

Mastering Apache Spark

Jacek Laskowski

Published
with GitBook



Table of Contents

Introduction	0
Overview of Spark	1
Anatomy of Spark Application	2
SparkConf - Configuration for Spark Applications	2.1
SparkContext - the door to Spark	2.2
RDD - Resilient Distributed Dataset	2.3
Operators - Transformations and Actions	2.3.1
mapPartitions	2.3.1.1
Partitions	2.3.2
Caching and Persistence	2.3.3
Shuffling	2.3.4
Checkpointing	2.3.5
Dependencies	2.3.6
Types of RDDs	2.3.7
ParallelCollectionRDD	2.3.7.1
MapPartitionsRDD	2.3.7.2
CoGroupedRDD	2.3.7.3
HadoopRDD	2.3.7.4
ShuffledRDD	2.3.7.5
BlockRDD	2.3.7.6
Spark Tools	3
Spark Shell	3.1
WebUI - UI for Spark Monitoring	3.2
Executors Tab	3.2.1
spark-submit	3.3
spark-class	3.4
Spark Architecture	4
Driver	4.1
Master	4.2
Workers	4.3

Executors	4.4
Spark Runtime Environment	5
DAGScheduler	5.1
Jobs	5.1.1
Stages	5.1.2
Task Scheduler	5.2
Tasks	5.2.1
TaskSets	5.2.2
TaskSetManager	5.2.3
TaskSchedulerImpl - Default TaskScheduler	5.2.4
Scheduler Backend	5.3
CoarseGrainedSchedulerBackend	5.3.1
Executor Backend	5.4
CoarseGrainedExecutorBackend	5.4.1
Shuffle Manager	5.5
Block Manager	5.6
HTTP File Server	5.7
Broadcast Manager	5.8
Dynamic Allocation	5.9
Data Locality	5.10
Cache Manager	5.11
Spark, Akka and Netty	5.12
OutputCommitCoordinator	5.13
RPC Environment (RpcEnv)	5.14
Netty-based RpcEnv	5.14.1
ContextCleaner	5.15
MapOutputTracker	5.16
ExecutorAllocationManager	5.17
Deployment Environments	6
Spark local	6.1
Spark on cluster	6.2
Spark Standalone	6.2.1
Master	6.2.1.1
web UI	6.2.1.2

Management Scripts for Standalone Master	6.2.1.3
Management Scripts for Standalone Workers	6.2.1.4
Checking Status	6.2.1.5
Example 2-workers-on-1-node Standalone Cluster (one executor per worker)	6.2.1.6
Spark on Mesos	6.2.2
Spark on YARN	6.2.3
Execution Model	7
Advanced Concepts of Spark	8
Broadcast variables	8.1
Accumulators	8.2
Security	9
Spark Security	9.1
Securing Web UI	9.2
Data Sources in Spark	10
Using Input and Output (I/O)	10.1
Spark and Parquet	10.1.1
Serialization	10.1.2
Using Apache Cassandra	10.2
Using Apache Kafka	10.3
Spark Application Frameworks	11
Spark Streaming	11.1
StreamingContext	11.1.1
Stream Operators	11.1.2
Windowed Operators	11.1.2.1
SaveAs Operators	11.1.2.2
Stateful Operators	11.1.2.3
web UI and Streaming Statistics Page	11.1.3
Streaming Listeners	11.1.4
Checkpointing	11.1.5
JobScheduler	11.1.6
JobGenerator	11.1.7
DStreamGraph	11.1.8
Discretized Streams (DStreams)	11.1.9

Input DStreams	11.1.9.1
ReceiverInputDStreams	11.1.9.2
ConstantInputDStreams	11.1.9.3
ForEachDStreams	11.1.9.4
WindowedDStreams	11.1.9.5
MapWithStateDStreams	11.1.9.6
StateDStreams	11.1.9.7
TransformedDStream	11.1.9.8
Receivers	11.1.10
ReceiverTracker	11.1.10.1
ReceiverSupervisors	11.1.10.2
ReceivedBlockHandlers	11.1.10.3
Ingesting Data from Kafka	11.1.11
KafkaRDD	11.1.11.1
RecurringTimer	11.1.12
Streaming DataFrames	11.1.13
Backpressure	11.1.14
Dynamic Allocation (Elastic Scaling)	11.1.15
Settings	11.1.16
Spark SQL	11.2
SQLContext	11.2.1
Dataset	11.2.2
DataFrame	11.2.3
DataFrameReader	11.2.3.1
DataFrameWriter	11.2.3.2
Standard Functions for DataFrames (functions object)	11.2.4
Streaming DataFrames (2.0.0)	11.2.5
ContinuousQueryManager	11.2.5.1
ContinuousQuery	11.2.5.2
Aggregation (GroupedData)	11.2.6
Joins	11.2.7
UDFs — User-Defined Functions	11.2.8
Windows in DataFrames	11.2.9

Catalyst optimizer	11.2.10
Into the depths	11.2.11
Datasets vs RDDs	11.2.12
Settings	11.2.13
Spark MLlib - Machine Learning in Spark	11.3
ML Pipelines	11.3.1
Persistance	11.3.1.1
Example — Text Classification	11.3.1.2
Example — Linear Regression	11.3.1.3
Distributed graph computations with GraphX	11.4
Monitoring, Tuning and Debugging	12
Logging	12.1
Performance Tuning	12.2
Spark Metrics System	12.3
Scheduler Listeners	12.4
Debugging Spark using sbt	12.5
Varia	13
Building Spark	13.1
Spark and Hadoop	13.2
Spark and software in-memory file systems	13.3
Spark and The Others	13.4
Distributed Deep Learning on Spark	13.5
Spark Packages	13.6
Spark Tips and Tricks	14
Access private members in Scala in Spark shell	14.1
SparkException: Task not serializable	14.2
Running Spark on Windows	14.3
Exercises	15
One-liners using PairRDDFunctions	15.1
Learning Jobs and Partitions Using take Action	15.2
Spark Standalone - Using ZooKeeper for High-Availability of Master	15.3
Spark's Hello World using Spark shell and Scala	15.4
WordCount using Spark shell	15.5
Your first complete Spark application (using Scala and sbt)	15.6

Spark (notable) use cases	15.7
Using Spark SQL to update data in Hive using ORC files	15.8
Developing Custom SparkListener to monitor DAGScheduler in Scala	15.9
Developing RPC Environment	15.10
Developing Custom RDD	15.11
Further Learning	16
 Courses	16.1
 Books	16.2
Commercial Products using Apache Spark	17
 IBM Analytics for Apache Spark	17.1
 Google Cloud Dataproc	17.2
Spark Advanced Workshop	18
 Requirements	18.1
 Day 1	18.2
 Day 2	18.3
Spark Talks Ideas (STI)	19
 10 Lesser-Known Tidbits about Spark Standalone	19.1
 Learning Spark internals using groupBy (to cause shuffle)	19.2
Glossary	

Mastering Apache Spark

Welcome to Mastering Apache Spark!

I'm [Jacek Laskowski](#), an **independent consultant** who offers development and training services for **Apache Spark** (and Scala, sbt, Akka Actors/Stream/HTTP with a bit of Apache Kafka, Apache Mesos, RxScala, Docker). I run [Warsaw Scala Enthusiasts](#) and [Warsaw Spark](#) meetups.

Contact me at jacek@japila.pl to discuss Spark opportunities, e.g. courses, workshops, or other mentoring or development services.

This collections of notes (what some may rashly call a "book") serves as the ultimate place of mine to collect all the nuts and bolts of using [Apache Spark](#). The notes aim to help me designing and developing better products with Spark. It is also a viable proof of my understanding of Apache Spark. I do eventually want to reach the highest level of mastery in Apache Spark.

It *may* become a book one day, but surely serves as **the study material** for trainings, workshops, videos and courses about Apache Spark. Follow me on twitter [@jaceklaskowski](#) to know it early. You will also learn about the upcoming events about Apache Spark.

Expect text and code snippets from [Spark's mailing lists](#), [the official documentation of Apache Spark](#), [StackOverflow](#), blog posts, [books from O'Reilly](#), press releases, YouTube/Vimeo videos, [Quora](#), [the source code of Apache Spark](#), etc. Attribution follows.

Overview of Spark

When you hear **Apache Spark** it can be two things - the Spark engine aka **Spark Core** or the Spark project - an "umbrella" term for Spark Core and the accompanying **Spark Application Frameworks**, i.e. [Spark SQL](#), [Spark Streaming](#), [Spark MLlib](#) and [Spark GraphX](#) that sit on top of Spark Core and the main data abstraction in Spark called [RDD - Resilient Distributed Dataset](#).

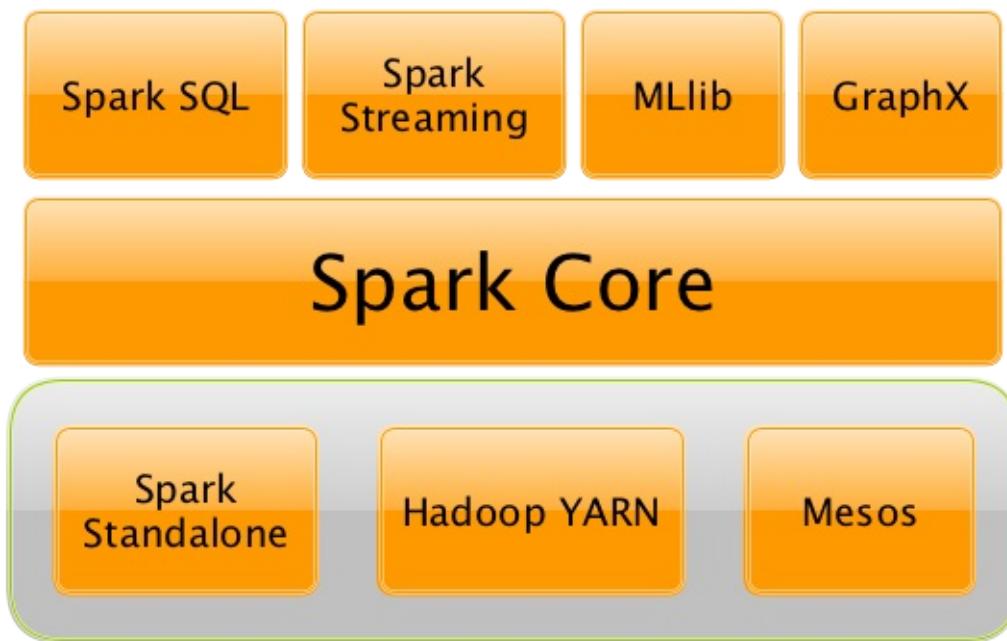


Figure 1. The Spark Platform

It is pretty much as Hadoop where it can mean different things for different people, and Spark has heavily been and still is influenced by Hadoop.

Why Spark

Let's list a few of the many reasons for Spark. We are doing it first, and then comes the overview that lends a more technical helping hand.

Diverse Workloads

As said by Matei Zaharia - the author of Apache Spark - in [Introduction to AmpLab Spark Internals video](#) (quoting with few changes):

One of the Spark project goals was to deliver a platform that supports a very wide array of **diverse workflows** - not only MapReduce **batch** jobs (there were available in Hadoop already at that time), but also **iterative computations** like graph algorithms or Machine Learning.

And also different scales of workloads from sub-second interactive jobs to jobs that run for many hours.

Spark also supports **near real-time streaming workloads** via [Spark Streaming](#) application framework.

ETL workloads and Analytics workloads are different, however Spark attempts to offer a unified platform for a wide variety of workloads.

Graph and Machine Learning algorithms are iterative by nature and less saves to disk or transfers over network means better performance.

There is also support for interactive workloads using Spark shell.

You should watch the video [What is Apache Spark?](#) by Mike Olson, Chief Strategy Officer and Co-Founder at Cloudera, who provides a very exceptional overview of Apache Spark, its rise in popularity in the open source community, and how Spark is primed to replace MapReduce as the general processing engine in Hadoop.

Leverages the Best in distributed batch data processing

When you think about **distributed batch data processing**, [Hadoop](#) naturally comes to mind as a viable solution.

Spark draws many ideas out of Hadoop MapReduce. They work together well - Spark on YARN and HDFS - while improving on the performance and simplicity of the distributed computing engine.

For many, Spark is Hadoop++, i.e. MapReduce done in a better way.

And it should **not** come as a surprise, without Hadoop MapReduce (its advances and deficiencies), Spark would not have been born at all.

RDD - Distributed Parallel Scala Collections

As a Scala developer, you may find Spark's RDD API very similar (if not identical) to [Scala's Collections API](#).

It is also exposed in Java, Python and R (as well as SQL, i.e. SparkSQL, in a sense).

So, when you have a need for distributed Collections API in Scala, Spark with RDD API should be a serious contender.

Rich Standard Library

Not only can you use `map` and `reduce` (as in Hadoop MapReduce jobs) in Spark, but also a vast array of other higher-level operators to ease your Spark queries and application development.

It expanded on the available computation styles beyond the only map-and-reduce available in Hadoop MapReduce.

Unified development and deployment environment for all

Regardless of the Spark tools you use - the Spark API for the many programming languages supported - Scala, Java, Python, R, or [the Spark shell](#), or the many [Spark Application Frameworks](#) leveraging the concept of [RDD](#), i.e. [Spark SQL](#), [Spark Streaming](#), [Spark MLlib](#) and [Spark GraphX](#), you still use the same development and deployment environment to for large data sets to yield a result, be it a prediction ([Spark MLlib](#)), a structured data queries ([Spark SQL](#)) or just a large distributed batch (Spark Core) or streaming (Spark Streaming) computation.

It's also very productive of Spark that teams can exploit the different skills the team members have acquired so far. Data analysts, data scientists, Python programmers, or Java, or Scala, or R, can all use the same Spark platform using tailor-made API. It makes for bringing skilled people with their expertise in different programming languages together to a Spark project.

Interactive exploration

It is also called *ad hoc queries*.

Using [the Spark shell](#) you can execute computations to process large amount of data (*The Big Data*). It's all interactive and very useful to explore the data before final production release.

Also, using the Spark shell you can access any [Spark cluster](#) as if it was your local machine. Just point the Spark shell to a 20-node of 10TB RAM memory in total (using `--master`) and use all the components (and their abstractions) like Spark SQL, Spark MLlib, [Spark Streaming](#), and Spark GraphX.

Depending on your needs and skills, you may see a better fit for SQL vs programming APIs or apply machine learning algorithms (Spark MLlib) from data in graph data structures (Spark GraphX).

Single environment

Regardless of which programming language you are good at, be it Scala, Java, Python or R, you can use the same single clustered runtime environment for prototyping, ad hoc queries, and deploying your applications leveraging the many ingestion data points offered by the Spark platform.

You can be as low-level as using RDD API directly or leverage higher-level APIs of Spark SQL (DataFrames), Spark MLlib (Pipelines), Spark GraphX (???), or [Spark Streaming](#) (DStreams).

Or use them all in a single application.

The single programming model and execution engine for different kinds of workloads simplify development and deployment architectures.

Rich set of supported data sources

Spark can read from many types of data sources - relational, NoSQL, file systems, etc.

Both, input and output data sources, allow programmers and data engineers use Spark as the platform with the large amount of data that is read from or saved to for processing, interactively (using Spark shell) or in applications.

Tools unavailable then, at your fingertips now

As much and often as it's recommended [to pick the right tool for the job](#), it's not always feasible. Time, personal preference, operating system you work on are all factors to decide what is right at a time (and using a hammer can be a reasonable choice).

Spark embraces many concepts in a single unified development and runtime environment.

- Machine learning that is so tool- and feature-rich in Python, e.g. SciKit library, can now be used by Scala developers (as Pipeline API in Spark MLlib or calling `pipe()`).
- DataFrames from R are available in Scala, Java, Python, R APIs.
- Single node computations in machine learning algorithms are migrated to their distributed versions in Spark MLlib.

This single platform gives plenty of opportunities for Python, Scala, Java, and R programmers as well as data engineers (SparkR) and scientists (using proprietary enterprise data warehousesthe with Thrift JDBC/ODBC server in Spark SQL).

Mind the proverb [if all you have is a hammer, everything looks like a nail](#), too.

Low-level Optimizations

Apache Spark uses a [directed acyclic graph \(DAG\)](#) of computation stages (aka **execution DAG**). It postpones any processing until really required for actions. Spark's **lazy evaluation** gives plenty of opportunities to induce low-level optimizations (so users have to know less to do more).

Mind the proverb [less is more](#).

Excels at low-latency iterative workloads

Spark supports diverse workloads, but successfully targets low-latency iterative ones. They are often used in Machine Learning and graph algorithms.

Many Machine Learning algorithms require plenty of iterations before the result models get optimal, like logistic regression. The same applies to graph algorithms to traverse all the nodes and edges when needed. Such computations can increase their performance when the interim partial results are stored in memory or at very fast solid state drives.

Spark can cache intermediate data in memory for faster model building and training. Once the data is loaded to memory (as an initial step), reusing it multiple times incurs no performance slowdowns.

Also, graph algorithms can traverse graphs one connection per iteration with the partial result in memory.

Less disk access and network can make a huge difference when you need to process lots of data, esp. when it is a BIG Data.

ETL done easier

Spark gives **Extract, Transform and Load (ETL)** a new look with the many programming languages supported - Scala, Java, Python (less likely R). You can use them all or pick the best for a problem.

Scala in Spark, especially, makes for a much less boiler-plate code (comparing to other languages and approaches like MapReduce in Java).

Unified API (for different computation models)

Spark offers one **unified API** for batch analytics, SQL queries, real-time analysis, machine learning and graph processing. Developers no longer have to learn many different processing engines per use case.

Different kinds of data processing using unified API

Spark offers three kinds of data processing using **batch**, **interactive**, and **stream processing** with the unified API and data structures.

Little to no disk use for better performance

In the no-so-long-ago times, when the most prevalent distributed computing framework was [Hadoop MapReduce](#), you could reuse a data between computation (even partial ones!) only after you've written it to an external storage like [Hadoop Distributed Filesystem \(HDFS\)](#). It can cost you a lot of time to compute even very basic multi-stage computations. It simply suffers from IO (and perhaps network) overhead.

One of the many motivations to build Spark was to have a framework that is good at data reuse.

Spark cuts it out in a way to keep as much data as possible in memory and keep it there until a job is finished. It doesn't matter how many stages belong to a job. What does matter is the available memory and how effective you are in using Spark API (so [no shuffle occur](#)).

The less network and disk IO, the better performance, and Spark tries hard to find ways to minimize both.

Fault Tolerance included

Faults are not considered a special case in Spark, but obvious consequence of being a parallel and distributed system. Spark handles and recovers from faults by default without particularly complex logic to deal with them.

Small Codebase Invites Contributors

Spark's design is fairly simple and the code that comes out of it is not huge comparing to the features it offers.

The reasonably small codebase of Spark invites project contributors - programmers who extend the platform and fix bugs in a more steady pace.

Overview

Apache Spark is an **open-source parallel distributed general-purpose cluster computing framework** with **in-memory big data processing engine** with programming interfaces (APIs) for the programming languages: Scala, Python, Java, and R.

Or, to have a one-liner, Apache Spark is a distributed, data processing engine for **batch and streaming modes** featuring SQL queries, graph processing, and Machine Learning.

In contrast to Hadoop's two-stage disk-based MapReduce processing engine, Spark's multi-stage in-memory computing engine allows for running most computations in memory, and hence very often provides better performance (there are reports about being up to 100 times faster - read [Spark officially sets a new record in large-scale sorting!](#)) for certain applications, e.g. iterative algorithms or interactive data mining.

Spark aims at speed, ease of use, and interactive analytics.

Spark is often called **cluster computing engine** or simply **execution engine**.

Spark is a **distributed platform for executing complex multi-stage applications**, like **machine learning algorithms**, and **interactive ad hoc queries**. Spark provides an efficient abstraction for in-memory cluster computing called [Resilient Distributed Dataset](#).

Using [Spark Application Frameworks](#), Spark simplifies access to machine learning and predictive analytics at scale.

Spark is mainly written in [Scala](#), but supports other languages, i.e. Java, Python, and R.

If you have large amounts of data that requires low latency processing that a typical MapReduce program cannot provide, Spark is an alternative.

- Access any data type across any data source.
- Huge demand for storage and data processing.

The Apache Spark project is an umbrella for [SQL](#) (with [DataFrames](#)), [streaming](#), [machine learning](#) (pipelines) and [graph](#) processing engines built atop Spark Core. You can run them all in a single application using a consistent API.

Spark runs locally as well as in clusters, on-premises or in cloud. It runs on top of Hadoop YARN, Apache Mesos, standalone or in the cloud (Amazon EC2 or IBM Bluemix).

Spark can access data from many [data sources](#).

Apache Spark's Streaming and SQL programming models with MLlib and GraphX make it easier for developers and data scientists to build applications that exploit machine learning and graph analytics.

At a high level, any Spark application creates **RDDs** out of some input, run ([lazy](#)) [transformations](#) of these RDDs to some other form (shape), and finally perform [actions](#) to collect or store data. Not much, huh?

You can look at Spark from programmer's, data engineer's and administrator's point of view. And to be honest, all three types of people will spend quite a lot of their time with Spark to finally reach the point where they exploit all the available features. Programmers use language-specific APIs (and work at the level of RDDs using transformations and actions), data engineers use higher-level abstractions like DataFrames or Pipelines APIs or external tools (that connect to Spark), and finally it all can only be possible to run because administrators set up Spark clusters to deploy Spark applications to.

It is Spark's goal to be a general-purpose computing platform with various specialized applications frameworks on top of a single unified engine.

In [Going from Hadoop to Spark: A Case Study, Sujee Maniyam 20150223](#):

Spark is like emacs - once you join emacs, you can't leave emacs.

Anatomy of Spark Application

Every Spark application starts at instantiating a [Spark context](#). Without a Spark context no computation can ever be started using Spark services.

Note

A Spark application is an instance of `SparkContext`. Or, put it differently, a Spark context constitutes a Spark application.

For it to work, you have to [create a Spark configuration using `SparkConf`](#) or use a [custom `SparkContext` constructor](#).

```
package pl.japila.spark

import org.apache.spark.{SparkContext, SparkConf}

object SparkMeApp {
  def main(args: Array[String]) {

    val masterURL = "local[*]" (1)

    val conf = new SparkConf() (2)
      .setAppName("SparkMe Application")
      .setMaster(masterURL)

    val sc = new SparkContext(conf) (3)

    val fileName = util.Try(args(0)).getOrElse("build.sbt")

    val lines = sc.textFile(fileName).cache() (4)

    val c = lines.count() (5)
    println(s"There are $c lines in $fileName")
  }
}
```

1. [Master URL](#) to connect the application to
2. Create Spark configuration
3. Create Spark context
4. Create `lines` RDD
5. Execute `count` action

Tip

[Spark shell](#) creates a Spark context and SQL context for you at startup.

When a Spark application starts (using [spark-submit script](#) or as a standalone application), it connects to [Spark master](#) as described by [master URL](#). It is part of [Spark context's initialization](#).

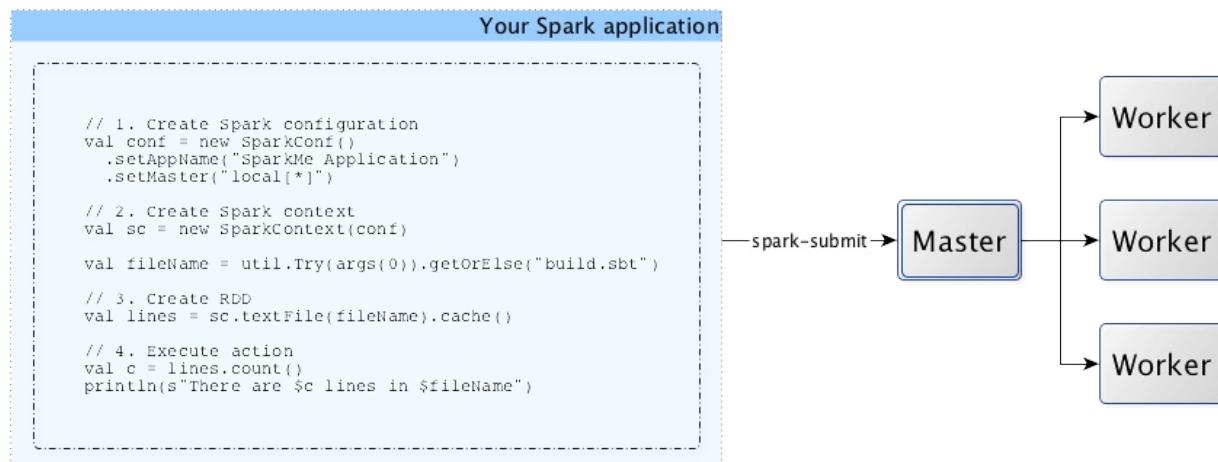


Figure 1. Submitting Spark application to master using master URL

Note	Your Spark application can run locally or on the cluster which is based on the cluster manager and the deploy mode (<code>--deploy-mode</code>). Refer to Deployment Modes .
------	--

You can then [create RDDs](#), [transform them to other RDDs](#) and ultimately [execute actions](#).

You can also [cache interim RDDs](#) to speed up data processing.

After all the data processing is completed, the Spark application finishes by [stopping the Spark context](#).

SparkConf - Configuration for Spark Applications

Tip	Refer to Spark Configuration for extensive coverage of how to configure Spark and user programs.
Caution	<p>TODO</p> <ul style="list-style-type: none"> • Describe <code>SparkConf</code> object for the application configuration. • the default configs • system properties

There are three ways to configure Spark and user programs:

- Spark Properties - use [Web UI](#) to learn the current properties.
- ...

Spark Properties

Every user program starts with creating an instance of `SparkConf` that holds the [master URL](#) to connect to (`spark.master`), the name for your Spark application (that is later displayed in [web UI](#) and becomes `spark.app.name`) and other Spark properties required for proper runs. An instance of `SparkConf` is then used to create [SparkContext](#).

Tip	<p>Start Spark shell with <code>--conf spark.logConf=true</code> to log the effective Spark configuration as INFO when <code>SparkContext</code> is started.</p> <pre>\$./bin/spark-shell --conf spark.logConf=true ... 15/10/19 17:13:49 INFO SparkContext: Running Spark version 1.6.0-SNAPSHOT 15/10/19 17:13:49 INFO SparkContext: Spark configuration: spark.app.name=Spark shell spark.home=/Users/jacek/dev/oss/spark spark.jars= spark.logConf=true spark.master=local[*] spark.repl.class.uri=http://10.5.10.20:64055 spark.submit.deployMode=client ...</pre> <p>Use <code>sc.getConf.toDebugString</code> to have a richer output once <code>SparkContext</code> has finished initializing.</p>
-----	---

You can query for the values of Spark properties in [Spark shell](#) as follows:

```
scala> sc.getConf.getOption("spark.local.dir")
res0: Option[String] = None

scala> sc.getConf.getOption("spark.app.name")
res1: Option[String] = Some(Spark shell)

scala> sc.getConf.get("spark.master")
res2: String = local[*]
```

Setting up Properties

There are the following ways to set up properties for Spark and user programs (in the order of importance from the least important to the most important):

- `conf/spark-defaults.conf` - the default
- `--conf` - the command line option used by `spark-shell` and `spark-submit`
- `SparkConf`

Default Configuration

The default Spark configuration is created when you execute the following code:

```
import org.apache.spark.SparkConf
val conf = new SparkConf
```

It merely loads any `spark.*` system properties.

You can use `conf.toDebugString` or `conf.getAll` to have the `spark.*` system properties loaded printed out.

```
scala> conf.getAll
res0: Array[(String, String)] = Array((spark.app.name,Spark shell), (spark.jars,""), (spa

scala> conf.toDebugString
res1: String =
spark.app.name=Spark shell
spark.jars=
spark.master=local[*]
spark.submit.deployMode=client

scala> println(conf.toDebugString)
spark.app.name=Spark shell
spark.jars=
spark.master=local[*]
spark.submit.deployMode=client
```



SparkContext - the door to Spark

SparkContext (aka **Spark context**) represents the connection to a [Spark execution environment \(deployment mode\)](#).

You have to create a Spark context before using Spark features and services in your application. A Spark context can be used to [create RDDs, accumulators and broadcast variables](#), access Spark services and [run jobs](#).

A Spark context is essentially a client of Spark's execution environment and acts as the *master of your Spark application* (don't get confused with the other meaning of [Master](#) in Spark, though).

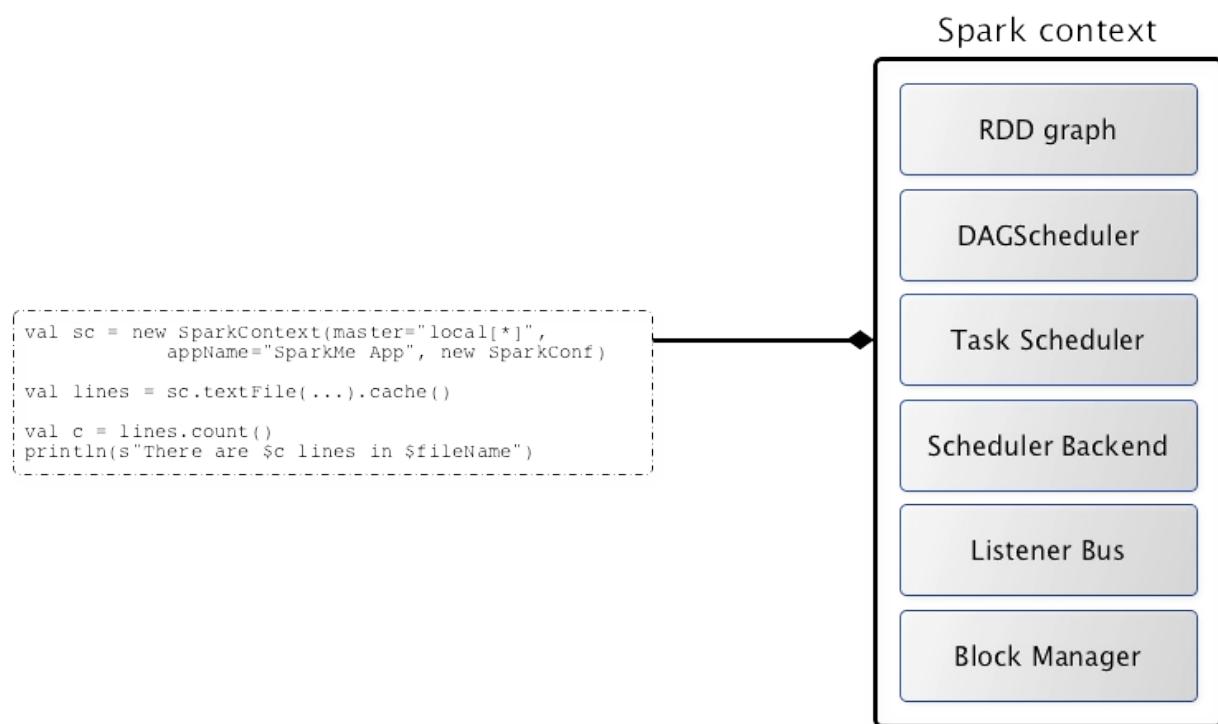


Figure 1. Spark context acts as the master of your Spark application

`SparkContext` offers the following functions:

- [Default Level of Parallelism](#)
- [Specifying mandatory master URL](#)
- [Specifying mandatory application name](#)
- [Creating RDDs](#)
- [Creating accumulators](#)
- [Creating broadcast variables](#)

- Accessing services, e.g. [Task Scheduler](#), [Listener Bus](#), [Block Manager](#), [Scheduler Backends](#), [Shuffle Manager](#).
- Running jobs
- Setting up custom Scheduler Backend, Task Scheduler and DAGScheduler
- Closure Cleaning
- Submitting Jobs Asynchronously
- Unpersisting RDDs, i.e. marking RDDs as non-persistent

Read the scaladoc of [org.apache.spark.SparkContext](#).

Master URL

Caution	FIXME
---------	-----------------------

[Connecting to a cluster](#)

Application Name

Caution	FIXME
---------	-----------------------

[Specifying mandatory application name](#)

Default Level of Parallelism

Default level of parallelism is the number of [partitions](#) when not specified explicitly by a user.

It is used for the methods like `SparkContext.parallelize`, `SparkContext.range` and `SparkContext.makeRDD` (as well as [Spark Streaming's](#) `DStream.countByValue` and `DStream.countByValueAndWindow` and few other places). It is also used to instantiate [HashPartitioner](#) or for the minimum number of partitions in [HadoopRDDs](#).

`SparkContext` queries [TaskScheduler](#) for the default level of parallelism (refer to [TaskScheduler Contract](#)).

SparkContext.makeRDD

Caution	FIXME
---------	-----------------------

Submitting Jobs Asynchronously

`SparkContext.submitJob` submits a job in an asynchronous, non-blocking way (using `DAGScheduler.submitJob` method).

It cleans the `processPartition` input function argument and returns an instance of `SimpleFutureAction` that holds the `JobWaiter` instance (it has received from `DAGScheduler.submitJob`).

Caution	FIXME What are <code>resultFunc</code> ?
---------	--

It is used in:

- [AsyncRDDActions](#) methods
- [Spark Streaming](#) for `ReceiverTrackerEndpoint.startReceiver`

Spark Configuration

Caution	FIXME
---------	-----------------------

Creating SparkContext

You create a `SparkContext` instance using a [SparkConf](#) object.

```
scala> import org.apache.spark.SparkConf
import org.apache.spark.SparkConf

scala> val conf = new SparkConf().setMaster("local[*]").setAppName("SparkMe App")
conf: org.apache.spark.SparkConf = org.apache.spark.SparkConf@7a8f69d6

scala> import org.apache.spark.SparkContext
import org.apache.spark.SparkContext

scala> val sc = new SparkContext(conf)  (1)
sc: org.apache.spark.SparkContext = org.apache.spark.SparkContext@50ee2523
```

1. You can also use the other constructor of `SparkContext` , i.e. `new SparkContext(master="local[*]", appName="SparkMe App", new SparkConf)` , with master and application name specified explicitly

When a Spark context starts up you should see the following INFO in the logs (amongst the other messages that come from services):

```
INFO SparkContext: Running Spark version 1.6.0-SNAPSHOT
```

Only one SparkContext may be running in a single JVM (check out [SPARK-2243 Support multiple SparkContexts in the same JVM](#)). Sharing access to a SparkContext in the JVM is the solution to share data within Spark (without relying on other means of data sharing using external data stores).

spark.driver.allowMultipleContexts

Quoting the scaladoc of [org.apache.spark.SparkContext](#):

Only one SparkContext may be active per JVM. You must `stop()` the active SparkContext before creating a new one.

The above quote is not necessarily correct when `spark.driver.allowMultipleContexts` is `true` (default: `false`). If `true`, Spark logs warnings instead of throwing exceptions when multiple SparkContexts are active, i.e. multiple SparkContext are running in this JVM. When creating an instance of `SparkContext`, Spark marks the current thread as having it being created (very early in the instantiation process).

Caution

It's not guaranteed that Spark will work properly with two or more SparkContexts. Consider the feature a work in progress.

SparkContext and RDDs

You use a Spark context to create RDDs (see [Creating RDD](#)).

When an RDD is created, it belongs to and is completely owned by the Spark context it originated from. RDDs can't by design be shared between SparkContexts.

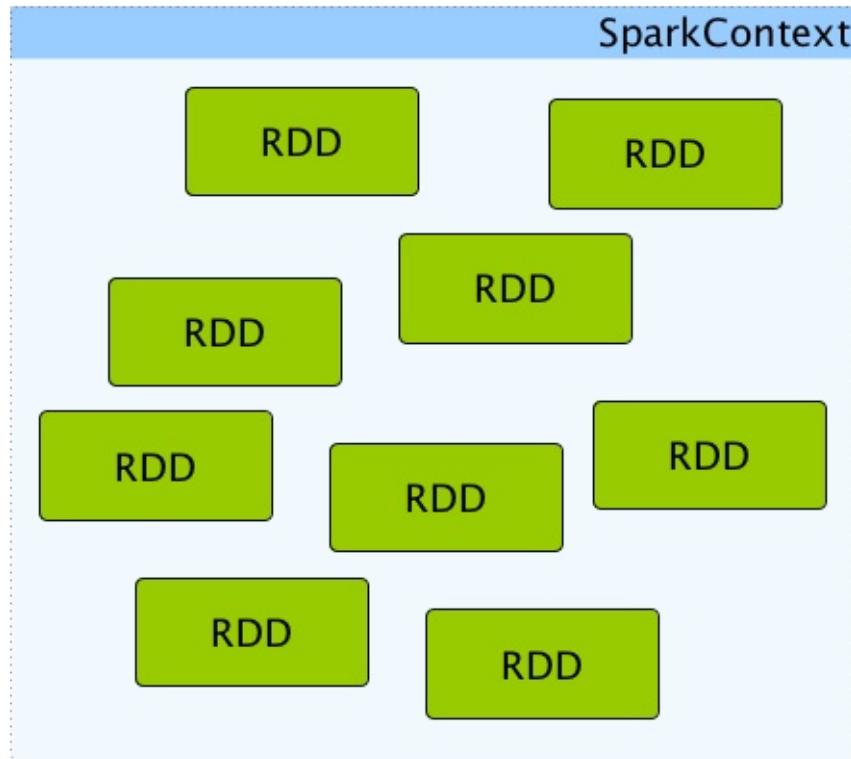


Figure 2. A Spark context creates a living space for RDDs.

SparkContext in Spark shell

In [Spark shell](#), an instance of `sparkContext` is automatically created for you under the name `sc`.

Read [Spark shell](#).

Creating RDD

`SparkContext` allows you to create many different RDDs from input sources like:

- Scala's collections, i.e. `sc.parallelize(0 to 100)`
- local or remote filesystems, i.e. `sc.textFile("README.md")`
- Any Hadoop `InputSource` using `sc.newAPIHadoopFile`

Read [Creating RDDs in RDD - Resilient Distributed Dataset](#).

Unpersisting RDDs (Marking RDDs as non-persistent)

It removes an RDD from the master's [Block Manager](#) (calls `removeRdd(rddId: Int, blocking: Boolean)`) and the internal `persistentRdds` mapping.

It finally posts an unpersist notification (as `SparkListenerUnpersistRDD` event) to `listenerBus`.

Setting Checkpoint Directory (`setCheckpointDir` method)

```
setCheckpointDir(directory: String)
```

`setCheckpointDir` method is used to set up the checkpoint directory...[FIXME](#)

Caution	FIXME
---------	-----------------------

Creating accumulators

Caution	FIXME
---------	-----------------------

Read [Use accumulators](#).

Creating Broadcast Variables

Tip	Consult Broadcast Variables to learn about broadcast variables.
-----	---

SparkContext comes with `broadcast` method to broadcast a value among Spark executors.

```
def broadcast[T: ClassTag](value: T): Broadcast[T]
```

It returns a `Broadcast[T]` instance that is a handle to a shared memory on executors.

```
scala> sc.broadcast("hello")
INFO MemoryStore: Ensuring 1048576 bytes of free space for block broadcast_0(free: 535953
INFO MemoryStore: Ensuring 80 bytes of free space for block broadcast_0(free: 535953408,
INFO MemoryStore: Block broadcast_0 stored as values in memory (estimated size 80.0 B, fr
INFO MemoryStore: Ensuring 34 bytes of free space for block broadcast_0_piece0(free: 5359
INFO MemoryStore: Block broadcast_0_piece0 stored as bytes in memory (estimated size 34.0
INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on localhost:61505 (size: 34.0
INFO SparkContext: Created broadcast 0 from broadcast at <console>:25
res0: org.apache.spark.broadcast.Broadcast[String] = Broadcast(0)
```

Spark transfers the value to Spark executors *once*, and tasks can share it without incurring repetitive network transmissions when requested multiple times.

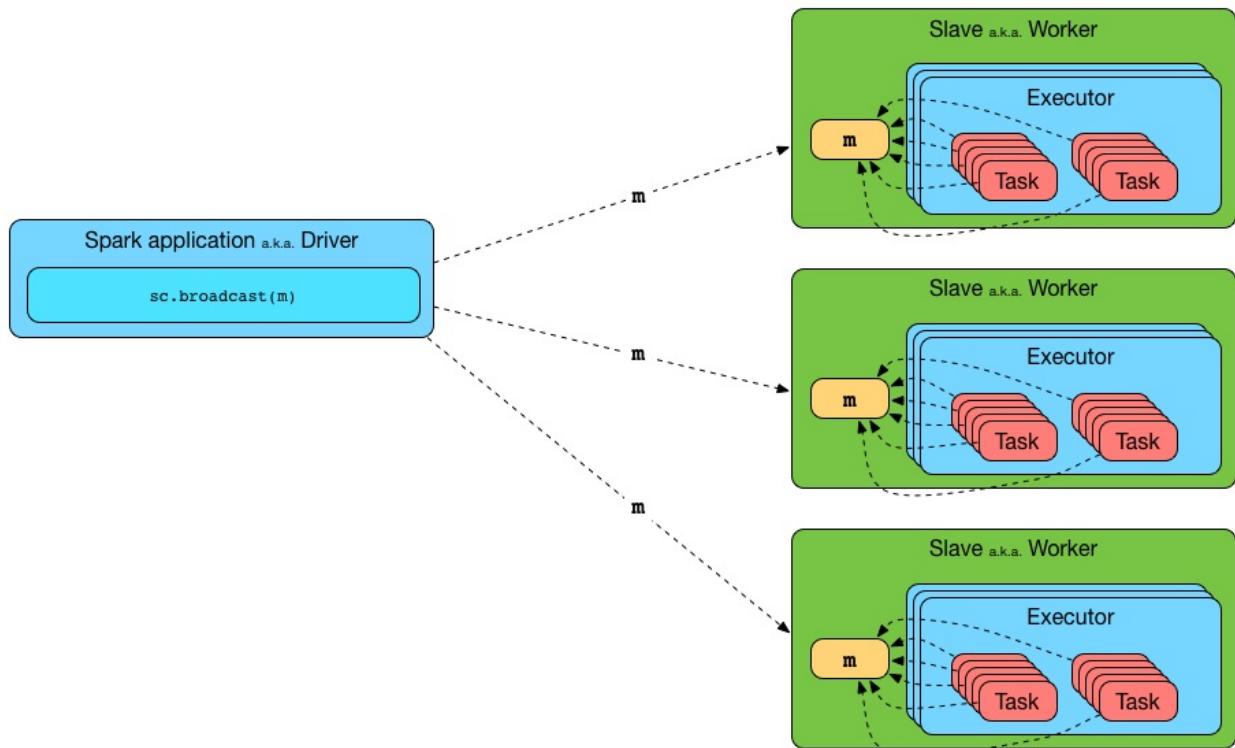


Figure 3. Broadcasting a value to executors

When a broadcast value is created the following INFO message appears in the logs:

```
INFO SparkContext: Created broadcast [id] from broadcast at <console>:25
```

You should *not* broadcast a RDD to use in tasks and Spark will warn you. It will not stop you, though. Consult [SPARK-5063 Display more helpful error messages for several invalid operations](#).

```
scala> val rdd = sc.parallelize(0 to 10)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>

scala> sc.broadcast(rdd)
WARN SparkContext: Can not directly broadcast RDDs; instead, call collect() and broadcast
```

Distribute JARs to workers

The jar you specify with `SparkContext.addJar` will be copied to all the worker nodes.

The configuration setting `spark.jars` is a comma-separated list of jar paths to be included in all tasks executed from this SparkContext. A path can either be a local file, a file in HDFS (or other Hadoop-supported filesystems), an HTTP, HTTPS or FTP URI, or `local:/path` for a file on every worker node.

```
scala> sc.addJar("build.sbt")
15/11/11 21:54:54 INFO SparkContext: Added JAR build.sbt at http://192.168.1.4:49427/jars
```

Caution

FIXME Why is HttpFileServer used for addJar?

SparkContext as the global configuration for services

SparkContext keeps track of:

- shuffle ids using `nextShuffleId` internal field for [registering shuffle dependencies to Shuffle Service](#).

