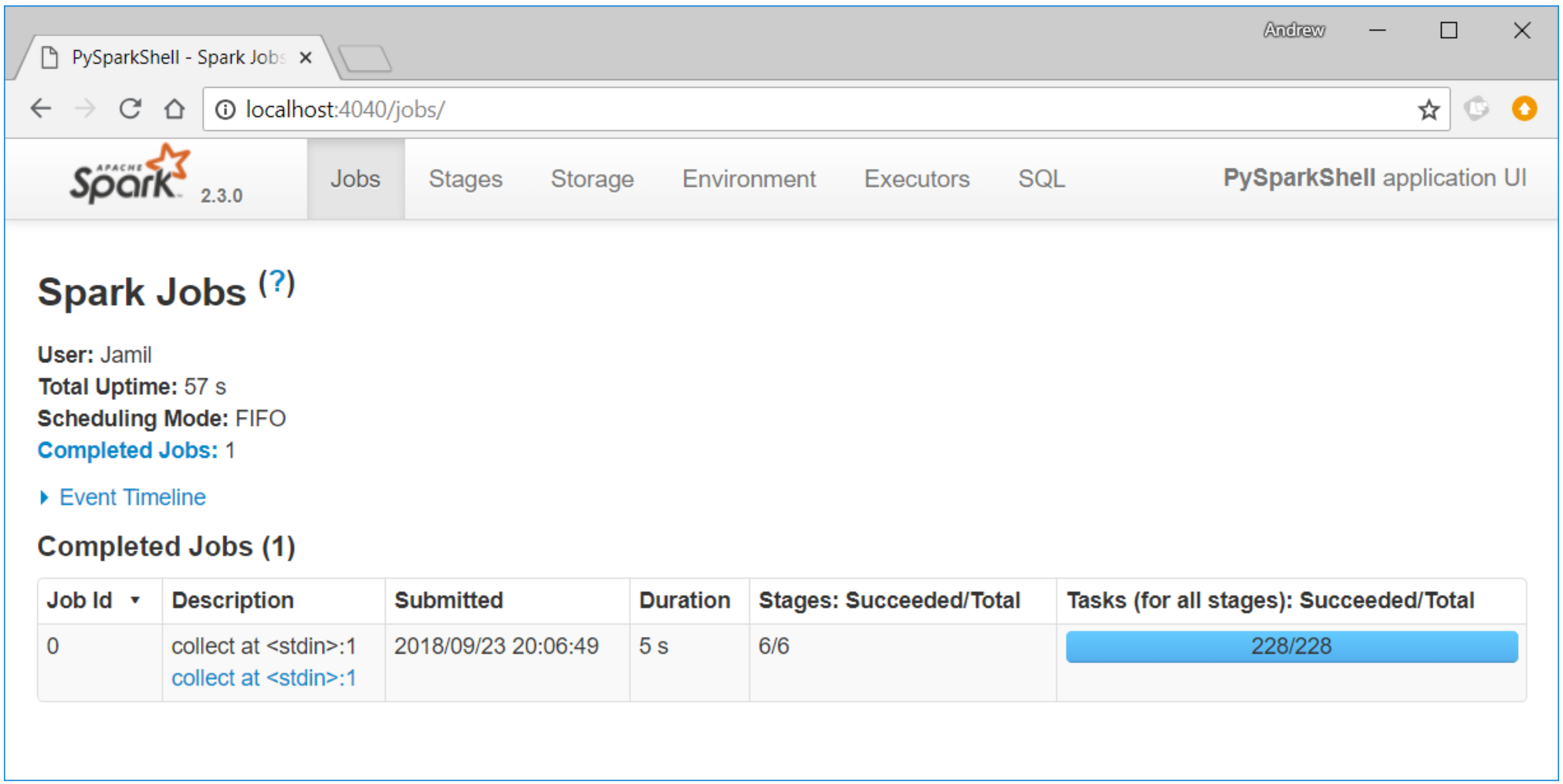
Output of step4.explain()

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")
step4.collect() # 250000000000
step4.explain()
```

