APACHE SPARK OVERVIEW

Apache Spark Overview

- **Apache Spark**
- Resilient Distributed Datasets (RDDs)
- Distributed Operations on RDDs
- Applying a Function
- Working with Records of Pairs
- Partitioning
- Broadcast Variables
- Running Spark Applications

Apache Spark

- Apache Spark allows us writing distributed programs, which process vast amounts of data
  - in a parallel and distributed manner
  - in-memory
  - on large (of thousands of nodes) clusters
  - reliably

Apache Spark

- Spark can integrate closely with Hadoop
  - Spark can run in Hadoop clusters
  - Spark can access to any Hadoop data source
  - Spark applications can be submitted to **YARN**
    - **Mesos** is also possible
    - Spark also comes with its own **Standalone** cluster management component
- It is important to understand that Spark does not require Hadoop, it simply has support for storage systems implementing the Hadoop APIs (thus local filesystem, Amazon S3, Cassandra, Hive, HBase, …)

Apache Spark

- Spark comes with an easier-to-use API than the MapReduce framework:
  - In the native Scala API, running the distributed operations on on distributed collections (RDDs) is very similar to running higher order functions over an ordinary collection

Apache Spark

- Spark is more general than MapReduce, and it is more of an alternative to the entire Hadoop Ecosystem with its unified stack for supporting a wide range of workloads:
  - General purpose Spark cluster computing (M/R alternative)
  - Spark SQL (for submitting SQL queries to the RDDs)
  - Spark MlLib (for running machine learning applications)
  - Spark Streaming (for making Stream computations)
  - Spark GraphX (for Graph processing)

analyticscenter

Apache Spark Overview

- Apache Spark
- **Resilient Distributed Datasets (RDDs)**
- Distributed Operations on RDDs
- Applying a Function
- Working with Records of Pairs
- Partitioning
- Broadcast Variables
- Running Spark Applications

Resilient Distributed Datasets

- In Spark, the core abstraction representing a distributed dataset is called an RDD
- Distributed computations are expressed as high level operations on distributed data collections–Resilient Distributed Datasets (RDDs)
- RDDs are distributed collections of
  - Records (of the same type)
  - Records of pairs (as it is in the MapReduce framework)

Resilient Distributed Datasets

- An RDD is created by:
  - Loading from an external data source,
  - Partitioning and distributing an existing collection
- All work is then expressed as a series of transformations and an action, where
  - A transformation returns a new RDD from an existing one
  - An action actually computes a result

Resilient Distributed Datasets

- RDDs are of partitions distributed to multiple nodes
- RDDs are lazily computed, that is, they are only actually materialized if they are used in an action
  - The intermediate RDDs through a series of operations with an action are not required to be materialized, unless Spark is specifically instructed to persist them
  - Plus, this persistence does not have to be into disk, and in fact it is kept in memory until it is full and then spilled into disk

Resilient Distributed Datasets

- As new RDDs are derived from the previous ones, Spark keeps track of the dependencies between the RDDs
  - This is called a lineage graph
    - It allows on-demand computation of an RDD
    - It allows recovery of lost data

Resilient Distributed Datasets

- Records of an RDD are typed, and certain functions are available only on certain types (such as mean is only available for RDDs of numeric records)
- RDDs are immutable, and the operations running on it are described in a functional manner
  - A transformed RDD is a new RDD