Running Jobs

All [RDD actions](#) in Spark launch [jobs](#) (that are run on one or many partitions of the RDD) using `SparkContext.runJob(rdd: RDD[T], func: Iterator[T] => U): Array[U]` .

Tip

For some actions like `first()` and `lookup()`, there is no need to compute all the partitions of the RDD in a job. And Spark knows it.

```
scala> import org.apache.spark.TaskContext
import org.apache.spark.TaskContext

scala> sc.runJob(lines, (t: TaskContext, i: Iterator[String]) => 1) (1)
res0: Array[Int] = Array(1, 1) (2)
```

1. Run a job using `runJob` on `lines` RDD with a function that returns 1 for every partition (of `lines` RDD).
2. What can you say about the number of partitions of the `lines` RDD? Is your result `res0` different than mine? Why?

Running a job is essentially executing a `func` function on all or a subset of partitions in an `rdd` RDD and returning the result as an array (with elements being the results per partition).

`SparkContext.runJob` prints out the following INFO message:

```
INFO Starting job: ...
```

And it follows up on [spark.logLineage](#) and then hands over the execution to [DAGScheduler.runJob\(\)](#).

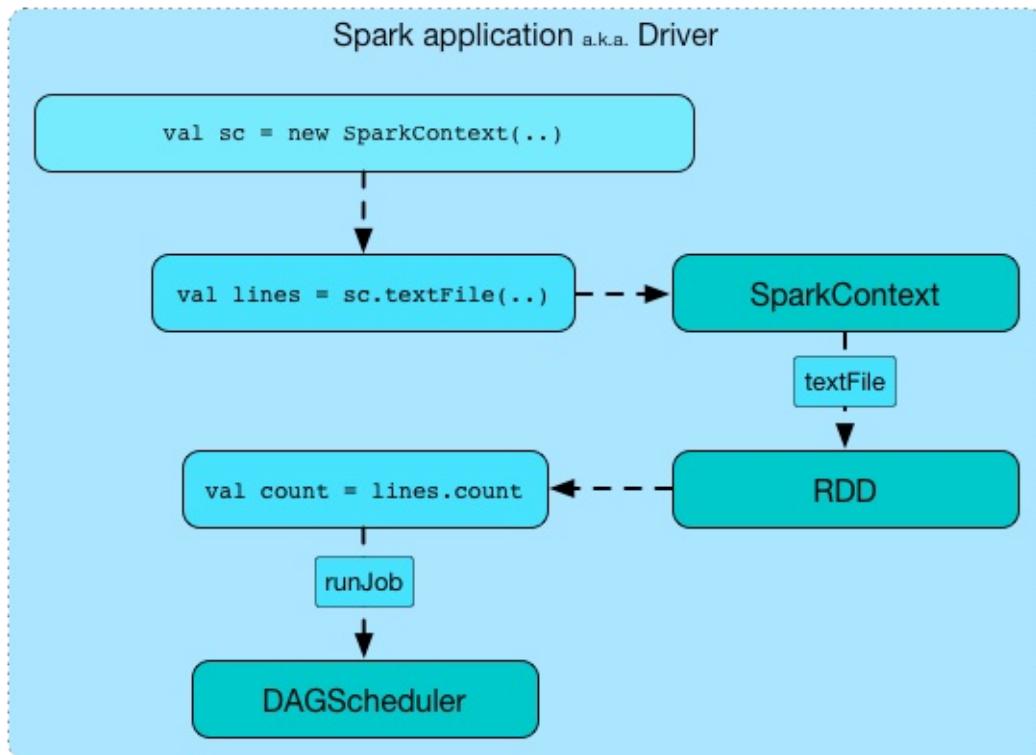


Figure 4. Executing action

Before the method finishes, it does [checkpointing](#) and posts `JobSubmitted` event (see [Event loop](#)).

Caution

Spark can only run jobs when a Spark context is available and active, i.e. started. See [Stopping Spark context](#).

Since `SparkContext` runs inside a Spark driver, i.e. a Spark application, it must be alive to run jobs.

Stopping `SparkContext`

You can stop a Spark context using `sparkContext.stop()` method. Stopping a Spark context stops the [Spark Runtime Environment](#) and effectively shuts down the entire Spark application (see [Anatomy of Spark Application](#)).

Calling `stop` many times leads to the following INFO message in the logs:

```
INFO SparkContext: SparkContext already stopped.
```

An attempt to use a stopped `SparkContext`'s services will result in

```
java.lang.IllegalStateException: SparkContext has been shutdown .
```

```
scala> sc.stop

scala> sc.parallelize(0 to 5)
java.lang.IllegalStateException: Cannot call methods on a stopped SparkContext.
```

When a SparkContext is being stopped, it does the following:

- Posts a application end event `SparkListenerApplicationEnd` to [Listener Bus](#)
- Stops [web UI](#)
- Requests [MetricSystem](#) to report metrics from all registered sinks (using `MetricsSystem.report()`)
- `metadataCleaner.cancel()`
- Stops [ContextCleaner](#)
- Stops [ExecutorAllocationManager](#)
- Stops [DAGScheduler](#)
- Stops [Listener Bus](#)
- Stops [EventLoggingListener](#)
- Stops [HeartbeatReceiver](#)
- Stops [ConsoleProgressBar](#)
- Stops [SparkEnv](#)

If all went fine you should see the following INFO message in the logs:

```
INFO SparkContext: Successfully stopped SparkContext
```

HeartbeatReceiver

Caution	FIXME
---------	-----------------------

`HeartbeatReceiver` is a [SparkListener](#).

Custom SchedulerBackend, TaskScheduler and DAGScheduler

By default, `SparkContext` uses (`private[spark] class`)

`org.apache.spark.scheduler.DAGScheduler`, but you can develop your own custom `DAGScheduler` implementation, and use (`private[spark]`) `SparkContext.dagScheduler_= (ds: DAGScheduler)` method to assign yours.

It is also applicable to `SchedulerBackend` and `TaskScheduler` using `schedulerBackend_= (sb: SchedulerBackend)` and `taskScheduler_= (ts: TaskScheduler)` methods, respectively.

Caution

FIXME Make it an advanced exercise.

Creating SchedulerBackend and TaskScheduler

`SparkContext.createTaskScheduler` is executed as part of [SparkContext's initialization](#) to create [Task Scheduler](#) and [Scheduler Backend](#).

It uses the [master URL](#) to select right implementations.

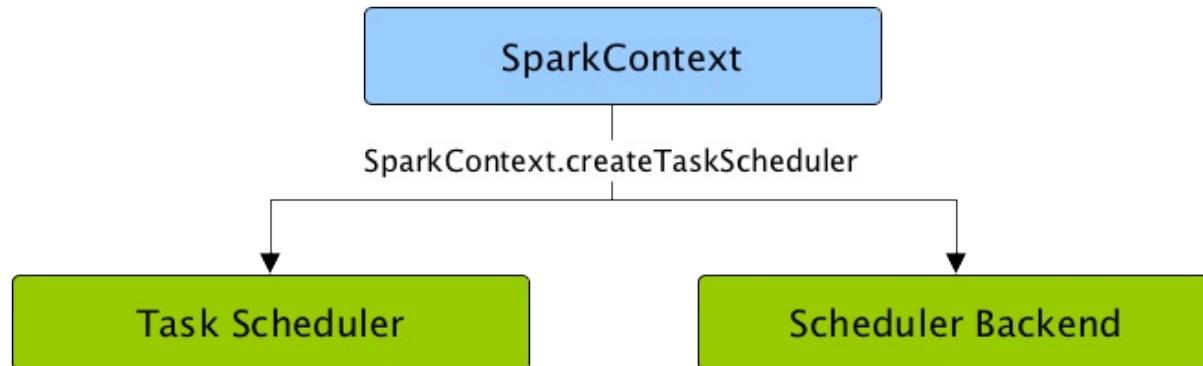


Figure 5. `SparkContext` creates Task Scheduler and Scheduler Backend

Events

When a Spark context starts, it triggers `SparkListenerEnvironmentUpdate` and `SparkListenerApplicationStart` events.

Refer to the section [SparkContext's initialization](#).

Persisted RDDs

FIXME When is the internal field `persistentRdds` used?

Setting Default Log Level Programmatically

You can use `SparkContext.setLogLevel(logLevel: String)` to adjust logging level in a Spark application, e.g. [Spark shell](#).

Tip

`sc.setLogLevel("INFO")` becomes `org.apache.log4j.Level.toLevel(logLevel)` and `org.apache.log4j.Logger.getRootLogger().setLevel(1)` internally.
See [org/apache/spark/SparkContext.scala](#).

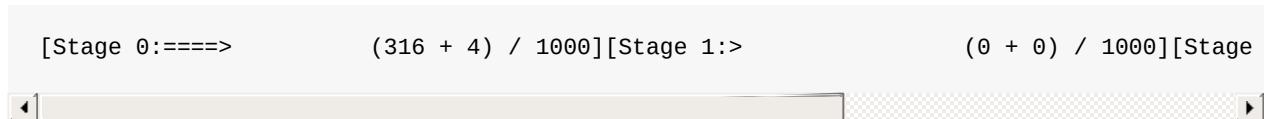
SparkStatusTracker

`SparkStatusTracker` requires a Spark context to work. It is created as part of `SparkContext's initialization`.

`SparkStatusTracker` is only used by [ConsoleProgressBar](#).

ConsoleProgressBar

`ConsoleProgressBar` shows the progress of active stages in console (to `stderr`). It polls the status of stages from `SparkStatusTracker` periodically and prints out active stages with more than one task. It keeps overwriting itself to hold in one line for at most 3 first concurrent stages at a time.



The progress includes the stage's id, the number of completed, active, and total tasks.

It is useful when you `ssh` to workers and want to see the progress of active stages.

It is only instantiated if the value of the boolean property `spark.ui.showConsoleProgress` (default: `true`) is `true` and the log level of `org.apache.spark.SparkContext` logger is `WARN` or higher (refer to [Logging](#)).

```
import org.apache.log4j._
Logger.getLogger("org.apache.spark.SparkContext").setLevel(Level.WARN)
```

To print the progress nicely `ConsoleProgressBar` uses `COLUMNS` environment variable to know the width of the terminal. It assumes `80` columns.

The progress bar prints out the status after a stage has ran at least `500ms`, every `200ms` (the values are not configurable).

See the progress bar in Spark shell with the following:

```
$ ./bin/spark-shell --conf spark.ui.showConsoleProgress=true (1)

scala> sc.setLogLevel("OFF") (2)

scala> Logger.getLogger("org.apache.spark.SparkContext").setLevel(Level.WARN) (3)

scala> sc.parallelize(1 to 4, 4).map { n => Thread.sleep(500 + 200 * n); n }.count (4)
[Stage 2:> (0 + 4) / 4]
[Stage 2:=====> (1 + 3) / 4]
[Stage 2:=====> (2 + 2) / 4]
[Stage 2:=====> (3 + 1) / 4]
```

1. Make sure `spark.ui.showConsoleProgress` is `true`. It is by default.
2. Disable (`OFF`) the root logger (that includes Spark's logger)
3. Make sure `org.apache.spark.SparkContext` logger is at least `WARN`.
4. Run a job with 4 tasks with 500ms initial sleep and 200ms sleep chunks to see the progress bar.

[Watch the short video](#) that show ConsoleProgressBar in action.

You may want to use the following example to see the progress bar in full glory - all 3 concurrent stages in console (borrowed from a comment to [\[SPARK-4017\] show progress bar in console #3029](#)):

```
> ./bin/spark-shell --conf spark.scheduler.mode=FAIR
scala> val a = sc.makeRDD(1 to 1000, 10000).map(x => (x, x)).reduceByKey(_ + _)
scala> val b = sc.makeRDD(1 to 1000, 10000).map(x => (x, x)).reduceByKey(_ + _)
scala> a.union(b).count()
```

Closure Cleaning (clean method)

Every time an action is called, Spark cleans up the closure, i.e. the body of the action, before it is serialized and sent over the wire to executors.

SparkContext comes with `clean(f: F, checkSerializable: Boolean = true)` method that does this. It in turn calls `closureCleaner.clean` method.

Not only does `closureCleaner.clean` method clean the closure, but also does it transitively, i.e. referenced closures are cleaned transitively.

A closure is considered serializable as long as it does not explicitly reference unserializable objects. It does so by traversing the hierarchy of enclosing closures and null out any references that are not actually used by the starting closure.

Tip

Enable `DEBUG` logging level for `org.apache.spark.util.ClosureCleaner` logger to see what happens inside the class.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.util.ClosureCleaner=DEBUG
```

With `DEBUG` logging level you should see the following messages in the logs:

```
+++ Cleaning closure [func] ([func.getClass.getName]) ===+
+ declared fields: [declaredFields.size]
 [field]
 ...
+++ closure [func] ([func.getClass.getName]) is now cleaned +++
```

Serialization is verified using a new instance of `Serializer` (as `SparkEnv.get.closureSerializer`). Refer to [Serialization](#).

Caution

[FIXME](#) an example, please.

Creating a SparkContext

Let's walk through a typical initialization code of `SparkContext` in a Spark application and see what happens under the covers.

```
import org.apache.spark.{SparkConf, SparkContext}

// 1. Create Spark configuration
val conf = new SparkConf()
.setAppName("SparkMe Application")
.setMaster("local[*]")

// 2. Create Spark context
val sc = new SparkContext(conf)
```

Note

The example uses Spark in [local mode](#), i.e. `setMaster("local[*]")` , but the initialization with [the other cluster modes](#) would follow similar steps.

It all starts with checking [whether SparkContexts can be shared or not](#) using `spark.driver.allowMultipleContexts` .

The very first information printed out is the version of Spark as an INFO message:

```
INFO SparkContext: Running Spark version 1.6.0-SNAPSHOT
```

An instance of [Listener Bus](#) is created (but not started yet).

The current user name is computed, i.e. read from a value of `SPARK_USER` environment variable or the currently logged-in user. It is available as later on as `sparkUser`.

```
scala> sc.sparkUser
res0: String = jacek
```

Caution

[FIXME](#) Where is `sparkUser` useful?

The initialization then checks whether a master URL as `spark.master` and an application name as `spark.app.name` are defined. `SparkException` is thrown if not.

When `spark.logConf` is `true` (default: `false`) [SparkConf.toDebugString](#) is called.

Note

`SparkConf.toDebugString` is called very early in the initialization process and other settings configured afterwards are not included. Use `sc.getConf.toDebugString` once `SparkContext` is initialized.

The Spark driver host (`spark.driver.host` to `localhost`) and port (`spark.driver.port` to `0`) system properties are set unless they are already defined.

`spark.executor.id` is set as `driver`.

Tip

Use `sc.getConf.get("spark.executor.id")` to know where the code is executed - [driver or executors](#).

It sets the jars and files based on `spark.jars` and `spark.files`, respectively. These are files that are required for proper task execution on executors.

If `spark.eventLog.enabled` was `true` (default: `false`), the internal field `_eventLogDir` is set to the value of `spark.eventLog.dir` property or simply `/tmp/spark-events`. Also, if `spark.eventLog.compress` is `true` (default: `false`), the short name of the `CompressionCodec` is assigned to `_eventLogCodec`. The config key is `spark.io.compression.codec` (default: `snappy`). The supported codecs are: `lz4`, `lzf`, and `snappy` or their short class names.

It sets `spark.externalBlockStore.folderName` to the value of `externalBlockStoreFolderName`.

Caution

[FIXME](#): What's `externalBlockStoreFolderName`?

For `yarn-client` master URL, the system property `SPARK_YARN_MODE` is set to `true`.

An instance of [JobProgressListener](#) is created and registered to [Listener Bus](#).

A [Spark execution environment](#) (`SparkEnv`) is created (using `createSparkEnv` that in turn calls `SparkEnv.createDriverEnv`).

`MetadataCleaner` is created.

Caution	FIXME What's <code>MetadataCleaner</code> ?
---------	---

Optional [ConsoleProgressBar](#) with [SparkStatusTracker](#) are created.

`SparkUI.createLiveUI` gets called to set `_ui` if the property `spark.ui.enabled` (default: `true`) is `true`.

Caution	FIXME Step through <code>SparkUI.createLiveUI</code> . Where's <code>_ui</code> used?
---------	---

A Hadoop configuration is created. See [Hadoop Configuration](#).

If there are jars given through the `SparkContext` constructor, they are added using `addJar`. Same for files using `addFile`.

At this point in time, the amount of memory to allocate to each executor (as `_executorMemory`) is calculated. It is the value of `spark.executor.memory` setting, or `SPARK_EXECUTOR_MEMORY` environment variable (or currently-deprecated `SPARK_MEM`), or defaults to `1024`.

`_executorMemory` is later available as `sc.executorMemory` and used for [LOCAL_CLUSTER_REGEX](#), [Spark Standalone's SparkDeploySchedulerBackend](#), to set `executorEnvs("SPARK_EXECUTOR_MEMORY")`, [MesosSchedulerBackend](#), [CoarseMesosSchedulerBackend](#).

The value of `SPARK_PREPEND_CLASSES` environment variable is included in `executorEnvs`.

Caution	FIXME <ul style="list-style-type: none"> • What's <code>_executorMemory</code> ? • What's the unit of the value of <code>_executorMemory</code> exactly? • What are "SPARK_TESTING", "spark.testing"? How do they contribute to <code>executorEnvs</code> ? • What's <code>executorEnvs</code> ?
---------	--

The Mesos scheduler backend's configuration is included in `executorEnvs`, i.e. `SPARK_EXECUTOR_MEMORY`, `_conf.getExecutorEnv`, and `SPARK_USER`.

HeartbeatReceiver RPC Endpoint is created using `HeartbeatReceiver` (as `_heartbeatReceiver`). Refer to [HeartbeatReceiver](#).

`SparkContext.createTaskScheduler` is executed (using the master URL).

Caution

FIXME Why is `_heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet)` important?

[Task Scheduler](#) is started.

The internal fields, `_applicationId` and `_applicationAttemptId`, are set. Application and attempt ids are specific to the implementation of [Task Scheduler](#).

The setting `spark.app.id` is set to `_applicationId` and Web UI gets notified about the new value (using `setappId(_applicationId)`). And also Block Manager (using `initialize(_applicationId)`).

```
scala> sc.getConf.get("spark.app.id")
res1: String = local-1447834845413
```

Caution

FIXME Why should UI and Block Manager know about the application id?

[Metric System](#) is started (after the application id is set using `spark.app.id`).

Caution

FIXME Why does Metric System need the application id?

The driver's metrics (servlet handler) are attached to the web ui after the metrics system is started.

`_eventLogger` is created and started if `isEventLogEnabled`. It uses [EventLoggingListener](#) that gets registered to [Listener Bus](#).

Caution

FIXME Why is `_eventLogger` required to be the internal field of `SparkContext`? Where is this used?

If [dynamic allocation](#) is enabled, `_executorAllocationManager` is set to `ExecutorAllocationManager` and started.

`_cleaner` is set to [ContextCleaner](#) if `spark.cleaner.referenceTracking` is `true` (default: `true`).

Caution

FIXME It'd be quite useful to have all the properties with their default values in `sc.getConf.toDebugString`, so when a configuration is not included but does change Spark runtime configuration, it should be added to `_conf`.

`setupAndStartListenerBus` registers user-defined listeners and starts [Listener Bus](#) that starts event delivery to the listeners.

`postEnvironmentUpdate` is called to post `SparkListenerEnvironmentUpdate` event over [Listener Bus](#) with information about Task Scheduler's scheduling mode, added jar and file paths, and other environmental details. They are displayed in [Web UI's Environment tab](#).

`postApplicationStart` is called to post `SparkListenerApplicationStart` event over [Listener Bus](#).

`TaskScheduler.postStartHook()` is called (see [TaskScheduler Contract](#))

Two new metrics sources are registered (via `_env.metricsSystem`):

- [BlockManagerSource](#)
- [ExecutorAllocationManagerSource](#) (only when `_executorAllocationManager` is set)

`ShutdownHookManager.addShutdownHook()` is called to do `SparkContext`'s cleanup.

Caution	FIXME What exactly does <code>ShutdownHookManager.addShutdownHook()</code> do?
---------	--

Any non-fatal Exception leads to termination of the `Spark context` instance.

Caution	FIXME What does <code>NonFatal</code> represent in Scala?
---------	---

`nextShuffleId` and `nextRddId` start with `0`.

Caution	FIXME Where are <code>nextShuffleId</code> and <code>nextRddId</code> used?
---------	---

A new instance of `Spark context` is created and ready for operation.

Hadoop Configuration

While a [SparkContext is created](#), so is a Hadoop configuration (as an instance of `org.apache.hadoop.conf.Configuration` that is available as `_hadoopConfiguration`).

Note	SparkHadoopUtil.get.newConfiguration is used.
------	---

If a `SparkConf` is provided it is used to build the configuration as described. Otherwise, the default `Configuration` object is returned.

If `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` are both available, the following settings are set for the Hadoop configuration:

- `fs.s3.awsAccessKeyId`, `fs.s3n.awsAccessKeyId`, `fs.s3a.access.key` are set to the value of `AWS_ACCESS_KEY_ID`
- `fs.s3.awsSecretAccessKey`, `fs.s3n.awsSecretAccessKey`, and `fs.s3a.secret.key` are set to the value of `AWS_SECRET_ACCESS_KEY`

Every `spark.hadoop.` setting becomes a setting of the configuration with the prefix `spark.hadoop.` removed for the key.

The value of `spark.buffer.size` (default: `65536`) is used as the value of `io.file.buffer.size`.

Environment Variables

- `SPARK_EXECUTOR_MEMORY` sets the amount of memory to allocate to each executor. See [Executor Memory](#).

RDD - Resilient Distributed Dataset

Introduction

The origins of RDD

The original paper that gave birth to the concept of RDD is [Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing](#) by Matei Zaharia, et al.

Resilient Distributed Dataset (RDD) is the primary data abstraction in Spark. It is a distributed collection of items. It is the bread and butter of Spark, and mastering the concept is of utmost importance to become a Spark pro. *And you wanna be a Spark pro, don't you?*

With RDD the creators of Spark managed to hide data partitioning and so distribution that in turn allowed them to design parallel computational framework with a higher-level programming interface (API) for four mainstream programming languages.

Learning about RDD by its name:

- **Resilient**, i.e. fault-tolerant and so able to recompute missing or damaged partitions on node failures with the help of [RDD lineage graph](#).
- **Distributed** across [clusters](#).
- **Dataset** is a collection of [partitioned data](#).

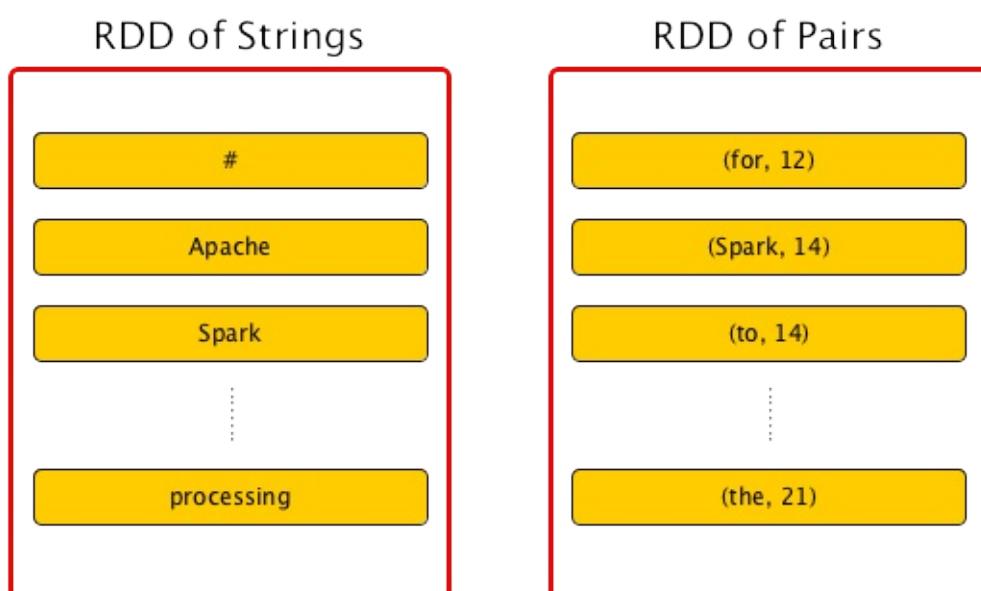


Figure 1. RDDs

From the scaladoc of [org.apache.spark.rdd.RDD](#):

A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable, partitioned collection of elements that can be operated on in parallel.

From the original paper about RDD - [Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing](#):

Resilient Distributed Datasets (RDDs) are a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.

Beside the above traits (that are directly embedded in the name of the data abstraction - RDD) it has the following additional traits:

- **Immutable**, i.e. it does not change once created.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.

RDDs are distributed by design and to achieve even **data distribution** as well as leverage **data locality** (in distributed systems like HDFS or Cassandra in which data is partitioned by default), they are **partitioned** to a fixed number of **partitions** - logical chunks (parts) of data. The logical division is for processing only and internally it is not divided whatsoever. Each partition comprises of **records**.

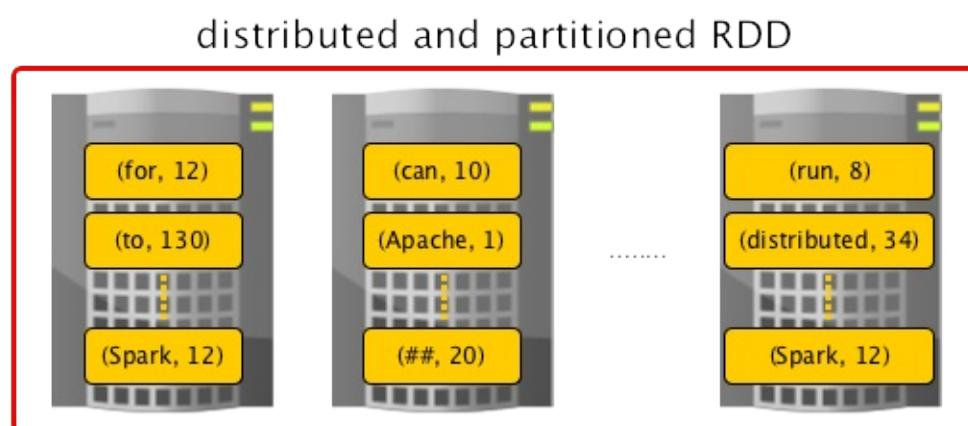


Figure 2. RDDs

Partitions are the units of parallelism. You can control the number of partitions of a RDD using `repartition` or `coalesce` operations. Spark tries to be as close to data as possible without wasting time to send data across network by means of **RDD shuffling**, and creates

as many partitions as required to follow the storage layout and thus optimize data access. It leads to a one-to-one mapping between (physical) data in distributed data storage, e.g. HDFS or Cassandra, and partitions.

RDDs support two kinds of operations:

- [transformations](#) - lazy operations that return another RDD.
- [actions](#) - operations that trigger computation and return values.

The motivation to create RDD were ([after the authors](#)) two types of applications that current computing frameworks handle inefficiently:

- **iterative algorithms** in machine learning and graph computations.
- **interactive data mining tools** as ad-hoc queries on the same dataset.

The goal is to reuse intermediate in-memory results across multiple data-intensive workloads with no need for copying large amounts of data over the network.

Each RDD is characterized by five main properties:

- An array of [partitions](#) that a dataset is divided to
- A function to do a computation for a partition
- List of [parent RDDs](#)
- An optional [partitioner](#) that defines how keys are hashed, and the pairs partitioned (for key-value RDDs)
- Optional [preferred locations](#), i.e. hosts for a partition where the data will have been loaded.

This RDD abstraction supports an expressive set of operations without having to modify scheduler for each one.

An RDD is a named (by [name](#)) and uniquely identified (by [id](#)) entity inside a [SparkContext](#). It lives in a SparkContext and as a SparkContext creates a logical boundary, RDDs can't be shared between SparkContexts (see [SparkContext and RDDs](#)).

An RDD can optionally have a friendly name accessible using `name` that can be changed using `=`:

```

scala> val ns = sc.parallelize(0 to 10)
ns: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:

scala> ns.id
res0: Int = 2

scala> ns.name
res1: String = null

scala> ns.name = "Friendly name"
ns.name: String = Friendly name

scala> ns.name
res2: String = Friendly name

scala> ns.toDebugString
res3: String = (8) Friendly name ParallelCollectionRDD[2] at parallelize at <console>:24

```

RDDs are a container of instructions on how to materialize big (arrays of) distributed data, and how to split it into partitions so Spark (using [executors](#)) can hold some of them.

In general, data distribution can help executing processing in parallel so a task processes a chunk of data that it could eventually keep in memory.

Spark does jobs in parallel, and RDDs are split into partitions to be processed and written in parallel. Inside a partition, data is processed sequentially.

Saving partitions results in part-files instead of one single file (unless there is a single partition).

Types of RDDs

There are some of the most interesting types of RDDs:

- [ParallelCollectionRDD](#)
- [CoGroupedRDD](#)
- [HadoopRDD](#) is an RDD that provides core functionality for reading data stored in HDFS using the older MapReduce API. The most notable use case is the return RDD of `SparkContext.textFile`.
- **MapPartitionsRDD** - a result of calling operations like `map`, `flatMap`, `filter`, `mapPartitions`, etc.
- **CoalescedRDD** - a result of calling operations like `repartition` and `coalesce`

- **ShuffledRDD** - a result of shuffling, e.g. after `repartition` and `coalesce`
- **PipedRDD** - an RDD created by piping elements to a forked external process.
- **PairRDD** (implicit conversion as `org.apache.spark.rdd.PairRDDFunctions`) that is an RDD of key-value pairs that is a result of `groupByKey` and `join` operations.
- **DoubleRDD** (implicit conversion as `org.apache.spark.rdd.DoubleRDDFunctions`) that is an RDD of `Double` type.
- **SequenceFileRDD** (implicit conversion as `org.apache.spark.rdd.SequenceFileRDDFunctions`) that is an RDD that can be saved as a SequenceFile .

Appropriate operations of a given RDD type are automatically available on a RDD of the right type, e.g. `RDD[(Int, Int)]`, through implicit conversion in Scala.

Transformations

A **transformation** is a lazy operation on a RDD that returns another RDD, like `map` , `flatMap` , `filter` , `reduceByKey` , `join` , `cogroup` , etc.

Go in-depth in the section [Transformations in Operations - Transformations and Actions](#).

Actions

An **action** is an operation that triggers execution of [RDD transformations](#) and returns a value (to a Spark driver - the user program).

Go in-depth in the section [Actions in Operations - Transformations and Actions](#).

Creating RDDs

SparkContext.parallelize

One way to create a RDD is with `SparkContext.parallelize` method. It accepts a collection of elements as shown below (`sc` is a `SparkContext` instance):

```
scala> val rdd = sc.parallelize(1 to 1000)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>
```

You may also want to randomize the sample data:

```
scala> val data = Seq.fill(10)(util.Random.nextInt)
data: Seq[Int] = List(-964985204, 1662791, -1820544313, -383666422, -111039198, 310967683

scala> val rdd = sc.parallelize(data)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>
```

Given the reason to use Spark to process more data than your own laptop could handle, `SparkContext.parallelize` is mainly used to learn Spark in the Spark shell. `SparkContext.parallelize` requires all the data to be available on a single machine - the Spark driver - that eventually hits the limits of your laptop.

SparkContext.makeRDD

Caution

[FIXME](#) What's the use case for `makeRDD` ?

```
scala> sc.makeRDD(0 to 1000)
res0: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at makeRDD at <console>:25
```

SparkContext.textFile

One of the easiest ways to create an RDD is to use `SparkContext.textFile` to read files. You can use the local `README.md` file (and then `map` it over to have an RDD of sequences of words):

```
scala> val words = sc.textFile("README.md").flatMap(_.split("\\s+")).cache()
words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[27] at flatMap at <console>:24
```

Note

You `cache()` it so the computation is not performed every time you work with `words`.

Creating RDDs from Input

Refer to [Using Input and Output \(I/O\)](#) to learn about the IO API to create RDDs.

Transformations

RDD transformations by definition transform an RDD into another RDD and hence are the way to create new ones.

Refer to [Transformations](#) section to learn more.

RDDs in Web UI

It is quite informative to look at RDDs in the Web UI that is at <http://localhost:4040> for [Spark shell](#).

Execute the following Spark application (type all the lines in `spark-shell`):

```
val ints = sc.parallelize(1 to 100) (1)
ints.setName("Hundred ints") (2)
ints.cache (3)
ints.count (4)
```

1. Creates an RDD with hundreds of numbers (with as many partitions as possible)
2. Sets the name of the RDD
3. Caches the RDD (so it shows up in Storage in UI)
4. Executes action (and materializes the RDD)

With the above executed, you should see the following in the Web UI:

The screenshot shows the Spark Web UI interface. At the top, there's a navigation bar with tabs for Jobs, Stages, Storage, Environment, Executors, and SQL. The Storage tab is currently selected. Below the navigation bar, there's a sub-header "Storage". Underneath, there's a table titled "RDDs". The table has columns for "RDD Name", "Storage Level", "Cached Partitions", "Fraction Cached", "Size in Memory", "Size in ExternalBlockStore", and "Size on Disk". A single row is visible in the table, corresponding to the "Hundreds ints" RDD created in the code. The "RDD Name" column shows "Hundreds ints", the "Storage Level" column shows "Memory Deserialized 1x Replicated", and the "Cached Partitions" column shows "8".

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore	Size on Disk
Hundreds ints	Memory Deserialized 1x Replicated	8	100%	2.1 KB	0.0 B	0.0 B

Figure 3. RDD with custom name

Click the name of the RDD (under **RDD Name**) and you will get the details of how the RDD is cached.

RDD Storage Info for Hundred ints

Storage Level: Memory Deserialized 1x Replicated
 Cached Partitions: 8
 Total Partitions: 8
 Memory Size: 2.1 KB
 Disk Size: 0.0 B

Data Distribution on 1 Executors

Host	Memory Usage	Disk Usage
localhost:56166	2.1 KB (530.0 MB Remaining)	0.0 B

8 Partitions

Block Name ▲	Storage Level	Size in Memory	Size on Disk	Executors
rdd_2_0	Memory Deserialized 1x Replicated	256.0 B	0.0 B	localhost:56166
rdd_2_1	Memory Deserialized 1x Replicated	280.0 B	0.0 B	localhost:56166
rdd_2_2	Memory Deserialized 1x Replicated	256.0 B	0.0 B	localhost:56166
rdd_2_3	Memory Deserialized 1x Replicated	280.0 B	0.0 B	localhost:56166
rdd_2_4	Memory Deserialized 1x Replicated	256.0 B	0.0 B	localhost:56166
rdd_2_5	Memory Deserialized 1x Replicated	280.0 B	0.0 B	localhost:56166
rdd_2_6	Memory Deserialized 1x Replicated	256.0 B	0.0 B	localhost:56166
rdd_2_7	Memory Deserialized 1x Replicated	280.0 B	0.0 B	localhost:56166

Figure 4. RDD Storage Info

Execute the following Spark job and you will see how the number of partitions decreases.

```
ints.repartition(2).count
```

Stages for All Jobs

Completed Stages: 5
 Completed Stages (5)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	count at <console>:27	+details 2015/09/23 13:29:42	34 ms	2/2			2.4 KB	
3	repartition at <console>:27	+details 2015/09/23 13:29:42	45 ms	8/8	2.1 KB			2.4 KB

Figure 5. Number of tasks after repartition

Computing Partition of RDD (using compute method)

The `RDD` abstract class defines abstract `compute(split: Partition, context: TaskContext): Iterator[T]` method that computes a given `split` `partition` to produce a collection of values.

It has to be implemented by any type of RDD in Spark and is called unless RDD is `checkpointed` (and the result can be read from a checkpoint).

When an RDD is [cached](#), for specified [storage levels](#) (i.e. all but `NONE`) [CacheManager](#) is requested to get or compute partitions.

`compute` method runs on a [driver](#).

Preferred Locations

A **preferred location** (aka *locality preferences* or *placement preferences*) is a block location for an HDFS file where to compute each partition on.

```
def getPreferredLocations(split: Partition): Seq[String]
```

specifies placement preferences for a partition in an RDD.

RDD Lineage Graph

A **RDD Lineage Graph** (aka *RDD operator graph*) is a graph of the parent RDD of a RDD. It is built as a result of applying transformations to the RDD.

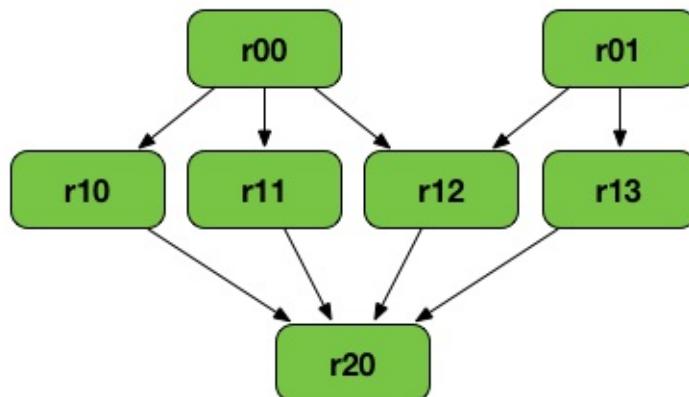


Figure 6. RDD lineage

The above RDD graph could be the result of the following series of transformations:

```

val r00 = sc.parallelize(0 to 9)
val r01 = sc.parallelize(0 to 90 by 10)
val r10 = r00 cartesian r01
val r11 = r00.map(n => (n, n))
val r12 = r00 zip r01
val r13 = r01.keyBy(_ / 20)
val r20 = Seq(r11, r12, r13).foldLeft(r10)(_ union _)
  
```

A RDD lineage graph is hence a graph of what transformations need to be executed after an action has been called.

You can learn about a RDD lineage graph using [RDD.toDebugString](#) method.

toDebugString

You can learn about a [RDD lineage graph](#) using `RDD.toDebugString` method.

```
scala> val wordsCount = sc.textFile("README.md").flatMap(_.split("\\s+")).map((_, 1)).reduceByKey(_ + _)
wordsCount: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[24] at reduceByKey at <console>:24

scala> wordsCount.toDebugString
res2: String =
(2) ShuffledRDD[24] at reduceByKey at <console>:24 []
+- (2) MapPartitionsRDD[23] at map at <console>:24 []
  |  MapPartitionsRDD[22] at flatMap at <console>:24 []
  |  MapPartitionsRDD[21] at textFile at <console>:24 []
  |  README.md HadoopRDD[20] at textFile at <console>:24 []
```

spark.logLineage

Enable `spark.logLineage` (assumed: `false`) to see a RDD lineage graph using [RDD.toDebugString](#) method every time an action on a RDD is called.

```
$ ./bin/spark-shell -c spark.logLineage=true

scala> sc.textFile("README.md", 4).count
...
15/10/17 14:46:42 INFO SparkContext: Starting job: count at <console>:25
15/10/17 14:46:42 INFO SparkContext: RDD's recursive dependencies:
(4) MapPartitionsRDD[1] at textFile at <console>:25 []
  |  README.md HadoopRDD[0] at textFile at <console>:25 []
```

Execution Plan

Execution Plan starts with the earliest RDDs (those with no dependencies on other RDDs or reference cached data) and ends with the RDD that produces the result of the action that has been called to execute.

Operators - Transformations and Actions

RDDs have two fundamental types of operations: [Transformations](#) and [Actions](#).

Transformations

Transformations are lazy operations on a RDD that return RDD objects or collections of RDDs, e.g. `map`, `filter`, `reduceByKey`, `join`, `cogroup`, `randomSplit`, etc.

Transformations are lazy and are not executed immediately, but only after an action has been executed.

Note

There are a couple of transformations that do trigger jobs, e.g. `sortBy`, `zipWithIndex`, etc.

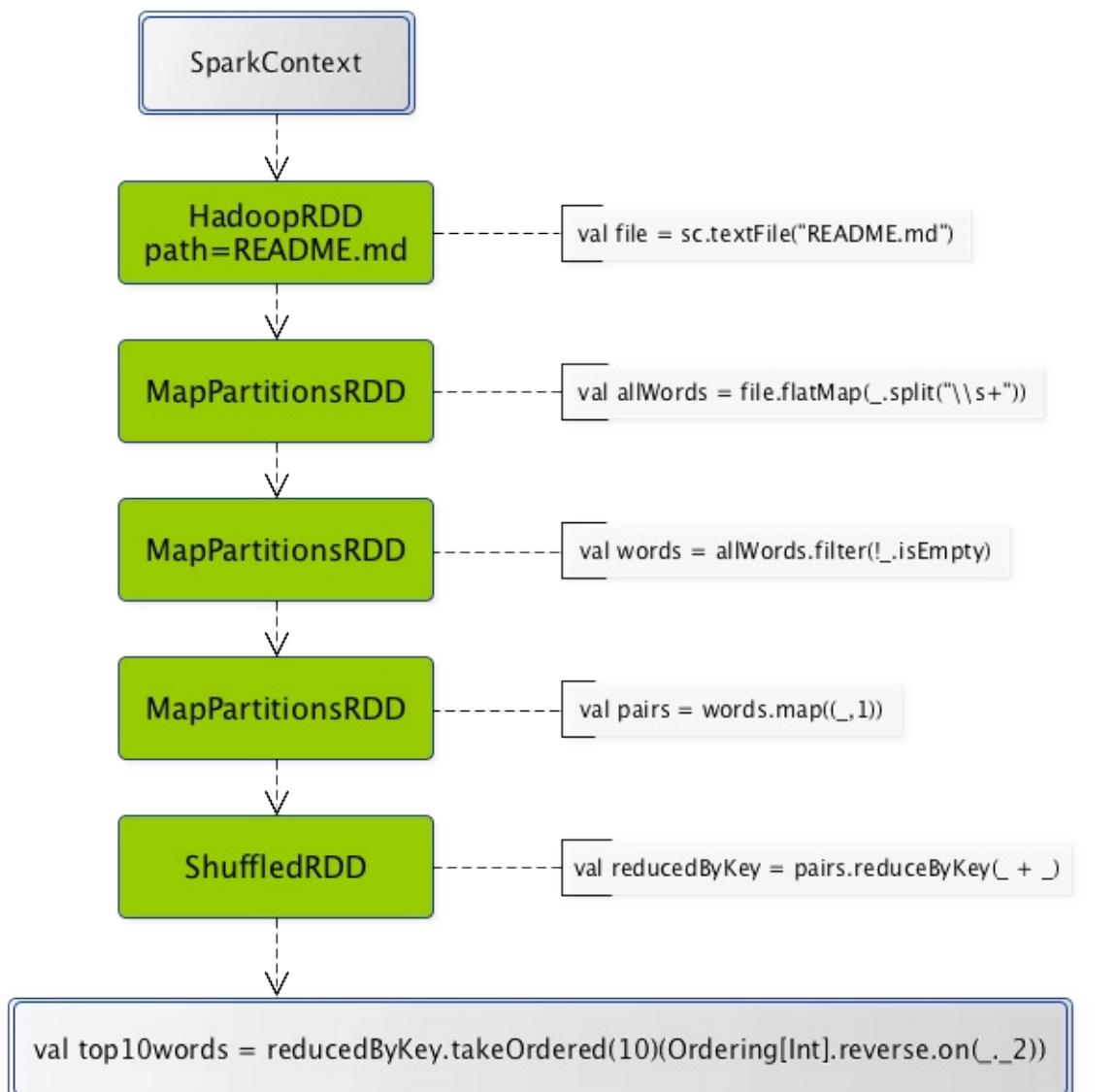


Figure 1. From SparkContext by transformations to the result

You can chain transformations to create **pipelines** of lazy computations.