```
== Physical Plan ==
*HashAggregate(keys=[], functions=[sum(id#15
+- Exchange SinglePartition
   +- *HashAggregate(keys=[], functions=[partial_sum(id#15L)])
      +- *Project [id#15L]
         +- *SortMergeJoin [id#15L], [id#10L], Inner
            :- *Sort [id#15L ASC NULLS FIRST], false, 0
            :  +- Exchange hashpartitioning(id#15L, 200)
            :     +- *Project [(id#7L * 5) AS id#15L]
            :        +- Exchange RoundRobinPartitioning(5)
            :           +- *Range (2, 10000000, step=2, splits=8)
            +- *Sort [id#10L ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(id#10L, 200)
                  +- Exchange RoundRobinPartitioning(6)
                     +- *Range (2, 10000000, step=4, splits=8)
```

Spark UI

- If code is run on the local machine, you can view Spark UI at url localhost:4040

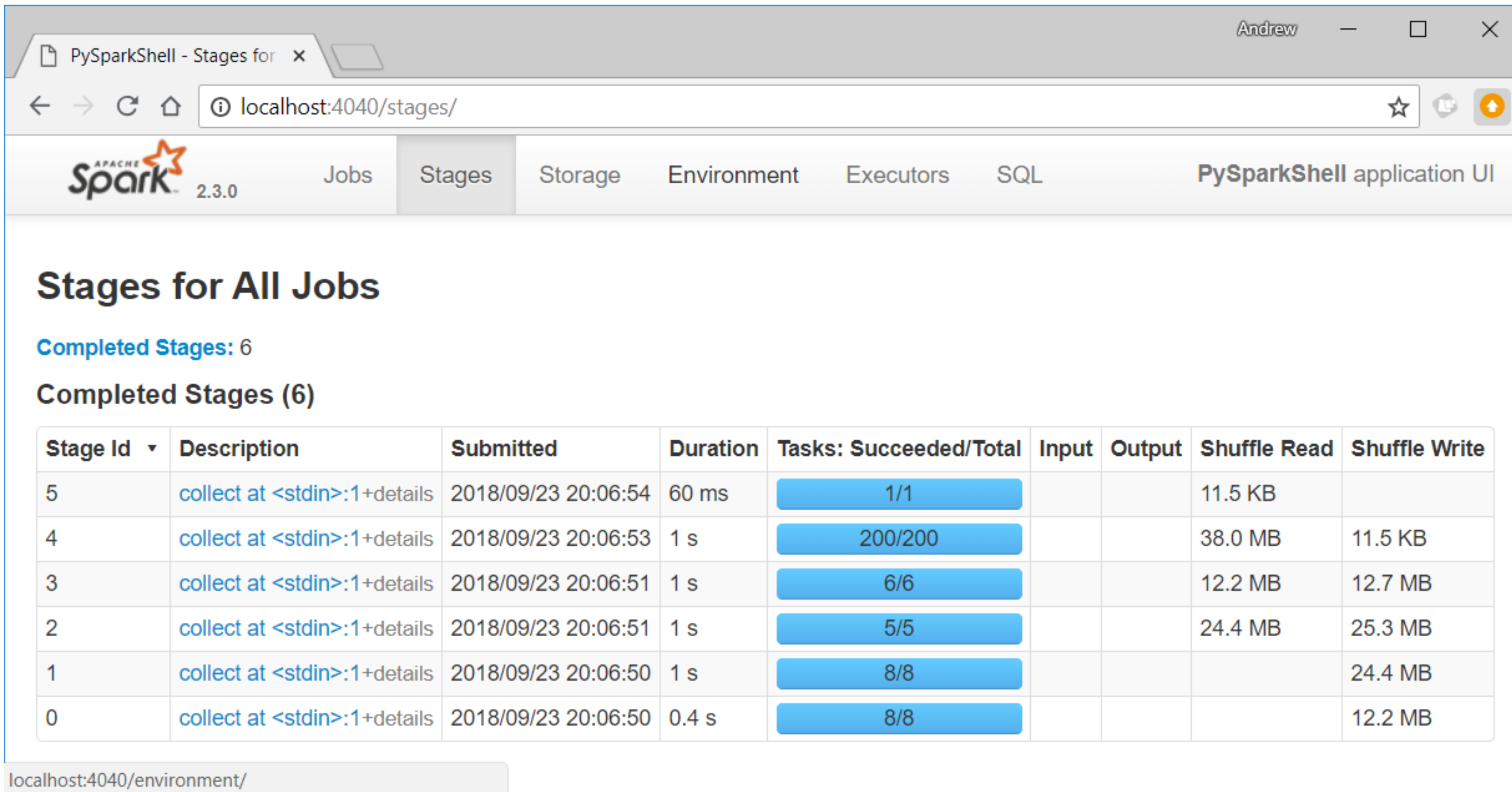


A Spark Job

- This job breaks down into the following stages and tasks:
  - Stage 1 with 8 Tasks
  - Stage 2 with 8 Tasks
  - Stage 3 with 6 Tasks
  - Stage 4 with 5 Tasks
  - Stage 5 with 200 Tasks
  - Stage 6 with 1 Task

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")
step4.collect() # 2500000000000
```