RDD Persistence

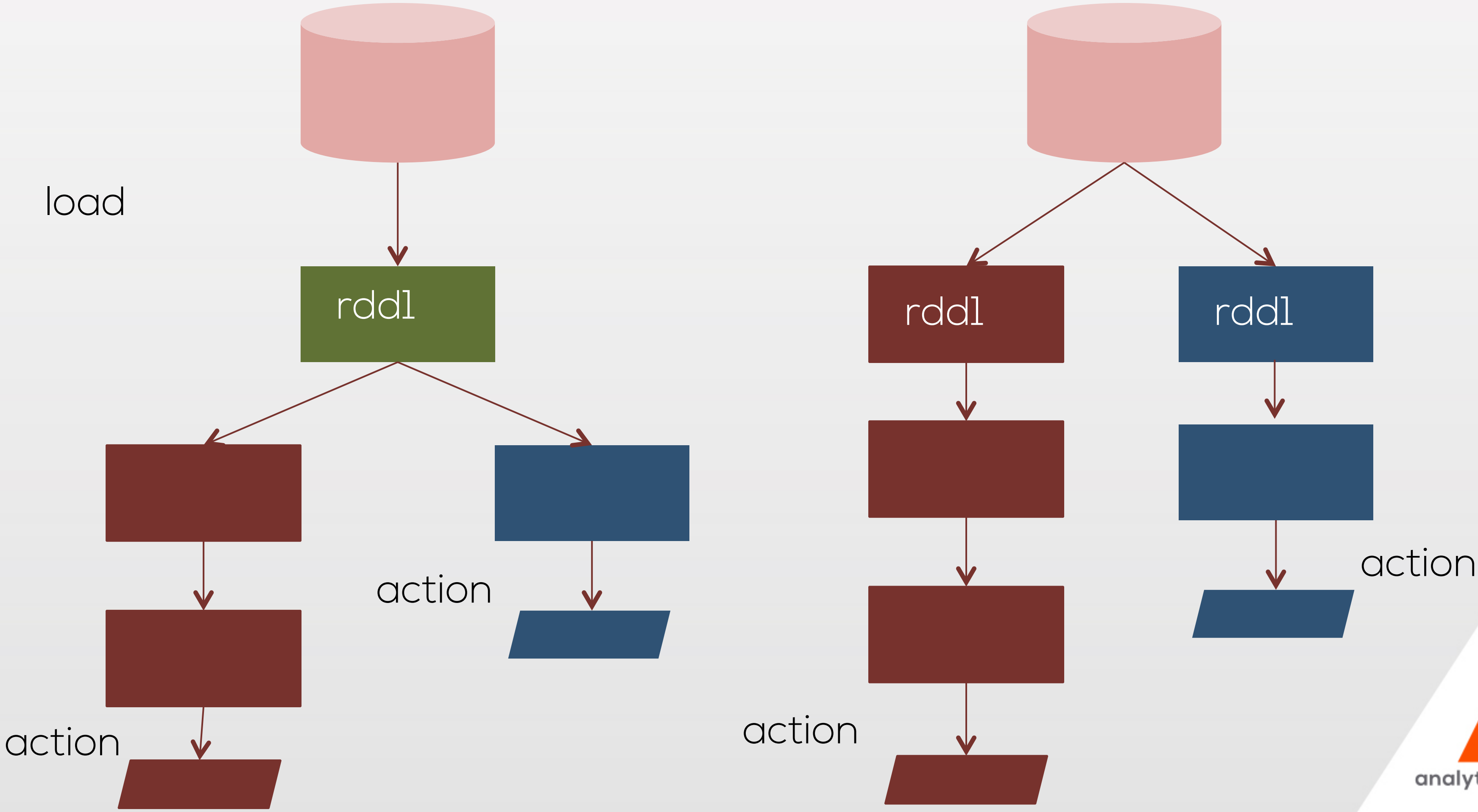
- Since they are lazily evaluated, using an RDD multiple times would cause Spark recompute the RDD (and its parents) each time an action requires it
- We can ask Spark to persist an RDD, so that it can be further reused in multiple actions
- Persisting is done by the nodes storing their partitions
- An RDD can be persisted with RDD#persist call, and where the RDD will be stored for further usage can be specified:
  - Memory
  - Memory and disk (spills to disk if the data do not fit in memory)
  - Disk

RDD Persistence

- An RDD can be persisted with **`RDD#persist`** call, and where the RDD will be stored for further usage can be specified:
  - Memory
  - Memory and disk (spills to disk if the data do not fit in memory)
  - Disk
- **`RDD#cache`** persists and RDD with MEMORY_ONLY storage level
- **`RDD#unpersist`** call removes all blocks from memory and disk

RDD Persistence

Persisted vs. not



Apache Spark Overview

- Apache Spark
- Resilient Distributed Datasets (RDDs)
- **Distributed Operations on RDDs**
- Applying a Function
- Working with Records of Pairs
- Partitioning
- Broadcast Variables
- Running Spark Applications

Distributed Operations on RDDs

- Typically, within a driver program, we load an RDD and launch a series of distributed operations
- A driver program holds a **`SparkContext`** instance, which is the main entry for Spark functionality
- **`SparkContext`** encapsulates the connection to a Spark cluster, and it can be used for creating an RDD, a broadcast object, etc.
- **`SparkContext`** is where jobs are submitted through (such as when we call an action operation on an RDD)

analyticscenter

Distributed Operations on RDDs

- The distributed operations can be transformations and actions:
  - Transformations return a new RDD from an RDD, such as mapping and filtering

```
rdd.filter(r=> r>10)
```

  - Transformations are not executed immediately, making applying a chain of transformations very efficient
  - Loading data is also a lazy operation (data is not loaded until it is necessary)