```
scala> val file = sc.textFile("README.md")
file: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[54] at textFile at <console>:24

scala> val allWords = file.flatMap(_.split("\\s+"))
allWords: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[55] at flatMap at <console>

scala> val words = allWords.filter(!_.isEmpty)
words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[56] at filter at <console>:28

scala> val pairs = words.map((_,1))
pairs: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[57] at map at <console>

scala> val reducedByKey = pairs.reduceByKey(_ + _)
reducedByKey: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[59] at reduceByKey at

scala> val top10words = reducedByKey.takeOrdered(10)(Ordering[Int].reverse.on(_._2))
INFO SparkContext: Starting job: takeOrdered at <console>:34
...
INFO DAGScheduler: Job 18 finished: takeOrdered at <console>:34, took 0.074386 s
top10words: Array[(String, Int)] = Array((the,21), (to,14), (Spark,13), (for,11), (and,10)
```

There are two kinds of transformations:

- [narrow transformations](#)
- [wide transformations](#)

Narrow Transformations

Narrow transformations are the result of `map`, `filter` and such that is from the data from a single partition only, i.e. it is self-sustained.

An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.

Spark groups narrow transformations as a stage.

Wide Transformations

Wide transformations are the result of `groupByKey` and `reduceByKey`. The data required to compute the records in a single partition may reside in many partitions of the parent RDD.

All of the tuples with the same key must end up in the same partition, processed by the same task. To satisfy these operations, Spark must execute [RDD shuffle](#), which transfers data across cluster and results in a new stage with a new set of partitions.

Actions

Actions are operations that return values, i.e. any RDD operation that returns a value of any type but `RDD[T]` is an action.

Note	Actions are synchronous. You can use AsyncRDDActions to release a calling thread while calling actions.
------	---

They trigger execution of [RDD transformations](#) to return values. Simply put, an action evaluates the [RDD lineage graph](#).

You can think of actions as a valve and until no action is fired, the data to be processed is not even in the pipes, i.e. transformations. Only actions can materialize the entire processing pipeline with real data.

Actions in [org.apache.spark.rdd.RDD](#):

- `aggregate`
- `collect`
- `count`
- `countApprox*`
- `countByValue*`
- `first`
- `fold`
- `foreach`
- `foreachPartition`
- `max`
- `min`
- `reduce`
- [saveAs* actions](#), e.g. `saveAsTextFile` , `saveAsHadoopFile`
- `take`

- `takeOrdered`
- `takeSample`
- `toLocalIterator`
- `top`
- `treeAggregate`
- `treeReduce`

Actions run `jobs` using [SparkContext.runJob](#) or directly [DAGScheduler.runJob](#).

```
scala> words.count (1)
res0: Long = 502
```

1. `words` is an RDD of `string`.

Tip

You should `cache` an RDD you work with when you want to execute two or more actions on it for better performance. Refer to [RDD Caching / Persistence](#).

Before calling an action, Spark does closure/function cleaning (using `SparkContext.clean`) to make it ready for serialization and sending over the wire to executors. Cleaning can throw a `SparkException` if the computation cannot be cleaned.

Note

Spark uses `closureCleaner` to clean closures.

AsyncRDDActions

`AsyncRDDActions` class offers asynchronous actions that you can use on RDDs (thanks to the implicit conversion `rddToAsyncRDDActions` in `RDD` class). The methods return a [FutureAction](#).

The following asynchronous methods are available:

- `countAsync`
- `collectAsync`
- `takeAsync`
- `foreachAsync`
- `foreachPartitionAsync`

FutureActions

Caution	FIXME
---------	-------

Gotchas - things to watch for

Even if you don't access it explicitly it cannot be referenced inside a closure as it is serialized and carried around across executors.

See <https://issues.apache.org/jira/browse/SPARK-5063>

mapPartitions Operator

Caution	FIXME
---------	-------

Partitions

Introduction

Depending on how you look at Spark (programmer, devop, admin), an RDD is about the content (developer's and data scientist's perspective) or how it gets spread out over a cluster (performance), i.e. how many partitions an RDD represents.

A **partition** (aka *split*) is...[FIXME](#)

Caution	<p>FIXME</p> <ol style="list-style-type: none">1. How does the number of partitions map to the number of tasks? How to verify it?2. How does the mapping between partitions and tasks correspond to data locality if any?
---------	--

Spark manages data using partitions that helps parallelize distributed data processing with minimal network traffic for sending data between executors.

By default, Spark tries to read data into an RDD from the nodes that are close to it. Since Spark usually accesses distributed partitioned data, to optimize transformation operations it creates partitions to hold the data chunks.

There is a one-to-one correspondence between how data is laid out in data storage like HDFS or Cassandra (it is partitioned for the same reasons).

Features:

- size
- number
- partitioning scheme
- node distribution
- repartitioning

Tip	<p>Read the following documentations to learn what experts say on the topic:</p> <ul style="list-style-type: none">• How Many Partitions Does An RDD Have?• Tuning Spark (the official documentation of Spark)
-----	---

By default, a partition is created for each HDFS partition, which by default is 64MB (from [Spark's Programming Guide](#)).

RDDs get partitioned automatically without programmer intervention. However, there are times when you'd like to adjust the size and number of partitions or the partitioning scheme according to the needs of your application.

You use `def getPartitions: Array[Partition]` method on a RDD to know the set of partitions in this RDD.

As noted in [View Task Execution Against Partitions Using the UI](#):

When a stage executes, you can see the number of partitions for a given stage in the Spark UI.

Start `spark-shell` and see it yourself!

```
scala> sc.parallelize(1 to 100).count
res0: Long = 100
```

When you execute the Spark job, i.e. `sc.parallelize(1 to 100).count`, you should see the following in [Spark shell application UI](#).

The screenshot shows the "Stages for All Jobs" section of the Spark UI. It displays one completed stage with the following details:

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
0	count at <console>:25	+details 2015/09/23 09:24:21	0.1 s	8/8				

Figure 1. The number of partition as Total tasks in UI

The reason for 8 Tasks in Total is that I'm on a 8-core laptop and by default the number of partitions is the number of *all* available cores.

```
$ sysctl -n hw.ncpu
8
```

You can request for the minimum number of partitions, using the second input parameter to many transformations.

```
scala> sc.parallelize(1 to 100, 2).count
res1: Long = 100
```

Completed Stages: 2

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	count at <console>:25	+details 2015/09/23 09:35:11	6 ms	2/2				
0	count at <console>:25	+details 2015/09/23 09:24:21	0.1 s	8/8				

Figure 2. Total tasks in UI shows 2 partitions

You can always ask for the number of partitions using `partitions` method of a RDD:

```
scala> val ints = sc.parallelize(1 to 100, 4)
ints: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>

scala> ints.partitions.size
res2: Int = 4
```

In general, smaller/more numerous partitions allow work to be distributed among more workers, but larger/fewer partitions allow work to be done in larger chunks, which may result in the work getting done more quickly as long as all workers are kept busy, due to reduced overhead.

Increasing partitions count will make each partition to have less data (or not at all!)

Spark can only run 1 concurrent task for every partition of an RDD, up to the number of cores in your cluster. So if you have a cluster with 50 cores, you want your RDDs to at least have 50 partitions (and probably [2-3x times that](#)).

As far as choosing a "good" number of partitions, you generally want at least as many as the number of executors for parallelism. You can get this computed value by calling

```
sc.defaultParallelism .
```

Also, the number of partitions determines how many files get generated by actions that save RDDs to files.

The maximum size of a partition is ultimately limited by the available memory of an executor.

In the first RDD transformation, e.g. reading from a file using `sc.textFile(path, partition)` , the `partition` parameter will be applied to all further transformations and actions on this RDD.

Partitions get redistributed among nodes whenever `shuffle` occurs. Repartitioning may cause `shuffle` to occur in some situations, but it is not guaranteed to occur in all cases. And it usually happens during action stage.

When creating an RDD by reading a file using `rdd = SparkContext().textFile("hdfs://.../file.txt")` the number of partitions may be smaller. Ideally, you would get the same number of blocks as you see in HDFS, but if the lines in your file are too long (longer than the block size), there will be fewer partitions.

Preferred way to set up the number of partitions for an RDD is to directly pass it as the second input parameter in the call like `rdd = sc.textFile("hdfs://.../file.txt", 400)`, where `400` is the number of partitions. In this case, the partitioning makes for 400 splits that would be done by the Hadoop's `TextInputFormat`, not Spark and it would work much faster. It's also that the code spawns 400 concurrent tasks to try to load `file.txt` directly into 400 partitions.

It will only work as described for uncompressed files.

When using `textFile` with compressed files (`file.txt.gz` not `file.txt` or similar), Spark disables splitting that makes for an RDD with only 1 partition (as reads against gzipped files cannot be parallelized). In this case, to change the number of partitions you should do [repartitioning](#).

Some operations, e.g. `map`, `flatMap`, `filter`, don't preserve partitioning.

`map`, `flatMap`, `filter` operations apply a function to every partition.

Repartitioning

- `def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null)` does repartitioning with exactly `numPartitions` partitions. It uses `coalesce` and `shuffle` to redistribute data.

With the following computation you can see that `repartition(5)` causes 5 tasks to be started using `NODE_LOCAL` [data locality](#).

```
scala> lines.repartition(5).count
...
15/10/07 08:10:00 INFO DAGScheduler: Submitting 5 missing tasks from ResultStage 7 (MapPa
15/10/07 08:10:00 INFO TaskSchedulerImpl: Adding task set 7.0 with 5 tasks
15/10/07 08:10:00 INFO TaskSetManager: Starting task 0.0 in stage 7.0 (TID 17, localhost,
15/10/07 08:10:00 INFO TaskSetManager: Starting task 1.0 in stage 7.0 (TID 18, localhost,
15/10/07 08:10:00 INFO TaskSetManager: Starting task 2.0 in stage 7.0 (TID 19, localhost,
15/10/07 08:10:00 INFO TaskSetManager: Starting task 3.0 in stage 7.0 (TID 20, localhost,
15/10/07 08:10:00 INFO TaskSetManager: Starting task 4.0 in stage 7.0 (TID 21, localhost,
...

```

You can see a change after executing `repartition(1)` causes 2 tasks to be started using `PROCESS_LOCAL` [data locality](#).

```
scala> lines.repartition(1).count
...
15/10/07 08:14:09 INFO DAGScheduler: Submitting 2 missing tasks from ShuffleMapStage 8 (M
15/10/07 08:14:09 INFO TaskSchedulerImpl: Adding task set 8.0 with 2 tasks
15/10/07 08:14:09 INFO TaskSetManager: Starting task 0.0 in stage 8.0 (TID 22, localhost,
15/10/07 08:14:09 INFO TaskSetManager: Starting task 1.0 in stage 8.0 (TID 23, localhost,
...

```

Please note that Spark disables splitting for compressed files and creates RDDs with only 1 partition. In such cases, it's helpful to use `sc.textFile('demo.gz')` and do repartitioning using `rdd.repartition(100)` as follows:

```
rdd = sc.textFile('demo.gz')
rdd = rdd.repartition(100)
```

With the lines, you end up with `rdd` to be exactly 100 partitions of roughly equal in size.

- `rdd.repartition(N)` does a `shuffle` to split data to match `N`
 - partitioning is done on round robin basis

Tip	If partitioning scheme doesn't work for you, you can write your own custom partitioner.
-----	---

Tip	It's useful to get familiar with Hadoop's TextInputFormat .
-----	---

coalesce transformation

```
coalesce(numPartitions: Int, shuffle: Boolean = false)(implicit ord: Ordering[T] = null):
```

The `coalesce` transformation is used to change the number of partitions. It can trigger **RDD shuffling** depending on the second `shuffle` boolean input parameter (defaults to `false`).

In the following sample, you `parallelize` a local 10-number sequence and `coalesce` it first without and then with shuffling (note the `shuffle` parameter being `false` and `true`, respectively). You use `toDebugString` to check out the **RDD's lineage graph**.

```
scala> val rdd = sc.parallelize(0 to 10, 8)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>

scala> rdd.partitions.size
res0: Int = 8

scala> rdd.coalesce(numPartitions=8, shuffle=false)    (1)
res1: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[1] at coalesce at <console>:27

scala> res1.toDebugString
res2: String =
(8) CoalescedRDD[1] at coalesce at <console>:27 []
|  ParallelCollectionRDD[0] at parallelize at <console>:24 []

scala> rdd.coalesce(numPartitions=8, shuffle=true)
res3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[5] at coalesce at <console>:27

scala> res3.toDebugString
res4: String =
(8) MapPartitionsRDD[5] at coalesce at <console>:27 []
|  CoalescedRDD[4] at coalesce at <console>:27 []
|  ShuffledRDD[3] at coalesce at <console>:27 []
+- (8) MapPartitionsRDD[2] at coalesce at <console>:27 []
   |  ParallelCollectionRDD[0] at parallelize at <console>:24 []
```

1. `shuffle` is `false` by default and it's explicitly used here for demo's purposes. Note the number of partitions that remains the same as the number of partitions in the source RDD `rdd`.

PairRDDFunctions - `groupByKey`, `reduceByKey`, `partitionBy`

You may want to look at the number of partitions from another angle.

It may often not be important to have a given number of partitions upfront (at RDD creation time upon [loading data from data sources](#)), so only "regrouping" the data by key after it is an RDD might be...the key (*pun not intended*).

You can use `groupByKey` or another `PairRDDFunctions` method to have a key in one processing flow.

You could use `partitionBy` that is available for RDDs to be RDDs of tuples, i.e. `PairRDD`:

```
rdd.keyBy(_.kind)
    .partitionBy(new HashPartitioner(PARTITIONS))
    .foreachPartition(...)
```

Think of situations where `kind` has low cardinality or highly skewed distribution and using the technique for partitioning might be not an optimal solution.

You could do as follows:

```
rdd.keyBy(_.kind).reduceByKey(....)
```

or `mapValues` or plenty of other solutions. [FIXME](#), *man.*

Partitioner

Caution	FIXME
---------	-----------------------

A **partitioner** captures data distribution at the output. A scheduler can optimize future operations based on this.

`val partitioner: Option[Partitioner]` specifies how the RDD is partitioned.

HashPartitioner

Caution	FIXME
---------	-----------------------

`HashPartitioner` is the default partitioner for `coalesce` operation when shuffle is allowed, e.g. calling `repartition`.

`spark.default.parallelism` - when set, it sets up the number of partitions to use for HashPartitioner.

RDD Caching and Persistence

RDDs can be cached (using RDD's `cache()` operation) or persisted (using RDD's `persist(newLevel: StorageLevel)` operation).

The `cache()` operation is a synonym of `persist()` that uses the default storage level `MEMORY_ONLY`.

RDDs can be [unpersisted](#).

Storage Levels

`StorageLevel` describes how an RDD is persisted (and addresses the following concerns):

- Does RDD use disk?
- How much of RDD is in memory?
- Does RDD use off-heap memory?
- Should an RDD be serialized (while persisting)?
- How many replicas (default: `1`) to use (can only be less than `40`)?

There are the following `StorageLevel` (number `_2` in the name denotes 2 replicas):

- `NONE` (default)
- `DISK_ONLY`
- `DISK_ONLY_2`
- `MEMORY_ONLY` (default for `cache()` operation)
- `MEMORY_ONLY_2`
- `MEMORY_ONLY_SER`
- `MEMORY_ONLY_SER_2`
- `MEMORY_AND_DISK`
- `MEMORY_AND_DISK_2`
- `MEMORY_AND_DISK_SER`
- `MEMORY_AND_DISK_SER_2`

- OFF_HEAP

You can check out the storage level using `getStorageLevel()` operation.

```
scala> val lines = sc.textFile("README.md")
lines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at textFile at <console>:24

scala> lines.getStorageLevel
res2: org.apache.spark.storage.StorageLevel = StorageLevel(false, false, false, false, 1)
```

Unpersisting RDDs (Clearing Blocks)

When `unpersist(blocking: Boolean = true)` method is called, you should see the following INFO message in the logs:

```
INFO [RddName]: Removing RDD [id] from persistence list
```

It then calls `SparkContext.unpersistRDD(id, blocking)` and sets `StorageLevel.NONE` as the storage level.

RDD shuffling

Tip

Read the official documentation about the topic [Shuffle operations](#). It is *still* better than this page.

Shuffling is a process of [repartitioning](#) (redistributing) data across partitions and may cause moving it across JVMs or even network when it is redistributed among executors.

Tip

Avoid shuffling at all cost. Think about ways to leverage existing partitions. Leverage partial aggregation to reduce data transfer.

By default, shuffling doesn't change the number of partitions, but their content.

- Avoid `groupByKey` and use `reduceByKey` or `combineByKey` instead.
 - `groupByKey` shuffles all the data, which is slow.
 - `reduceByKey` shuffles only the results of sub-aggregations in each partition of the data.

Example - join

PairRDD offers [join](#) transformation that (quoting the official documentation):

When called on datasets of type (K, V) and (K, W) , returns a dataset of $(K, (V, W))$ pairs with all pairs of elements for each key.

Let's have a look at an example and see how it works under the covers:

```

scala> val kv = (0 to 5) zip Stream.continually(5)
kv: scala.collection.immutable.IndexedSeq[(Int, Int)] = Vector((0,5), (1,5), (2,5), (3,5)

scala> val kw = (0 to 5) zip Stream.continually(10)
kw: scala.collection.immutable.IndexedSeq[(Int, Int)] = Vector((0,10), (1,10), (2,10), (3,10)

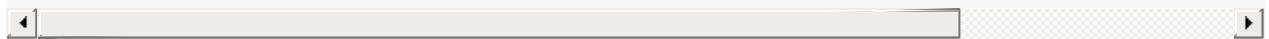
scala> val kvR = sc.parallelize(kv)
kvR: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[3] at parallelize at <console>:26

scala> val kwR = sc.parallelize(kw)
kwR: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[4] at parallelize at <console>:27

scala> val joined = kvR join kwR
joined: org.apache.spark.rdd.RDD[(Int, (Int, Int))] = MapPartitionsRDD[10] at join at <console>:28

scala> joined.toDebugString
res7: String =
(8) MapPartitionsRDD[10] at join at <console>:32 []
|  MapPartitionsRDD[9] at join at <console>:32 []
|  CoGroupedRDD[8] at join at <console>:32 []
+- (8) ParallelCollectionRDD[3] at parallelize at <console>:26 []
+- (8) ParallelCollectionRDD[4] at parallelize at <console>:26 []

```



It doesn't look good when there is an "angle" between "nodes" in an operation graph. It appears before the `join` operation so shuffle is expected.

Here is how the job of executing `joined.count` looks in Web UI.

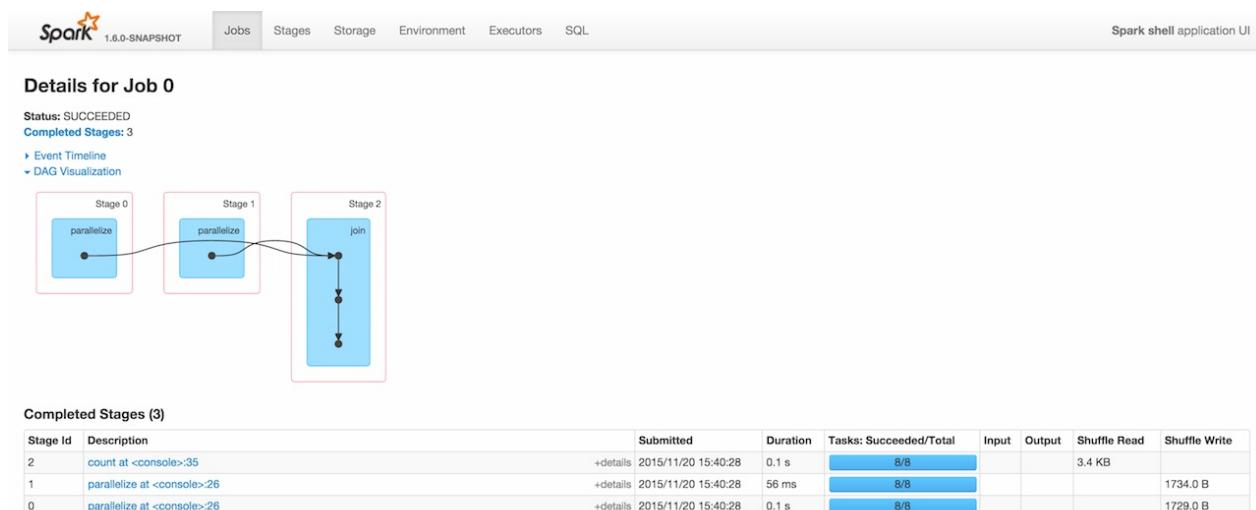


Figure 1. Executing `joined.count`

The screenshot of Web UI shows 3 stages with two `parallelize` to Shuffle Write and `count` to Shuffle Read. It means shuffling has indeed happened.

Caution

FIXME Just learnt about `sc.range(0, 5)` as a shorter version of `sc.parallelize(0 to 5)`

`join` operation is one of the **cogroup operations** that uses `defaultPartitioner`, i.e. walks through [the RDD lineage graph](#) (sorted by the number of partitions decreasing) and picks the partitioner with positive number of output partitions. Otherwise, it checks `spark.default.parallelism` setting and if defined picks [HashPartitioner](#) with the default parallelism of the [scheduler backend](#).

`join` is almost `CoGroupedRDD.mapValues`.

Caution

[FIXME](#) the default parallelism of scheduler backend

Checkpointing

Introduction

Checkpointing is a process of truncating [RDD lineage graph](#) and saving it to a reliable distributed (HDFS) or local file system.

There are two types of checkpointing:

- **reliable** - in Spark (core), RDD checkpointing that saves the actual intermediate RDD data to a reliable distributed file system, e.g. HDFS.
- **local** - in [Spark Streaming](#) or GraphX - RDD checkpointing that truncates [RDD lineage graph](#).

It's up to a Spark application developer to decide when and how to checkpoint using `RDD.checkpoint()` method.

Before checkpointing is used, a Spark developer has to set the checkpoint directory using `SparkContext.setCheckpointDir(directory: String)` method.

Reliable Checkpointing

You call `SparkContext.setCheckpointDir(directory: String)` to set the **checkpoint directory** - the directory where RDDs are checkpointed. The `directory` must be a HDFS path if running on a cluster. The reason is that the driver may attempt to reconstruct the checkpointed RDD from its own local file system, which is incorrect because the checkpoint files are actually on the executor machines.

You mark an RDD for checkpointing by calling `RDD.checkpoint()`. The RDD will be saved to a file inside the checkpoint directory and all references to its parent RDDs will be removed. This function has to be called before any job has been executed on this RDD.

Note

It is strongly recommended that a checkpointed RDD is persisted in memory, otherwise saving it on a file will require recomputation.

When an action is called on a checkpointed RDD, the following INFO message is printed out in the logs:

```
15/10/10 21:08:57 INFO ReliableRDDCheckpointData: Done checkpointing RDD 5 to file:/Users
```

ReliableRDDCheckpointData

When `RDD.checkpoint()` operation is called, all the information related to RDD checkpointing are in `ReliableRDDCheckpointData`.

`spark.cleaner.referenceTracking.cleanCheckpoints` (default: `false`) - whether clean checkpoint files if the reference is out of scope.

ReliableCheckpointRDD

After `RDD.checkpoint` the RDD has `ReliableCheckpointRDD` as the new parent with the exact number of partitions as the RDD.

Local Checkpointing

Beside the `RDD.checkpoint()` method, there is similar one - `RDD.localCheckpoint()` that marks the RDD for **local checkpointing** using Spark's existing caching layer.

This `RDD.localCheckpoint()` method is for users who wish to truncate [RDD lineage graph](#) while skipping the expensive step of replicating the materialized data in a reliable distributed file system. This is useful for RDDs with long lineages that need to be truncated periodically, e.g. GraphX.

Local checkpointing trades fault-tolerance for performance.

The checkpoint directory set through `SparkContext.setCheckpointDir` is not used.

LocalRDDCheckpointData

[FIXME](#)

LocalCheckpointRDD

[FIXME](#)

Dependencies

Dependency (represented by [Dependency](#) class) is a connection between RDDs after applying a transformation.

You can use `RDD.dependencies` method to know the collection of dependencies of a RDD (`Seq[Dependency[_]]`).

```
scala> val r1 = sc.parallelize(0 to 9)
r1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[20] at parallelize at <console>

scala> val r2 = sc.parallelize(0 to 9)
r2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[21] at parallelize at <console>

scala> val r3 = sc.parallelize(0 to 9)
r3: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[22] at parallelize at <console>

scala> val r4 = sc.union(r1, r2, r3)
r4: org.apache.spark.rdd.RDD[Int] = UnionRDD[23] at union at <console>:24

scala> r4.dependencies
res0: Seq[org.apache.spark.Dependency[_]] = ArrayBuffer(org.apache.spark.RangeDependency@

scala> r4.toDebugString
res1: String =
(24) UnionRDD[23] at union at <console>:24 []
|  ParallelCollectionRDD[20] at parallelize at <console>:18 []
|  ParallelCollectionRDD[21] at parallelize at <console>:18 []
|  ParallelCollectionRDD[22] at parallelize at <console>:18 []

scala> r4.collect
...
res2: Array[Int] = Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1
```

Kinds of Dependencies

`Dependency` is the base abstract class with a single `def rdd: RDD[T]` method.

```
scala> val r = sc.parallelize(0 to 9).groupByKey(identity)
r: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[12] at groupBy at <consol

scala> r.dependencies.map(_.rdd).foreach(println)
MapPartitionsRDD[11] at groupBy at <console>:18
```

There are the following more specialized `Dependency` extensions:

- `NarrowDependency`
 - `OneToOneDependency`
 - `PruneDependency`
 - `RangeDependency`
- `ShuffleDependency`

ShuffleDependency

A `ShuffleDependency` represents a dependency on the output of a `shuffle map stage`.

```
scala> val r = sc.parallelize(0 to 9).groupBy(identity)
r: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[12] at groupBy at <consol

scala> r.dependencies
res0: Seq[org.apache.spark.Dependency[_]] = List(org.apache.spark.ShuffleDependency@493b0
```

A `ShuffleDependency` belongs to a single pair RDD (available as `rdd` of type `RDD[Product2[K, V]]`).

A `ShuffleDependency` has a `shuffleId` ([FIXME](#) from `SparkContext.newShuffleId`).

It uses `partitioner` to partition the shuffle output. It also uses `ShuffleManager` to register itself (using `ShuffleManager.registerShuffle`) and `ContextCleaner` to register itself for cleanup (using `ContextCleaner.registerShuffleForCleanup`).

Every `ShuffleDependency` is registered to `MapOutputTracker` by the shuffle's id and the number of the partitions of a RDD (using `MapOutputTrackerMaster.registerShuffle`).

The places where `ShuffleDependency` is used:

- `CoGroupedRDD` and `SubtractedRDD` when `partitioner` differs among RDDs
- `ShuffledRDD` and `ShuffledRowRDD` that are RDDs from a shuffle

The RDD operations that may or may not use the above RDDs and hence shuffling:

- `coalesce`
 - `repartition`
- `cogroup`

- intersection
- subtractByKey
 - subtract
- sortByKey
 - sortBy
- repartitionAndSortWithinPartitions
- combineByKeyWithClassTag
 - combineByKey
 - aggregateByKey
 - foldByKey
 - reduceByKey
 - countApproxDistinctByKey
 - groupByKey
- partitionBy

Note	There may be other dependent methods that use the above.
------	--

NarrowDependency

`NarrowDependency` is an abstract extension of `Dependency` with *narrow* (limited) number of partitions of the parent RDD that are required to compute a partition of the child RDD. Narrow dependencies allow for pipelined execution.

`NarrowDependency` extends the base with the additional method:

```
def getParents(partitionId: Int): Seq[Int]
```

to get the parent partitions for a partition `partitionId` of the child RDD.

OneToOneDependency

`OneToOneDependency` is a narrow dependency that represents a one-to-one dependency between partitions of the parent and child RDDs.

```

scala> val r1 = sc.parallelize(0 to 9)
r1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[13] at parallelize at <console>

scala> val r3 = r1.map((_, 1))
r3: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[19] at map at <console>:20

scala> r3.dependencies
res32: Seq[org.apache.spark.Dependency[_]] = List(org.apache.spark.OneToOneDependency@735

scala> r3.toDebugString
res33: String =
(8) MapPartitionsRDD[19] at map at <console>:20 []
| ParallelCollectionRDD[13] at parallelize at <console>:18 []

```

PruneDependency

`PruneDependency` is a narrow dependency that represents a dependency between the `PartitionPruningRDD` and its parent.

RangeDependency

`RangeDependency` is a narrow dependency that represents a one-to-one dependency between ranges of partitions in the parent and child RDDs.

It is used in `UnionRDD` for `SparkContext.union`, `RDD.union` transformation to list only a few.

```

scala> val r1 = sc.parallelize(0 to 9)
r1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[13] at parallelize at <console>

scala> val r2 = sc.parallelize(10 to 19)
r2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[14] at parallelize at <console>

scala> val unioned = sc.union(r1, r2)
unioned: org.apache.spark.rdd.RDD[Int] = UnionRDD[16] at union at <console>:22

scala> unioned.dependencies
res19: Seq[org.apache.spark.Dependency[_]] = ArrayBuffer(org.apache.spark.RangeDependency

scala> unioned.toDebugString
res18: String =
(16) UnionRDD[16] at union at <console>:22 []
| ParallelCollectionRDD[13] at parallelize at <console>:18 []
| ParallelCollectionRDD[14] at parallelize at <console>:18 []

```

ParallelCollectionRDD

ParallelCollectionRDD is an RDD of a collection of elements with `numSlices` partitions and optional `locationPrefs`.

`ParallelCollectionRDD` is the result of `SparkContext.parallelize` and `SparkContext.makeRDD` methods.

The data collection is split on to `numSlices` slices.

It uses `ParallelCollectionPartition`.

MapPartitionsRDD

MapPartitionsRDD is an RDD that applies the provided function `f` to every partition of the parent RDD.

By default, it does not preserve partitioning — the last input parameter `preservesPartitioning` is `false`. If it is `true`, it retains the original RDD's partitioning.

`MapPartitionsRDD` is the result of the following transformations:

- `RDD.map`
- `RDD.flatMap`
- `RDD.filter`
- `RDD.glom`
- `RDD.mapPartitions`
- `RDD.mapPartitionsWithIndex`
- `PairRDDFunctions.mapValues`
- `PairRDDFunctions.flatMapValues`

CoGroupedRDD

A RDD that cogroups its pair RDD parents. For each key k in parent RDDs, the resulting RDD contains a tuple with the list of values for that key.

Use `RDD.cogroup(...)` to create one.

HadoopRDD

[HadoopRDD](#) is an RDD that provides core functionality for reading data stored in HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI using the older MapReduce API ([org.apache.hadoop.mapred](#)).

HadoopRDD is created as a result of calling the following methods in [SparkContext](#):

- `hadoopFile`
- `textFile` (the most often used in examples!)
- `sequenceFile`

Partitions are of type `HadoopPartition`.

When an HadoopRDD is computed, i.e. an action is called, you should see the INFO message `Input split:` in the logs.

```
scala> sc.textFile("README.md").count
...
15/10/10 18:03:21 INFO HadoopRDD: Input split: file:/Users/jacek/dev/oss/spark/README.md:
15/10/10 18:03:21 INFO HadoopRDD: Input split: file:/Users/jacek/dev/oss/spark/README.md:
...

```

The following properties are set upon partition execution:

- **mapred.tip.id** - task id of this task's attempt
- **mapred.task.id** - task attempt's id
- **mapred.task.is.map** as `true`
- **mapred.task.partition** - split id
- **mapred.job.id**

Spark settings for `HadoopRDD`:

- **spark.hadoop.cloneConf** (default: `false`) - `shouldCloneJobConf` - should a Hadoop job configuration `JobConf` object be cloned before spawning a Hadoop job. Refer to [\[SPARK-2546\] Configuration object thread safety issue](#). When `true`, you should see a DEBUG message `Cloning Hadoop Configuration`.

You can register callbacks on TaskContext.

HadoopRDDs are not checkpointed. They do nothing when `checkpoint()` is called.

Caution	<p>FIXME</p> <ul style="list-style-type: none"> • What are <code>InputMetrics</code> ? • What is <code>JobConf</code> ? • What are the InputSplits: <code>FileSplit</code> and <code>combineFileSplit</code> ? * What are <code>InputFormat</code> and <code>Configurable</code> subtypes? • What's <code>InputFormat</code>'s RecordReader? It creates a key and a value. What are they? • What does TaskContext do? • What's Hadoop Split? input splits for Hadoop reads? See <code>InputFormat.getSplits</code>
---------	---

getPartitions

The number of partition for HadoopRDD, i.e. the return value of `getPartitions`, is calculated using `InputFormat.getSplits(jobConf, minPartitions)` where `minPartitions` is only a hint of how many partitions one may want at minimum. As a hint it does not mean the number of partitions will be exactly the number given.

For `SparkContext.textFile` the input format class is [org.apache.hadoop.mapred.TextInputFormat](#).

The [javadoc of org.apache.hadoop.mapred.FileInputFormat](#) says:

FileInputFormat is the base class for all file-based InputFormats. This provides a generic implementation of `getSplits(JobConf, int)`. Subclasses of FileInputFormat can also override the `isSplitable(FileSystem, Path)` method to ensure input-files are not split-up and are processed as a whole by Mappers.

Tip	You may find the sources of org.apache.hadoop.mapred.FileInputFormat.getSplits enlightening.
-----	--

ShuffledRDD

ShuffledRDD is an RDD of (key, value) pairs. It is a shuffle step (the result RDD) for transformations that trigger `shuffle` at execution. Such transformations ultimately call `coalesce` transformation with `shuffle` input parameter `true` (default: `false`).

By default, the map-side combining flag (`mapSideCombine`) is `false`. It can however be changed using `ShuffledRDD.setMapSideCombine(mapSideCombine: Boolean)` method (and is used in `PairRDDFunctions.combineByKeyWithClassTag` that sets it `true` by default).

The only dependency of ShuffledRDD is a single-element collection of `ShuffleDependency`. Partitions are of type `ShuffledRDDPartition`.

Let's have a look at the below example with `groupBy` transformation:

```
scala> val r = sc.parallelize(0 to 9, 3).groupBy(_ / 3)
r: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[2] at groupBy at <console>
scala> r.toDebugString
res0: String =
(3) ShuffledRDD[2] at groupBy at <console>:18 []
 +- (3) MapPartitionsRDD[1] at groupBy at <console>:18 []
   |  ParallelCollectionRDD[0] at parallelize at <console>:18 []
```

As you may have noticed, `groupBy` transformation adds `ShuffledRDD` RDD that will execute shuffling at execution time (as depicted in the following screenshot).

Completed Stages (2)								
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	count at <console>:21	+details 2015/12/07 06:37:39	41 ms	3/3			620.0 B	
0	groupBy at <console>:18	+details 2015/12/07 06:37:39	79 ms	3/3				620.0 B

Figure 1. Two stages in a job due to shuffling

It can be the result of RDD transformations using Scala implicits:

- `repartitionAndSortWithinPartitions`
- `sortByKey` (be very careful due to [SPARK-1021] `sortByKey()` launches a cluster job when it shouldn't)
- `partitionBy` (only when the input partitioner is different from the current one in an RDD)

It uses `Partitioner`.

It uses [MapOutputTrackerMaster](#) to get preferred locations for a shuffle, i.e. a `ShuffleDependency`.

PairRDDFunctions.combineByKeyWithClassTag

`PairRDDFunctions.combineByKeyWithClassTag` function assumes `mapSideCombine` as `true` by default. It then creates `ShuffledRDD` with the value of `mapSideCombine` when the input partitioner is different from the current one in an RDD.

The function is a generic base function for `combineByKey`-based functions, `combineByKeyWithClassTag`-based functions, `aggregateByKey`, `foldByKey`, `reduceByKey`, `countApproxDistinctByKey`, `groupByKey`, `combineByKeyWithClassTag`-based functions.

BlockRDD

Caution	FIXME
---------	-------

Spark Streaming calls `BlockRDD.removeBlocks()` while clearing metadata.

Spark shell

Spark shell is an interactive shell for learning about Apache Spark, ad-hoc queries and developing Spark applications. It is a very convenient tool to explore the many things available in Spark and one of the many reasons why Spark is so helpful even for very simple tasks (see [Why Spark](#)).

There are variants of Spark for different languages: `spark-shell` for Scala and `pyspark` for Python.

Note	This document uses <code>spark-shell</code> only.
------	---

`spark-shell` is based on Scala REPL with automatic instantiation of [Spark context](#) as `sc` and [SQL context](#) as `sqlContext`.

Note	<p>When you execute <code>spark-shell</code> it executes Spark submit as follows:</p> <pre>org.apache.spark.deploy.SparkSubmit --class org.apache.spark.repl.Main --name \$</pre> <p>Set <code>SPARK_PRINT_LAUNCH_COMMAND</code> to see the entire command to be executed. Refer Command of Spark Scripts.</p>
------	--

Spark shell boils down to executing [Spark submit](#) and so command-line arguments of Spark submit become Spark shell's, e.g. `--verbose`.

Using Spark shell

You start Spark shell using `spark-shell` script (available in `bin` directory).

```
$ ./bin/spark-shell
Spark context available as sc.
SQL context available as sqlContext.
Welcome to

   __
  / _/_
  _\ \V_ _\V_ _`/ _/ '
 /__/ .__/\_,/_/_ /_/\_\  version 1.6.0-SNAPSHOT
  /_/

Using Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_66)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Spark shell gives you the `sc` value which is the [SparkContext](#) for the session.

```
scala> sc
res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@2ac0cb64
```

Besides, there is also `sqlContext` which is an instance of `org.apache.spark.sql.SQLContext` to use Spark SQL. Refer to [Spark SQL](#).

```
scala> sqlContext  
res1: org.apache.spark.sql.SQLContext = org.apache.spark.sql.hive.HiveContext@60ae950f
```

To close Spark shell, you press `ctrl+D` or type in `:q` (or any subset of `:quit`).

scala> :quit

Learning Spark interactively

One way to learn about a tool like **the Spark shell** is to read its error messages. Together with the source code it may be a viable tool to reach mastery.

Let's give it a try using `spark-shell` .

While trying it out using an incorrect value for the master's URL, you're told about `--help` and `--verbose` options.

```
→ spark git:(master) ✘ ./bin/spark-shell --master mss  
Error: Master must start with yarn, spark, mesos, or local  
Run with --help for usage help or --verbose for debug output
```

You're also told about the acceptable values for `--master`.

Let's see what `--verbose` gives us.

```
→ spark git:(master) ✘ ./bin/spark-shell --verbose --master mss
Using properties file: null
Parsed arguments:
  master          mss
  deployMode      null
  executorMemory null
  executorCores  null
  totalExecutorCores null
  propertiesFile  null
  driverMemory    null
  driverCores     null
  driverExtraClassPath null
  driverExtraLibraryPath null
  driverExtraJavaOptions null
  supervise        false
  queue           null
  numExecutors    null
  files           null
  pyFiles          null
  archives         null
  mainClass        org.apache.spark.repl.Main
  primaryResource  spark-shell
  name             Spark shell
  childArgs        []
  jars             null
  packages         null
  packagesExclusions null
  repositories    null
  verbose          true
```

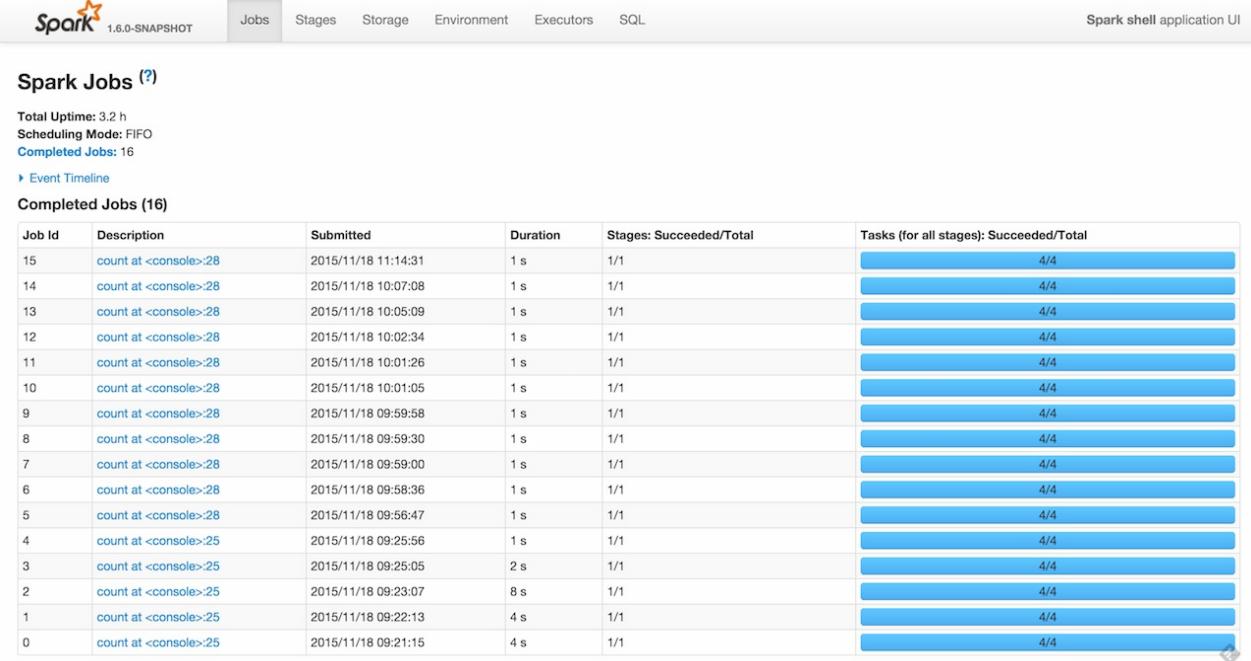
Spark properties used, including those specified through
`--conf` and those from the properties file `null`:

```
Error: Master must start with yarn, spark, mesos, or local
Run with --help for usage help or --verbose for debug output
```

Tip	These 'null's could instead be replaced with some other, more meaningful values.
-----	--

WebUI - UI for Spark Monitoring

Spark comes with **Web UI** (aka **webUI**) to inspect job executions using a browser.



The screenshot shows the Spark Web UI interface. At the top, there's a navigation bar with tabs: Jobs (selected), Stages, Storage, Environment, Executors, and SQL. To the right of the tabs, it says "Spark shell application UI". Below the navigation bar, there's a section titled "Spark Jobs (?)". It displays system statistics: Total Uptime: 3.2 h, Scheduling Mode: FIFO, and Completed Jobs: 16. There's also a link to "Event Timeline". The main area is titled "Completed Jobs (16)" and contains a table with 16 rows of job details. Each row includes columns for Job Id, Description, Submitted, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total. All tasks are shown as 4/4 completed.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
15	count at <console>:28	2015/11/18 11:14:31	1 s	1/1	4/4
14	count at <console>:28	2015/11/18 10:07:08	1 s	1/1	4/4
13	count at <console>:28	2015/11/18 10:05:09	1 s	1/1	4/4
12	count at <console>:28	2015/11/18 10:02:34	1 s	1/1	4/4
11	count at <console>:28	2015/11/18 10:01:26	1 s	1/1	4/4
10	count at <console>:28	2015/11/18 10:01:05	1 s	1/1	4/4
9	count at <console>:28	2015/11/18 09:59:58	1 s	1/1	4/4
8	count at <console>:28	2015/11/18 09:59:30	1 s	1/1	4/4
7	count at <console>:28	2015/11/18 09:59:00	1 s	1/1	4/4
6	count at <console>:28	2015/11/18 09:58:36	1 s	1/1	4/4
5	count at <console>:28	2015/11/18 09:56:47	1 s	1/1	4/4
4	count at <console>:25	2015/11/18 09:25:56	1 s	1/1	4/4
3	count at <console>:25	2015/11/18 09:25:05	2 s	1/1	4/4
2	count at <console>:25	2015/11/18 09:23:07	8 s	1/1	4/4
1	count at <console>:25	2015/11/18 09:22:13	4 s	1/1	4/4
0	count at <console>:25	2015/11/18 09:21:15	4 s	1/1	4/4

Figure 1. Welcome page - Jobs page

Every SparkContext launches its own instance of Web UI which is available at

`http://[master]:4040` by default (the port can be changed using `spark.ui.port`).

It offers pages (tabs) with the following information:

- Jobs
- Stages
- Storage (with RDD size and memory use)
- [Environment](#)
- [Executors](#)
- [SQL](#)

This information is available only until the application is running by default.