Stages



PySparkShell - Stages for

localhost:4040/stages/

Spark 2.3.0 | Jobs | Stages | Storage | Environment | Executors | SQL | PySparkShell application UI

Stages for All Jobs

Completed Stages: 6

Completed Stages (6)

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	collect at <stdin>:1+details	2018/09/23 20:06:54	60 ms	1/1			11.5 KB	
4	collect at <stdin>:1+details	2018/09/23 20:06:53	1 s	200/200			38.0 MB	11.5 KB
3	collect at <stdin>:1+details	2018/09/23 20:06:51	1 s	6/6			12.2 MB	12.7 MB
2	collect at <stdin>:1+details	2018/09/23 20:06:51	1 s	5/5			24.4 MB	25.3 MB
1	collect at <stdin>:1+details	2018/09/23 20:06:50	1 s	8/8				24.4 MB
0	collect at <stdin>:1+details	2018/09/23 20:06:50	0.4 s	8/8				12.2 MB

localhost:4040/environment/

Stages

- A stage is a group of tasks that can be executed together to compute the same operation on multiple machines
- Spark will try to pack as much work as possible into the same stage
- starts new stages after shuffle operations
- A shuffle represents a physical repartitioning of the data
  - sorting or grouping by key
- keeps track of the order stages must run

Stages (cont.)

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
```

```
Stage 1 with 8 Tasks
Stage 2 with 8 Tasks
```

- By default, when range is used to create a DF, it has eight partitions

Stages (cont.)

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
```

```
Stage 1 with 8 Tasks
Stage 2 with 8 Tasks
Stage 3 with 6 Tasks
Stage 4 with 5 Tasks
```

- By default, when range is used to create a DF, it has eight partitions
- Repartitioning changes the number of partitions by shuffling the data

Stages (cont.)

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
```

```
Stage 1 with 8 Tasks
Stage 2 with 8 Tasks
Stage 3 with 6 Tasks
Stage 4 with 5 Tasks
```

- By default, when range is used to create a DF, it has eight partitions
- Repartitioning changes the number of partitions by shuffling the data
- The DFs are shuffled into 5 and 6 partitions, corresponding to the number of tasks in stages 3 and 4

Stages (cont.)

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
```

```
Stage 1 with 8 Tasks
Stage 2 with 8 Tasks
Stage 3 with 6 Tasks
Stage 4 with 5 Tasks
```

Stages (cont.)

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
```

```
Stage 1 with 8 Tasks
Stage 2 with 8 Tasks
Stage 3 with 6 Tasks
Stage 4 with 5 Tasks
```

```
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")
step4.collect() # 2500000000000
```

Stages (cont.)

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
```

```
Stage 1 with 8 Tasks
Stage 2 with 8 Tasks
Stage 3 with 6 Tasks
Stage 4 with 5 Tasks
```

```
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")
step4.collect() # 2500000000000
```

```
Stage 5 with 200 Tasks
Stage 6 with 1 Task
```

Stages (cont.)

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
```

```
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")
step4.collect() # 2500000000000
```

```
Stage 1 with 8 Tasks
Stage 2 with 8 Tasks
Stage 3 with 6 Tasks
Stage 4 with 5 Tasks
```

```
Stage 5 with 200 Tasks
Stage 6 with 1 Task
```

- `step2` is a value-by-value manipulation in stage 4

Stages (cont.)

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
```

```
Stage 1 with 8 Tasks
Stage 2 with 8 Tasks
Stage 3 with 6 Tasks
Stage 4 with 5 Tasks
```

```
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")
step4.collect() # 2500000000000
```

```
Stage 5 with 200 Tasks
Stage 6 with 1 Task
```

- `step2` is a value-by-value manipulation in stage 4
- `step2.join(step12, ["id"])` is a join (shuffle)

Stages (cont.)