Spark

- Some Spark transformations:
  - Record wise projectors and filters
  - Record wise multi-row generating operations (the flatMap)
  - Single table set operations (such as distinct)
  - Multi table set operations (such as join, cartesian)

Spark

- The distributed operations can be transformations and actions:
  - These operations can be transformations and actions:
    - Actions return a result to the driver or write the result to storage, such as returning an aggregate and collecting RDD into the driver

```
rdd.count()
```

```
rdd.saveAsSequenceFile(path)
```

    - An action, unlike a transformation, kicks off a computation

analyticscenter

Spark

- Some Spark actions:
  - reduce (a particular type of aggregation)
  - fold (another particular type of aggregation)
  - aggregate (general purpose aggregation)
  - Pre-defined aggregations (count, sum, …)
  - Per-key aggregations (reduceByKey, …)
  - Collecting and RDD into a local data structure (collect)
  - Collecting a small list of records of an RDD (take)
  - Per-key actions (foreach)

Spark

- It is the **actions** that causes the evaluation of the transformations that are required to be computed for the action to be executed
- Any action requires the entire RDD to be computed from scratch, that is, an action cannot use the same intermediate RDD created for another action
  - This is inefficient for iterative (and interactive) workloads
  - The persistence mechanism allows Spark to persist an RDD (possibly with a replication) to the memory, and spilled to disk if necessary (or disk only)

Apache Spark Overview

- Apache Spark
- Resilient Distributed Datasets (RDDs)
- Distributed Operations on RDDs
- **Applying a Function**
- Working with Records of Pairs
- Partitioning
- Broadcast Variables
- Running Spark Applications

Applying a Function

- Usually, we pass functions to transformations (and sometimes actions), to describe the computation to be done on the records of the RDD
  - For example, we might want to pass an external boolen function to a filter, some transformation for mapping, …
- For these functions to be able to applied on the partitions distributed to different nodes, the function we pass (and the data referenced in it) should be serializable
- Inline functions, references to methods, or static functions can be passed

Applying a Function

- Care should be taken to avoid passing a method (or field) of an object, since this would cause to serialize the whole object containing the function
  - This might cause a **`NotSerializableException`**
- Instead, inline functions, local variabes, and static methods in a global singleton object should be preferred

analyticscenter

Apache Spark Overview

- Apache Spark
- Resilient Distributed Datasets (RDDs)
- Distributed Operations on RDDs
- Applying a Function
- **Working with Records of Pairs**
- Partitioning
- Broadcast Variables
- Running Spark Applications

Working with Records of Pairs

- When the elements of an RDD are of type **`Tuple`**, Spark provides special operations for working with them
- Such RDDs are called Pair RDDs
- Pair RDDs provide additional distributed computation mechanisms, such as **`reduceByKey`**, using which one can implement MapReduce-like aggregation-per-key operations

analyticscenter

Working with Records of Pairs

- A Pair RDD can be created at load time, or by a transformation returning **`Tuple`**s as records
  - In Scala, once we have a Pair RDD, additional operations such as **`reduceByKey`** and **`join`** are automatically available through implicit conversions (just import **`SparkContext._`**)

analyticscenter

Working with Records of Pairs

- Example pair transformations on an RDD (rdd) **`{('a', 1), ('b', 2), ('b', 3)}`**

`rdd.reduceByKey((x, y) => x + y)`	`{('a',1), ('b',5)}`
`rdd.groupByKey()`	`{('a',[1]), ('b',[2,3])}`
`rdd.mapValues(v => v * 2)`	`{('a',2), ('b',4), ('b',6)}`
`rdd.keys()`	`{'a', 'b', 'b'}`

Working with Records of Pairs

- Multi dataset operations that are available only on Pair RDDs are as following

```
rdd.subtractByKey(other)
rdd.join(other)
rdd.rightOuterJoin(other)
rdd.leftOuterJoin(other)
rdd.cogroup(other)
```

Working with Records of Pairs

- WordCount in Spark

```
//sc is the SparkContext instance

val input = sc.textFile("hdfs://path/to/file")
val words = input.flatMap(line => line.split(" "))
val wordCounts = words.map(word => (word, 1))
                      .reduceByKey((i, j) => i + j)
```

- No need to specify if a combiner (local aggregation) should be used, **`reduce`** function applies local aggregations automatically

How Aggregations By Key Work

- Most of by-key-aggregations are implemented on top of `combineByKey`
- Such aggregations would require a MapReduce-like computation, that is, for per-key aggregations to be performed, the records read should be shuffled through the aggregation (reducer, if you want) nodes
  - Shuffle is not necessary if the dataset is already shuffled, i.e. if they are partitioned in a way that one key can only exist in one partition