You can view the web UI after the fact after setting `spark.eventLog.enabled` to `true` before starting the application.

Environment Tab

The screenshot shows the Apache Spark Web UI with the 'Environment' tab selected. The 'Runtime Information' section contains a table with key-value pairs for Java Home, Java Version, and Scala Version. The 'Spark Properties' section contains a larger table with key-value pairs for various Spark configuration parameters like spark.app.id, spark.app.name, spark.driver.host, etc.

Name	Value
Java Home	/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre
Java Version	1.8.0_66 (Oracle Corporation)
Scala Version	version 2.11.7

Name	Value
spark.app.id	local-1447834845413
spark.app.name	Spark shell
spark.driver.host	192.168.1.4
spark.driver.port	62703
spark.executor.id	driver
spark.externalBlockStore.folderName	spark-3d0ae652-01d0-4a8a-ad6b-e33b44f99f5e
spark.filesServer.url	http://192.168.1.4:62705
spark.home	/Users/jacek/dev/oss/spark
spark.jars	
spark.master	local[""]
spark.repl.class.uri	http://192.168.1.4:62702
spark.scheduler.mode	FIFO
spark.submit.deployMode	client
spark.ui.showConsoleProgress	true

Figure 2. Environment tab in Web UI

JobProgressListener

`JobProgressListener` is [the listener](#) for Web UI.

It tracks information to be displayed in the UI.

Caution	FIXME What information does this track?
---------	---

Settings

- `spark.ui.enabled` (default: `true`) setting controls whether the web UI is started at all.
- `spark.ui.port` (default: `4040`) controls the port Web UI binds to.

If multiple SparkContexts attempt to run on the same host (it is not possible to have two or more Spark contexts on a single JVM, though), they will bind to successive ports beginning with `spark.ui.port`.

- `spark.ui.killEnabled` (default: `true`) - whether or not you can kill stages in web UI.

Executors Tab

Caution	FIXME
---------	-----------------------

spark-submit script

You use `spark-submit` script to launch a Spark application, i.e. submit the application to a Spark deployment environment.

You can find `spark-submit` script in `bin` directory of the Spark distribution.

Deploy Modes

Using `--deploy-mode` command-line option you can specify two different deploy modes:

- `client` (default)
- `cluster`

It is only used for [clustered modes](#).

Command-line Options

Execute `./bin/spark-submit --help` to know about the command-line options supported.

```
→ spark git:(master) ✘ ./bin/spark-submit --help
Usage: spark-submit [options] <app jar | python file> [app arguments]
Usage: spark-submit --kill [submission ID] --master [spark://...]
Usage: spark-submit --status [submission ID] --master [spark://...]

Options:
  --master MASTER_URL          spark://host:port, mesos://host:port, yarn, or local.
  --deploy-mode DEPLOY_MODE    Whether to launch the driver program locally ("client") or
                               on one of the worker machines inside the cluster ("cluster")
                               (Default: client).
  --class CLASS_NAME           Your application's main class (for Java / Scala apps).
  --name NAME                  A name of your application.
  --jars JARS                  Comma-separated list of local jars to include on the driver
                               and executor classpaths.
  --packages                   Comma-separated list of maven coordinates of jars to includ
                               on the driver and executor classpaths. Will search the loca
                               maven repo, then maven central and any additional remote
                               repositories given by --repositories. The format for the
                               coordinates should be groupId:artifactId:version.
  --exclude-packages           Comma-separated list of groupId:artifactId, to exclude whil
                               resolving the dependencies provided in --packages to avoid
                               dependency conflicts.
  --repositories               Comma-separated list of additional remote repositories to
                               search for the maven coordinates given with --packages.
  --py-files PY_FILES          Comma-separated list of .zip, .egg, or .py files to place
                               on the PYTHONPATH for Python apps.
```

--files FILES	Comma-separated list of files to be placed in the working directory of each executor.
--conf PROP=VALUE	Arbitrary Spark configuration property.
--properties-file FILE	Path to a file from which to load extra properties. If not specified, this will look for conf/spark-defaults.conf.
--driver-memory MEM	Memory for driver (e.g. 1000M, 2G) (Default: 1024M).
--driver-java-options	Extra Java options to pass to the driver.
--driver-library-path	Extra library path entries to pass to the driver.
--driver-class-path	Extra class path entries to pass to the driver. Note that jars added with --jars are automatically included in the classpath.
--executor-memory MEM	Memory per executor (e.g. 1000M, 2G) (Default: 1G).
--proxy-user NAME	User to impersonate when submitting the application.
--help, -h	Show this help message and exit
--verbose, -v	Print additional debug output
--version,	Print the version of current Spark
Spark standalone with cluster deploy mode only:	
--driver-cores NUM	Cores for driver (Default: 1).
Spark standalone or Mesos with cluster deploy mode only:	
--supervise	If given, restarts the driver on failure.
--kill SUBMISSION_ID	If given, kills the driver specified.
--status SUBMISSION_ID	If given, requests the status of the driver specified.
Spark standalone and Mesos only:	
--total-executor-cores NUM	Total cores for all executors.
Spark standalone and YARN only:	
--executor-cores NUM	Number of cores per executor. (Default: 1 in YARN mode, or all available cores on the worker in standalone mode)
YARN-only:	
--driver-cores NUM	Number of cores used by the driver, only in cluster mode (Default: 1).
--queue QUEUE_NAME	The YARN queue to submit to (Default: "default").
--num-executors NUM	Number of executors to launch (Default: 2).
--archives ARCHIVES	Comma separated list of archives to be extracted into the working directory of each executor.
--principal PRINCIPAL	Principal to be used to login to KDC, while running on secure HDFS.
--keytab KEYTAB	The full path to the file that contains the keytab for the principal specified above. This keytab will be copied to the node running the Application Master via the Secure Distributed Cache, for renewing the login tickets and the delegation tokens periodically.

- `--class`
- `--conf` or `-c`
- `--deploy-mode`
- `--driver-class-path`
- `--driver-cores` for **Standalone cluster mode** only
- `--driver-java-options`
- `--driver-library-path`
- `--driver-memory`
- `--executor-memory`
- `--files`
- `--jars`
- `--kill` for **Standalone cluster mode** only
- `--master`
- `--name`
- `--packages`
- `--exclude-packages`
- `--properties-file`
- `--proxy-user`
- `--py-files`
- `--repositories`
- `--status` for **Standalone cluster mode** only
- `--total-executor-cores`

List of switches, i.e. command-line options that do not take parameters:

- `--help` or `-h`
- `--supervise` for **Standalone cluster mode** only
- `--usage-error`
- `--verbose` or `-v`

- `--version`

YARN-only options:

- `--archives`
- `--executor-cores`
- `--keytab`
- `--num-executors`
- `--principal`
- `--queue`

Environment Variables

The following is the list of environment variables that are considered when command-line options are not specified:

- `MASTER` for `--master`
- `SPARK_DRIVER_MEMORY` for `--driver-memory`
- `SPARK_EXECUTOR_MEMORY` (see [Environment Variables](#) in the `SparkContext` document)
- `SPARK_EXECUTOR_CORES`
- `DEPLOY_MODE`
- `SPARK_YARN_APP_NAME`
- `_SPARK_CMD_USAGE`

External packages and custom repositories

The `spark-submit` utility supports specifying external packages using Maven coordinates using `--packages` and custom repositories using `--repositories`.

```
./bin/spark-submit \
--packages my:awesome:package \
--repositories s3n://$aws_ak:$aws_sak@bucket/path/to/repo
```

[FIXME](#) Why should I care?

Low-level details of spark-submit

Note

Set `SPARK_PRINT_LAUNCH_COMMAND` to see the final command to be executed, e.g.

```
SPARK_PRINT_LAUNCH_COMMAND=1 ./bin/spark-shell
```

Refer to [Print Launch Command of Spark Scripts](#).

The source code of the script lies in <https://github.com/apache/spark/blob/master/bin/spark-submit>.

When you execute the `spark-submit` script you basically launch `org.apache.spark.deploy.SparkSubmit` class (via another `spark-class` script) passing on the command line arguments.

At startup, the `spark-class` script loads additional environment settings (see [section on spark-env.sh in this document](#)).

And then `spark-class` searches for so-called **the Spark assembly jar** (`spark-assembly.hadoop..jar`) in `SPARK_HOME/lib` or `SPARK_HOME/assembly/target/scala-$SPARK_SCALA_VERSION` for a binary distribution or Spark built from sources, respectively.

Note

Set `SPARK_PREPEND_CLASSES` to have the Spark launcher classes (from `$SPARK_HOME/launcher/target/scala-$SPARK_SCALA_VERSION/classes`) to appear before the Spark assembly jar. It's useful for development so your changes don't require rebuilding Spark from the beginning.

As the last step in the process, `org.apache.spark.launcher.Main` class is executed with `org.apache.spark.deploy.SparkSubmit` and the other command line arguments (given to `spark-submit` at the very beginning). The Main class programmatically computes the final command to be executed.

TODO (further review)

- OptionParser class
- `spark-defaults.conf` in `SPARK_CONF_DIR` or `$SPARK_HOME/conf`
- SparkSubmit itself

spark-env.sh - load additional environment settings

- `spark-env.sh` consists of environment settings to configure Spark for your site.

```
export JAVA_HOME=/your/directory/java
export HADOOP_HOME=/usr/lib/hadoop
export SPARK_WORKER_CORES=2
export SPARK_WORKER_MEMORY=1G
```

- `spark-env.sh` is loaded at the startup of Spark's command line scripts.
- `SPARK_ENV_LOADED` env var is to ensure the `spark-env.sh` script is loaded once.
- `SPARK_CONF_DIR` points at the directory with `spark-env.sh` or `$SPARK_HOME/conf` is used.
- `spark-env.sh` is executed if it exists.
- `$SPARK_HOME/conf` directory has `spark-env.sh.template` file that serves as a template for your own custom configuration.

Consult [Environment Variables](#) in the official documentation.

spark-class

`bin/spark-class` shell script is the script launcher for internal Spark classes.

Note	Ultimately, any shell script in Spark calls <code>spark-class</code> script.
------	--

When started, it loads `$SPARK_HOME/bin/load-spark-env.sh`, searches for the Spark assembly jar, and starts [org.apache.spark.launcher.Main](#).

org.apache.spark.launcher.Main

[org.apache.spark.launcher.Main](#) is the command-line launcher used in Spark scripts, like `spark-class`.

It uses `SPARK_PRINT_LAUNCH_COMMAND` to print launch command to standard output.

It builds the command line for a Spark class using the environment variables:

- `SPARK_DAEMON_JAVA_OPTS` and `SPARK_MASTER_OPTS` to be added to the command line of the command.
- `SPARK_DAEMON_MEMORY` (default: `1g`) for `-Xms` and `-Xmx`.

Spark Architecture

Spark uses a **master/worker architecture**. There is a [driver](#) that talks to a single coordinator called [master](#) that manages [workers](#) in which [executors](#) run.

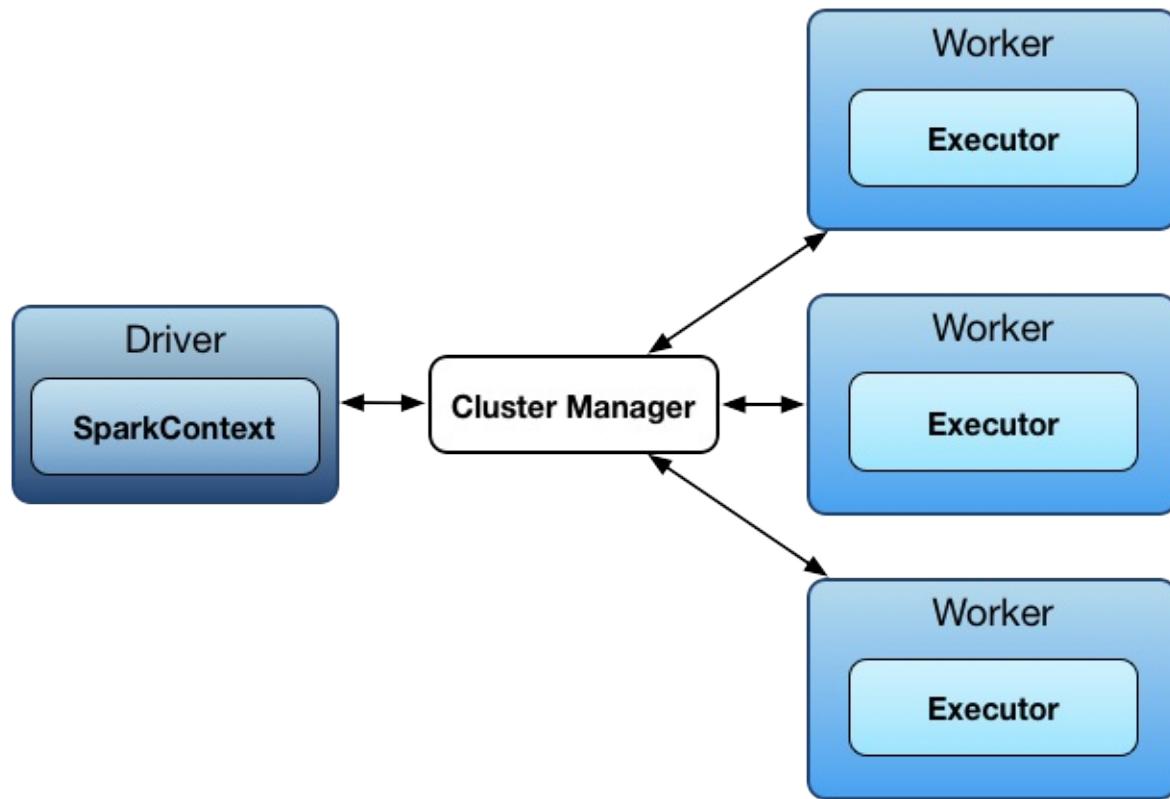


Figure 1. Spark architecture

The driver and the executors run in their own Java processes. You can run them all on the same (*horizontal cluster*) or separate machines (*vertical cluster*) or in a mixed machine configuration.

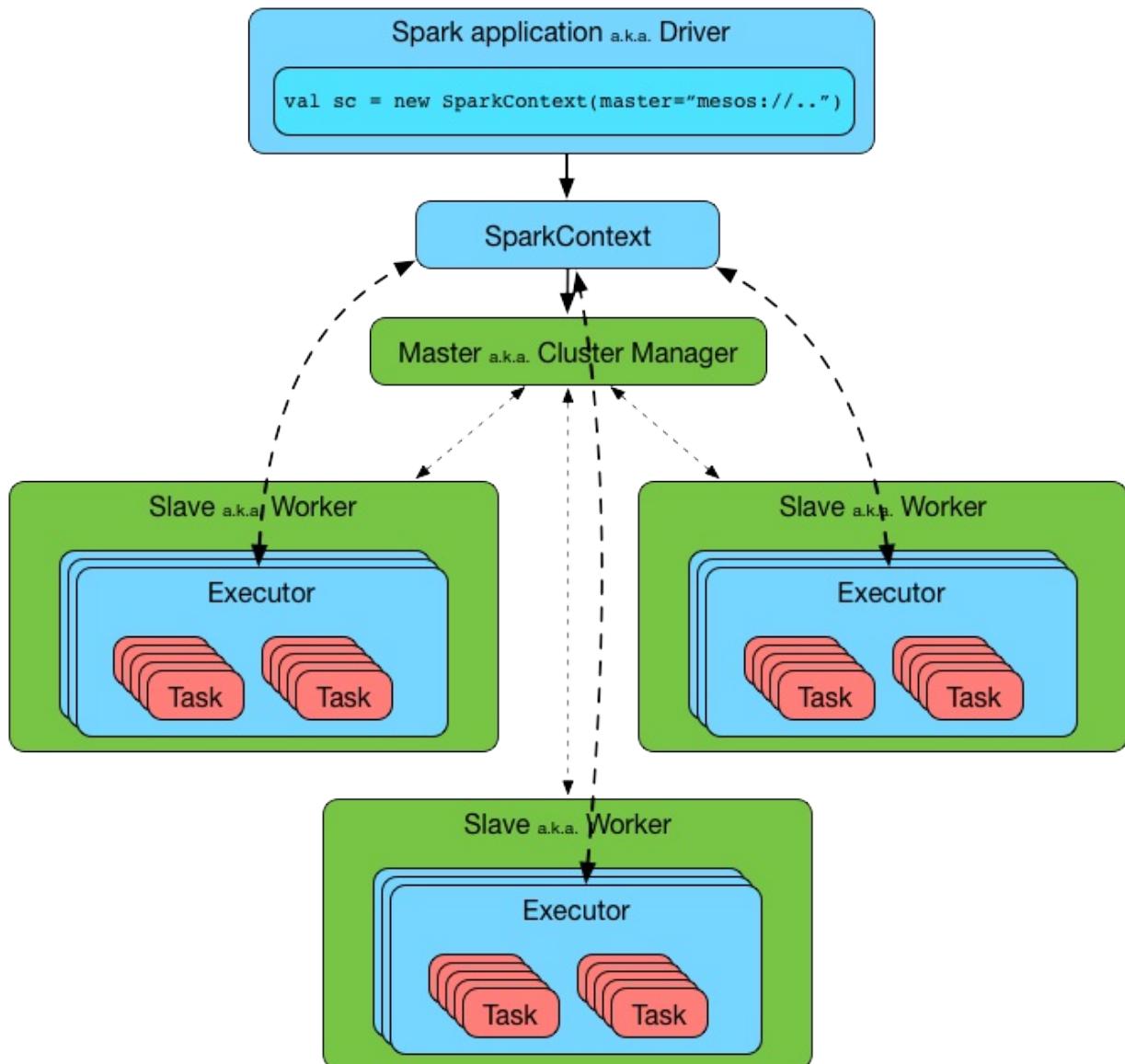


Figure 2. Spark architecture in detail

Physical machines are called **hosts** or **nodes**.

Driver

A Spark **driver** is the process that creates and owns an instance of `SparkContext`. It is your Spark application that launches the `main` method in which the instance of `SparkContext` is created. It is the cockpit of jobs and tasks execution (using `DAGScheduler` and `Task Scheduler`). It hosts [Web UI](#) for the environment.

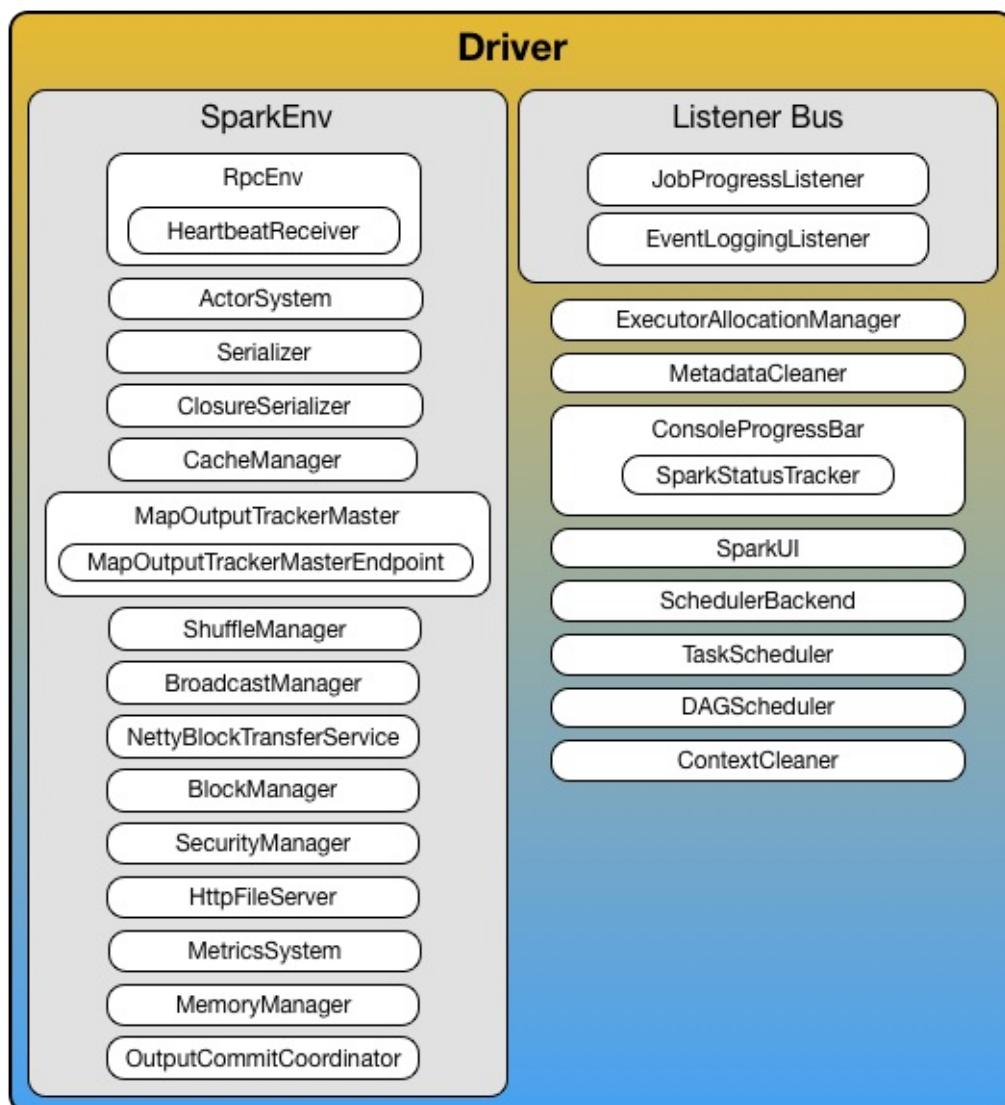


Figure 1. Driver with the services

It splits a Spark application into tasks and schedules them to run on executors.

A driver is where the task scheduler lives and spawns tasks across workers.

A driver coordinates workers and overall execution of tasks.

Driver requires the additional services (beside the common ones like `ShuffleManager`, `MemoryManager`, `BlockTransferService`, `BroadcastManager`, `CacheManager`):

- Listener Bus
- driverActorSystemName
- [RPC Environment](#) (for Netty and Akka)
- [MapOutputTrackerMaster](#) with the name **MapOutputTracker**
- [BlockManagerMaster](#) with the name **BlockManagerMaster**
- [HttpFileServer](#)
- [MetricsSystem](#) with the name **driver**
- [OutputCommitCoordinator](#) with the endpoint's name **OutputCommitCoordinator**

Caution

[FIXME](#) Diagram of RpcEnv for a driver (and later executors). Perhaps it should be in the notes about RpcEnv?

Master

A **master** is a running Spark instance that connects to a cluster manager for resources.

The master acquires cluster nodes to run executors.

Caution

[FIXME](#) Add it to the Spark architecture figure above.

Workers

Workers (aka **slaves**) are running Spark instances where executors live to execute tasks. They are the compute nodes in Spark.

Caution	FIXME Are workers perhaps part of Spark Standalone only?
---------	--

Caution	FIXME How many executors are spawned per worker?
---------	--

A worker receives serialized tasks that it runs in a thread pool.

It hosts a local [Block Manager](#) that serves blocks to other workers in a Spark cluster. Workers communicate among themselves using their Block Manager instances.

Caution	FIXME Diagram of a driver with workers as boxes.
---------	--

Explain task execution in Spark and understand Spark's underlying execution model.

New vocabulary often faced in Spark UI

[When you create SparkContext](#), each worker starts an executor. This is a separate process (JVM), and it loads your jar, too. The executors connect back to your driver program. Now the driver can send them commands, like `flatMap`, `map` and `reduceByKey`. When the driver quits, the executors shut down.

A new process is not started for each step. A new process is started on each worker when the `SparkContext` is constructed.

The executor deserializes the command (this is possible because it has loaded your jar), and executes it on a partition.

Shortly speaking, an application in Spark is executed in three steps:

1. Create RDD graph, i.e. DAG (directed acyclic graph) of RDDs to represent entire computation.
2. Create stage graph, i.e. a DAG of stages that is a logical execution plan based on the RDD graph. Stages are created by breaking the RDD graph at shuffle boundaries.
3. Based on the plan, schedule and execute tasks on workers.

In the [WordCount example](#), the RDD graph is as follows:

file → lines → words → per-word count → global word count → output

Based on this graph, two stages are created. The **stage** creation rule is based on the idea of **pipelining** as many **narrow transformations** as possible. RDD operations with "narrow" dependencies, like `map()` and `filter()`, are pipelined together into one set of tasks in each stage.

In the end, every stage will only have shuffle dependencies on other stages, and may compute multiple operations inside it.

In the WordCount example, the narrow transformation finishes at per-word count. Therefore, you get two stages:

- file → lines → words → per-word count
- global word count → output

Once stages are defined, Spark will generate tasks from stages. The first stage will create a series of **ShuffleMapTask** and the last stage will create **ResultTasks** because in the last stage, one action operation is included to produce results.

The number of tasks to be generated depends on how your files are distributed. Suppose that you have 3 three different files in three different nodes, the first stage will generate 3 tasks: one task per partition.

Therefore, you should not map your steps to tasks directly. A task belongs to a stage, and is related to a partition.

The number of tasks being generated in each stage will be equal to the number of partitions.

Cleanup

Caution	FIXME
---------	-------

Settings

- `spark.worker.cleanup.enabled` (default: `false`) **Cleanup** enabled.

Executors

Executors are distributed agents responsible for [executing tasks](#). They *typically* (i.e. not always) run for the entire lifetime of a Spark application. Executors send [active task metrics](#) to a [driver](#). They also inform [executor backends](#) about task status updates (task results including).

Note	Executors are managed solely by executor backends .
------	---

Executors provide in-memory storage for RDDs that are cached in Spark applications (via [Block Manager](#)).

When executors are started they register themselves with the driver and communicate directly to execute tasks.

Executor offers are described by executor id and the host on which an executor runs (see [Resource Offers](#) in this document).

Executors use a [thread pool](#) for [sending metrics](#) and [launching tasks](#) (by means of [TaskRunner](#)).

Each executor can run multiple tasks over its lifetime, both in parallel and sequentially.

It is recommended to have as many executors as data nodes and as many cores as you can get from the cluster.

Executors are described by their **id**, **hostname**, **environment** (as `SparkEnv`), and **classpath** (and, less importantly, and more for internal optimization, whether they run in [local](#) or [cluster mode](#)).

Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.executor.Executor</code> logger to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre style="background-color: #f0f0f0; padding: 5px;">log4j.logger.org.apache.spark.executor.Executor=INFO</pre> <p>Refer to Logging.</p>
------------	--

When an executor is started you should see the following INFO messages in the logs:

```
INFO Executor: Starting executor ID [executorId] on host [executorHostname]
INFO Executor: Using REPL class URI: http://[executorHostname]:56131
```

It creates an RPC endpoint for communication with the driver.

(only for non-local mode) Executors initialize the application using [BlockManager.initialize\(\)](#).

Executors maintain a mapping between task ids and running tasks (as instances of [TaskRunner](#)) in `runningTasks` internal map. Consult [Launching Tasks](#) section.

A worker requires the additional services (beside the common ones like ...):

- `executorActorSystemName`
- [RPC Environment](#) (for Akka only)
- [MapOutputTrackerWorker](#)
- [MetricsSystem](#) with the name `executor`

Caution

[**FIXME**](#) How many cores are assigned per executor?

Coarse-Grained Executors

Coarse-grained executors are executors that use [CoarseGrainedExecutorBackend](#) for task scheduling.

Launching Tasks

`Executor.launchTask` creates a [TaskRunner](#) that is then executed on [Executor task launch worker Thread Pool](#).

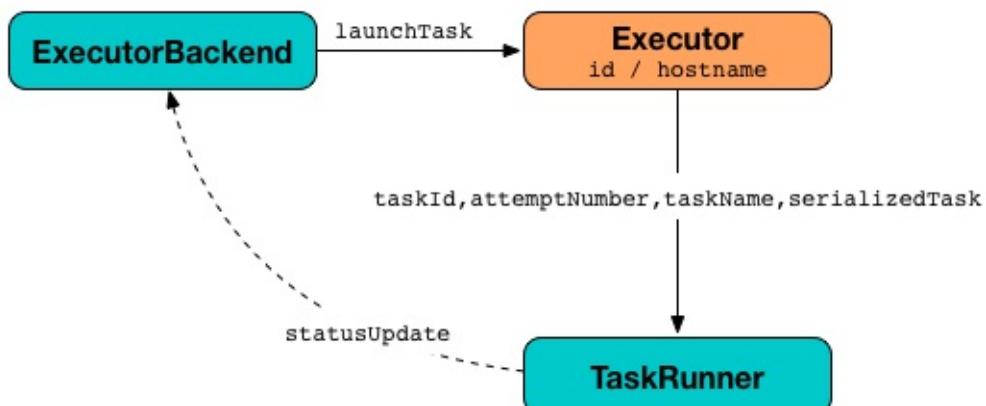


Figure 1. Launching tasks on executor using TaskRunners

It accepts the context (as [ExecutorBackend](#)), attempt number, and the id, name and serialized form of the task.

The newly-created `TaskRunner` is registered in the internal `runningTasks` map.

Heartbeats with Active Tasks Metrics

Executors keep sending [active tasks metrics](#) to a driver every `spark.executor.heartbeatInterval`.

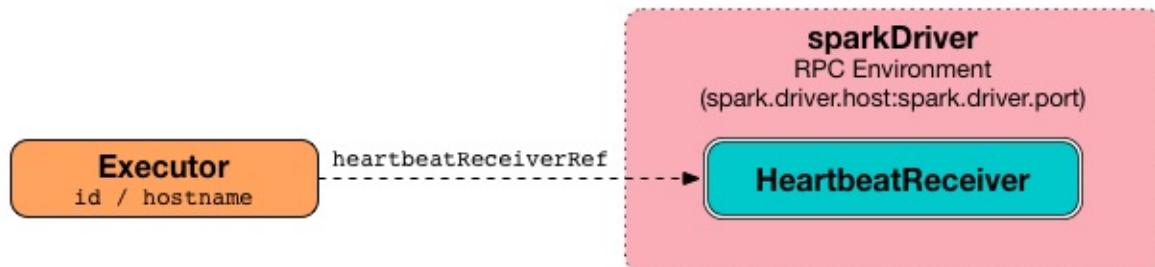


Figure 2. Executors use HeartbeatReceiver endpoint to report task metrics

The structure sent is an array of `(Long, TaskMetrics)`.

	FIXME
Caution	<ul style="list-style-type: none"> • What's in <code>taskRunner.task.metrics</code> ? • What's in <code>Heartbeat</code> ? Why is <code>blockManagerId</code> sent? • What's in <code>RpcUtils.makeDriverRef</code> ?

Executors use internal `driver-heartbeater` daemon single-thread scheduled pool executor, i.e. `ScheduledThreadPoolExecutor`.

TaskRunner

TaskRunner is a thread of execution that runs a task. It requires a [ExecutorBackend](#) (to send status updates to), task and attempt ids, task name, and serialized version of the task (as `ByteBuffer`). It sends updates about task execution to the [ExecutorBackend](#).

Sending task status updates to [ExecutorBackend](#) is a mechanism to inform the executor backend about task being started (`TaskState.RUNNING`), task finish together with the serialized result (`TaskState.FINISHED`), task being killed (`TaskState.KILLED`) and task failures (`TaskState.FAILED`).

When a `TaskRunner` starts running, it prints the following INFO message to the logs:

```
INFO Executor: Running [taskName] (TID [taskId])
```

`taskId` is the id of the task being executed in [Executor task launch worker-\[taskId\]](#).

Information about it is sent to the driver using [ExecutorBackend](#) as `TaskState.RUNNING` and zero-byte `ByteBuffer`.

TaskRunner can run a single task only. When TaskRunner finishes, it is removed from the internal `runningTasks` map.

Caution	FIXME TaskMemoryManager and task serialization and deserialization
---------	--

It first deserializes the task (using `Task.deserializeWithDependencies`), `updateDependencies(taskFiles, taskJars)`, and deserializes the task bytes into a `Task` instance (using the globally-configured `Serializer`). The `Task` instance has the [TaskMemoryManager](#) set.

This is the moment when a task can stop its execution if it was killed while being deserialized. If not killed, `TaskRunner` continues executing the task.

You should see the following DEBUG message in the logs:

```
DEBUG Task [taskId]'s epoch is [task.epoch]
```

TaskRunner sends update of the epoch of the task to [MapOutputTracker](#).

Caution	FIXME Why is <code>MapOutputTracker.updateEpoch</code> needed?
---------	--

Task runs (with `taskId`, `attemptNumber`, and the globally-configured `MetricsSystem`). See [Task Execution](#).

When a task finishes, it returns a value and `accumUpdates`.

Caution	FIXME What are <code>accumUpdates</code> ? It should perhaps be described in Task Execution .
---------	---

The result value is serialized (using the other instance of `Serializer`, i.e. `serializer` - there are two `Serializer` instances in `SparkContext.env`).

A `DirectTaskResult` that contains the serialized result and `accumUpdates` is serialized.

If `maxResultSize` is set and the size of the serialized result exceeds the value, a `SparkException` is reported.

```
scala> sc.getConf.get("spark.driver.maxResultSize")
res5: String = 1m

scala> sc.parallelize(0 to 1024*1024+10, 1).collect
...
INFO DAGScheduler: Job 3 failed: collect at <console>:25, took 0.075073 s
org.apache.spark.SparkException: Job aborted due to stage failure: Total size of serializ
at org.apache.spark.scheduler.DAGScheduler.org$apache$spark$scheduler$DAGScheduler$$fai
```

If however the size exceeds `akkaFrameSize`, ...[FIXME](#).

A successful execution is "announced" as INFO to the logs:

```
INFO Executor: Finished [taskName] (TID [taskId]). [resultSize] bytes result sent to driv
```

The serialized result is sent to the driver using [ExecutorBackend](#) as `TaskState.FINISHED` and `serializedResult`.

FetchFailedException

Caution	FIXME
---------	-----------------------

`FetchFailedException` exception is thrown when an executor (more specifically [TaskRunner](#)) has failed to fetch a shuffle block.

It contains the following:

- the unique identifier for a BlockManager (as `BlockManagerId`)
- `shuffleId`
- `mapId`
- `reduceId`
- `message` - a short exception message
- `cause` - a `Throwable` object

TaskRunner catches it and informs [ExecutorBackend](#) about the case (using `statusUpdate` with `TaskState.FAILED` task state).

Caution	FIXME Image with the call to ExecutorBackend.
---------	---

Resource Offers

Read [resourceOffers](#) in [TaskSchedulerImpl](#) and [resourceOffer](#) in [TaskSetManager](#).

Executor task launch worker Thread Pool

Executors use daemon cached thread pools called **Executor task launch worker-ID** (with `ID` being the task id) for [launching tasks](#).

Executor Memory - `spark.executor.memory` or `SPARK_EXECUTOR_MEMORY` settings

You can control the amount of memory per executor using `spark.executor.memory` setting. It sets the available memory equally for all executors per application.

Note	The amount of memory per executor is looked up when <code>SparkContext</code> is created.
------	---

You can change the assigned memory per executor per node in [standalone cluster](#) using `SPARK_EXECUTOR_MEMORY` environment variable.

You can find the value displayed as **Memory per Node** in [web UI](#) for standalone Master (as depicted in the figure below).

Spark 2.0.0-SNAPSHOT **Spark Master at spark://localhost:7077**

URL: spark://localhost:7077
REST URL: spark://localhost:6066 (*cluster mode*)
Alive Workers: 1
Cores in use: 2 Total, 2 Used
Memory in use: 2.0 GB Total, 2.0 GB Used
Applications: 1 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20160109142947-192.168.1.12-53888	192.168.1.12:53888	ALIVE	2 (2 Used)	2.0 GB (2.0 GB Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160109143144-0001 (kill)	Spark shell	2	2.0 GB	2016/01/09 14:31:44	jacek	RUNNING	52 s

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160109143059-0000	Spark shell	2	1024.0 MB	2016/01/09 14:30:59	jacek	FINISHED	24 s

Figure 3. Memory per Node in Spark Standalone's web UI

The above figure shows the result of running [Spark shell](#) with the amount of memory per executor defined explicitly (on command line), i.e.

```
./bin/spark-shell --master spark://localhost:7077 -c spark.executor.memory=2g
```

Metrics

Executors use [Metrics System](#) (via `ExecutorSource`) to report metrics about internal status.

Note	Metrics are only available for cluster modes, i.e. <code>local</code> mode turns metrics off.
------	---

The name of the source is **executor**.

It emits the following numbers:

- **threadpool.activeTasks** - the approximate number of threads that are actively executing tasks (using [ThreadPoolExecutor.getActiveCount\(\)](#))
- **threadpool.completeTasks** - the approximate total number of tasks that have completed execution (using [ThreadPoolExecutor.getCompletedTaskCount\(\)](#))
- **threadpool.currentPool_size** - the current number of threads in the pool (using [ThreadPoolExecutor.getPoolSize\(\)](#))
- **threadpool.maxPool_size** - the maximum allowed number of threads that have ever simultaneously been in the pool (using [ThreadPoolExecutor.getMaximumPoolSize\(\)](#))
- **filesystem.hdfs / read_bytes** using [FileSystem.getAllStatistics\(\)](#) and `getBytesRead()`
- **filesystem.hdfs.write_bytes** using [FileSystem.getAllStatistics\(\)](#) and `getBytesWritten()`
- **filesystem.hdfs.read_ops** using [FileSystem.getAllStatistics\(\)](#) and `getReadOps()`
- **filesystem.hdfs.largeRead_ops** using [FileSystem.getAllStatistics\(\)](#) and `getLargeReadOps()`
- **filesystem.hdfs.write_ops** using [FileSystem.getAllStatistics\(\)](#) and `getWriteOps()`
- **filesystem.file.read_bytes**
- **filesystem.file.write_bytes**
- **filesystem.file.read_ops**
- **filesystem.file.largeRead_ops**
- **filesystem.file.write_ops**

Settings

- `spark.executor.cores` - the number of cores for an executor
- `spark.executor.extraClassPath` - a list of URLs representing the user classpath. Each entry is separated by system-dependent path separator, i.e. `:` on Unix/MacOS systems and `;` on Microsoft Windows.
- `spark.executor.extraJavaOptions` - extra Java options for executors

- `spark.executor.extraLibraryPath` - a list of additional library paths separated by system-dependent path separator, i.e. `:` on Unix/MacOS systems and `;` on Microsoft Windows.
- `spark.executor.userClassPathFirst` (default: `false`) controls whether to load classes in user jars before those in Spark jars.
- `spark.executor.heartbeatInterval` (default: `10s`) - the interval after which an executor reports heartbeat and metrics for active tasks to the driver. Refer to [Sending heartbeats and partial metrics for active tasks](#).
- `spark.executor.id`
- `spark.executor.instances` - the number of executors. When greater than `0`, it disables [Dynamic Allocation](#).
- `spark.executor.logs.rolling.maxSize`
- `spark.executor.logs.rolling.maxRetainedFiles`
- `spark.executor.logs.rolling.strategy`
- `spark.executor.logs.rolling.time.interval`
- `spark.executor.memory` (default: `1024 mebibytes`) - the amount of memory to use per executor process (equivalent to `SPARK_EXECUTOR_MEMORY` environment variable). See [Executor Memory - spark.executor.memory setting](#) in this document.
- `spark.executor.port`
- `spark.executor.uri` - equivalent to `SPARK_EXECUTOR_URI`
- `spark.repl.class.uri` (default: `null`) used when in `spark-shell` to create REPL ClassLoader to load new classes defined in the Scala REPL as a user types code.

Enable `INFO` logging level for `org.apache.spark.executor.Executor` logger to have the value printed out to the logs:

```
INFO Using REPL class URI: [classUri]
```

- `spark.akka.frameSize` (default: `128 MB`, maximum: `2047 MB`) - the configured max frame size for Akka messages. If a task result is bigger, executors use [block manager](#) to send results back.
- `spark.driver.maxResultSize` (default: `1g`)

Caution

FIXME `spark.driver.maxResultSize` is used in few other pages so decide where it should belong to and link the other places.

Spark Runtime Environment

Spark Runtime Environment is the runtime environment with Spark services that interact with each other to build Spark computing platform.

Spark Runtime Environment is represented by a [SparkEnv](#) object that holds all the required services for a running Spark instance, i.e. a master or an executor.

SparkEnv

SparkEnv holds all runtime objects for a running Spark instance, using [SparkEnv.createDriverEnv\(\)](#) for a driver and [SparkEnv.createExecutorEnv\(\)](#) for an executor.

Tip

Enable `DEBUG` logging level for `org.apache.spark.SparkEnv` logger to learn the low-level details of `SparkEnv`.

Add the following line to `conf/log4j.properties` with requested `DEBUG` log level:

```
log4j.logger.org.apache.spark.SparkEnv=DEBUG
```

You can access the Spark environment using `SparkEnv.get`.

```
scala> import org.apache.spark._  
import org.apache.spark._  
  
scala> SparkEnv.get  
res0: org.apache.spark.SparkEnv = org.apache.spark.SparkEnv@2220c5f7
```

SparkEnv.create()

`SparkEnv.create` is a common initialization procedure to create a Spark execution environment for either a driver or an executor.

It is called by [SparkEnv.createDriverEnv\(\)](#) and [SparkEnv.createExecutorEnv\(\)](#).

When executed, it creates a Serializer (based on `spark.serializer`) and, at `DEBUG` logging level, prints out the following message to the logs:

```
DEBUG Using serializer: [serializer.getClass]
```

It creates another Serializer (based on `spark.closure.serializer`).

It creates a ShuffleManager based on `spark.shuffle.manager` setting.

It creates a MemoryManager based on `spark.memory.useLegacyMode` setting.

Caution

FIXME What's MemoryManager?

It creates a NettyBlockTransferService.

It creates a BlockManagerMaster.

It creates a BlockManager.

It creates a BroadcastManager.

It creates a CacheManager.

It creates a MetricsSystem different for a driver and a worker.

It initializes `userFiles` temporary directory used for downloading dependencies for a driver while this is the executor's current working directory for an executor.

An OutputCommitCoordinator is created.

SparkEnv.createDriverEnv()

`SparkEnv.createDriverEnv` creates a **driver's (execution) environment** that is the Spark execution environment for a driver.

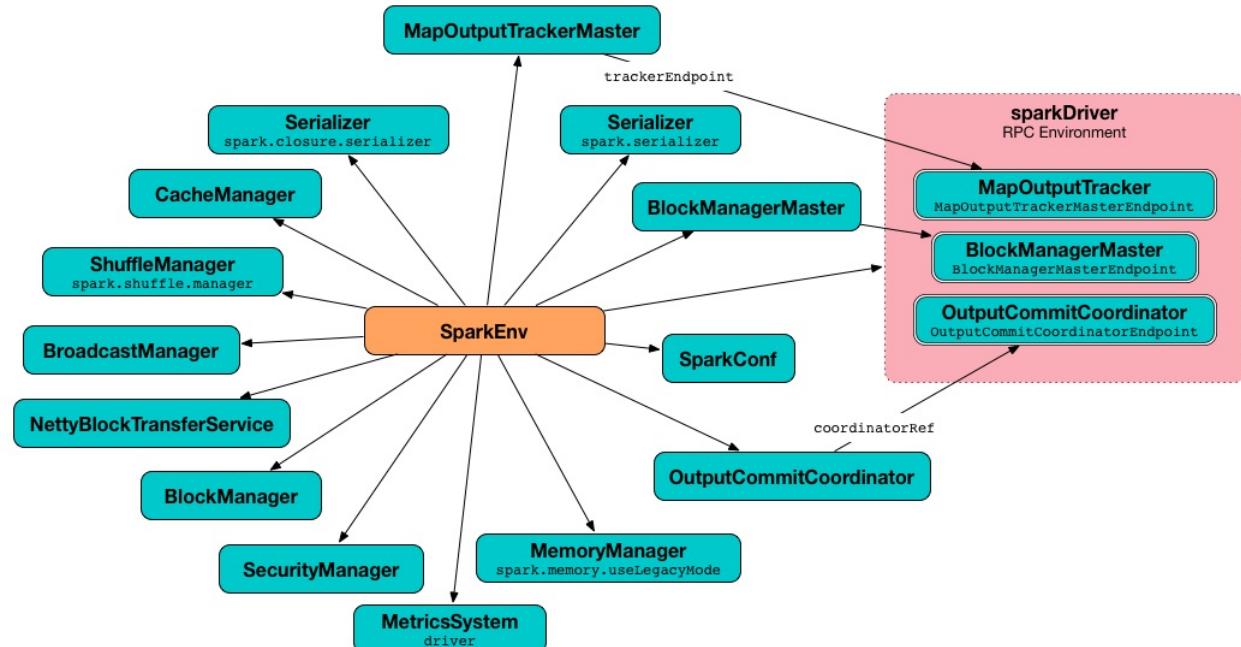


Figure 1. Spark Environment for driver

The method accepts an instance of [SparkConf](#), whether it runs in local mode or not, an instance of [listener bus](#), the number of driver's cores to use for execution in local mode or `0` otherwise, and a [OutputCommitCoordinator](#) (default: none).