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
```

```
Stage 1 with 8 Tasks
Stage 2 with 8 Tasks
Stage 3 with 6 Tasks
Stage 4 with 5 Tasks
```

```
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")
step4.collect() # 2500000000000
```

```
Stage 5 with 200 Tasks
Stage 6 with 1 Task
```

- `step2` is a value-by-value manipulation in stage 4
- `step2.join(step12, ["id"])` is a join (shuffle)
- By default, when a shuffle is performed during execution, it outputs 200 shuffle partitions

Stages (cont.)

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
```

```
Stage 1 with 8 Tasks
Stage 2 with 8 Tasks
Stage 3 with 6 Tasks
Stage 4 with 5 Tasks
```

```
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")
step4.collect() # 2500000000000
```

```
Stage 5 with 200 Tasks
Stage 6 with 1 Task
```

- `step2` is a value-by-value manipulation in stage 4
- `step2.join(step12, ["id"])` is a join (shuffle)
- By default, when a shuffle is performed during execution, it outputs 200 shuffle partitions
- `sum(id)` aggregates individual partitions and brings results to 1 partition

Stages (cont.)

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
```

```
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")
step4.collect() # 2500000000000
```

```
Stage 1 with 8 Tasks
Stage 2 with 8 Tasks
Stage 3 with 6 Tasks
Stage 4 with 5 Tasks
```

```
Stage 5 with 200 Tasks
Stage 6 with 1 Task
```

- `step2` is a value-by-value manipulation in stage 4
- `step2.join(step12, ["id"])` is a join (shuffle)
- By default, when a shuffle is performed during execution, it outputs 200 shuffle partitions
- `sum(id)` aggregates individual partitions and brings results to 1 partition
- `collect()` sends the final result to the driver

Changing Number of Partitions for a Shuffle

- By default, a shuffle outputs 200 partitions
- This code changes that:

```
spark.conf.set("spark.sql.shuffle.partitions", "10")
```

Changing Number of Partitions for a Shuffle

- By default, a shuffle outputs 200 partitions
- This code changes that:

```
spark.conf.set("spark.sql.shuffle.partitions", "10")
```

- A good rule of thumb:
  - Typically, 2-4 partitions for each CPU in your cluster

Tasks

- A stage consists of tasks

Tasks

- A stage consists of tasks
- Each task corresponds to a set of transformations applied to a unit of data (the partition)

Tasks

- A stage consists of tasks
- Each task corresponds to a set of transformations applied to a unit of data (the partition)
- A task runs on a single executor

Tasks

- A stage consists of tasks
- Each task corresponds to a set of transformations applied to a unit of data (the partition)
- A task runs on a single executor
- If there is one big partition in our dataset, we will have one task

Tasks

- A stage consists of tasks
- Each task corresponds to a set of transformations applied to a unit of data (the partition)
- A task runs on a single executor
- If there is one big partition in our dataset, we will have one task
- If there are many little partitions, we will have many tasks that can be executed in parallel

Tasks

- A stage consists of tasks
- Each task corresponds to a set of transformations applied to a unit of data (the partition)
- A task runs on a single executor
- If there is one big partition in our dataset, we will have one task
- If there are many little partitions, we will have many tasks that can be executed in parallel
- Partitioning your data into a greater number of partitions means that more parallelism

Pipelining

- Pipelining is automatic optimization done by Spark

Pipelining

- Pipelining is automatic optimization done by Spark
- Any sequence of operations that feed data directly into each other, without needing to move it across nodes, is pipelined into a **single stage** of tasks

Pipelining

- Pipelining is automatic optimization done by Spark
- Any sequence of operations that feed data directly into each other, without needing to move it across nodes, is pipelined into a **single stage** of tasks
- Ex: a map/select operation followed by a filter followed by a map/select

Pipelining

- Pipelining is automatic optimization done by Spark
- Any sequence of operations that feed data directly into each other, without needing to move it across nodes, is pipelined into a single stage of tasks
- Ex: a map/select operation followed by a filter followed by a map/select
- Spark is fast as it performs as many steps as possible before writing data to memory or disk

Shuffle Persistence

- Another optimization technique done by Spark
- Whenever a shuffle operation is performed, Spark writes the result to permanent storage
- Any subsequent operation that depends on that shuffle will launch by reading the data from storage
- Shuffle operations do not need to be executed again
- This also makes Spark run faster