How Aggregations By Key Work

- The user usually defines an initialValue (of result type), a merge behavior of a record (how a new record would be merged into the current aggregate), and merge behavior of multiple aggregates (how the local results should be combined)
- As in MapReduce, the **`combineByKey`** allows the partitioning behavior to be set, and whether or not a **`Combiner`** should be used to be specified
- A specific aggregation is combining values associated with a key together in a collection (**`groupByKey`**), which is an example aggregation that does not use map side combiners

Apache Spark Overview

- Apache Spark
- Resilient Distributed Datasets (RDDs)
- Distributed Operations on RDDs
- Applying a Function
- Working with Records of Pairs
- **Partitioning**
- Broadcast Variables
- Running Spark Applications

Partitioning

- Recall that if an RDD is partitioned by key,
  - per-key aggregations can be performed map-side
  - joins can be performed efficiently
- When a dataset is reused many times in such operations after being read, the user can (should, actually) partition the dataset manually. Two examples are:
  - **`HashPartitioner`** (equivalent to Hadoop's **`HashPartitioner`**)
  - **`RangePartitioner`** (equivalent to Hadoop's **`TotalOrderPartitioner`**)
- A Pair RDD can be partitioned using **`partitionBy`** method
  - This is a transformation, and returns a new RDD (which is partitioned)

Partitioning

- When a dataset is reused many times in such operations after being read, the user can (should, actually) partition the dataset manually. Two examples are:
  - `HashPartitioner` (equivalent to Hadoop's `HashPartitioner`)
    - When, say, an RDD is hash partitioned into 100, keys with the same hash value modulo 100 would go to the same partition
  - `RangePartitioner` (equivalent to Hadoop's `TotalOrderPartitioner`)
    - Records within the same key-range (sorted range of keys) would appear in the same partition
    - Range is determined automatically by Spark (based on sampling of the RDD)

Apache Spark Overview

- Apache Spark
- Resilient Distributed Datasets (RDDs)
- Distributed Operations on RDDs
- Applying a Function
- Working with Records of Pairs
- Partitioning
- **Broadcast Variables**
- Running Spark Applications

Broadcast Variables

- Users can broadcast a read-only variable to be cached on each machine (instead of shipping a copy of it with tasks)
- `SparkContext#broadcast` can be used to broadcast the variable
- The value can be retrieved using its `get` method afterwards

Accumulators

- Another type of shared variables in Spark is the accumulators
- They are global variables that can be accumulated using associative operations
- The idea is similar to MapReduce counters, and this is the way to define a global variable and accumulate into it from each task
- Creation of an accumulator is simple, **`SparkContext#accumulator`**, and the accumulation is performed by **`+=`** method (or **`add`**)