Two driver-related properties `spark.driver.host` and `spark.driver.port` are expected in the Spark configuration.

For Akka-based RPC Environment (obsolete since Spark 1.6.0-SNAPSHOT), the name of the actor system for the driver is **sparkDriver**. See [clientMode](#) how it is created in detail.

It creates `MapOutputTrackerMaster` object and registers `MapOutputTracker` RPC endpoint as an instance of `MapOutputTrackerMasterEndpoint`. See [MapOutputTracker](#).

It creates a MetricsSystem for **driver**.

An OutputCommitCoordinator is created and **OutputCommitCoordinator** RPC endpoint registered.

SparkEnv.createExecutorEnv()

`SparkEnv.createExecutorEnv` creates an **executor's (execution) environment** that is the Spark execution environment for an executor.

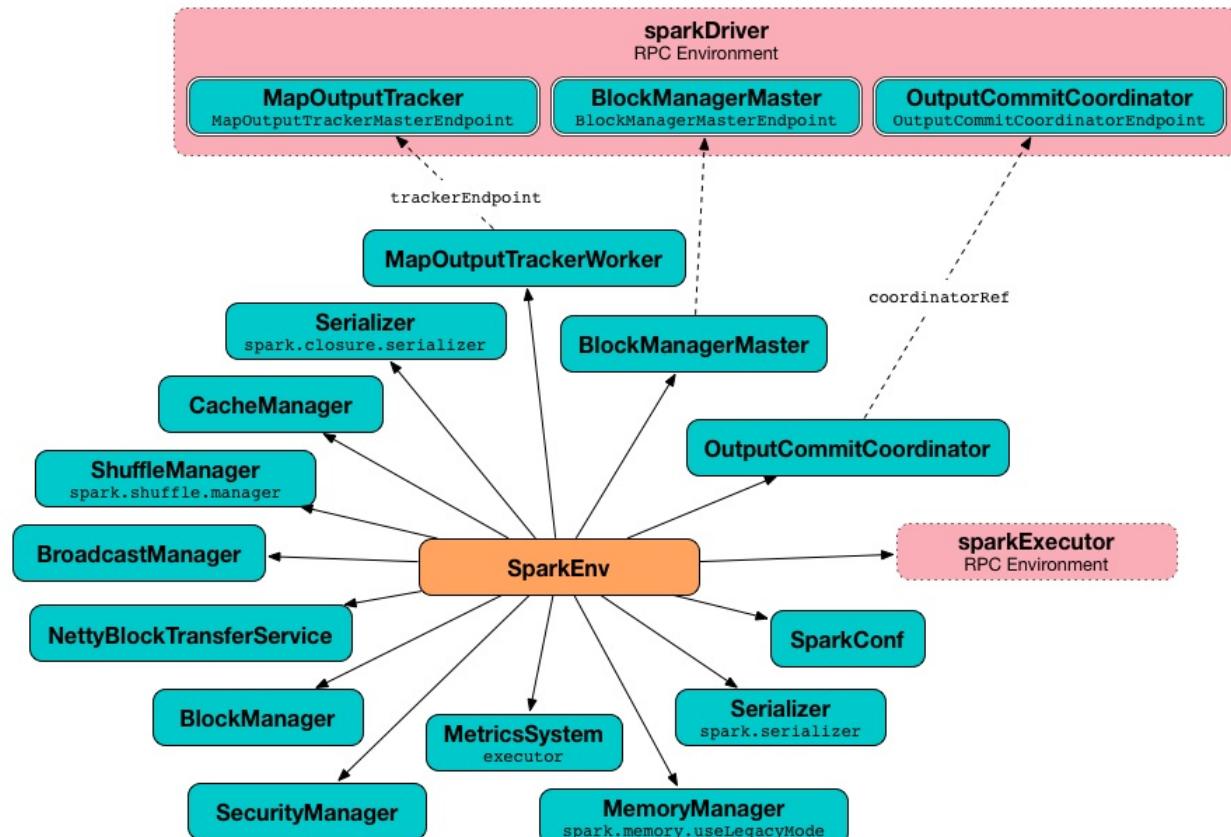


Figure 2. Spark Environment for executor

It uses `SparkConf`, the executor's identifier, hostname, port, the number of cores, and whether or not it runs in local mode.

For Akka-based RPC Environment (obsolete since Spark 1.6.0-SNAPSHOT), the name of the actor system for an executor is `sparkExecutor`.

It creates an `MapOutputTrackerWorker` object and looks up `MapOutputTracker` RPC endpoint. See [MapOutputTracker](#).

It creates a `MetricsSystem` for `executor` and starts it.

An `OutputCommitCoordinator` is created and `OutputCommitCoordinator` RPC endpoint looked up.

Settings

- `spark.driver.host` - the name of the machine where the (active) driver runs.
- `spark.driver.port` - the port of the driver.
- `spark.serializer` (`default: org.apache.spark.serializer.JavaSerializer`) - the Serializer.
- `spark.closure.serializer` (`default: org.apache.spark.serializer.JavaSerializer`) - the Serializer.
- `spark.shuffle.manager` (`default: sort`) - one of the three available implementations of [ShuffleManager](#) or a fully-qualified class name of a custom implementation of `ShuffleManager`.
 - `hash` OR `org.apache.spark.shuffle.hash.HashShuffleManager`
 - `sort` OR `org.apache.spark.shuffle.sort.SortShuffleManager`
 - `tungsten-sort` OR `org.apache.spark.shuffle.sort.TungstenSortShuffleManager`
- `spark.memory.useLegacyMode` (`default: false`) - `StaticMemoryManager` (`true`) or `UnifiedMemoryManager` (`false`).

DAGScheduler

Note

The introduction that follows was highly influenced by the scaladoc of [org.apache.spark.scheduler.DAGScheduler](#). As DAGScheduler is a private class it does not appear in the official API documentation. You are strongly encouraged to read [the sources](#) and only then read this and the related pages afterwards.

"Reading the sources", I say?! Yes, I am kidding!

Introduction

DAGScheduler is the scheduling layer of Apache Spark that implements **stage-oriented scheduling**, i.e. after an RDD action has been called it becomes a job that is then transformed into a set of stages that are submitted as TaskSets for execution (see [Execution Model](#)).

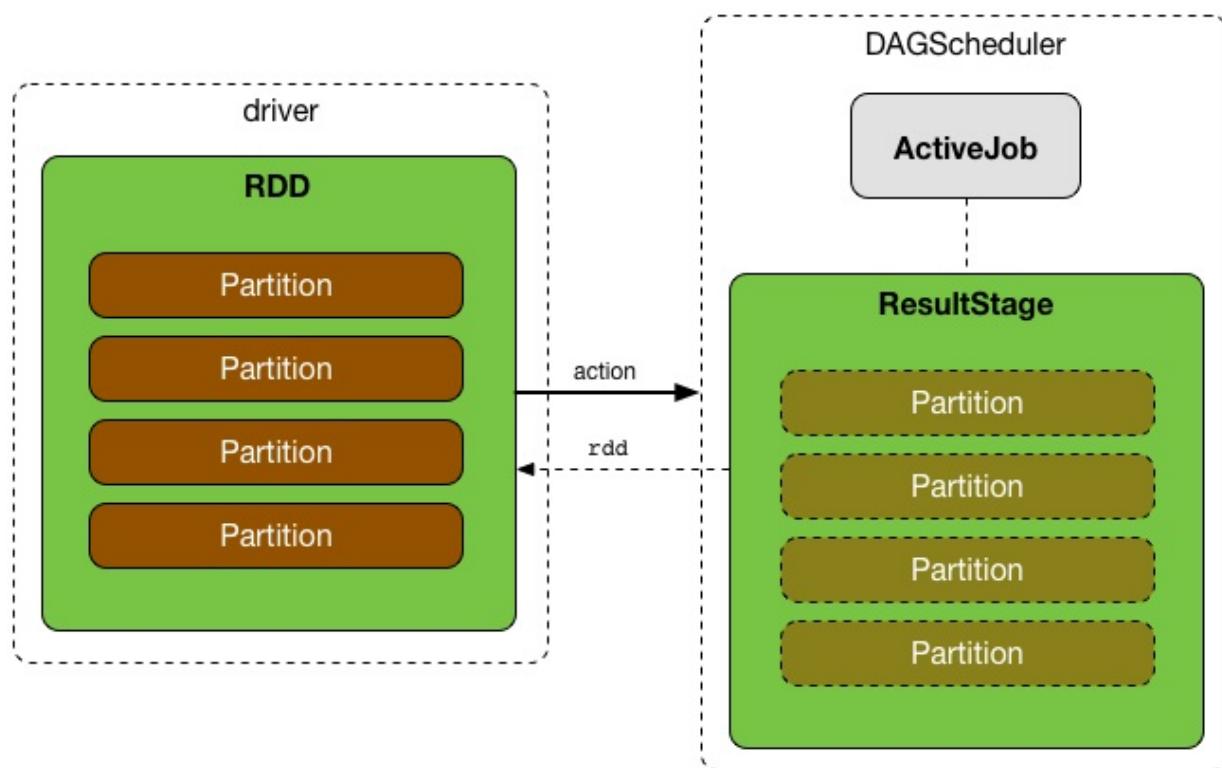


Figure 1. Executing action leads to new ResultStage and ActiveJob in DAGScheduler
The fundamental concepts of DAGScheduler are **jobs** and **stages** (refer to [Jobs](#) and [Stages](#) respectively).

DAGScheduler works on a driver. It is created as part of [SparkContext's initialization](#), right after [TaskScheduler](#) and [SchedulerBackend](#) are ready.

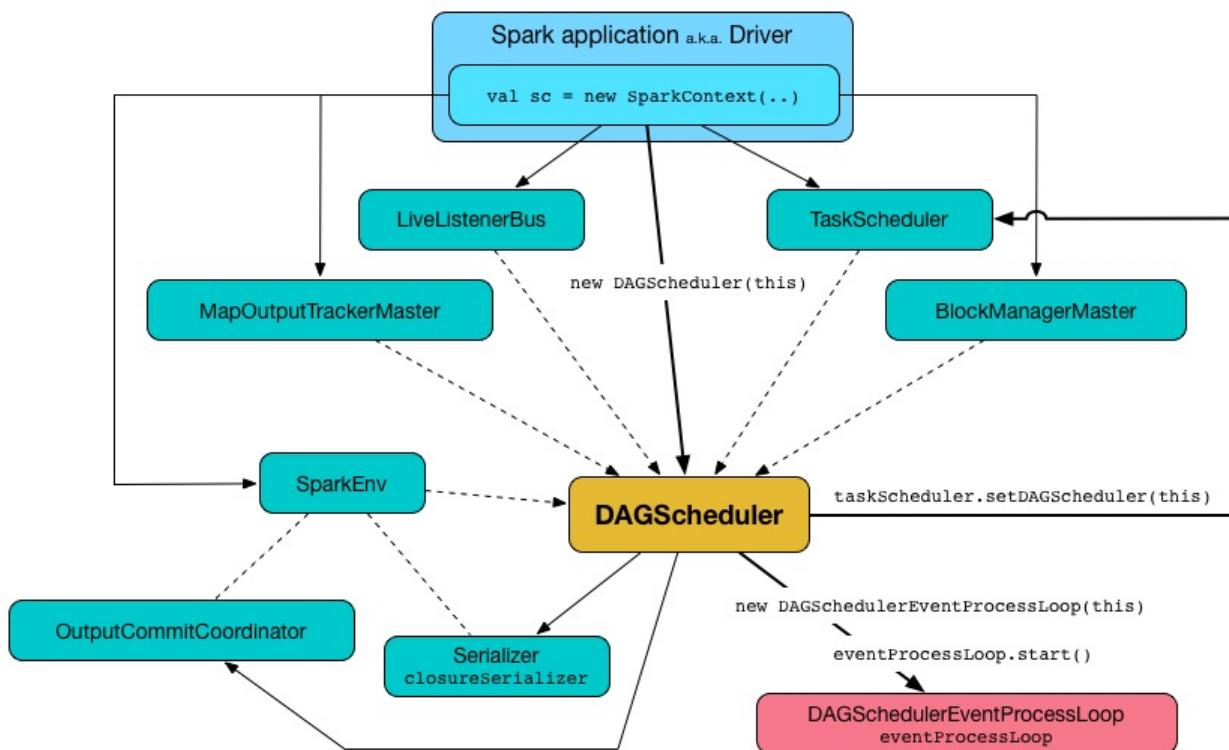


Figure 2. DAGScheduler as created by SparkContext with other services
DAGScheduler does three things in Spark (thorough explanations follow):

- Computes an **execution DAG**, i.e. DAG of stages, for a job.
- Determines the **preferred locations** to run each task on.
- Handles failures due to **shuffle output files** being lost.

It computes a **directed acyclic graph (DAG)** of stages for each job, keeps track of which RDDs and stage outputs are materialized, and finds a minimal schedule to run jobs. It then submits stages to [TaskScheduler](#).

In addition to coming up with the execution DAG, DAGScheduler also determines the preferred locations to run each task on, based on the current cache status, and passes the information to [TaskScheduler](#).

Furthermore, it handles failures due to shuffle output files being lost, in which case old stages may need to be resubmitted. Failures within a stage that are not caused by shuffle file loss are handled by the TaskScheduler itself, which will retry each task a small number of times before cancelling the whole stage.

DAGScheduler uses an **event queue architecture** in which a thread can post [DAGSchedulerEvent](#) events, e.g. a new job or stage being submitted, that DAGScheduler reads and executes sequentially. See the section [Internal Event Loop - dag-scheduler-event-loop](#).

DAGScheduler runs stages in topological order.

Tip

Turn `DEBUG` or more detailed `TRACE` logging level to see what happens inside `DAGScheduler`.

Add the following line to `conf/log4j.properties` with requested logging level - `DEBUG` or `TRACE`:

```
log4j.logger.org.apache.spark.scheduler.DAGScheduler=TRACE
```

`DAGScheduler` needs [SparkContext](#), [Task Scheduler](#), [Listener Bus](#), [MapOutputTracker](#) and [Block Manager](#) to work. However, at the very minimum, `DAGScheduler` needs `SparkContext` only (and asks `SparkContext` for the other services).

`DAGScheduler` reports metrics about its execution (refer to the section [Metrics](#)).

When `DAGScheduler` schedules a job as a result of [executing an action on a RDD](#) or [calling `SparkContext.runJob\(\)` method directly](#), it spawns parallel tasks to compute (partial) results per partition.

Internal Registries

`DAGScheduler` maintains the following information in internal registries:

- `nextJobId` for the next job id
- `numTotalJobs` (alias of `nextJobId`) for the total number of submitted
- `nextStageId` for the next stage id
- `jobIdToStageIds` for a mapping between jobs and their stages
- `stageIdToStage` for a mapping between stage ids to stages
- `shuffleToMapStage` for a mapping between ids to `ShuffleMapStages`
- `jobIdToActiveJob` for a mapping between job ids to `ActiveJobs`
- `waitingStages` for stages with parents to be computed
- `runningStages` for stages currently being run
- `failedStages` for stages that failed due to fetch failures (as reported by [CompletionEvents for FetchFailed end reasons](#)) and are going to be [resubmitted](#).
- `activeJobs` for a collection of `ActiveJob` instances

- `cacheLocs` is a mapping between RDD ids and their cache preferences per partition (as arrays indexed by partition numbers). Each array value is the set of locations where that RDD partition is cached on. See [Cache Tracking](#).
- `failedEpoch` is a mapping between failed executors and the epoch number when the failure was caught per executor.

Caution	FIXME Review <ul style="list-style-type: none"> • <code>cleanupStateForJobAndIndependentStages</code>
---------	---

DAGScheduler.resubmitFailedStages

`resubmitFailedStages()` is called to go over `failedStages` collection (of failed stages) and submit them (using [submitStage](#)).

If the failed stages collection contains any stage, the following INFO message appears in the logs:

```
INFO Resubmitting failed stages
```

`cacheLocs` and `failedStages` are cleared, and failed stages are [submitStage](#) one by one, ordered by job ids (in an increasing order).

Ultimately, all waiting stages are submitted (using [submitWaitingStages](#)).

DAGScheduler.runJob

When executed, `DAGScheduler.runJob` is given the following arguments:

- A **RDD** to run job on.
- A **function** to run on each partition of the RDD.
- A set of **partitions** to run on (not all partitions are always required to compute a job for actions like `first()` or `take()`).
- A callback **function** `resultHandler` to pass results of executing the function to.
- **Properties** to attach to a job.

It calls [DAGScheduler.submitJob](#) and then waits until a result comes using a [JobWaiter](#) object. A job can succeed or fail.

When a job succeeds, the following INFO shows up in the logs:

```
INFO Job [jobId] finished: [callSite], took [time] s
```

When a job fails, the following INFO shows up in the logs:

```
INFO Job [jobId] failed: [callSite], took [time] s
```

The method finishes by throwing an exception.

DAGScheduler.submitJob

`DAGScheduler.submitJob` is called by `SparkContext.submitJob` and `DAGScheduler.runJob`.

When called, it does the following:

- Checks whether the set of partitions to run a function on are in the range of available partitions of the RDD.
- Increments the internal `nextJobId` job counter.
- Returns a 0-task `JobWaiter` when no partitions are passed in.
- Or posts `JobSubmitted` event to `dag-scheduler-event-loop` and returns a `JobWaiter`.

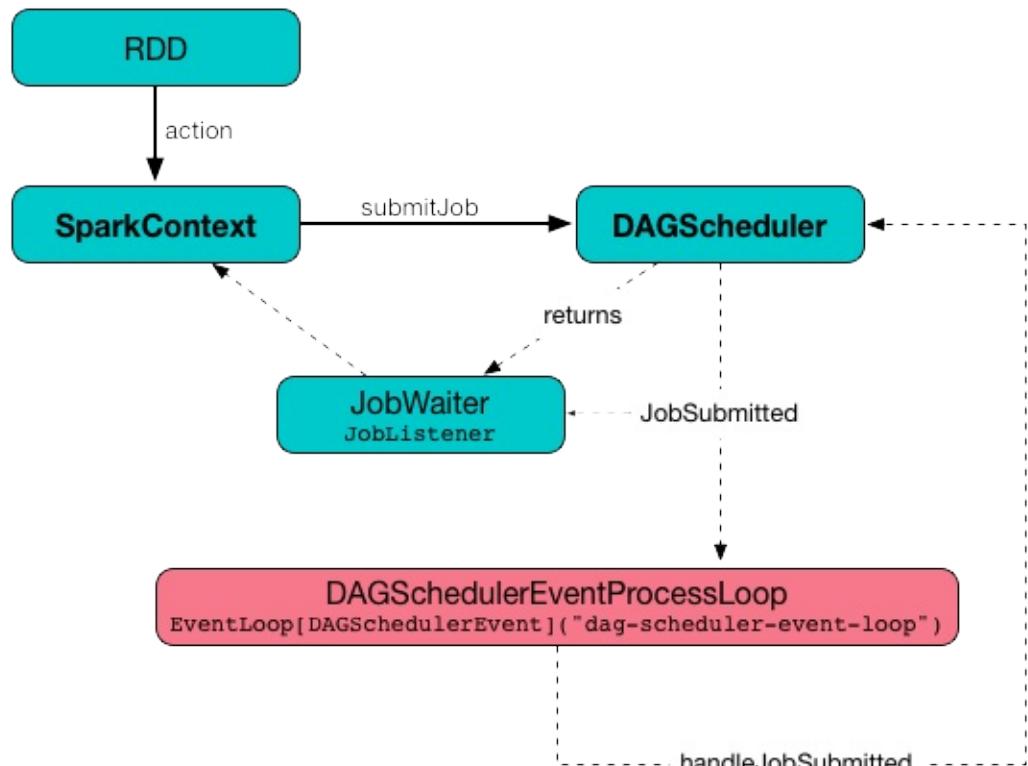
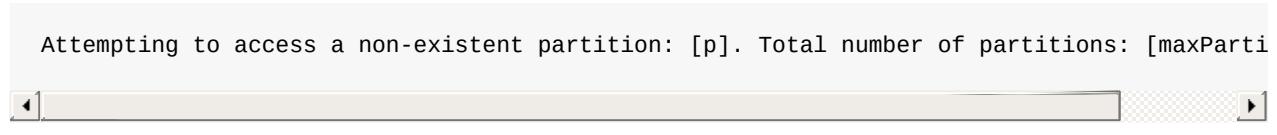


Figure 3. `DAGScheduler.submitJob`

You may see an exception thrown when the partitions in the set are outside the range:



Attempting to access a non-existent partition: [p]. Total number of partitions: [maxPartitions]

JobListener and Completion Events

You can listen for job completion or failure events after submitting a job to the DAGScheduler using `JobListener`. It is a `private[spark]` contract (a Scala trait) with the following two methods:

```
private[spark] trait JobListener {
    def taskSucceeded(index: Int, result: Any)
    def jobFailed(exception: Exception)
}
```

A job listener is notified each time a task succeeds (by `def taskSucceeded(index: Int, result: Any)`), as well as if the whole job fails (by `def jobFailed(exception: Exception)`).

An instance of `JobListener` is used in the following places:

- In `ActiveJob` as a listener to notify if tasks in this job finish or the job fails.
- In `DAGScheduler.handleJobSubmitted`
- In `DAGScheduler.handleMapStageSubmitted`
- In `JobSubmitted`
- In `MapStageSubmitted`

The following are the job listeners used:

- `JobWaiter` waits until DAGScheduler completes the job and passes the results of tasks to a `resultHandler` function.
- `ApproximateActionListener` [FIXME](#)

JobWaiter

A `JobWaiter` is an extension of `JobListener`. It is used as the return value of `DAGScheduler.submitJob` and `DAGScheduler.submitMapStage`. You can use a JobWaiter to block until the job finishes executing or to cancel it.

While the methods execute, `JobSubmitted` and `MapStageSubmitted` events are posted that reference the JobWaiter.

Since a `JobWaiter` object is a `JobListener` it gets notifications about `taskSucceeded` and `jobFailed`. When the total number of tasks (that equals the number of partitions to compute) equals the number of `taskSucceeded`, the `JobWaiter` instance is marked succeeded. A `jobFailed` event marks the `JobWaiter` instance failed.

- **FIXME** Who's using `submitMapStage` ?

DAGScheduler.executorAdded

`executorAdded(execId: String, host: String)` method simply posts a `ExecutorAdded` event to `eventProcessLoop`.

DAGScheduler.taskEnded

`taskEnded(task: Task[_], reason: TaskEndReason, result: Any, accumUpdates: Map[Long, Any], taskInfo: TaskInfo, taskMetrics: TaskMetrics)` method simply posts a `CompletionEvent` event to `eventProcessLoop`.

Note	DAGScheduler.taskEnded method is called by a TaskSetManager to report task completions, failures including.
------	---

dag-scheduler-event-loop - Internal Event Loop

`DAGScheduler.eventProcessLoop` (of type `DAGSchedulerEventProcessLoop`) - is the event process loop to which Spark (by `DAGScheduler.submitJob`) posts jobs to schedule their execution. Later on, `TaskSetManager` talks back to DAGScheduler to inform about the status of the tasks using the same "communication channel".

It allows Spark to release the current thread when posting happens and let the event loop handle events on a separate thread - asynchronously.

...IMAGE...**FIXME**

Internally, `DAGSchedulerEventProcessLoop` uses `java.util.concurrent.LinkedBlockingDeque` blocking deque that grows indefinitely (i.e. up to `Integer.MAX_VALUE` events).

The name of the single "logic" thread that reads events and takes decisions is **dag-scheduler-event-loop**.

```
"dag-scheduler-event-loop" #89 daemon prio=5 os_prio=31 tid=0x000007f809bc0a000 nid=0xc903
```

The following are the current types of `DAGSchedulerEvent` events that are handled by `DAGScheduler`:

- `JobSubmitted` - posted when an action job is submitted to DAGScheduler (via `submitJob` or `runApproximateJob`).
- `MapStageSubmitted` - posted when a ShuffleMapStage is submitted (via `submitMapStage`).
- `StageCancelled`
- `JobCancelled`
- `JobGroupCancelled`
- `AllJobsCancelled`
- `BeginEvent` - posted when `TaskSetManager` reports that a task is starting.
`dagScheduler.handleBeginEvent` is executed in turn.
- `GettingResultEvent` - posted when `TaskSetManager` reports that a task has completed and results are being fetched remotely.
`dagScheduler.handleGetTaskResult` executes in turn.
- `CompletionEvent` - posted when `TaskSetManager` reports that a task has completed successfully or failed.
- `ExecutorAdded` - executor (`execId`) has been spawned on a host (`host`). Remove it from the failed executors list if it was included, and `submitWaitingStages()`.
- `ExecutorLost`
- `TaskSetFailed`
- `ResubmitFailedStages`

FIXME**Caution**

- What is an approximate job (as in `DAGScheduler.runApproximateJob`)?
- statistics? `MapOutputStatistics` ?

JobCancelled and handleJobCancellation

`JobCancelled(jobId: Int)` event is posted to cancel a job if it is scheduled or still running. It triggers execution of `DAGScheduler.handleStageCancellation(stageId)` .

Note

It seems that although `SparkContext.cancelJob(jobId: Int)` calls `DAGScheduler.cancelJob`, no feature/code in Spark calls `SparkContext.cancelJob(jobId: Int)`. A dead code?

When `JobWaiter.cancel` is called, it calls `DAGScheduler.cancelJob`. You should see the following INFO message in the logs:

```
INFO Asked to cancel job [jobId]
```

It is a signal to the DAGScheduler to cancel the job.

Caution	FIXME
---------	-----------------------

ExecutorAdded and handleExecutorAdded

`ExecutorAdded(execId, host)` event triggers execution of `DAGScheduler.handleExecutorAdded(execId: String, host: String)`.

It checks `failedEpoch` for the executor id (using `execId`) and if it is found the following INFO message appears in the logs:

```
INFO Host added was in lost list earlier: [host]
```

The executor is removed from the list of failed nodes.

At the end, `DAGScheduler.submitWaitingStages()` is called.

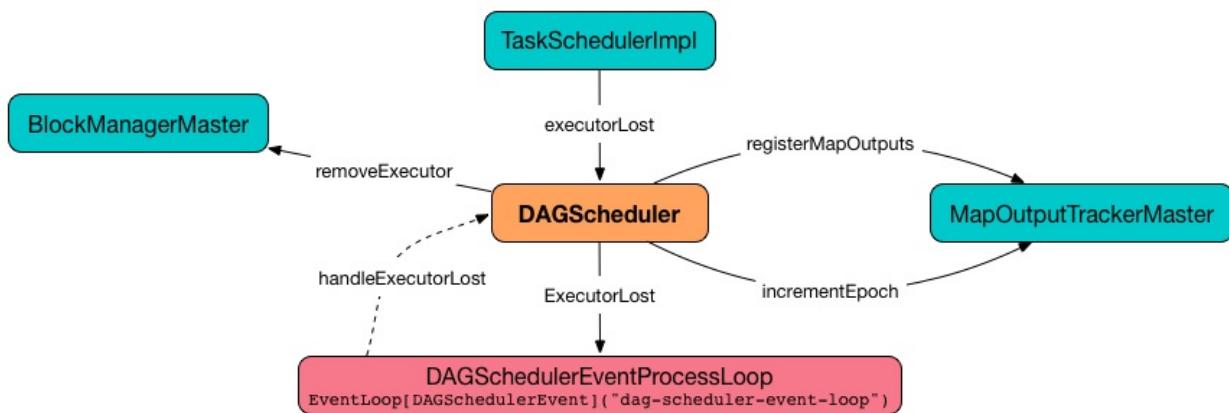
ExecutorLost and handleExecutorLost (with fetchFailed being false)

`ExecutorLost(execId)` event triggers execution of `DAGScheduler.handleExecutorLost(execId: String, fetchFailed: Boolean, maybeEpoch: Option[Long] = None)` with `fetchFailed` being `false`.

Note	<code>handleExecutorLost</code> recognizes two cases (by means of <code>fetchFailed</code>):
------	---

- fetch failures (`fetchFailed` is `true`) from executors that are indirectly assumed lost. See <>handleTaskCompletion-FetchFailed, FetchFailed case in handleTaskCompletion>.
- lost executors (`fetchFailed` is `false`) for executors that did not report being alive in a given timeframe

The current epoch number could be provided (as `maybeEpoch`) or it is calculated by requesting it from `MapOutputTrackerMaster` (using `MapOutputTrackerMaster.getEpoch`).

Figure 4. `DAGScheduler.handleExecutorLost`

Recurring `ExecutorLost` events merely lead to the following DEBUG message in the logs:

```
DEBUG Additional executor lost message for [execId] (epoch [currentEpoch])
```

If however the executor is not in the list of executor lost or the failed epoch number is smaller than the current one, the executor is added to `failedEpoch`.

The following INFO message appears in the logs:

```
INFO Executor lost: [execId] (epoch [currentEpoch])
```

`BlockManagerMaster.removeExecutor(execId)` is called.

If no external shuffle service is in use or the `ExecutorLost` event was for a map output fetch operation, all `ShuffleMapStages` (using `shuffleToMapStage`) are called (in order):

- `ShuffleMapStage.removeOutputsOnExecutor(execId)`
- `MapOutputTrackerMaster.registerMapOutputs(shuffleId, stage.outputLocInMapOutputTrackerFormat(), changeEpoch = true)`

For no `ShuffleMapStages` (in `shuffleToMapStage`), `MapOutputTrackerMaster.incrementEpoch` is called.

`cacheLocs` is cleared.

At the end, `DAGScheduler.submitWaitingStages()` is called.

StageCancelled and handleStageCancellation

`StageCancelled(stageId: Int)` event is posted to cancel a stage and all jobs associated with it. It triggers execution of `DAGScheduler.handleStageCancellation(stageId)`.

It is the result of executing `sparkContext.cancelStage(stageId: Int)` that is called from the web UI (controlled by `spark.ui.killEnabled`).

Caution

[FIXME](#) Image of the tab with kill

`DAGScheduler.handleStageCancellation(stageId)` checks whether the `stageId` stage exists and for each job associated with the stage, it calls `handleJobCancellation(jobId, s"because Stage [stageId] was cancelled")`.

Note

A stage knows what jobs it is part of using the internal set `jobIds`.

`def handleJobCancellation(jobId: Int, reason: String = "")` checks whether the job exists in `jobIdToStageIds` and if not, prints the following DEBUG to the logs:

```
DEBUG Trying to cancel unregistered job [jobId]
```

However, if the job exists, the job and all the stages that are only used by it (using `failJobAndIndependentStages` method).

For each running stage associated with the job (`jobIdToStageIds`), if there is only one job for the stage (`stageIdToStage`), [TaskScheduler.cancelTasks](#) is called, `outputCommitCoordinator.stageEnd(stage.id)`, and `SparkListenerStageCompleted` is posted. The stage is no longer a running one (removed from `runningStages`).

Caution

[FIXME](#) Image please with the call to TaskScheduler.

- `spark.job.interruptOnCancel` (default: `false`) - controls whether or not to interrupt a job on cancel.

In case [TaskScheduler.cancelTasks](#) completed successfully, [JobListener](#) is informed about job failure, `cleanupStateForJobAndIndependentStages` is called, and `SparkListenerJobEnd` posted.

Caution

[FIXME](#) `cleanupStateForJobAndIndependentStages` code review.

Caution

[FIXME](#) Where are `job.properties` assigned to a job?

```
"Job %d cancelled %s".format(jobId, reason)
```

If no stage exists for `stageId`, the following INFO message shows in the logs:

```
INFO No active jobs to kill for Stage [stageId]
```

At the end, `DAGScheduler.submitWaitingStages()` is called.

MapStageSubmitted and handleMapStageSubmitted

When a **MapStageSubmitted** event is posted, it triggers execution of `DAGScheduler.handleMapStageSubmitted` method.

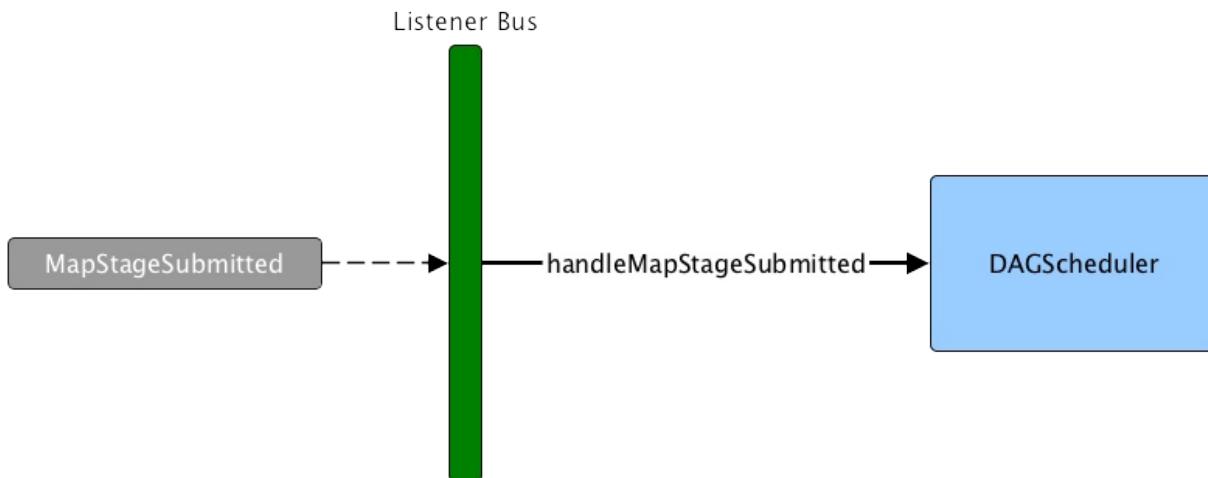


Figure 5. `DAGScheduler.handleMapStageSubmitted` handles `MapStageSubmitted` events. It is called with a job id (for a new job to be created), a [ShuffleDependency](#), and a [JobListener](#).

You should see the following INFOs in the logs:

```

Got map stage job %s (%s) with %d output partitions
Final stage: [finalStage] ([finalStage.name])
Parents of final stage: [finalStage.parents]
Missing parents: [list of stages]
  
```

A `SparkListenerJobStart` event is posted to [Listener Bus](#) (so other event listeners know about the event - not only `DAGScheduler`).

The execution procedure of `MapStageSubmitted` events is then exactly ([FIXME ?](#)) as for [JobSubmitted](#).

The difference between `handleMapStageSubmitted` and `handleJobSubmitted`:

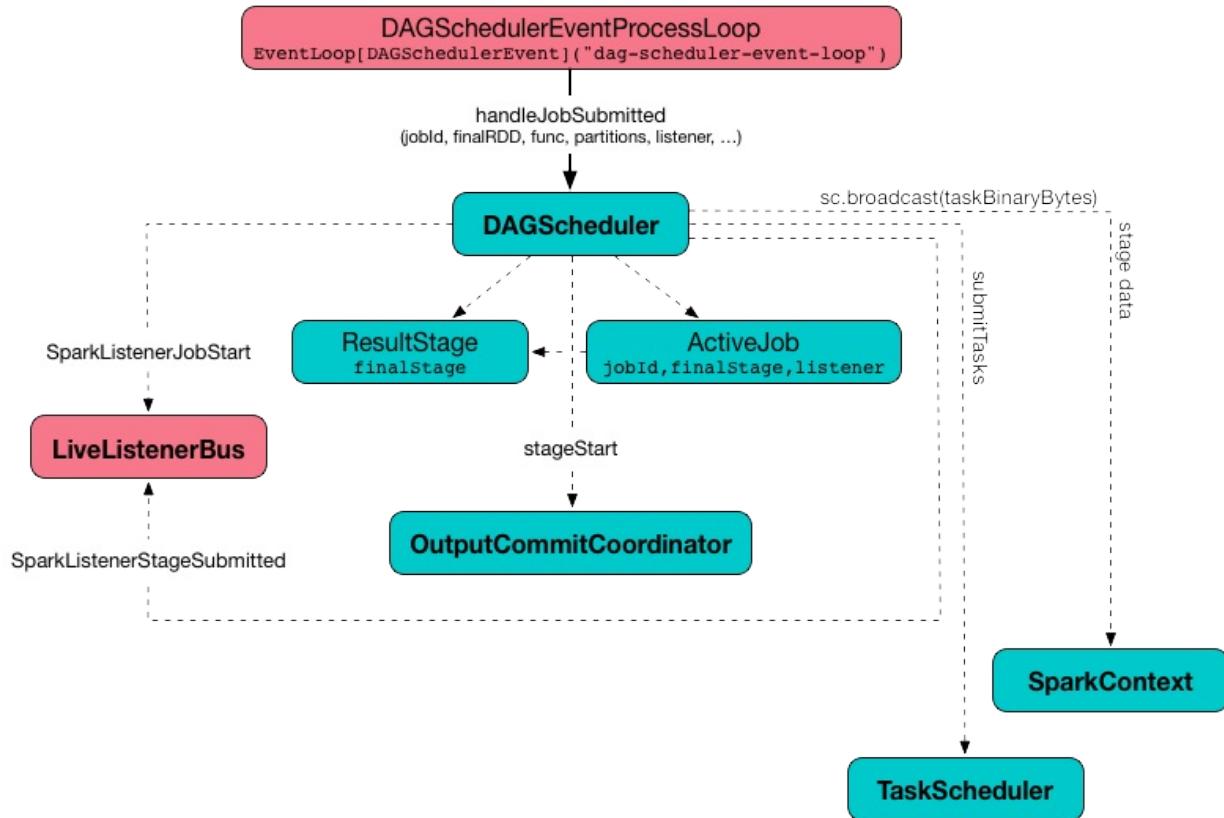
- `handleMapStageSubmitted` has `shuffleDependency` among the input parameters while `handleJobSubmitted` has `finalRDD`, `func`, and `partitions`.
 - `handleMapStageSubmitted` initializes `finalStage` as
`getShuffleMapStage(dependency, jobId)` while `handleJobSubmitted` as `finalStage = newResultStage(finalRDD, func, partitions, jobId, callSite)`
 - `handleMapStageSubmitted` INFO logs Got map stage job %s (%s) with %d output partitions with `dependency.rdd.partitions.length` while `handleJobSubmitted` does Got job %s (%s) with %d output partitions with `partitions.length`.
- Tip**
- **FIXME:** Could the above be cut to `ActiveJob.numPartitions` ?
 - `handleMapStageSubmitted` adds a new job with `finalStage.addActiveJob(job)` while `handleJobSubmitted` sets with `finalStage.setActiveJob(job)` .
 - `handleMapStageSubmitted` checks if the final stage has already finished, tells the listener and removes it using the code:

```
if (finalStage.isAvailable) {
    markMapStageJobAsFinished(job, mapOutputTracker.getStatistics(dependency))
}
```

JobSubmitted and handleJobSubmitted

When DAGScheduler receives **JobSubmitted** event it calls

`DAGScheduler.handleJobSubmitted` method.

Figure 6. `DAGScheduler.handleJobSubmitted`

`handleJobSubmitted` has access to the final RDD, the partitions to compute, and the `JobListener` for the job, i.e. `JobWaiter`.

It creates a new `ResultStage` (as `finalStage` on the picture) and instantiates `ActiveJob`.

Caution	FIXME review <code>newResultStage</code>
---------	--

You should see the following INFO messages in the logs:

```

INFO DAGScheduler: Got job [jobId] ([callSite.shortForm]) with [partitions.length] output
INFO DAGScheduler: Final stage: [finalStage] ([finalStage.name])
INFO DAGScheduler: Parents of final stage: [finalStage.parents]
INFO DAGScheduler: Missing parents: [getMissingParentStages(finalStage)]
  
```

Then, the `finalStage` stage is given the `ActiveJob` instance and some housekeeping is performed to track the job (using `jobIdToActiveJob` and `activeJobs`).

`SparkListenerJobStart` event is posted to [Listener Bus](#).

Caution	FIXME <code> jobIdToStageIds</code> and <code> stageIdToStage</code> - they're already computed. When? Where?
---------	--

When `DAGScheduler` executes a job it first submits the final stage (using `submitStage`).

Right before `handleJobSubmitted` finishes, `DAGScheduler.submitWaitingStages()` is called.

CompletionEvent and handleTaskCompletion

Using `CompletionEvent` event is the mechanism of DAGScheduler to be informed about task completions. It is handled by `handleTaskCompletion(event: CompletionEvent)`.

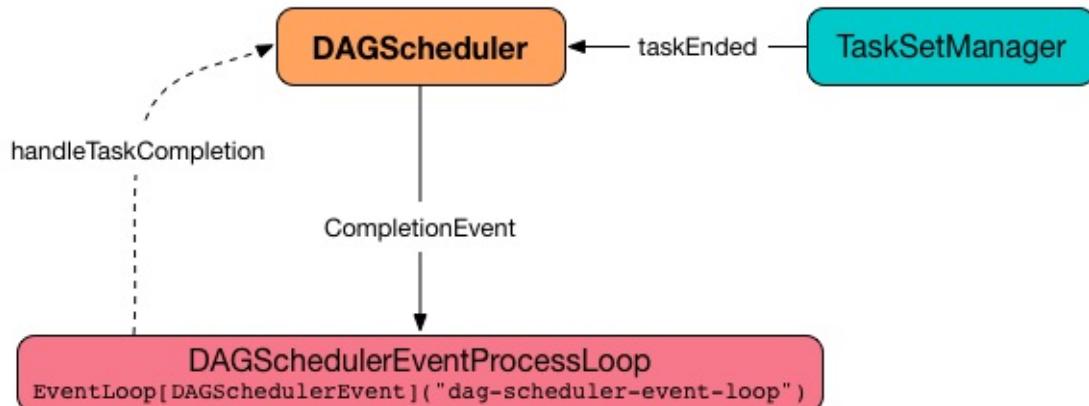


Figure 7. DAGScheduler and CompletionEvent

Note	<code>CompletionEvent</code> holds contextual information about the completed task.
------	---

The task knows about the stage it belongs to (using `Task.stageId`), the partition it works on (using `Task.partitionId`), and the stage attempt (using `Task.stageAttemptId`).

`OutputCommitCoordinator.taskCompleted` is called.

If the reason for task completion is not `Success`, `SparkListenerTaskEnd` is posted on [Listener Bus](#). The only difference with [TaskEndReason: Success](#) is how the stage attempt id is calculated. Here, it is `Task.stageAttemptId` (not `Stage.latestInfo.attemptId`).

Caution	<code>FIXME</code> What is the difference between stage attempt ids?
---------	--

If the stage the task belongs to has been cancelled, `stageIdToStage` should not contain it, and the method quits.

The main processing begins now depending on `TaskEndReason` - the reason for task completion (using `event.reason`). The method skips processing `TaskEndReasons` : `TaskCommitDenied`, `ExceptionFailure`, `TaskResultLost`, `ExecutorLostFailure`, `TaskKilled`, and `UnknownReason`, i.e. it does nothing.

TaskEndReason: Success

`SparkListenerTaskEnd` is posted on [Listener Bus](#).

The partition the task worked on is removed from `pendingPartitions` of the stage.

The processing splits per task type - `ResultTask` or `ShuffleMapTask` - and `submitWaitingStages()` is called.

ResultTask

For `ResultTask`, the stage is `ResultStage`. If there is no job active for the stage (using `resultStage.activeJob`), the following INFO message appears in the logs:

```
INFO Ignoring result from [task] because its job has finished
```

Otherwise, check whether the task is marked as running for the job (using `job.finished`) and proceed. The method skips execution when the task has already been marked as completed in the job.

Caution	<code>FIXME</code> When could a task that has just finished be ignored, i.e. the job has already marked <code>finished</code> ? Could it be for stragglers?
---------	--

`updateAccumulators(event)` is called.

The partition is marked as `finished` (using `job.finished`) and the number of partitions calculated increased (using `job.numFinished`).

If the whole job has finished (when `job.numFinished == job.numPartitions`), then:

- `markStageAsFinished` is called
- `cleanupStateForJobAndIndependentStages(job)`
- `SparkListenerJobEnd` is posted on [Listener Bus](#) with `JobSucceeded`

The `JobListener` of the job (using `job.listener`) is informed about the task completion (using `job.listener.taskSucceeded(rt.outputId, event.result)`). If the step fails, i.e. throws an exception, the `JobListener` is informed about it (using `job.listener.jobFailed(new SparkDriverExecutionException(e))`).

Caution	<code>FIXME</code> When would <code>job.listener.taskSucceeded</code> throw an exception? How?
---------	---

ShuffleMapTask

For `ShuffleMapTask`, the stage is `ShuffleMapStage`.

`updateAccumulators(event)` is called.

`event.result` is `MapStatus` that knows the executor id where the task has finished (using `status.location.executorId`).

You should see the following DEBUG message in the logs:

```
DEBUG ShuffleMapTask finished on [execId]
```

If `failedEpoch` contains the executor and the epoch of the `ShuffleMapTask` is not greater than that in `failedEpoch`, you should see the following INFO message in the logs:

```
INFO Ignoring possibly bogus [task] completion from executor [executorId]
```

Otherwise, `shuffleStage.addOutputLoc(smt.partitionId, status)` is called.

The method does more processing only if the internal `runningStages` contains the `ShuffleMapStage` with no more pending partitions to compute (using `shuffleStage.pendingPartitions`).

`markStageAsFinished(shuffleStage)` is called.

The following INFO logs appear in the logs:

```
INFO looking for newly runnable stages
INFO running: [runningStages]
INFO waiting: [waitingStages]
INFO failed: [failedStages]
```

`mapOutputTracker.registerMapOutputs` with `changeEpoch` is called.

`cacheLocs` is cleared.

If the map stage is ready, i.e. all partitions have shuffle outputs, map-stage jobs waiting on this stage (using `shuffleStage.mapStageJobs`) are marked as finished.

`MapOutputTrackerMaster.getStatistics(shuffleStage.shuffleDep)` is called and every map-stage job is `markMapStageJobAsFinished(job, stats)`.

Otherwise, if the map stage is *not* ready, the following INFO message appears in the logs:

```
INFO Resubmitting [shuffleStage] ([shuffleStage.name]) because some of its tasks had fail
```

`submitStage(shuffleStage)` is called.

Caution	FIXME All "...is called" above should be rephrased to use links to appropriate sections.
---------	---

TaskEndReason: Resubmitted

For `Resubmitted` case, you should see the following INFO message in the logs:

```
INFO Resubmitted [task], so marking it as still running
```

The task (by `task.partitionId`) is added to the collection of pending partitions of the stage (using `stage.pendingPartitions`).

Tip

A stage knows how many partitions are yet to be calculated. A task knows about the partition id for which it was launched.

TaskEndReason: FetchFailed

`FetchFailed(bmAddress, shuffleId, mapId, reduceId, failureMessage)` comes with `BlockManagerId` (as `bmAddress`) and the other self-explanatory values.

Note

A task knows about the id of the stage it belongs to.

When `FetchFailed` happens, `stageIdToStage` is used to access the failed stage (using `task.stageId` and the `task` is available in `event` in `handleTaskCompletion(event: CompletionEvent)`). `shuffleToMapStage` is used to access the map stage (using `shuffleId`).

If `failedStage.latestInfo.attemptId != task.stageAttemptId`, you should see the following INFO in the logs:

```
INFO Ignoring fetch failure from [task] as it's from [failedStage] attempt [task.stageAtt
```

Caution

FIXME What does `failedStage.latestInfo.attemptId != task.stageAttemptId` mean?

And the case finishes. Otherwise, the case continues.

If the failed stage is in `runningStages`, the following INFO message shows in the logs:

```
INFO Marking [failedStage] ([failedStage.name]) as failed due to a fetch failure from [ma
```

`markStageAsFinished(failedStage, Some(failureMessage))` is called.

Caution

FIXME What does `markStageAsFinished` do?

If the failed stage is not in `runningStages`, the following DEBUG message shows in the logs:

```
DEBUG Received fetch failure from [task], but its from [failedStage] which is no longer r
```

When `disallowStageRetryForTest` is set, `abortStage(failedStage, "Fetch failure will not retry stage due to testing config", None)` is called.

Caution	FIXME Describe <code>disallowStageRetryForTest</code> and <code>abortStage</code> .
---------	---

If the number of fetch failed attempts for the stage exceeds the allowed number (using `Stage.failedOnFetchAndShouldAbort`), the following method is called:

```
abortStage(failedStage, s"$failedStage (${failedStage.name}) has failed the maximum allow
```

If there are no failed stages reported (`failedStages` is empty), the following INFO shows in the logs:

```
INFO Resubmitting [mapStage] ([mapStage.name]) and [failedStage] ([failedStage.name]) due
```

And the following code is executed:

```
messageScheduler.schedule(
  new Runnable {
    override def run(): Unit = eventProcessLoop.post(ResubmitFailedStages)
  }, DAGScheduler.RESUBMIT_TIMEOUT, TimeUnit.MILLISECONDS)
```

Caution	FIXME What does the above code do?
---------	--

For all the cases, the failed stage and map stages are both added to `failedStages` set.

If `mapId` (in the `FetchFailed` object for the case) is provided, the map stage output is cleaned up (as it is broken) using `mapStage.removeOutputLoc(mapId, bmAddress)` and `MapOutputTrackerMaster.unregisterMapOutput(shuffleId, mapId, bmAddress)` methods.

Caution	FIXME What does <code>mapStage.removeOutputLoc</code> do?
---------	---

If `bmAddress` (in the `FetchFailed` object for the case) is provided, `handleExecutorLost(bmAddress.executorId, fetchFailed = true, Some(task.epoch))` is called. See [ExecutorLost](#) and [handleExecutorLost](#) (with `fetchFailed` being false).

Caution	FIXME What does <code>handleExecutorLost</code> do?
---------	---

Submit Waiting Stages (using `submitWaitingStages`)

`DAGScheduler.submitWaitingStages` method checks for waiting or failed stages that could now be eligible for submission.

The following `TRACE` messages show in the logs when the method is called:

```
TRACE DAGScheduler: Checking for newly runnable parent stages
TRACE DAGScheduler: running: [runningStages]
TRACE DAGScheduler: waiting: [waitingStages]
TRACE DAGScheduler: failed: [failedStages]
```

The method clears the internal `waitingStages` set with stages that wait for their parent stages to finish.

It goes over the waiting stages sorted by job ids in increasing order and calls `submitStage` method.

submitStage - Stage Submission

Caution	FIXME
---------	-----------------------

`DAGScheduler.submitStage(stage: Stage)` is called when `stage` is ready for submission.

It recursively submits any missing parents of the stage.

There has to be an `ActiveJob` instance for the stage to proceed. Otherwise the stage and all the dependent jobs are aborted (using `abortStage`) with the message:

```
Job aborted due to stage failure: No active job for stage [stage.id]
```

For a stage with `ActiveJob` available, the following `DEBUG` message show up in the logs:

```
DEBUG DAGScheduler: submitStage([stage])
```

Only when the stage is not in `waiting` (`waitingStages`), `running` (`runningStages`) or `failed` states can this stage be processed.

A list of missing parent stages of the stage is calculated (see [Calculating Missing Parent Stages](#)) and the following `DEBUG` message shows up in the logs:

```
DEBUG DAGScheduler: missing: [missing]
```

When the stage has no parent stages missing, it is submitted and the `INFO` message shows up in the logs:

```
INFO DAGScheduler: Submitting [stage] ([stage.rdd]), which has no missing parents
```

And `submitMissingTasks` is called. That finishes the stage submission.

If however there are missing parent stages for the stage, all stages are processed recursively (using `submitStage`), and the stage is added to `waitingStages` set.

Calculating Missing Parent Map Stages

`DAGScheduler.getMissingParentStages(stage: Stage)` calculates missing parent map stages for a given `stage`.

It starts with the stage's target RDD (as `stage.rdd`). If there are `uncached partitions`, it traverses the dependencies of the RDD (as `RDD.dependencies`) that can be the instances of `ShuffleDependency` or `NarrowDependency`.

For each `ShuffleDependency`, the method searches for the corresponding `ShuffleMapStage` (using `getShuffleMapStage`) and if unavailable, the method adds it to a set of missing (map) stages.

Caution	FIXME Review <code>getShuffleMapStage</code>
---------	--

Caution	FIXME ...IMAGE with <code>ShuffleDependencies</code> queried
---------	--

It continues traversing the chain for each `NarrowDependency` (using `Dependency.rdd`).

Fault recovery - stage attempts

A single stage can be re-executed in multiple **attempts** due to fault recovery. The number of attempts is configured ([FIXME](#)).

If `TaskScheduler` reports that a task failed because a map output file from a previous stage was lost, the `DAGScheduler` resubmits that lost stage. This is detected through a `CompletionEvent` with `FetchFailed`, or an `ExecutorLost` event. `DAGScheduler` will wait a small amount of time to see whether other nodes or tasks fail, then resubmit `TaskSets` for any lost stage(s) that compute the missing tasks.

Please note that tasks from the old attempts of a stage could still be running.

A stage object tracks multiple `stageInfo` objects to pass to Spark listeners or the web UI.

The latest `stageInfo` for the most recent attempt for a stage is accessible through `latestInfo`.

Cache Tracking

DAGScheduler tracks which RDDs are cached to avoid recomputing them and likewise remembers which shuffle map stages have already produced output files to avoid redoing the map side of a shuffle.

DAGScheduler is only interested in cache location coordinates, i.e. host and executor id, per partition of an RDD.

Caution

FIXME: A diagram, please

If [the storage level of an RDD is NONE](#), there is no caching and hence no partition cache locations are available. In such cases, whenever asked, DAGScheduler returns a collection with empty-location elements for each partition. The empty-location elements are to mark **uncached partitions**.

Otherwise, a collection of `RDDBlockId` instances for each partition is created and [BlockManagerMaster](#) is asked for locations (using `BlockManagerMaster.getLocations`). The result is then mapped to a collection of `TaskLocation` for host and executor id.

Preferred Locations

DAGScheduler computes where to run each task in a stage based on [the preferred locations of its underlying RDDs](#), or [the location of cached or shuffle data](#).

Adaptive Query Planning

See [SPARK-9850 Adaptive execution in Spark](#) for the design document. The work is currently in progress.

[DAGScheduler.submitMapStage](#) method is used for adaptive query planning, to run map stages and look at statistics about their outputs before submitting downstream stages.

ScheduledExecutorService daemon services

DAGScheduler uses the following ScheduledThreadPoolExecutors (with the policy of removing cancelled tasks from a work queue at time of cancellation):

- `dag-scheduler-message` - a daemon thread pool using `j.u.c.ScheduledThreadPoolExecutor` with core pool size `1`. It is used to post `ResubmitFailedStages` when `FetchFailed` is reported.

They are created using `ThreadUtils.newDaemonSingleThreadScheduledExecutor` method that uses Guava DSL to instantiate a ThreadFactory.

submitMissingTasks for Stage and Job

`DAGScheduler.submitMissingTasks(stage: Stage, jobId: Int)` is called when the parent stages of the current `stage` have already been finished and it is now possible to run tasks for it.

In the logs, you should see the following DEBUG message:

```
DEBUG DAGScheduler: submitMissingTasks([stage])
```

`pendingPartitions` internal field of the stage is cleared (it is later filled out with the partitions to run tasks for).

The stage is asked for partitions to compute (see [findMissingPartitions](#) in Stages).

The method adds the stage to `runningStages`.

The stage is told to be started to [OutputCommitCoordinator](#) (using
`outputCommitCoordinator.stageStart`)

Caution

[FIXME](#) Review `outputCommitCoordinator.stageStart`

The mapping between task ids and task preferred locations is computed (see [getPreferredLocs - Computing Preferred Locations for Tasks and Partitions](#)).

A new stage attempt is created (using `Stage.makeNewStageAttempt`).

`SparkListenerStageSubmitted` is posted.

Caution

[FIXME](#) `SparkEnv.get.closureSerializer.newInstance()`

The stage is serialized and broadcast to workers using [SparkContext.broadcast](#) method, i.e. it is `Serializer.serialize` to calculate `taskBinaryBytes` - an array of bytes of `(rdd, func)` for `ResultStage` and `(rdd, shuffleDep)` for `ShuffleMapStage`.

Caution

[FIXME](#) Review `taskBinaryBytes`.

When serializing the stage fails, the stage is removed from the internal `runningStages` set, `abortStage` is called and the method stops.

Caution

[FIXME](#) Review `abortStage`.

At this point in time, the stage is on workers.

For each partition to compute for the stage, a collection of [ShuffleMapTask](#) for `ShuffleMapStage` or `ResultTask` for `ResultStage` is created.

Caution	FIXME Image with creating tasks for partitions in the stage.
---------	--

If there are tasks to launch (there are missing partitions in the stage), the following INFO and DEBUG messages are in the logs:

```
INFO DAGScheduler: Submitting [tasks.size] missing tasks from [stage] ([stage.rdd])
DEBUG DAGScheduler: New pending partitions: [stage.pendingPartitions]
```

All tasks in the collection become a [TaskSet](#) for [TaskScheduler.submitTasks](#).

In case of no tasks to be submitted for a stage, a DEBUG message shows up in the logs.

For [ShuffleMapStage](#):

```
DEBUG DAGScheduler: Stage [stage] is actually done; (available: ${stage.isAvailable}, avai
```

For [ResultStage](#):

```
DEBUG DAGScheduler: Stage ${stage} is actually done; (partitions: ${stage.numPartitions})
```

getPreferredLocs - Computing Preferred Locations for Tasks and Partitions

Caution	FIXME Review + why does the method return a sequence of TaskLocations?
---------	--

Note	Task ids correspond to partition ids.
------	---------------------------------------

Stopping

When a DAGScheduler stops (via `stop()`), it stops the internal `dag-scheduler-message` thread pool, [dag-scheduler-event-loop](#), and [TaskScheduler](#).

Metrics

Spark's DAGScheduler uses [Spark Metrics System](#) (via `DAGSchedulerSource`) to report metrics about internal status.

The name of the source is **DAGScheduler**.

It emits the following numbers:

- **stage.failedStages** - the number of failed stages
- **stage.runningStages** - the number of running stages
- **stage.waitingStages** - the number of waiting stages
- **job.allJobs** - the number of all jobs
- **job.activeJobs** - the number of active jobs

Settings

- `spark.test.noStageRetry` (default: `false`) - if enabled, `FetchFailed` will not cause stage retries, in order to surface the problem. Used for testing.

Jobs

A **job** (aka *action job* or *active job*) is a top-level work item (computation) submitted to [DAGScheduler](#) to [compute the result of an action](#).



Figure 1. RDD actions submit jobs to DAGScheduler

Computing a job is equivalent to computing the partitions of the RDD the action has been executed upon. The number of partitions in a job depends on the type of a stage - [ResultStage](#) or [ShuffleMapStage](#).

A job starts with a single target RDD, but can ultimately include other RDDs that are all part of [the target RDD's lineage graph](#).

The parent stages are the instances of [ShuffleMapStage](#).

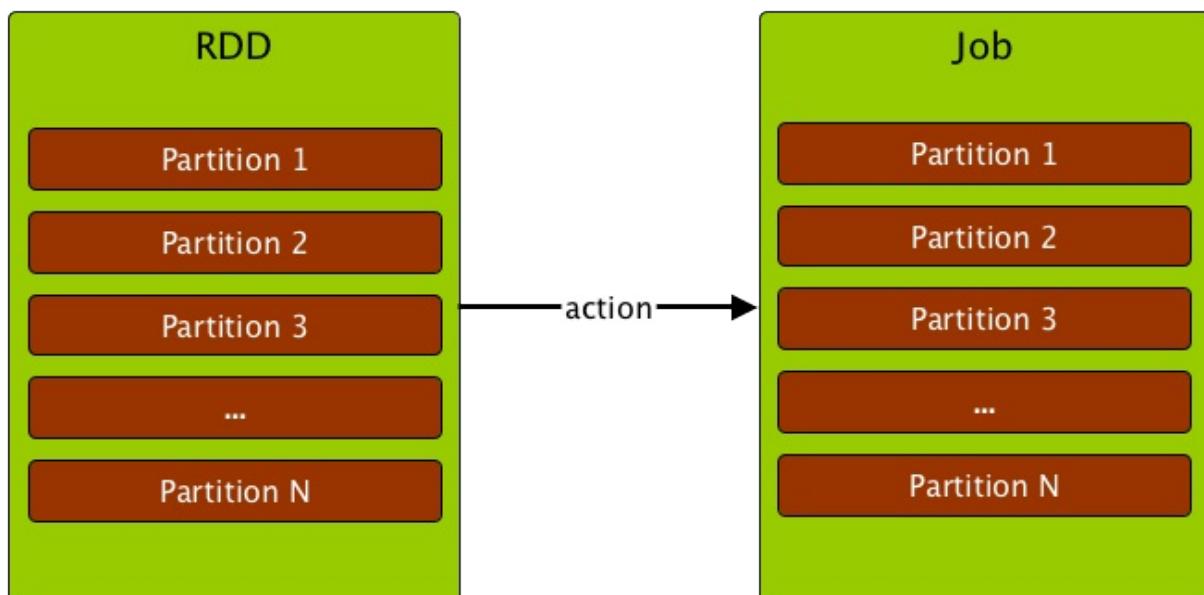


Figure 2. Computing a job is computing the partitions of an RDD

Note	Note that not all partitions have always to be computed for ResultStages for actions like <code>first()</code> and <code>lookup()</code> .
------	--

Internally, a job is represented by an instance of [private\[spark\]](#) class [org.apache.spark.scheduler.ActiveJob](#).

Caution	FIXME <ul style="list-style-type: none"> Where are instances of ActiveJob used?
---------	---

A job can be one of two logical types (that are only distinguished by an internal `finalStage` field of `ActiveJob`):

- **Map-stage job** that computes the map output files for a [ShuffleMapStage](#) (for `submitMapStage`) before any downstream stages are submitted.

It is also used for [adaptive query planning](#), to look at map output statistics before submitting later stages.

- **Result job** that computes a [ResultStage](#) to execute an action.

Jobs track how many partitions have already been computed (using `finished` array of `Boolean` elements).

Stages

Introduction

A **stage** is a set of parallel tasks, one per partition of an RDD, that compute partial results of a function executed as part of a Spark job.

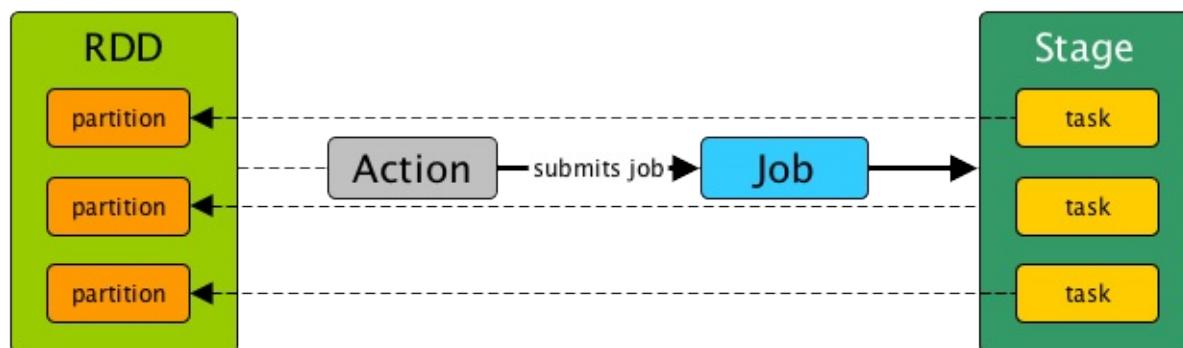


Figure 1. Stage, tasks and submitting a job

A stage is uniquely identified by `id`. When a stage is created, `DAGScheduler` increments internal counter `nextStageId` to track the number of `stage` submissions.

A stage can only work on the partitions of a single RDD (identified by `rdd`), but can be associated with many other dependent parent stages (via internal field `parents`), with the boundary of a stage marked by shuffle dependencies.

Submitting a stage can therefore trigger execution of a series of dependent parent stages (refer to [RDDs, Job Execution, Stages, and Partitions](#)).

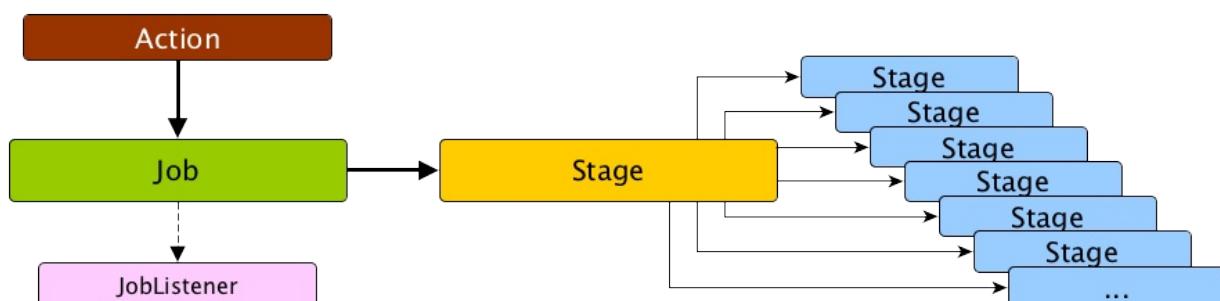


Figure 2. Submitting a job triggers execution of the stage and its parent stages

Finally, every stage has a `firstJobId` that is the id of the job that submitted the stage.

There are two types of stages:

- **ShuffleMapStage** is an intermediate stage (in the execution DAG) that produces data for other stage(s). It writes **map output files** for a shuffle. It can also be the final stage in a job in [adaptive query planning](#).

- [ResultStage](#) is the final stage that executes a [Spark action](#) in a user program by running a function on an RDD.

When a job is submitted, a new stage is created with the parent ShuffleMapStages linked — they can be created from scratch or linked to, i.e. shared, if other jobs use them already.

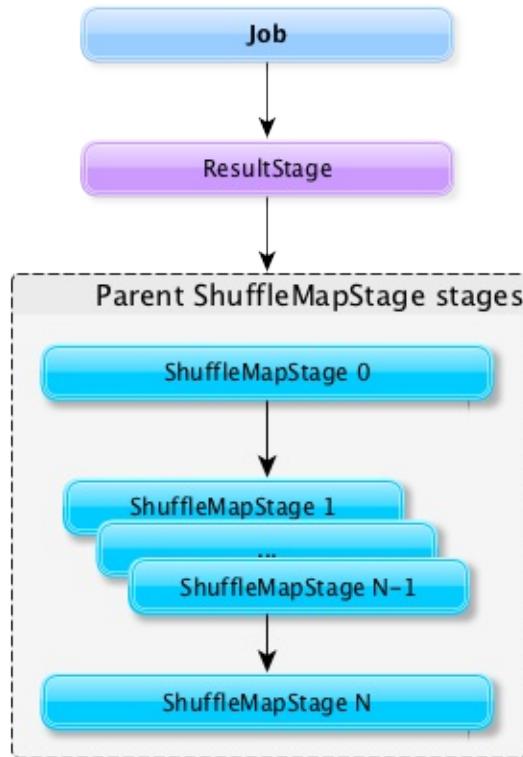


Figure 3. DAGScheduler and Stages for a job

A stage knows about the jobs it belongs to (using the internal field `jobIds`).

DAGScheduler splits up a job into a collection of stages. Each stage contains a sequence of [narrow transformations](#) that can be completed without [shuffling](#) the entire data set, separated at **shuffle boundaries**, i.e. where shuffle occurs. Stages are thus a result of breaking the RDD graph at shuffle boundaries.

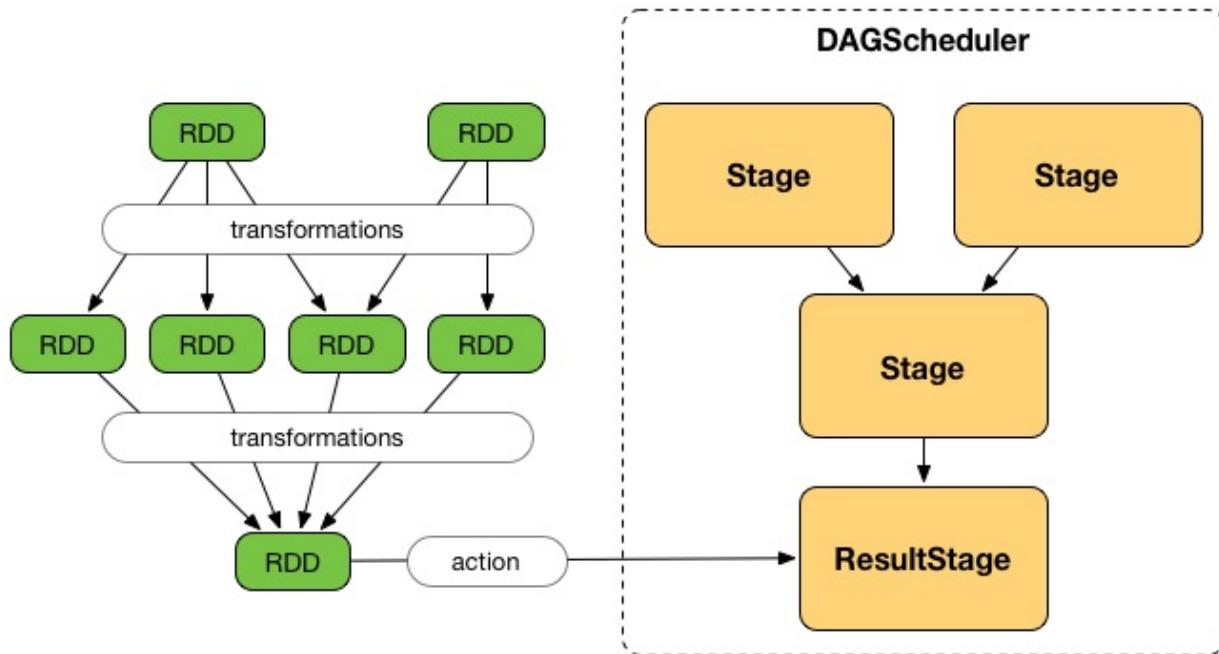


Figure 4. Graph of Stages

Shuffle boundaries introduce a barrier where stages/tasks must wait for the previous stage to finish before they fetch map outputs.

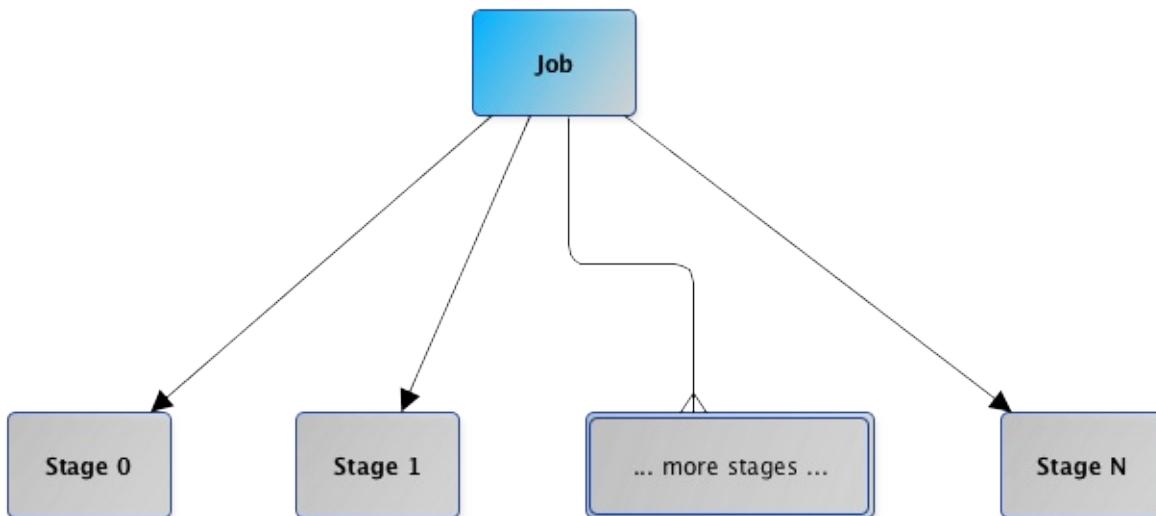


Figure 5. DAGScheduler splits a job into stages

RDD operations with [narrow dependencies](#), like `map()` and `filter()`, are pipelined together into one set of tasks in each stage, but operations with shuffle dependencies require multiple stages, i.e. one to write a set of map output files, and another to read those files after a barrier.

In the end, every stage will have only shuffle dependencies on other stages, and may compute multiple operations inside it. The actual pipelining of these operations happens in the `RDD.compute()` functions of various RDDs, e.g. `MappedRDD`, `FilteredRDD`, etc.

At some point of time in a stage's life, every partition of the stage gets transformed into a task - `ShuffleMapTask` or `ResultTask` for `ShuffleMapStage` and `ResultStage`, respectively.

Partitions are computed in jobs, and result stages may not always need to compute all partitions in their target RDD, e.g. for actions like `first()` and `lookup()`.

`DAGScheduler` prints the following INFO message when there are tasks to submit:

```
INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 36 (ShuffledRDD[86] at red
```



There is also the following DEBUG message with pending partitions:

```
DEBUG DAGScheduler: New pending partitions: Set(0)
```

Tasks are later submitted to [Task Scheduler](#) (via `taskScheduler.submitTasks`).

When no tasks in a stage can be submitted, the following DEBUG message shows in the logs:

```
FIXME
```

numTasks - where and what

Caution

[FIXME](#) Why do stages have `numTasks`? Where is this used? How does this correspond to the number of partitions in a RDD?

ShuffleMapStage

A **ShuffleMapStage** (aka **shuffle map stage**, or simply **map stage**) is an intermediate stage in the execution DAG that produces data for [shuffle operation](#). It is an input for the other following stages in the DAG of stages. That is why it is also called a **shuffle dependency's map side** (see [ShuffleDependency](#))

ShuffleMapStages usually contain multiple pipelined operations, e.g. `map` and `filter`, before shuffle operation.

Caution

[FIXME](#): Show the example and the logs + figures

A single ShuffleMapStage can be part of many jobs — refer to the section [ShuffleMapStage sharing](#).

A `ShuffleMapStage` is a stage with a [ShuffleDependency](#) - the shuffle that it is part of and `outputLocs` and `numAvailableOutputs` track how many map outputs are ready.

Note

`ShuffleMapStages` can also be submitted independently as jobs with `DAGScheduler.submitMapStage` for [Adaptive Query Planning](#).

When executed, `ShuffleMapStages` save **map output files** that can later be fetched by reduce tasks.

Caution

[FIXME](#) Figure with `ShuffleMapStages` saving files

The number of the partitions of an RDD is exactly the number of the tasks in a `ShuffleMapStage`.

The output locations (`outputLocs`) of a `ShuffleMapStage` are the same as used by its [ShuffleDependency](#). Output locations can be missing, i.e. partitions have not been cached or are lost.

`ShuffleMapStages` are registered to `DAGScheduler` that tracks the mapping of shuffles (by their ids from `SparkContext`) to corresponding `ShuffleMapStages` that compute them, stored in `shuffleToMapStage`.

A new `ShuffleMapStage` is created from an input [ShuffleDependency](#) and a job's id (in `DAGScheduler#newOrUsedShuffleStage`).

[FIXME](#): Where's `shuffleToMapStage` used?

- `getShuffleMapStage` - see [Stage sharing](#)
- `getAncestorShuffleDependencies`
- `cleanupStateForJobAndIndependentStages`
- `handleExecutorLost`

When there is no `ShuffleMapStage` for a shuffle id (of a `ShuffleDependency`), one is created with the ancestor shuffle dependencies of the RDD (of a `ShuffleDependency`) that are registered to [MapOutputTrackerMaster](#).

[FIXME](#) Where is `shuffleMapStage` used?

- `newShuffleMapStage` - the proper way to create shuffle map stages (with the additional setup steps)
- [MapStageSubmitted](#)
- `getShuffleMapStage` - see [Stage sharing](#)

	FIXME
Caution	<ul style="list-style-type: none"> • What's <code>ShuffleMapStage.outputLocs</code> and <code>MapStatus</code> ? • <code>newShuffleMapStage</code>

ResultStage

A **ResultStage** is the final stage in running any job that applies a function on some partitions of the target RDD to compute the result of an action.

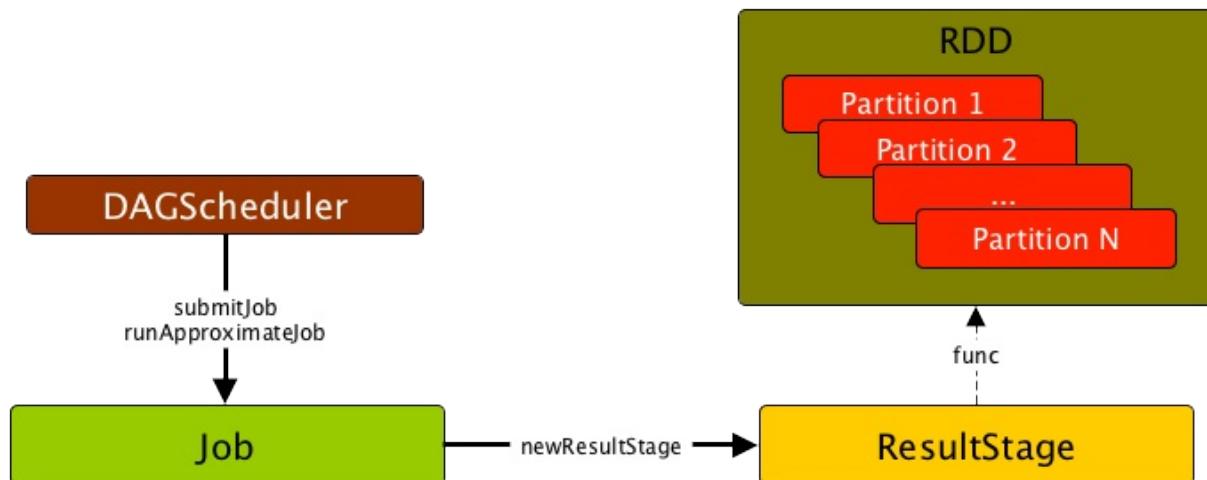


Figure 6. Job creates ResultStage as the first stage

The partitions are given as a collection of partition ids (`partitions`) and the function `func`:
`(TaskContext, Iterator[_]) ⇒ _` .

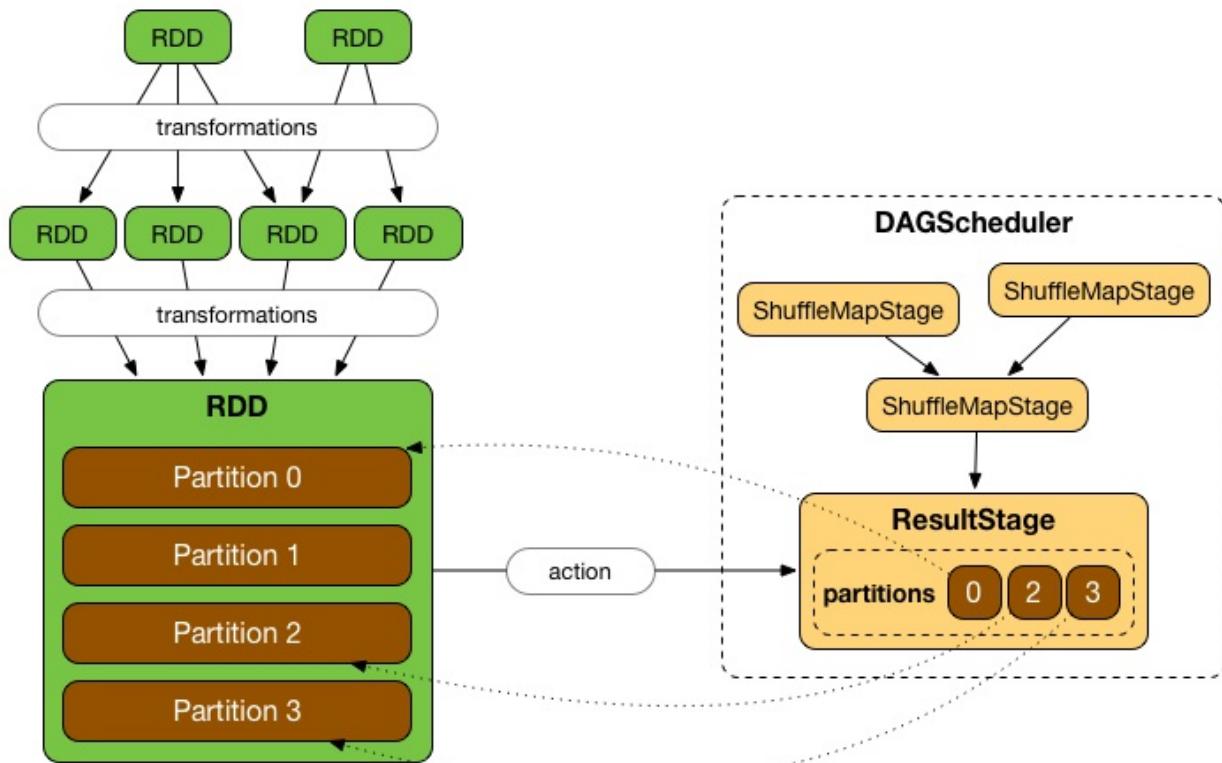


Figure 7. ResultStage and partitions

ShuffleMapStage Sharing

ShuffleMapStages can be shared across multiple jobs, if these jobs reuse the same RDDs.

When a ShuffleMapStage is submitted to DAGScheduler to execute, `getShuffleMapStage` is called (as part of `handleMapStageSubmitted` while `newResultStage` - note the `new` part - for `handleJobSubmitted`).

```
scala> val rdd = sc.parallelize(0 to 5).map((_,1)).sortByKey() (1)
scala> rdd.count  (2)
scala> rdd.count  (3)
```

1. Shuffle at `sortByKey()`
2. Submits a job with two stages with two being executed
3. Intentionally repeat the last action that submits a new job with two stages with one being shared as already-being-computed

Details for Job 3

Status: SUCCEEDED
Completed Stages: 1
Skipped Stages: 1

Event Timeline
DAG Visualization

Completed Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
6	count at <console>:27	+details 2015/11/08 14:43:06	11 ms	7/7			1573.0 B	

Skipped Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	map at <console>:24	+details Unknown	Unknown	0/8				

Figure 8. Skipped Stages are already-computed ShuffleMapStages

Stage.findMissingPartitions

`Stage.findMissingPartitions()` calculates the ids of the missing partitions, i.e. partitions for which the `ActiveJob` knows they are not finished (and so they are missing).

A `ResultStage` stage knows it by querying the active job about partition ids (`numPartitions`) that are not finished (using `ActiveJob.finished` array of booleans).

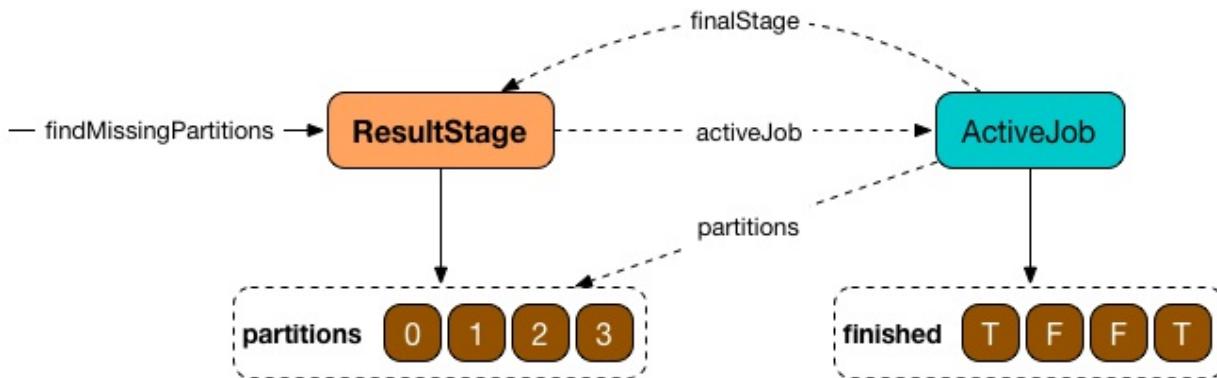


Figure 9. ResultStage.findMissingPartitions and ActiveJob

In the above figure, partitions 1 and 2 are not finished (`F` is false while `T` is true).

Stage.failedOnFetchAndShouldAbort

`Stage.failedOnFetchAndShouldAbort(stageAttemptId: Int): Boolean` checks whether the number of fetch failed attempts (using `fetchFailedAttemptIds`) exceeds the number of consecutive failures allowed for a given stage (that should then be aborted)

Note	The number of consecutive failures for a stage is not configurable.
------	---

Task Schedulers

A **Task Scheduler** schedules [tasks](#) for a [single Spark application](#) according to [scheduling mode](#) (aka [order task policy](#)).

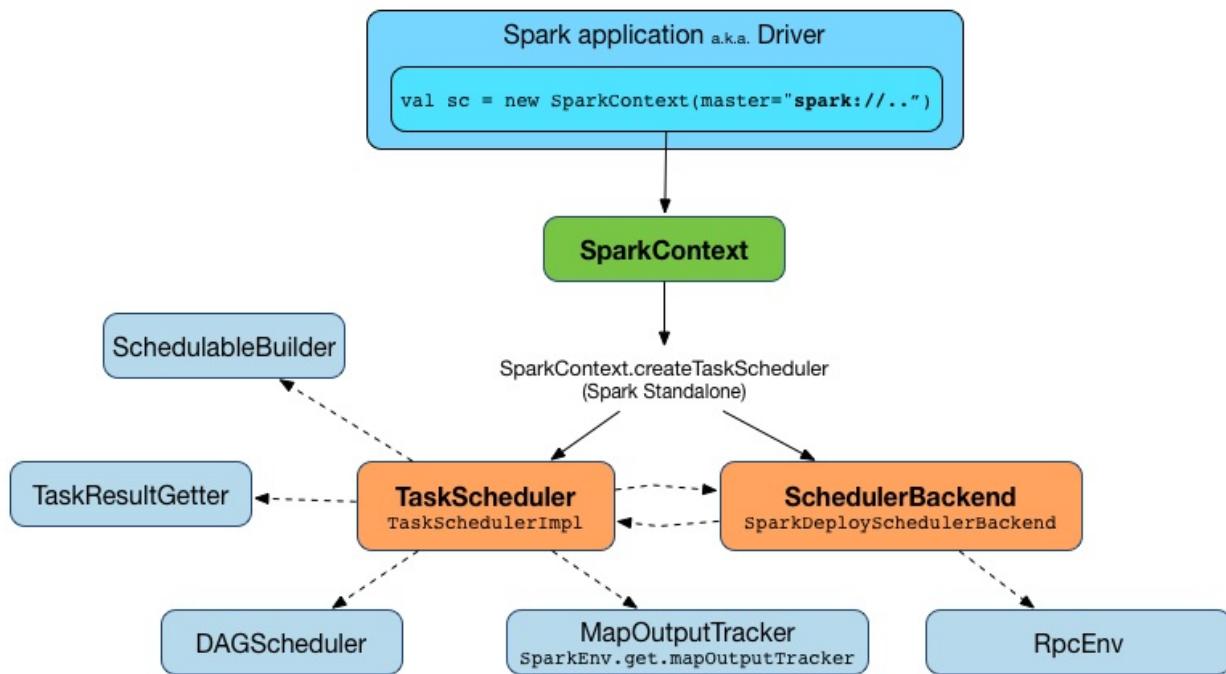


Figure 1. TaskScheduler works for a single SparkContext

A TaskScheduler gets sets of tasks (as [TaskSets](#)) submitted to it from the [DAGScheduler](#) for each stage, and is responsible for sending the tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers.