Apache Spark Overview

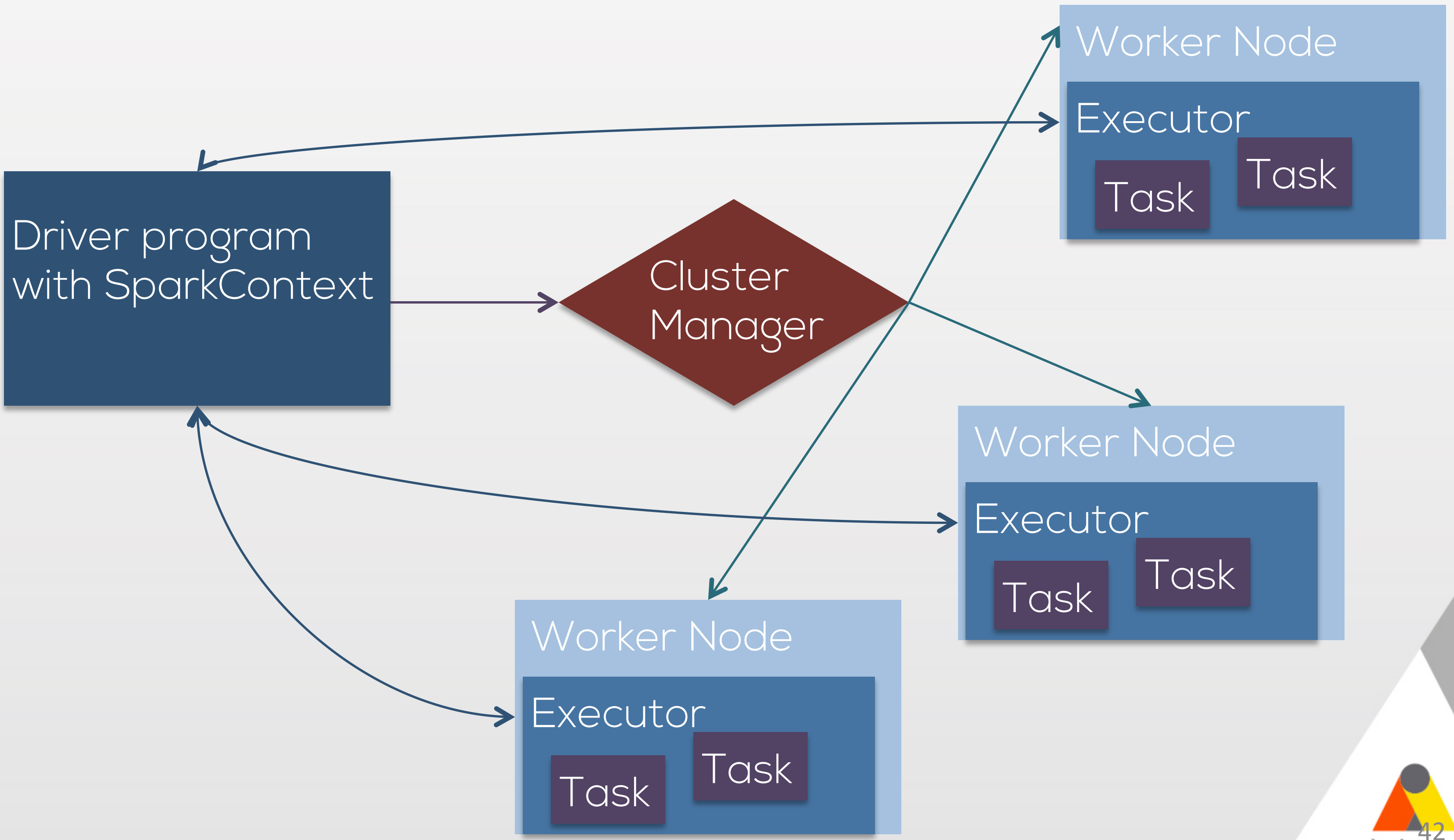
- Apache Spark
- Resilient Distributed Datasets (RDDs)
- Distributed Operations on RDDs
- Applying a Function
- Working with Records of Pairs
- Partitioning
- Broadcast Variables
- **Running Spark Applications**

A Spark Cluster

- A Spark application is submitted by a driver program (the program with the **`SparkContext`**), and distributed tasks are run by executors launched on Worker nodes
  - Executor process is per application (This also means different applications cannot communicate data)
  - Spark is agnostic to the underlying cluster manager
  - Driver should be able to communicate with the cluster (to submit and monitor the applications)
  - It is good to have HDFS data nodes colocated with Spark worker nodes,
    - Spark would create partitions from an HDFS file based on **`InputSplits`** computed by **`FileInputFormat`**, which maps HDFS blocks to InputSplits

A Spark Cluster

- Here is what a Spark application looks like on a cluster



Spark Shell

- Spark comes with a shell, through which users can perform Spark operations (creating/loading RDDs and running distributed operations on them) in an interactive fashion

```
$ ./bin/spark-shell

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.2.1
      /_/

scala> val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5, 6), 2)
scala> rdd.map(_*2).
       sum

...
res3: Double = 42.0
```

Spark Application

- A Spark application is a Scala (or Java or Python) application with a main method containing a `SparkContext`
- This is the driver application (shell itself is a driver, actually), and based on the `SparkConf` object (with app name, master host address, how many executors would be created, which additional jars should be used, etc.) passed to the `SparkContext`, when this application is run it is submitted to the cluster
- Another way to run a Spark application is packaging it into a jar with all of its dependencies (it is important not to include Spark jar as a dependceny), and using the `spark-submit` utility
  - Tools like Apache maven might come handy to package an application with the correct dependencies

Submitting Spark Applications

```
# Several example spark-submit runs (from spark.apache.org),
# last argument (100 and 1000) is the argument to the main
# method of the SparkPi app
# Run spark-submit with --help argument

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

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


analyticscenter

Submitting Spark Applications

```
# Several example spark-submit runs (from spark.apache.org),
# last argument (100 and 1000) is the argument to the main
# method of the SparkPi app
# Run spark-submit with --help argument

# Run on YARN

export HADOOP_CONF_DIR=XXXX
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn-cluster \
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000
```


analyticscenter

Demo

Example Interactive Analyses with Spark Shell, and Using the spark-submit Utility

Apache Spark Overview

End of Chapter