TaskScheduler Contract

Note	org.apache.spark.scheduler.TaskScheduler is a <code>private[spark]</code> Scala trait.
------	--

Every task scheduler has to offer the following services:

- Return `Pool` (using `rootPool`)
- Return `SchedulingMode` (using `schedulingMode`) that determines policy to order tasks.
- Can be started (using `start()`) and stopped (using `stop()`)
- (optionally) `postStartHook()` if needed for additional post-start initialization. It does nothing by default. It is called at the very end of [SparkContext's initialization](#).
- `submitTasks(taskSet: TaskSet)`
- `cancelTasks(stageId: Int, interruptThread: Boolean)` to cancel all tasks in a stage.

- Setting a custom [DAGScheduler](#) (using `setDAGScheduler(dagScheduler: DAGScheduler)`).
- Return the default level of parallelism (using `defaultParallelism()`)
- `executorHeartbeatReceived`
- (optionally) Return an application id for the current job (using `applicationId()`). It returns `spark-application-[System.currentTimeMillis]` by default.
- Handle executor lost events (using `executorLost(executorId: String, reason: ExecutorLossReason)`)
- Return an application attempt ID associated with the job (using `applicationAttemptId`)

Caution

[FIXME](#) Have an exercise to create a SchedulerBackend.

Available Implementations

Spark comes with the following task schedulers:

- [TaskSchedulerImpl](#)
- [YarnScheduler](#) for [Spark on YARN](#)
- [YarnClusterScheduler](#) for [Spark on YARN](#)

A TaskScheduler emits events to the DAGScheduler.

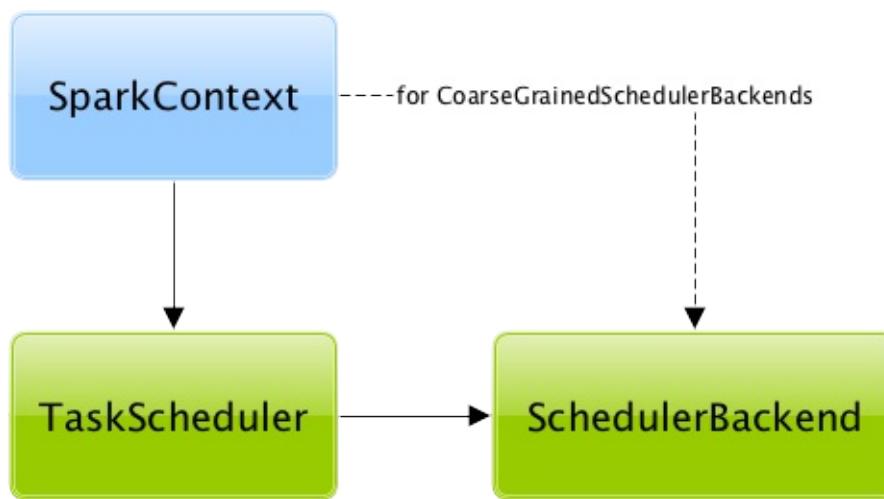


Figure 2. TaskScheduler uses SchedulerBackend for different clusters

Schedulable Contract

`Schedulable` is an interface for schedulable entities.

It assumes that each `Schedulable`:

- Is identified by `name`
- Knows about its parent `Pool`
- Has a `schedulableQueue`, a `schedulingMode`, `weight`, `minShare`, `runningTasks`, `priority`, `stageId`
- Adds or removes `Schedulables`
- Returns a `Schedulable` by name
- Can be informed about lost executors (using `executorLost` method)
- Checks speculatable tasks (using `checkSpeculatableTasks`)
- Tracks `TaskSetManagers` (using `getSortedTaskSetQueue`)

Pool

`Pool` is a [Schedulable](#). It requires a name, a scheduling mode, initial `minshare` and weight.

TaskContextImpl

Caution	FIXME
---------	-----------------------

- `stage`
- `partition`
- `task attempt`
- `attempt number`
- `runningLocally = false`

TaskMemoryManager

Caution	FIXME
---------	-----------------------

TaskMetrics

Caution	FIXME
---------	-----------------------

Tasks

Task (aka *command*) is an individual unit of work for executors to run.

It is an individual unit of physical execution (computation) that runs on a single machine for parts of your Spark application on a data. All tasks in a stage should be completed before moving on to another stage.

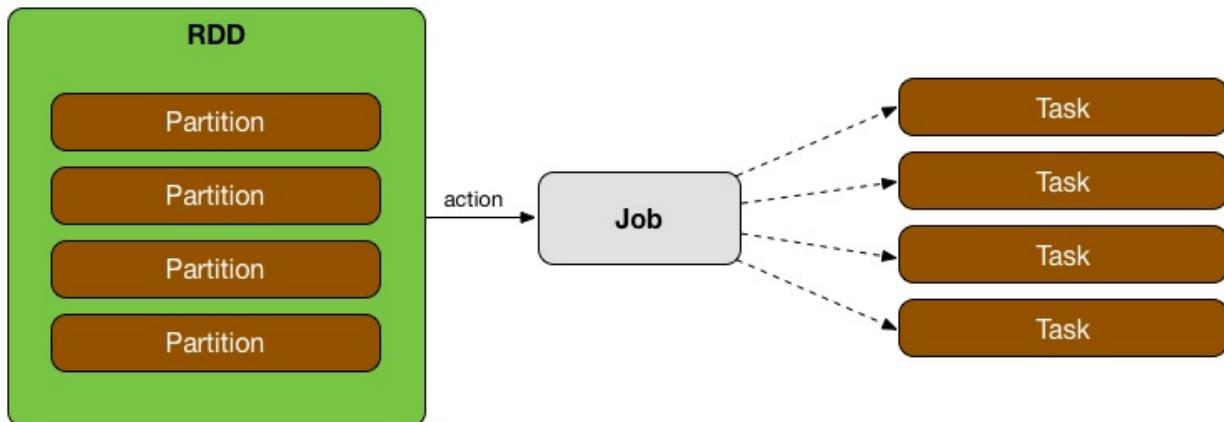


Figure 1. Tasks correspond to partitions in RDD

A task can also be considered a computation in a stage on a partition in a given job attempt.

A Task belongs to a single stage and operates on a single partition (a part of an RDD).

Tasks are spawned one by one for each stage and data partition.

Caution

[FIXME](#) What are `stageAttemptId` and `taskAttemptId` ?

There are two kinds of tasks:

- **ShuffleMapTask** that executes a task and divides the task's output to multiple buckets (based on the task's partitioner). See [ShuffleMapTask](#).
- **ResultTask** that executes a task and sends the task's output back to the driver application.

The very last stage in a job consists of multiple `ResultTasks`, while earlier stages consist of [ShuffleMapTask](#)'s.

Task Execution

Caution

[FIXME](#) What does `Task.run` do?

Consult [TaskRunner](#).

Task States

A task can be in one of the following states:

- LAUNCHING
- RUNNING
- FINISHED
- FAILED - when `FetchFailedException` (see [FetchFailedException](#)),
`CommitDeniedException` or any `Throwable` occur
- KILLED - when an executor kills a task
- LOST

States are the values of `org.apache.spark.TaskState`.

Task is finished when it is in one of `FINISHED`, `FAILED`, `KILLED`, `LOST`

`LOST` and `FAILED` states are considered failures.

Note

Task states correspond to [org.apache.mesos.Protos.TaskState](#).

ShuffleMapTask

A **ShuffleMapTask** divides the elements of an RDD into multiple buckets (based on a partitioner specified in [ShuffleDependency](#)).

TaskSets

Introduction

A **TaskSet** is a collection of tasks that belong to a single [stage](#) and a [stage attempt](#). It has also **priority** and **properties** attributes. Priority is used in FIFO scheduling mode (see [Priority Field and FIFO Scheduling](#)) while properties are the properties of the first job in the stage.

Caution

[FIXME](#) Where are properties of a TaskSet used?

A TaskSet represents the missing partitions of a stage.

The pair of a stage and a stage attempt uniquely describes a TaskSet and that is what you can see in the logs when a TaskSet is used:

```
TaskSet [stageId].[stageAttemptId]
```

A TaskSet contains a fully-independent sequence of tasks that can run right away based on the data that is already on the cluster, e.g. map output files from previous stages, though it may fail if this data becomes unavailable.

TaskSet can be submitted (consult [TaskScheduler Contract](#)).

removeRunningTask

Caution

[FIXME](#) Review `TaskSet.removeRunningTask(tid)`

Where TaskSets are used

- [DAGScheduler.submitMissingTasks](#)
 - `TaskSchedulerImpl.submitTasks`
- `DAGScheduler.taskSetFailed`
- `DAGScheduler.handleTaskSetFailed`
- `TaskSchedulerImpl.createTaskSetManager`

Priority Field and FIFO Scheduling

A TaskSet has `priority` field that turns into the **priority** field's value of [TaskSetManager](#) (which is a Schedulable).

The `priority` field is used in `FIFOSchedulingAlgorithm` in which equal priorities give stages an advantage (not to say *priority*).

`FIFOSchedulingAlgorithm` is only used for `FIFO` scheduling mode in a `Pool` which is a Schedulable collection of `Schedulable's.

Effectively, the `priority` field is the job's id of the first job this stage was part of (for FIFO scheduling).

TaskSetManager

TaskSetManager manages execution of the tasks in a single **TaskSet** (after having it been handed over by [TaskScheduler](#)).

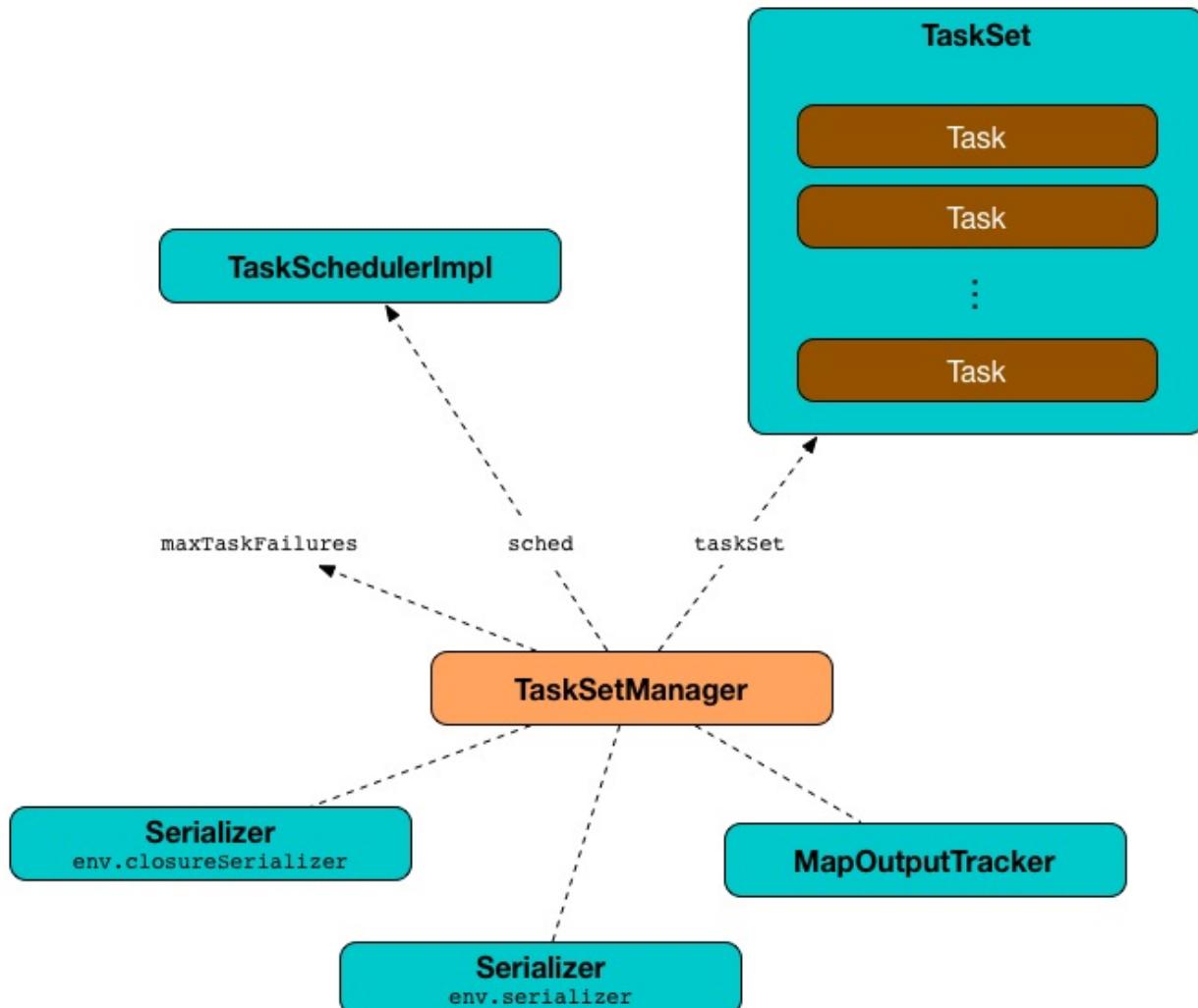


Figure 1. TaskSetManager and its Dependencies

The responsibilities of a TaskSetManager include (follow along the links to learn more in the corresponding sections):

- [Scheduling the tasks in a taskset](#)
- [Retrying tasks on failure](#)
- [Locality-aware scheduling via delay scheduling](#)

Tip

Enable `DEBUG` logging level for `org.apache.spark.scheduler.TaskSetManager` logger to see what happens under the covers in `TaskSetManager`.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.scheduler.TaskSetManager=DEBUG
```

TaskSetManager is Schedulable

`TaskSetManager` is a [Schedulable](#) with the following implementation:

- no `parent` assigned, i.e. it is always `null`.
- `schedulableQueue` returns no queue, i.e. `null`.
- `schedulingMode` returns `SchedulingMode.NONE`.
- `weight` is always `1`.
- `minShare` is always `0`.
- `runningTasks` is the number of running tasks in the internal `runningTasksSet`.
- `priority` is the priority of the TaskSet (using `taskSet.priority`).
- `stageId` is the stage id of the TaskSet (using `taskSet.stageId`).
- `name` is `TaskSet_[taskSet.stageId.toString]`
- `addSchedulable` does nothing.
- `removeSchedulable` does nothing.
- `getSchedulableByName` always returns `null`.
- [executorLost](#)
- [checkSpeculatableTasks](#)
- `getSortedTaskSetQueue` returns a one-element collection with the sole element being itself.

TaskSetManager.checkSpeculatableTasks

`checkSpeculatableTasks` checks whether there are speculatable tasks in the TaskSet.

Note

`checkSpeculatableTasks` is called by `TaskSchedulerImpl.checkSpeculatableTasks`.

If the TaskSetManager is `zombie` or has a single task in TaskSet, it assumes no speculatable tasks.

The method goes on with the assumption of no speculatable tasks by default.

It computes the minimum number of finished tasks for speculation (as `spark.speculation.quantile` of all the finished tasks).

You should see the DEBUG message in the logs:

```
DEBUG Checking for speculative tasks: minFinished = [minFinishedForSpeculation]
```

It then checks whether the number is equal or greater than the number of tasks completed successfully (using `tasksSuccessful`).

Having done that, it computes the median duration of all the successfully completed tasks (using `taskInfos`) and task length threshold using the median duration multiplied by `spark.speculation.multiplier` that has to be equal or less than `100` .

You should see the DEBUG message in the logs:

```
DEBUG Task length threshold for speculation: [threshold]
```

For each task (using `taskInfos`) that is not marked as successful yet (using `successful`) for which there is only one copy running (using `copiesRunning`) and the task takes more time than the calculated threshold, but it was not in `speculatableTasks` it is assumed **speculatable**.

You should see the following INFO message in the logs:

```
INFO Marking task [index] in stage [taskSet.id] (on [info.host]) as speculatable because
```

The task gets added to the internal `speculatableTasks` collection. The method responds positively.

TaskSetManager.executorLost

`executorLost(execId: String, host: String, reason: ExecutorLossReason)` is part of [Schedulable contract](#) that TaskSetManager follows.

It first checks whether the TaskSet is for a [ShuffleMapStage](#), i.e. all `TaskSet.tasks` are instances of [ShuffleMapTask](#). At the same time, it also checks whether an external shuffle server is used (using `env.blockManager.externalShuffleServiceEnabled`) that could serve the shuffle outputs.

If the above checks pass, tasks (using `taskInfos`) are checked for their executors and whether the failed one is among them.

For each task that executed on the failed executor, if it was not marked as successful already (using `successful`), it is:

- marked as failed (using `successful`)
- removed from `copiesRunning`
- the number of successful tasks is decremented (using `tasksSuccessful`)
- the task gets added to the collection of pending tasks (using `addPendingTask` method)
- DAGScheduler is informed about resubmission (using [DAGScheduler.taskEnded](#) with the `reason` being `Resubmitted`).

For the other cases, i.e. if the TaskSet is for [ResultStage](#) or an external shuffle service is used, for all running tasks for the failed executor, [handleFailedTask](#) is called with the task state being `FAILED`.

Eventually, [recomputeLocality\(\)](#) is called.

TaskSetManager.executorAdded

`executorAdded` simply calls [recomputeLocality](#) method.

TaskSetManager.recomputeLocality

`recomputeLocality` (re)computes locality levels as a indexed collection of task localities, i.e. `Array[TaskLocality.TaskLocality]`.

Note	<code>TaskLocality</code> is an enumeration with <code>PROCESS_LOCAL</code> , <code>NODE_LOCAL</code> , <code>NO_PREF</code> , <code>RACK_LOCAL</code> , <code>ANY</code> values.
------	---

The method starts with `currentLocalityIndex` being `0`.

It checks whether `pendingTasksForExecutor` has at least one element, and if so, it looks up [spark.locality.wait.*](#) for `PROCESS_LOCAL` and checks whether there is an executor for which `TaskSchedulerImpl.isExecutorAlive` is `true`. If the checks pass, `PROCESS_LOCAL` becomes an element of the result collection of task localities.

The same checks are performed for `pendingTasksForHost`, `NODE_LOCAL`, and `TaskSchedulerImpl.hasExecutorsAliveOnHost` to add `NODE_LOCAL` to the result collection of task localities.

Then, the method checks `pendingTasksWithNoPrefs` and if it's not empty, `NO_PREF` becomes an element of the levels collection.

If `pendingTasksForRack` is not empty, and the wait time for `RACK_LOCAL` is defined, and there is an executor for which `TaskSchedulerImpl.hasHostAliveOnRack` is `true`, `RACK_LOCAL` is added to the levels collection.

`ANY` is the last and always-added element in the levels collection.

Right before the method finishes, it prints out the following DEBUG to the logs:

```
DEBUG Valid locality levels for [taskSet]: [levels]
```

`myLocalityLevels`, `localityWaits`, and `currentLocalityIndex` are recomputed.

TaskSetManager.resourceOffer

Caution	FIXME Review <code>TaskSetManager.resourceOffer</code> + Does this have anything related to the following section about scheduling tasks?
---------	--

For every TaskSet submitted for execution, `TaskSchedulerImpl` creates a new instance of `TaskSetManager`. It then calls `SchedulerBackend.reviveOffers()` (refer to [submitTasks](#)).

Caution	FIXME picture of the calls between components
---------	--

`resourceOffer` method responds to an offer of a single executor from the scheduler by finding a task (as a `TaskDescription`). It works in [non-zombie state](#) only. It dequeues a pending task from the taskset by checking pending tasks per executor (using `pendingTasksForExecutor`), host (using `pendingTasksForHost`), with no localization preferences (using `pendingTasksWithNoPrefs`), rack (uses `TaskSchedulerImpl.getRackForHost` that seems to return "non-zero" value for [YarnScheduler](#) only)

From `TaskSetManager.resourceOffer`:

```
INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, 192.168.1.4, partition 0, PROC
```

If a serialized task is bigger than `100` kB (it is not a configurable value), a `WARN` message is printed out to the logs (only once per taskset):

```
WARN TaskSetManager: Stage [task.stageId] contains a task of very large size ([serialized
```

A task id is added to `runningTasksSet` set and [parent pool](#) notified (using `increaseRunningTasks(1)` up the chain of pools).

The following INFO message appears in the logs:

```
INFO TaskSetManager: Starting task [id] in stage [taskSet.id] (TID [taskId], [host], part
```

For example:

```
INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1, localhost, partition 1, PROCES
```

Scheduling Tasks in TaskSet

Caution	FIXME
---------	-----------------------

For each submitted [TaskSet](#), a new TaskSetManager is created. The TaskSetManager completely and exclusively owns a TaskSet submitted for execution.

Caution	FIXME A picture with TaskSetManager owning TaskSet
---------	--

Caution	FIXME What component knows about TaskSet and TaskSetManager. Isn't it that TaskSets are created by DAGScheduler while TaskSetManager is used by TaskSchedulerImpl only?
---------	--

TaskSetManager requests the current epoch from [MapOutputTracker](#) and sets it on all tasks in the taskset.

You should see the following DEBUG in the logs:

```
DEBUG Epoch for [taskSet]: [epoch]
```

Caution	FIXME What's epoch. Why is this important?
---------	--

TaskSetManager keeps track of the tasks pending execution per executor, host, rack or with no locality preferences.

Locality-Aware Scheduling aka Delay Scheduling

TaskSetManager computes locality levels for the TaskSet for delay scheduling. While computing you should see the following DEBUG in the logs:

```
DEBUG Valid locality levels for [taskSet]: [levels]
```

Caution	FIXME What's delay scheduling?
---------	--

Events

When a task has finished, [TaskSetManager](#) calls [DAGScheduler.taskEnded](#).

Caution	FIXME
---------	-----------------------

TaskSetManager.handleSuccessfulTask

`handleSuccessfulTask(tid: Long, result: DirectTaskResult[_])` method marks the task (by `tid`) as successful and notifies the DAGScheduler that the task has ended.

It is called by... when...[FIXME](#)

Caution	FIXME Describe <code>TaskInfo</code>
---------	--

It marks `TaskInfo` (using `taskInfos`) as successful (using `TaskInfo.markSuccessful()`).

It removes the task from `runningTasksSet`. It also decreases the number of running tasks in the parent pool if it is defined (using `parent` and `Pool.decreaseRunningTasks`).

It notifies DAGScheduler that the task ended successfully (using [DAGScheduler.taskEnded](#) with `Success` as `TaskEndReason`).

If the task was not marked as successful already (using `successful`), `tasksSuccessful` is incremented and the following INFO message appears in the logs:

```
INFO Finished task [info.id] in stage [taskSet.id] (TID [info.taskId]) in [info.duration]
```

◀	▶
-------------------	-------------------

Note	A TaskSet knows about the stage id it is associated with.
------	---

It also marks the task as successful (using `successful`). Finally, if the number of tasks finished successfully is exactly the number of tasks the TaskSetManager manages, the TaskSetManager turns zombie.

Otherwise, when the task was already marked as successful, the following INFO message appears in the logs:

```
INFO Ignoring task-finished event for [info.id] in stage [taskSet.id] because task [index
failedExecutors.remove(index) is called.
```

Caution**FIXME** What does `failedExecutors.remove(index)` mean?

At the end, the method checks whether the TaskSetManager is a zombie and no task is running (using `runningTasksSet`), and if so, it calls [TaskSchedulerImpl.taskSetFinished](#).

TaskSetManager.handleFailedTask

`handleFailedTask(tid: Long, state: TaskState, reason: TaskEndReason)` method is called by [TaskSchedulerImpl](#) or [executorLost](#).

Caution**FIXME** image with `handleFailedTask` (and perhaps the other parties involved)

The method first checks whether the task has already been marked as failed (using [taskInfos](#)) and if it has, it quits.

It removes the task from [runningTasksSet](#) and informs [the parent pool](#) to decrease its running tasks.

It marks the TaskInfo as failed and grabs its index so the number of copies running of the task is decremented (see [copiesRunning](#)).

Caution**FIXME** Describe `TaskInfo`

The method calculates the failure exception to report per `TaskEndReason`. See below for the possible cases of `TaskEndReason`.

Caution**FIXME** Describe `TaskEndReason`.

The executor for the failed task is added to [failedExecutors](#).

It informs DAGScheduler that the task ended (using [DAGScheduler.taskEnded](#)).

The task is then added to the list of pending tasks.

Caution**FIXME** Review `addPendingTask`

If the TaskSetManager is not a [zombie](#), and the task was not `KILLED`, and the task failure should be counted towards the maximum number of times the task is allowed to fail before the stage is aborted (`TaskFailedReason.countTowardsTaskFailures` is `true`), `numFailures` is

incremented and if the number of failures of the task equals or is greater than assigned to the TaskSetManager (`maxTaskFailures`), the ERROR appears in the logs:

```
ERROR Task [id] in stage [id] failed [maxTaskFailures] times; aborting job
```

And `abort` is called, and the method quits.

Otherwise, `TaskSchedulerImpl.taskSetFinished` is called when the TaskSetManager is `zombie` and there are no running tasks.

FetchFailed

For `FetchFailed`, it logs WARNING:

```
WARNING Lost task [id] in stage [id] (TID [id], [host]): [reason.toErrorString]
```

Unless it has already been marked as successful (in `successful`), the task becomes so and `tasksSuccessful` is incremented.

The TaskSetManager becomes `zombie`.

No exception is returned.

ExceptionFailure

For `ExceptionFailure`, it grabs `TaskMetrics` if available.

If it is a `NotSerializableException`, it logs ERROR:

```
ERROR Task [id] in stage [id] (TID [tid]) had a not serializable result: [exception.descr]
```

It calls `abort` and returns no failure exception.

It continues if not being a `NotSerializableException`.

It grabs the description and the time of the ExceptionFailure.

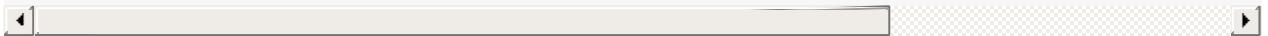
If the description, i.e. the ExceptionFailure, has already been reported (and is therefore a duplication), `spark.logging.exceptionPrintInterval` is checked before reprinting the duplicate exception in full.

For full printout of the ExceptionFailure, the following WARNING appears in the logs:

```
WARNING Lost task [id] in stage [id] (TID [id], [host]): [reason.toErrorString]
```

Otherwise, the following INFO appears in the logs:

```
INFO Lost task [id] in stage [id] (TID [id]) on executor [host]: [ef.className] ([ef.desc
```



The ExceptionFailure becomes failure exception.

ExecutorLostFailure

For `ExecutorLostFailure` if not `exitCausedByApp`, the following INFO appears in the logs:

```
INFO Task [tid] failed because while it was being computed, its executor exited for a rea
```



No failure exception is returned.

Other TaskFailedReasons

For the other TaskFailedReasons, the WARNING appears in the logs:

```
WARNING Lost task [id] in stage [id] (TID [id], [host]): [reason.toErrorString]
```

No failure exception is returned.

Other TaskEndReason

For the other TaskEndReasons, the ERROR appears in the logs:

```
ERROR Unknown TaskEndReason: [e]
```

No failure exception is returned.

Retrying Tasks on Failure

Caution	FIXME
---------	-------

Up to `spark.task.maxFailures` attempts

Task retries and spark.task.maxFailures

When you start Spark program you set up `spark.task.maxFailures` for the number of failures that are acceptable until TaskSetManager gives up and marks a job failed.

In Spark shell with local master, `spark.task.maxFailures` is fixed to `1` and you need to use [local-with-retries master](#) to change it to some other value.

In the following example, you are going to execute a job with two partitions and keep one failing at all times (by throwing an exception). The aim is to learn the behavior of retrying task execution in a stage in TaskSet. You will only look at a single task execution, namely

`0.0`.

```
$ ./bin/spark-shell --master "local[*, 5]"
...
scala> sc.textFile("README.md", 2).mapPartitionsWithIndex((idx, it) => if (idx == 0) throw
...
15/10/27 17:24:56 INFO DAGScheduler: Submitting 2 missing tasks from ResultStage 1 (MapPa
15/10/27 17:24:56 DEBUG DAGScheduler: New pending partitions: Set(0, 1)
15/10/27 17:24:56 INFO TaskSchedulerImpl: Adding task set 1.0 with 2 tasks
...
15/10/27 17:24:56 INFO TaskSetManager: Starting task 0.0 in stage 1.0 (TID 2, localhost,
...
15/10/27 17:24:56 INFO Executor: Running task 0.0 in stage 1.0 (TID 2)
...
15/10/27 17:24:56 ERROR Executor: Exception in task 0.0 in stage 1.0 (TID 2)
java.lang.Exception: Partition 2 marked failed
...
15/10/27 17:24:56 INFO TaskSetManager: Starting task 0.1 in stage 1.0 (TID 4, localhost,
15/10/27 17:24:56 INFO Executor: Running task 0.1 in stage 1.0 (TID 4)
15/10/27 17:24:56 INFO HadoopRDD: Input split: file:/Users/jacek/dev/oss/spark/README.md:
15/10/27 17:24:56 ERROR Executor: Exception in task 0.1 in stage 1.0 (TID 4)
java.lang.Exception: Partition 2 marked failed
...
15/10/27 17:24:56 ERROR Executor: Exception in task 0.4 in stage 1.0 (TID 7)
java.lang.Exception: Partition 2 marked failed
...
15/10/27 17:24:56 INFO TaskSetManager: Lost task 0.4 in stage 1.0 (TID 7) on executor loc
15/10/27 17:24:56 ERROR TaskSetManager: Task 0 in stage 1.0 failed 5 times; aborting job
15/10/27 17:24:56 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks have all compl
15/10/27 17:24:56 INFO TaskSchedulerImpl: Cancelling stage 1
15/10/27 17:24:56 INFO DAGScheduler: ResultStage 1 (count at <console>:25) failed in 0.05
15/10/27 17:24:56 DEBUG DAGScheduler: After removal of stage 1, remaining stages = 0
15/10/27 17:24:56 INFO DAGScheduler: Job 1 failed: count at <console>:25, took 0.085810 s
org.apache.spark.SparkException: Job aborted due to stage failure: Task 0 in stage 1.0 fa
```

Zombie state

TaskSetManager enters **zombie** state when all tasks in a taskset have completed successfully (regardless of the number of task attempts), or if the task set has been aborted (see [Aborting TaskSet](#)).

While in zombie state, TaskSetManager can launch no new tasks and [responds with no TaskDescription to resourceOffers](#).

TaskSetManager remains in the zombie state until all tasks have finished running, i.e. to continue to track and account for the running tasks.

Aborting TaskSet using abort Method

`abort(message: String, exception: Option[Throwable] = None)` method informs [DAGScheduler](#) that a TaskSet was aborted (using `DAGScheduler.taskSetFailed` method).

Caution	FIXME image with DAGScheduler call
---------	--

The TaskSetManager enters [zombie state](#).

Finally, `maybeFinishTaskSet` method is called.

Caution	FIXME Why is <code>maybeFinishTaskSet</code> method called? When is <code>runningTasks</code> <code>0</code> ?
---------	--

Internal Registries

- `copiesRunning`
- `successful`
- `numFailures`
- `failedExecutors` contains a mapping of TaskInfo's indices that failed to executor ids and the time of the failure. It is used in [handleFailedTask](#).
- `taskAttempts`
- `tasksSuccessful`
- `weight` (default: `1`)
- `minShare` (default: `0`)
- `priority` (default: `taskSet.priority`)
- `stageId` (default: `taskSet.stageId`)
- `name` (default: `TaskSet_[taskSet.stageId]`)

- `parent`
- `totalResultSize`
- `calculatedTasks`
- `runningTasksSet`
- `isZombie (default: false)`
- `pendingTasksForExecutor`
- `pendingTasksForHost`
- `pendingTasksForRack`
- `pendingTasksWithNoPrefs`
- `allPendingTasks`
- `speculatableTasks`
- `taskInfos` is the mapping between task ids and their `TaskInfo`
- `recentExceptions`

Settings

- `spark.scheduler.executorTaskBlacklistTime` (`default: 0L`) - time interval to pass after which a task can be re-launched on the executor where it has once failed. It is to prevent repeated task failures due to executor failures.
- `spark.speculation (default: false)`
- `spark.speculation.quantile` (`default: 0.75`) - the percentage of tasks that has not finished yet at which to start speculation.
- `spark.speculation.multiplier` (`default: 1.5`)
- `spark.driver.maxResultSize` (`default: 1g`) is the limit of bytes for total size of results. If the value is smaller than `1m` or `1048576` (`1024 * 1024`), it becomes 0.
- `spark.logging.exceptionPrintInterval` (`default: 10000`) - how frequently to reprint duplicate exceptions in full, in milliseconds
- `spark.locality.wait` (`default: 3s`) - for locality-aware delay scheduling for `PROCESS_LOCAL`, `NODE_LOCAL`, and `RACK_LOCAL` when locality-specific setting is not set.
- `spark.locality.wait.process` (`default: the value of spark.locality.wait`) - delay for `PROCESS_LOCAL`

- `spark.locality.wait.node` (default: the value of `spark.locality.wait`) - delay for `NODE_LOCAL`
- `spark.locality.wait.rack` (default: the value of `spark.locality.wait`) - delay for `RACK_LOCAL`

TaskSchedulerImpl - Default TaskScheduler

`TaskSchedulerImpl` is the default implementation of [TaskScheduler Contract](#). It can schedule tasks for multiple types of cluster managers by using a [Scheduler Backend](#).

When your Spark application starts, i.e. at the time when an instance of `SparkContext` is created, `TaskSchedulerImpl` with a `SchedulerBackend` and `DAGScheduler` are created and started, too.

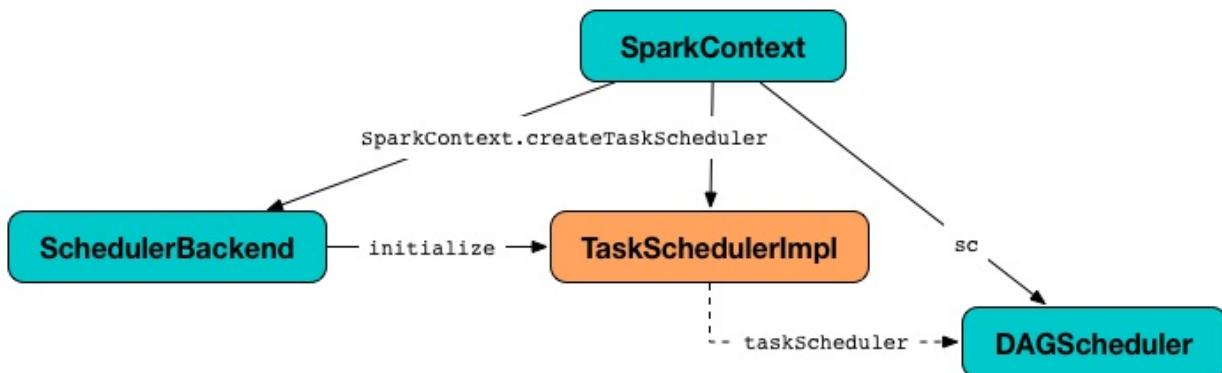


Figure 1. `TaskSchedulerImpl` and Other Services

Note	The source code is in org.apache.spark.scheduler.TaskSchedulerImpl .
------	--

TaskSchedulerImpl.initialize

`initialize(backend: SchedulerBackend)` method is responsible for initializing a `TaskSchedulerImpl`.

It has to be called after a `TaskSchedulerImpl` is created and before it can be [started](#) to know about the [SchedulerBackend](#) to use.

Note	<code>SparkContext</code> initializes <code>TaskSchedulerImpl</code> .
------	--

Besides being assigned the instance of `SchedulerBackend`, it sets up `rootPool` (as part of [TaskScheduler Contract](#)). It uses a scheduling mode (as configured by `spark.scheduler.mode`) to build a `SchedulableBuilder` and call `SchedulableBuilder.buildPools()`.

TaskSchedulerImpl.start

When `TaskSchedulerImpl` is started (using `start()` method), it starts the [scheduler backend](#) it manages.

Below is a figure of the method calls in Spark Standalone mode.

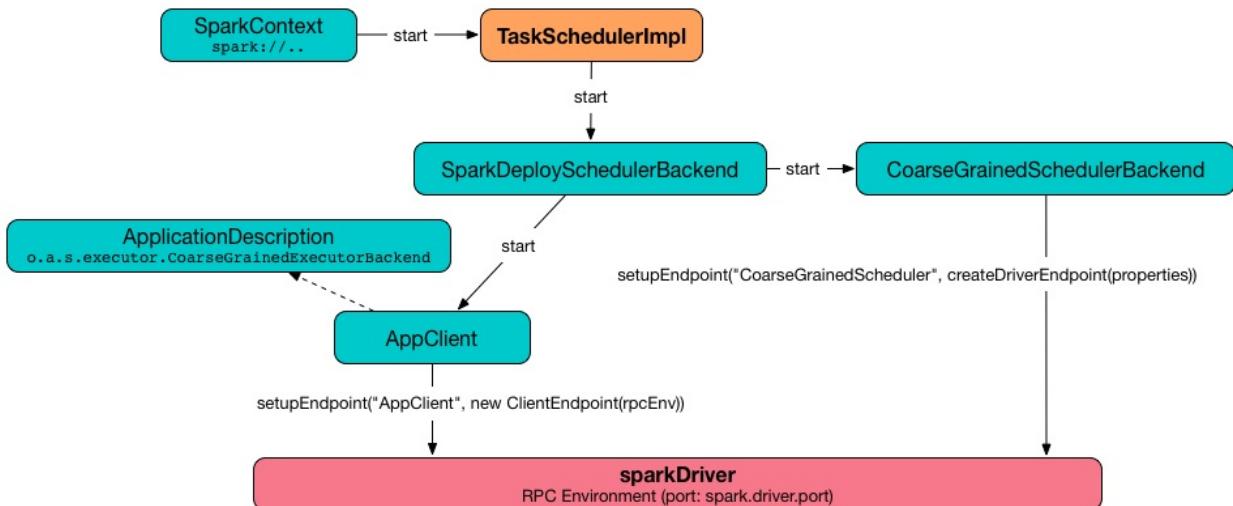


Figure 2. Starting TaskSchedulerImpl in Spark Standalone mode

Tip	SparkContext starts a TaskScheduler.
-----	--------------------------------------

It also starts the **task-scheduler-speculation** executor pool. See [Speculative Execution of Tasks](#).

Post-Start Initialization (using postStartHook)

`TaskSchedulerImpl` comes with a custom `postStartHook()` (see [TaskScheduler Contract](#)) to wait until a scheduler backend is ready (see [SchedulerBackend Contract](#)).

Internally, it uses `waitBackendReady()` to do the waiting and looping.

TaskSchedulerImpl.stop

When `TaskSchedulerImpl` is stopped (using `stop()` method), it does the following:

- Shuts down the internal `task-scheduler-speculation` thread pool executor (used for [Speculative execution of tasks](#)).
- Stops [SchedulerBackend](#).
- Stops [TaskResultGetter](#).
- Cancels `starvationTimer` timer.

Speculative Execution of Tasks

Speculative tasks (also **speculatable tasks** or **task strugglers**) are tasks that run slower than most ([FIXME](#) the setting) of the all tasks in a job.

Speculative execution of tasks is a health-check procedure that checks for tasks to be **speculated**, i.e. running slower in a stage than the median of all successfully completed tasks in a taskset ([FIXME](#) the setting). Such slow tasks will be re-launched in another worker. It will not stop the slow tasks, but run a new copy in parallel.

The thread starts as `TaskSchedulerImpl` starts in [clustered deployment modes](#) with `spark.speculation` enabled. It executes periodically every `spark.speculation.interval` after `spark.speculation.interval` passes.

When enabled, you should see the following INFO message in the logs:

```
INFO Starting speculative execution thread
```

It works as **task-scheduler-speculation** daemon thread pool using

`j.u.c.ScheduledThreadPoolExecutor` with core pool size `1`.

The job with speculatable tasks should finish while speculative tasks are running, and it will leave these tasks running - no KILL command yet.

It uses `checkSpeculatableTasks` method that asks `rootPool` to check for speculatable tasks. If there are any, SchedulerBackend is called for [reviveOffers](#).

Caution

[FIXME](#) How does Spark handle repeated results of speculative tasks since there are copies launched?

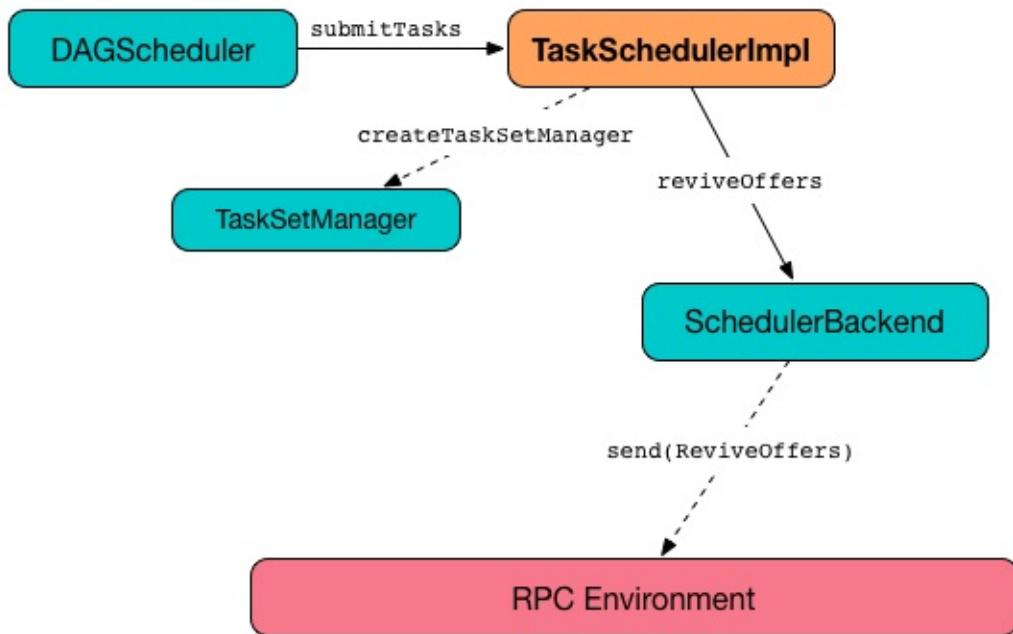
Default Level of Parallelism

Default level of parallelism is a hint for sizing jobs.

`TaskSchedulerImpl` uses [SchedulerBackend.defaultParallelism\(\)](#) to calculate the value, i.e. it just passes it along to a scheduler backend.

Task Submission (using submitTasks)

Tasks are submitted for execution as a [TaskSet](#) using `submitTasks(taskSet: TaskSet)` method.

Figure 3. `TaskSchedulerImpl.submitTasks`**Note**

If there are tasks to launch for missing partitions in a stage, `DAGScheduler` executes `submitTasks` (see [submitMissingTasks for Stage and Job](#)).

When this method is called, you should see the following INFO message in the logs:

```
INFO TaskSchedulerImpl: Adding task set [taskSet.id] with [tasks.length] tasks
```

It creates a new `TaskSetManager` for the given `TaskSet` `taskset` and the acceptable number of task failures (as `maxTaskFailures`).

Note

`maxTaskFailures` that is the acceptable number of task failures is given when a `TaskSchedulerImpl` is created.

`taskSetsByStageIdAndAttempt`, i.e. a mapping of stages and another mapping of attempt ids and `TaskSetManagers`, is checked for conflicting `TaskSetManagers`, i.e. `TaskSetManagers` for which the `TaskSets` are different and `TaskSetManager` is not zombie. If there is one, an `IllegalStateException` is thrown with the message:

```
more than one active taskSet for stage [stage]: [TaskSet ids]
```

Otherwise, `SchedulableBuilder.addTaskSetManager` is called with the `manager` being the `TaskSetManager` just created.

When the method is called the first time (`hasReceivedTask` is `false`) in cluster mode, `starvationTimer` is scheduled at fixed rate, i.e. every `spark.starvation.timeout` after the first `spark.starvation.timeout` passes (`hasReceivedTask` becomes `true`).

Every time the starvation timer thread is executed, it checks whether `hasLaunchedTask` is `false`, and logs the WARNING:

```
WARNING Initial job has not accepted any resources; check your cluster UI to ensure that
```

Otherwise, the timer thread cancels itself.

It then calls `SchedulerBackend.reviveOffers()`.

Tip	Use <code>dag-scheduler-event-loop</code> thread to step through the code in a debugger.
-----	--

Offering Resources (using `resourceOffers`)

`resourceOffers(offers: Seq[WorkerOffer])` method is called by a cluster manager or `LocalBackend` (for local mode) to offer free resources available on the executors to run tasks on.

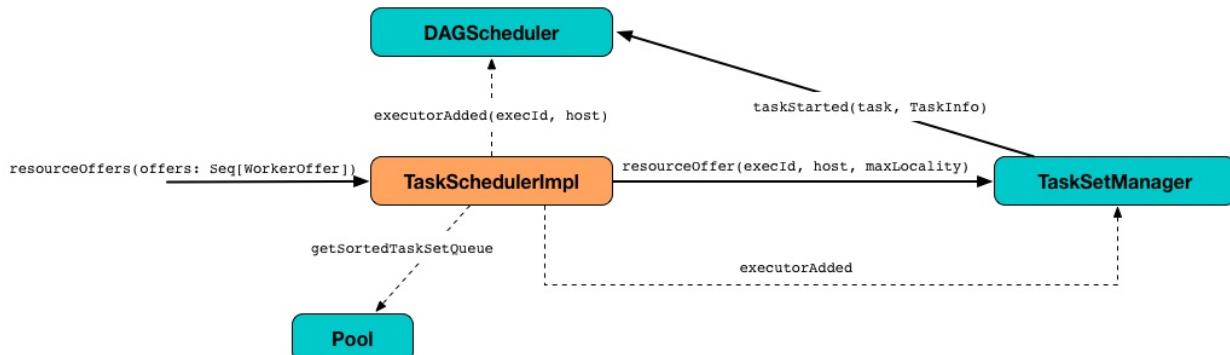


Figure 4. `TaskSchedulerImpl.resourceOffers` under the hood

A `workerOffer` is a 3-tuple with executor id, host, and free cores available.

For each offer, the `resourceOffers` method tracks hosts per executor id (using `executorIdToHost`) and sets `0` as the number of tasks running on the executor if there is no tasks running already (using `executorIdToTaskCount`). It also tracks executor id per host.

Warning	FIXME BUG? Why is the executor id not added to <code>executorsByHost</code> ?
---------	--

`DAGScheduler.executorAdded(executorId, host)` is called for a new host.

Warning	FIXME BUG? Why is <code>executorAdded</code> called for a new host added? Can't we have more executors on a host? The name of the method is misleading then.
---------	--

Caution	FIXME a picture with <code>executorAdded</code> call from <code>TaskSchedulerImpl</code> to <code>DAGScheduler</code> .
---------	---

Caution**FIXME** Why is `getRackForHost` important?

It builds a list of tasks (using `TaskDescription`) to assign to each worker.

`rootPool.getSortedTaskSetQueue` is called for available TaskSetManagers and for each TaskSetManager the DEBUG message is printed out to the logs:

```
DEBUG parentName: [taskSet.parent.name], name: [taskSet.name], runningTasks: [taskSet.run
```



Moreover, if a new host was added to the pool (using `newExecAvail` - **FIXME** when exactly?), TaskSetManagers get informed about the new executor (using `TaskSetManager.executorAdded()`).

Warning**FIXME BUG?** Why is the name `newExecAvail` since it's called for a new host added? Can't we have more executors on a host? The name of the method could be misleading.

For each taskset in `sortedTaskSets`, different locality preferences are checked...**FIXME**

Check whether the number of cores in an offer is more than the number of cores needed for a task (using `spark.task.cpus`).

When `resourceOffers` managed to launch a task, the internal field `hasLaunchedTask` becomes `true` (that effectively means what the name says "*There were executors and I managed to launch a task*").

TaskResultGetter

`TaskResultGetter` is a helper class for `TaskSchedulerImpl.statusUpdate`. It *asynchronously* fetches the task results of tasks that have finished successfully (using `enqueueSuccessfulTask`) or fetches the reasons of failures for failed tasks (using `enqueueFailedTask`). It then sends the "results" back to `TaskSchedulerImpl`.

Caution**FIXME** Image with the dependencies**Tip**Consult [Task States](#) in Tasks to learn about the different task states.

The only instance of `TaskResultGetter` is created as part of a `TaskSchedulerImpl` creation (as `taskResultGetter`). It requires a `SparkEnv` and `TaskSchedulerImpl`. It is stopped when `TaskSchedulerImpl` stops.

`TaskResultGetter` offers the following methods:

- enqueueSuccessfulTask
- enqueueFailedTask

The methods use the internal (daemon thread) thread pool **task-result-getter** (as `getTaskResultExecutor`) with `spark.resultGetter.threads` so they can be executed asynchronously.

TaskResultGetter.enqueueSuccessfulTask

`enqueueSuccessfulTask(taskSetManager: TaskSetManager, tid: Long, serializedData: ByteBuffer)` starts by deserializing `TaskResult` (from `serializedData` using `SparkEnv.closureSerializer`).

If the result is `DirectTaskResult`, the method checks `taskSetManager.canFetchMoreResults(serializedData.limit())` and possibly quits. If not, it deserializes the result (using `SparkEnv.serializer`).

Caution

FIXME Review

`taskSetManager.canFetchMoreResults(serializedData.limit())`.

If the result is `IndirectTaskResult`, the method checks

`taskSetManager.canFetchMoreResults(size)` and possibly removes the block id (using `SparkEnv.blockManager.master.removeBlock(blockId)`) and quits. If not, you should see the following DEBUG message in the logs:

```
DEBUG Fetching indirect task result for TID [tid]
```

`scheduler.handleTaskGettingResult(taskSetManager, tid)` gets called. And `sparkEnv.blockManager.getRemoteBytes(blockId)`.

Failure in getting task result from BlockManager results in calling `TaskSchedulerImpl.handleFailedTask(taskSetManager, tid, TaskState.FINISHED, TaskResultLost)` and quit.

The task result is deserialized to `DirectTaskResult` (using `SparkEnv.closureSerializer`) and `sparkEnv.blockManager.master.removeBlock(blockId)` is called afterwards.

`TaskSchedulerImpl.handleSuccessfulTask(taskSetManager, tid, result)` is called.

Caution

FIXME What is `TaskSchedulerImpl.handleSuccessfulTask` doing?

Any `ClassNotFoundException` or non fatal exceptions lead to `TaskSetManager.abort`.

TaskResultGetter.enqueueFailedTask

`enqueueFailedTask(taskSetManager: TaskSetManager, tid: Long, taskState: TaskState, serializedData: ByteBuffer)` checks whether `serializedData` contains any data and if it does it deserializes it to a `TaskEndReason` (using `SparkEnv.closureSerializer`).

Either `UnknownReason` or the deserialized instance is passed on to [TaskSchedulerImpl.handleFailedTask](#) as the reason of the failure.

Any `ClassNotFoundException` leads to printing out the ERROR message to the logs:

```
ERROR Could not deserialize TaskEndReason: ClassNotFoundException with classloader [loader]
```

TaskSchedulerImpl.statusUpdate

`statusUpdate(tid: Long, state: TaskState, serializedData: ByteBuffer)` is called by [scheduler backends](#) to inform about task state changes (see [Task States](#) in Tasks).

Caution	FIXME image with scheduler backends calling <code>TaskSchedulerImpl.statusUpdate</code> .
---------	---

It is called by:

- [CoarseGrainedSchedulerBackend](#) when `StatusUpdate(executorId, taskId, state, data)` comes.
- [MesosSchedulerBackend](#) when `org.apache.mesos.Scheduler.statusUpdate` is called.
- [LocalEndpoint](#) when `StatusUpdate(taskId, state, serializedData)` comes.

When `statusUpdate` starts, it checks the current state of the task and act accordingly.

If a task became `TaskState.LOST` and there is still an executor assigned for the task (it seems it may not given the check), the executor is marked as lost (or sometimes called failed). The executor is later announced as such using `DAGScheduler.executorLost` with [SchedulerBackend.reviveOffers\(\)](#) being called afterwards.

Caution	FIXME Why is <code>SchedulerBackend.reviveOffers()</code> called only for lost executors?
---------	---

Caution	FIXME Review <code>TaskSchedulerImpl.removeExecutor</code>
---------	--

The method looks up the [TaskSetManager](#) for the task (using `taskIdToTaskSetManager`).

When the TaskSetManager is found and the task is in finished state, the task is removed from the internal data structures, i.e. `taskIdToTaskSetManager` and `taskIdToExecutorId`, and the number of currently running tasks for the executor(s) is decremented (using `executorIdToTaskCount`).

For a `FINISHED` task, `TaskSet.removeRunningTask` is called and then `TaskResultGetter.enqueueSuccessfulTask`.

For a task in `FAILED`, `KILLED`, or `LOST` state, `TaskSet.removeRunningTask` is called (as for the `FINISHED` state) and then `TaskResultGetter.enqueueFailedTask`.

If the `TaskSetManager` could not be found, the following ERROR shows in the logs:

```
ERROR Ignoring update with state [state] for TID [tid] because its task set is gone (this
```

◀

▶

TaskSchedulerImpl.handleFailedTask

`TaskSchedulerImpl.handleFailedTask(taskSetManager: TaskSetManager, tid: Long, taskState: TaskState, reason: TaskEndReason)` is called when `TaskResultGetter.enqueueSuccessfulTask` failed to fetch bytes from BlockManager or as part of `TaskResultGetter.enqueueFailedTask`.

Either way there is an error related to task execution.

It calls `TaskSetManager.handleFailedTask`.

If the `TaskSetManager` is not a `zombie` and the task's state is not `KILLED`, `SchedulerBackend.reviveOffers` is called.

TaskSchedulerImpl.taskSetFinished

`taskSetFinished(manager: TaskSetManager)` method is called to inform `TaskSchedulerImpl` that all tasks in a `TaskSetManager` have finished execution.

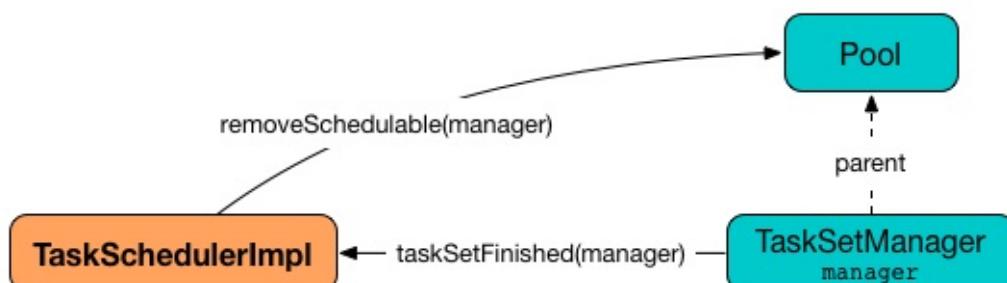


Figure 5. `TaskSchedulerImpl.taskSetFinished` is called when all tasks are finished

Note

`taskSetFinished` is called by `TaskSetManager` at the very end of `TaskSetManager.handleSuccessfulTask`.

`taskSetsByStageIdAndAttempt` internal mapping is queried by stage id (using `manager.taskSet.stageId`) for the corresponding `TaskSets` (`TaskSetManagers` in fact) to remove the currently-finished stage attempt (using `manager.taskSet.stageAttemptId`) and if it

was the only attempt, the stage id is completely removed from `taskSetsByStageIdAndAttempt`.

Note

A TaskSetManager owns a TaskSet that corresponds to a stage.

`Pool.removeSchedulable(manager)` is called for the `parent` of the TaskSetManager.

You should see the following INFO message in the logs:

```
INFO Removed TaskSet [manager.taskSet.id], whose tasks have all completed, from pool [man
```

Scheduling Modes

The scheduling mode in `TaskSchedulerImpl` is configured by `spark.scheduler.mode` setting that can be one of the following values:

- **FIFO** with no pools; one root pool with instances of `TaskSetManager`; lower priority gets `Schedulable` sooner or earlier stage wins.
- **FAIR**
- **NONE** means no sub-queues

See [SchedulableBuilder](#).

SchedulableBuilder

Caution**FIXME**

`SchedulableBuilder` offers the following methods:

- `rootPool` to return a `Pool`.
- `buildPools()`
- `addTaskSetManager(manager: Schedulable, properties: Properties)`

Note

`SchedulableBuilder.addTaskSetManager` is called by `TaskSchedulerImpl.submitTasks` when a TaskSet is submitted for execution.

There are two implementations available (click the links to go to their sections):

- [FIFOSchedulableBuilder](#)
- [FairSchedulableBuilder](#)

FIFOSchedulableBuilder

`FIFOSchedulableBuilder` is a *very basic* `SchedulableBuilder`.

- `rootPool` is given as a mandatory input parameter to the constructor.
- `buildPools` does nothing.
- `addTaskSetManager(manager: Schedulable, properties: Properties)` adds the `manager` `Schedulable` to `rootPool` (using `Pool.addSchedulable`).

FairSchedulableBuilder

Caution	FIXME
---------	-----------------------

TaskSchedulerImpl.executorAdded

`executorAdded(execId: String, host: String)` method simply passes the notification along to DAGScheduler (using [DAGScheduler.executorAdded](#))

Caution	FIXME Image with a call from TaskSchedulerImpl to DAGScheduler, please.
---------	---

Internal Registries

Caution	FIXME How/where are these mappings used?
---------	--

TaskSchedulerImpl tracks the following information in its internal data structures:

- the number of `tasks` already scheduled for execution (`nextTaskId`).
- `TaskSets` by stage and attempt ids (`taskSetsByStageIdAndAttempt`)
- `tasks` to their `TaskSetManagers` (`taskIdToTaskSetManager`)
- `tasks` to `executors` (`taskIdToExecutorId`)
- the number of `tasks` running per `executor` (`executorIdToTaskCount`)
- the set of `executors` on each host (`executorsByHost`)
- the set of hosts per rack (`hostsByRack`)
- executor ids to corresponding host (`executorIdToHost`).

Settings

- `spark.task.maxFailures` (default: 4 for [cluster mode](#) and 1 for [local](#) except [local-with-retries](#)) - The number of individual task failures before giving up on the entire TaskSet and the job afterwards.

It is used in [TaskSchedulerImpl](#) to initialize [TaskSetManager](#).

- `spark.task.cpus` (default: 1) - how many CPUs to request per task in a [SparkContext](#). You cannot have different number of CPUs per task in a single [SparkContext](#).
- `spark.scheduler.mode` (default: FIFO) can be of any of FAIR, FIFO, or NONE. Refer to [Scheduling Modes](#).
- `spark.speculation.interval` (default: 100ms) - how often to check for speculative tasks.
- `spark.starvation.timeout` (default: 15s) - Threshold above which Spark warns a user that an initial TaskSet may be starved.
- `spark.resultGetter.threads` (default: 4) - the number of threads for [TaskResultGetter](#).

Scheduler Backends

Introduction

Spark comes with a pluggable backend mechanism called **scheduler backend** (aka *backend scheduler*) to support various cluster managers, e.g. [Apache Mesos](#), [Hadoop YARN](#) or Spark's own [Spark Standalone](#) and [Spark local](#).

These cluster managers differ by their custom task scheduling modes and resource offers mechanisms, and Spark's approach is to abstract the differences in [SchedulerBackend Contract](#).

A scheduler backend is created and started as part of SparkContext's initialization (when [TaskSchedulerImpl](#) is started - see [Creating Scheduler Backend and Task Scheduler](#)).

Caution	FIXME Image how it gets created with SparkContext in play here or in SparkContext doc.
---------	--

Scheduler backends are started and stopped as part of TaskSchedulerImpl's initialization and stopping.

Being a scheduler backend in Spark assumes a [Apache Mesos](#)-like model in which "an application" gets **resource offers** as machines become available and can launch tasks on them. Once a scheduler backend obtains the resource allocation, it can start executors.

Tip	Understanding how Apache Mesos works can greatly improve understanding Spark.
-----	---

SchedulerBackend Contract

Note	<code>org.apache.spark.scheduler.SchedulerBackend</code> is a <code>private[spark]</code> Scala trait in Spark.
------	---

Every scheduler backend has to offer the following services:

- Can be started (using `start()`) and stopped (using `stop()`).
- Do [reviveOffers](#)
- Calculate [default level of parallelism](#).
- Be able to `killTask(taskId: Long, executorId: String, interruptThread: Boolean)`.
- Answers `isReady()` to tell about its started/stopped state. It returns `true` by default.

- Knows the application id for a job (using `applicationId()`).

Caution

FIXME `applicationId()` doesn't accept an input parameter. How is Scheduler Backend related to a job and an application?

- Knows an application attempt id (see [applicationAttemptId](#))
- Knows the URLs for the driver's logs (see [getDriverLogUrls](#)).

Caution

FIXME Screenshot the tab and the links

Caution

FIXME Have an exercise to create a SchedulerBackend.

reviveOffers

Note

It is used in `TaskSchedulerImpl` using `backend` internal reference.

Refer to [Task Submission a.k.a. reviveOffers](#) for local mode.

Refer to [reviveOffers](#) for CoarseGrainedSchedulerBackend.

Refer to [reviveOffers](#) for MesosSchedulerBackend.

No other custom implementations of `defaultParallelism()` exists.

Default Level of Parallelism

Default level of parallelism is used by [TaskScheduler](#) to use as a hint for sizing jobs.

Note

It is used in `TaskSchedulerImpl.defaultParallelism` .

Refer to [LocalBackend](#) for local mode.

Refer to [Default Level of Parallelism](#) for CoarseGrainedSchedulerBackend.

Refer to [Default Level of Parallelism](#) for CoarseMesosSchedulerBackend.

No other custom implementations of `defaultParallelism()` exists.

applicationAttemptId

`applicationAttemptId(): Option[String]` returns no application attempt id.

It is currently only supported by YARN cluster scheduler backend as the YARN cluster manager supports multiple attempts.

getDriverLogUrls

`getDriverLogUrls: Option[Map[String, String]]` returns no URLs by default.

It is currently only supported by [YarnClusterSchedulerBackend](#).

Available Implementations

Spark comes with the following scheduler backends:

- [LocalBackend](#) (local mode)
- [CoarseGrainedSchedulerBackend](#)
 - **SparkDeploySchedulerBackend** used in [Spark Standalone](#) (and local-cluster - [FIXME](#))
 - [YarnSchedulerBackend](#)
 - [YarnClientSchedulerBackend](#)
 - **YarnClusterSchedulerBackend** used in [Spark on YARN](#) in cluster mode
 - [CoarseMesosSchedulerBackend](#)
 - [SimrSchedulerBackend](#)
- [MesosSchedulerBackend](#)

CoarseGrainedSchedulerBackend

CoarseGrainedSchedulerBackend is a [scheduler backend](#) that waits for [coarse-grained executors](#) to connect to it (using [RPC Environment](#)) and [launch tasks](#). It talks to a cluster manager for resources for executors (register, remove).

This backend holds executors for the duration of the Spark job rather than relinquishing executors whenever a task is done and asking the scheduler to launch a new executor for each new task.

It requires a [Task Scheduler](#), and a [RPC Environment](#).

It uses [Listener Bus](#).

It registers [CoarseGrainedScheduler RPC Endpoint](#) that executors use for RPC communication.

It tracks:

- the total number of cores in the cluster (using `totalCoreCount`)
- the total number of executors that are currently registered
- executors (`ExecutorData`)
- executors to be removed (`executorsPendingToRemove`)
- hosts and the number of possible tasks possibly running on them
- lost executors with no real exit reason
- tasks per slaves (`taskIdsOnSlave`)

Caution

[FIXME](#) Where are these counters used?

Tip Enable `DEBUG` logging level for `org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend` logger to see what happens inside.

Tip Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend=DE
```

Launching Tasks

`launchTasks` gets called when making offers (using `makeOffers` method).

Caution

[FIXME](#) Why is there `Seq[Seq[TaskDescription]]` ?

Tasks are serialized and their size checked to fit [spark.akka.frameSize](#).

If the serialized task's size is over the frame size, the TaskSetManager of the task is called using [TaskSetManager.abort](#) method. It is [TaskSchedulerImpl](#) to be queried for the TaskSetManager of the task.

Caution

[FIXME](#) At that point, tasks have their executor assigned. When and how did that happen?

The number of free cores of the executor for the task is decremented (by [spark.task.cpus](#)) and [LaunchTask](#) message is sent to it.

Default Level of Parallelism

The default parallelism is controlled by [spark.default.parallelism](#) or is at least `2` or `totalCoreCount` .

reviveOffers

It sends [ReviveOffers](#) message to `driverEndpoint` .

Caution

[FIXME](#) Image

Stopping

`stop()` method stops [CoarseGrainedScheduler RPC Endpoint](#) and executors (using `stopExecutors()`).

Note

When called with no `driverEndpoint` both `stop()` and `stopExecutors()` do nothing. `driverEndpoint` is initialized in `start` and the initialization order matters.

It prints INFO to the logs:

```
INFO Shutting down all executors
```

It then sends [StopExecutors](#) message to `driverEndpoint` . It disregards the response.

It sends [StopDriver](#) message to `driverEndpoint` . It disregards the response.

isReady - Delaying Task Launching

`CoarseGrainedSchedulerBackend` comes with its own implementation of `isReady()` method (see [SchedulerBackend Contract](#)).

It starts by ensuring that sufficient resources are available to launch tasks using its own `sufficientResourcesRegistered` method. It gives custom implementations of itself a way to handle the question properly. It assumes resources are available by default.

It prints out the following INFO to the logs:

```
INFO SchedulerBackend is ready for scheduling beginning after reached minRegisteredResour
```

`minRegisteredRatio` in the logs above is in the range 0 to 1 (uses `spark.scheduler.minRegisteredResourcesRatio`) to denote the minimum ratio of registered resources to total expected resources before submitting tasks.

In case there are no sufficient resources available yet (the above requirement does not hold), it checks whether the time from the startup passed `spark.scheduler.maxRegisteredResourcesWaitingTime` to give a way to submit tasks (despite `minRegisteredRatio` not being reached yet).

You should see the following INFO in the logs:

```
INFO SchedulerBackend is ready for scheduling beginning after waiti
```

Otherwise, it keeps delaying task launching until enough resources register or `spark.scheduler.maxRegisteredResourcesWaitingTime` passes.

CoarseGrainedScheduler RPC Endpoint

When `CoarseGrainedSchedulerBackend` starts, it registers **CoarseGrainedScheduler** RPC endpoint (`driverEndpoint` internal field).

It is called **standalone scheduler's driver endpoint** internally.

Caution

[FIXME](#) Why is the field's name `driverEndpoint` ?

It tracks:

- Executor addresses (host and port) for executors (`addressToExecutorId`) - it is set when an executor connects to register itself. See [RegisterExecutor](#) RPC message.

- Total number of core count (`totalCoreCount`) - the sum of all cores on all executors.
See [RegisterExecutor](#) RPC message.
- The number of executors available (`totalRegisteredExecutors`). See [RegisterExecutor](#) RPC message.
- `ExecutorData` for each registered executor (`executorDataMap`). See [RegisterExecutor](#) RPC message.

It uses `driver-revive-thread` daemon single-thread thread pool for ...[FIXME](#)

Caution

[FIXME](#) A potential issue with
`driverEndpoint.asInstanceOf[NettyRpcEndpointRef].toURI` - doubles
`spark://` prefix.

- `spark.scheduler.revive.interval` (default: `1s`) - time between reviving offers.

RPC Messages

KillTask(taskId, executorId, interruptThread)

RemoveExecutor

RetrieveSparkProps

ReviveOffers

`ReviveOffers` calls `makeOffers()` that makes fake resource offers on all executors that are alive.

Caution

[FIXME](#) When is an executor alive? What other states can an executor be in?

Caution

[FIXME](#) What is **fake resource offers**?

A collection of `workerOffer` for each active executor is offered to `TaskSchedulerImpl` (refer to [resourceOffers](#) in `TaskSchedulerImpl`).

It then [launches tasks](#) for the resource offers.

StatusUpdate(executorId, taskId, state, data)

StopDriver

`StopDriver` message stops the RPC endpoint.

StopExecutors

`StopExecutors` message is receive-reply and blocking. When received, the following INFO message appears in the logs:

```
INFO Asking each executor to shut down
```

It then sends a `StopExecutor` message to every registered executor (from `executorDataMap`).

RegisterExecutor

`RegisterExecutor(executorId, executorRef, hostPort, cores, logUrls)` is sent by `CoarseGrainedExecutorBackend` to register itself.

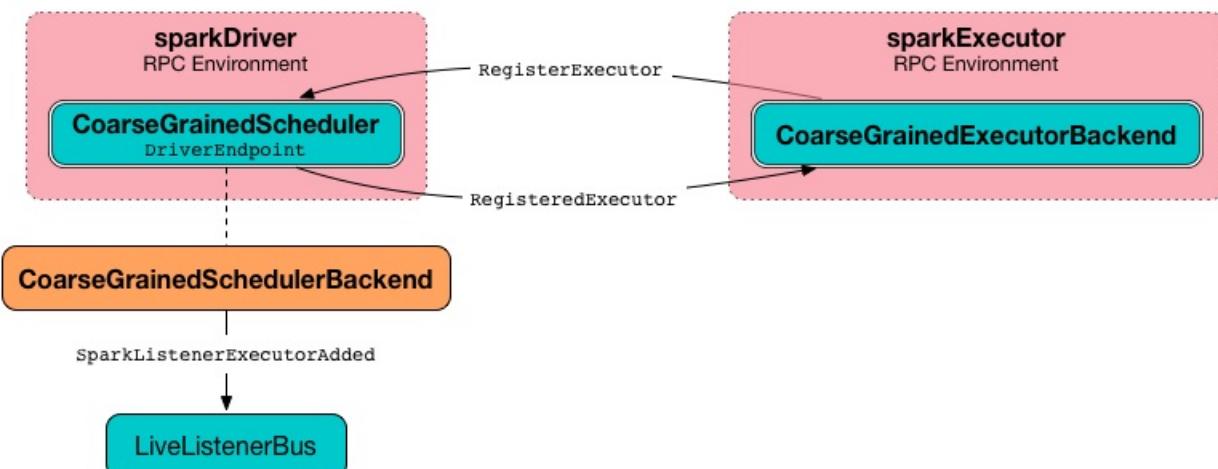


Figure 1. Executor registration (RegisterExecutor RPC message flow)

Only one executor can register as `executorId`.

```
INFO Registered executor [executorRef] ([executorAddress]) with ID [executorId]
```

It does internal bookkeeping like updating `addressToExecutorId`, `totalCoreCount`, and `totalRegisteredExecutors`, `executorDataMap`.

When `numPendingExecutors` is more than `0`, the following is printed out to the logs:

```
DEBUG Decrement number of pending executors ([numPendingExecutors] left)
```

It replies with `RegisteredExecutor(executorAddress.host)` (consult [RPC Messages](#) of `CoarseGrainedExecutorBackend`).

It then announces the new executor by posting [SparkListenerExecutorAdded](#) on [Listener Bus](#).

`makeOffers` is called. It is described as "*Make fake resource offers on all executors*".

Caution

[FIXME](#) What are **fake resource offers**? Review `makeOffers` in `DriverEndpoint`.

Settings

- `spark.akka.frameSize` (default: `128` and not greater than `2047m` - `200k` for extra data in an Akka message) - largest frame size for Akka messages (serialized tasks or task results) in MB.
- `spark.default.parallelism` (default: maximum of `totalCoreCount` and 2) - [default parallelism](#) for the scheduler backend.
- `spark.scheduler.minRegisteredResourcesRatio` (default: `0`) - a double value between 0 and 1 (including) that controls the minimum ratio of (registered resources / total expected resources) before submitting tasks. See [isReady](#).
- `spark.scheduler.maxRegisteredResourcesWaitingTime` (default: `30s`) - the time to wait for sufficient resources available. See [isReady](#).

Executor Backends

Executor Backend is a pluggable interface used by [executors](#) to send status updates to a cluster scheduler.

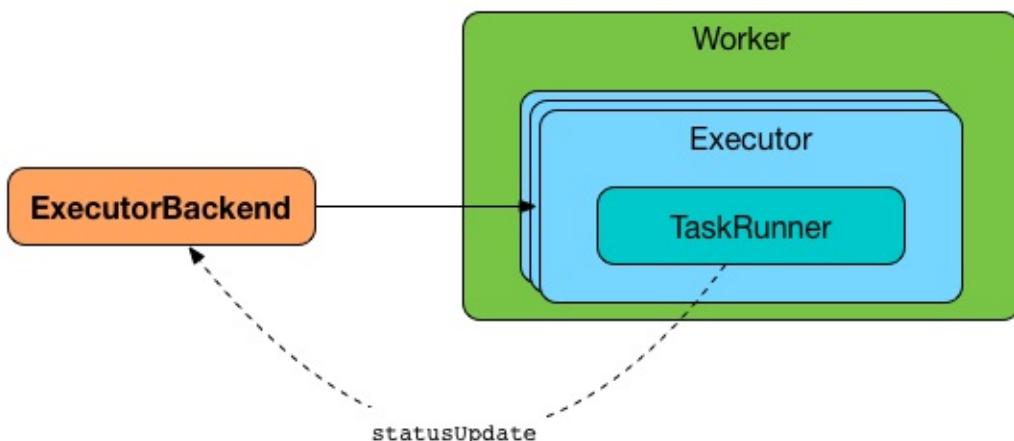


Figure 1. ExecutorBackends work on executors and communicate with driver

The interface comes with one method:

```
def statusUpdate(taskId: Long, state: TaskState, data: ByteBuffer)
```

It is effectively a bridge between the driver and an executor, i.e. there are two endpoints running.

Caution	FIXME What is cluster scheduler? Where is ExecutorBackend used?
---------	---

Status updates include information about tasks, i.e. id, [state](#), and data (as [ByteBuffer](#)).

At startup, an executor backend connects to the driver and creates an executor. It then launches and kills tasks. It stops when the driver orders so.

There are the following types of executor backends:

- [LocalBackend](#) (local mode)
- [CoarseGrainedExecutorBackend](#)
- [MesosExecutorBackend](#)

MesosExecutorBackend

Caution	FIXME
---------	-----------------------

CoarseGrainedExecutorBackend

CoarseGrainedExecutorBackend is an [executor backend](#) for coarse-grained executors that live until it terminates.

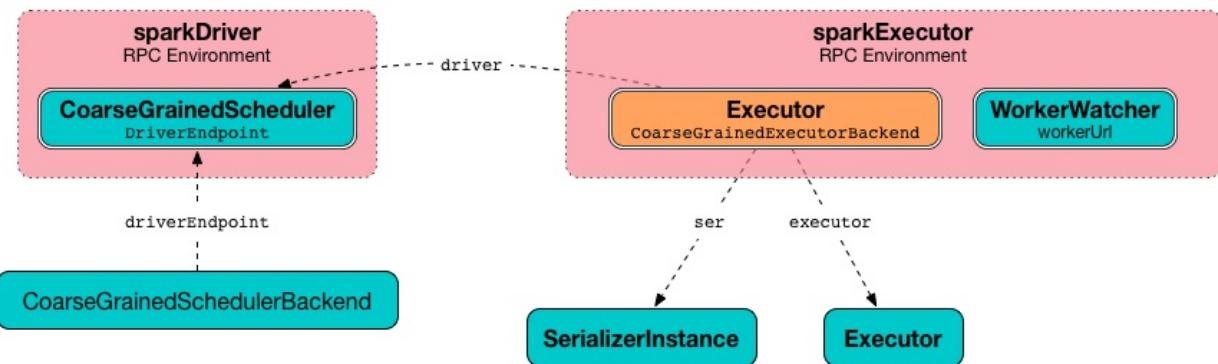


Figure 1. CoarseGrainedExecutorBackend and Others

CoarseGrainedExecutorBackend registers itself as a [RPC Endpoint](#) under the name **Executor**.

When started it connects to `driverUrl` (given as [an option on command line](#)), i.e. [CoarseGrainedSchedulerBackend](#), for tasks to run.

After the connection to driver is established ([RegisteredExecutor](#)), it spawns an [executor](#).

Caution	What are <code>RegisterExecutor</code> and <code>RegisterExecutorResponse</code> ? Why does <code>CoarseGrainedExecutorBackend</code> send it in <code>onStart</code> ?
---------	---

When it cannot connect to `driverUrl`, it terminates (with the exit code `1`).

Caution	What are <code>SPARK_LOG_URL</code> env vars? Who sets them?
---------	--

When the driver terminates, CoarseGrainedExecutorBackend exits (with exit code `1`).

ERROR Driver [remoteAddress] disassociated! Shutting down.
--

When **Executor** RPC Endpoint is started (`onstart`), it prints out INFO message to the logs:

INFO CoarseGrainedExecutorBackend: Connecting to driver: [driverUrl]
--

All task status updates are sent along to `driverRef` as `statusUpdate` messages.

Caution	<p>FIXME Review the use of:</p> <ul style="list-style-type: none"> • Used in <code>SparkContext.createTaskScheduler</code>
Tip	<p>Enable <code>INFO</code> logging level for <code>org.apache.spark.executor.CoarseGrainedExecutorBackend</code> logger to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code>:</p> <pre>log4j.logger.org.apache.spark.executor.CoarseGrainedExecutorBackend=INFO</pre>

Driver's URL

The driver's URL is of the format `spark://[RpcEndpoint name]@[hostname]:[port]`, e.g.
`spark://CoarseGrainedScheduler@192.168.1.6:64859`.

main

`CoarseGrainedExecutorBackend` is a command-line application (it comes with `main` method).

It accepts the following options:

- `--driver-url` (required) - the driver's URL. See [driver's URL](#).
- `--executor-id` (required) - the executor's id
- `--hostname` (required) - the name of the host
- `--cores` (required) - the number of cores (must be more than `0`)
- `--app-id` (required) - the id of the application
- `--worker-url` - the worker's URL, e.g. `spark://Worker@192.168.1.6:64557`
- `--user-class-path` - a URL/path to a resource to be added to CLASSPATH; can be specified multiple times.

Unrecognized options or required options missing cause displaying usage help and exit.

```
$ ./bin/spark-class org.apache.spark.executor.CoarseGrainedExecutorBackend

"Usage: CoarseGrainedExecutorBackend [options]

Options are:
--driver-url <driverUrl>
--executor-id <executorId>
--hostname <hostname>
--cores <cores>
--app-id <appid>
--worker-url <workerUrl>
--user-class-path <url>
```

It first fetches Spark properties from [CoarseGrainedSchedulerBackend](#) (using the `driverPropsFetcher` RPC Environment and the endpoint reference given in [driver's URL](#)).

For this, it creates `sparkConf`, reads `spark.executor.port` setting (defaults to `0`) and creates the `driverPropsFetcher` RPC Environment in [client mode](#). The RPC environment is used to resolve the driver's endpoint to post `RetrieveSparkProps` message.

It sends a (blocking) `RetrieveSparkProps` message to the driver (using the value for `driverUrl` command-line option). When the response (the driver's `sparkConf`) arrives it adds `spark.app.id` (using the value for `appid` command-line option) and creates a brand new `SparkConf`.

If `spark.yarn.credentials.file` is set, ...[FIXME](#)

A `SparkEnv` is created using [SparkEnv.createExecutorEnv](#) (with `isLocal` being `false`).

Caution	FIXME
---------	-----------------------

Usage

Caution	FIXME Where is <code>org.apache.spark.executor.CoarseGrainedExecutorBackend</code> used?
---------	--

It is used in:

- `SparkDeploySchedulerBackend`
- `CoarseMesosSchedulerBackend`
- `SparkClassCommandBuilder` - ???
- `ExecutorRunnable`

RPC Messages

RegisteredExecutor(hostname)

`RegisteredExecutor(hostname)` is received to confirm successful registration to a driver. This is when `executor` is created.

```
INFO CoarseGrainedExecutorBackend: Successfully registered with driver
```

RegisterExecutorFailed(message)

`RegisterExecutorFailed(message)` is to inform that registration to a driver failed. It exits `CoarseGrainedExecutorBackend` with exit code `1`.

```
ERROR CoarseGrainedExecutorBackend: Slave registration failed: [message]
```

LaunchTask(data)

`LaunchTask(data)` checks whether an executor has been created and prints the following ERROR if not:

```
ERROR CoarseGrainedExecutorBackend: Received LaunchTask command but executor was null
```

Otherwise, it deserializes `TaskDescription` (from `data`).

```
INFO CoarseGrainedExecutorBackend: Got assigned task [taskId]
```

Finally, it launches the task on the executor (calls `Executor.launchTask` method).

KillTask(taskId, _, interruptThread)

`KillTask(taskId, _, interruptThread)` message kills a task (calls `Executor.killTask`).

If an executor has not been initialized yet ([FIXME](#): why?), the following ERROR message is printed out to the logs and `CoarseGrainedExecutorBackend` exits:

```
ERROR Received KillTask command but executor was null
```

StopExecutor

`StopExecutor` message handler is receive-reply and blocking. When received, the handler prints the following INFO message to the logs:

```
INFO CoarseGrainedExecutorBackend: Driver commanded a shutdown
```

It then sends a `Shutdown` message to itself.

Shutdown

`Shutdown` stops the executor, itself and RPC Environment.

Shuffle Manager

Spark comes with a pluggable mechanism for **shuffle systems**.

Shuffle Manager (aka **Shuffle Service**) is a Spark service that tracks [shuffle dependencies](#) for [ShuffleMapStage](#). The driver and executors all have their own Shuffle Service.

The setting [spark.shuffle.manager](#) sets up the default shuffle manager.

The driver registers shuffles with a shuffle manager, and executors (or tasks running locally in the driver) can ask to read and write data.

It is network-addressable, i.e. it is available on a host and port.

There can be many shuffle services running simultaneously and a driver registers with all of them when [CoarseGrainedSchedulerBackend](#) is used.

The service is available under `SparkEnv.get.shuffleManager`.

When [ShuffledRDD](#) is computed it reads partitions from it.

The name appears [here](#), twice in [the build's output](#) and others.

Review the code in `network/shuffle` module.

- When is data eligible for shuffling?
- Get the gist of "*The shuffle files are not currently cleaned up when using Spark on Mesos with the external shuffle service*"

ShuffleManager Contract

Note	org.apache.spark.shuffle.ShuffleManager is a <code>private[spark]</code> Scala trait.
------	---

Every `ShuffleManager` implementation has to offer the following services:

- Is identified by a short name (as `shortName`)
- Registers shuffles so they are addressable by a `shuffleHandle` (using `registerShuffle`)
- Returns a `ShuffleWriter` for a partition (using `getWriter`)
- Returns a `ShuffleReader` for a range of partitions (using `getReader`)
- Removes shuffles (using `unregisterShuffle`)

- Returns a `ShuffleBlockResolver` (using `shuffleBlockResolver`)
- Can be stopped (using `stop`)

Available Implementations

Spark comes with two implementations of `ShuffleManager` contract:

- `org.apache.spark.shuffle.sort.SortShuffleManager` (short name: `sort` or `tungsten-sort`)
- `org.apache.spark.shuffle.hash.HashShuffleManager` (short name: `hash`)

Caution

`FIXME` Exercise for a custom implementation of Shuffle Manager using `private[spark] ShuffleManager trait`.

SortShuffleManager

`SortShuffleManager` is a shuffle manager with the short name being `sort`.

It uses `IndexShuffleBlockResolver` as the `shuffleBlockResolver`.

External Shuffle Service

`org.apache.spark.deploy.ExternalShuffleService` is the external shuffle service.

When `spark.shuffle.service.enabled` is enabled, the service provides a mechanism to manage shuffle output files so they are available for executors. It offers an uninterrupted access to the shuffle output files regardless of executors being killed or down.

Tip

Enable `INFO` logging level for `org.apache.spark.deploy.ExternalShuffleService` logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.deploy.ExternalShuffleService=INFO
```

You can start an external shuffle service instance using the following command:

```
./bin/spark-class org.apache.spark.deploy.ExternalShuffleService
```

When the server starts, you should see the following INFO message in the logs:

```
INFO ExternalShuffleService: Starting shuffle service on port [port] with useSasl = [useS
```

Enable `TRACE` logging level for
`org.apache.spark.network.shuffle.ExternalShuffleBlockHandler` logger to see what happens inside.

Tip Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.network.shuffle.ExternalShuffleBlockHandler=TRACE
```

Enable `DEBUG` logging level for
`org.apache.spark.network.shuffle.ExternalShuffleBlockResolver` logger to see what happens inside.

Tip Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.network.shuffle.ExternalShuffleBlockResolver=DEBUG
```

You should see the following INFO message in the logs:

```
INFO ExternalShuffleBlockResolver: Registered executor [AppExecId] with [executorInfo]
```

You should also see the following messages when a SparkContext is closed:

```
INFO ExternalShuffleBlockResolver: Application [appId] removed, cleanupLocalDirs = [clean
INFO ExternalShuffleBlockResolver: Cleaning up executor [AppExecId]'s [executor.localDirs
DEBUG ExternalShuffleBlockResolver: Successfully cleaned up directory: [localDir]
```

Settings

- `spark.shuffle.manager` (default: `sort`) sets up the default shuffle manager by a short name or the fully-qualified class name of a custom implementation.
- `spark.shuffle.service.enabled` (default: `false`) controls whether an external shuffle service should be used. When enabled, the Spark driver will register with the shuffle service. See [External Shuffle Service](#).

It is used in [CoarseMesosSchedulerBackend](#) to instantiate

```
MesosExternalShuffleClient .
```

- `spark.shuffle.service.port` (default: `7337`)

- `spark.shuffle.spill` (default: `true`) - no longer in used, and when `false` the following WARNING shows in the logs:

```
WARN SortShuffleManager: spark.shuffle.spill was set to false, but this configuration
```



Block Manager

Block Manager is a key-value store for blocks that acts as a cache. It runs on every node, i.e. a driver and executors, in a Spark runtime environment. It provides interfaces for putting and retrieving blocks both locally and remotely into various stores, i.e. memory, disk, and off-heap.

A BlockManager manages the storage for most of the data in Spark, i.e. block that represent a cached RDD partition, intermediate shuffle data, broadcast data, etc. See [BlockId](#).

It is created when a Spark application starts, i.e. as part of [SparkEnv.create\(\)](#).

A BlockManager must be [initialized](#) before it is fully operable.

A BlockManager relies on the following services:

- [RpcEnv](#)
- [BlockManagerMaster](#)
- [Serializer](#)
- [MemoryManager](#)
- [MapOutputTracker](#)
- [ShuffleManager](#)
- [BlockTransferService](#)
- [SecurityManager](#)

BlockManager is a [BlockDataManager](#).

Caution

[FIXME](#) Review [BlockDataManager](#)

Enable `TRACE` logging level for `org.apache.spark.storage.BlockManager` logger to see what happens in BlockManager.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.storage.BlockManager=TRACE
```

Tip

BlockManager.initialize

`initialize(appId: String)` method is called to initialize a BlockManager instance.

Note	The method must be called before a BlockManager can be fully operable.
------	--

It does the following:

- It initializes [BlockTransferService](#)
- It initializes a shuffle client, be it [ExternalShuffleClient](#) or [BlockTransferService](#).
- It creates an instance of [BlockManagerId](#) given an executor id, host name and port for [BlockTransferService](#).
- It creates the address of the server that serves this executor's shuffle files (using `shuffleServerId`)

If an external shuffle service is used, the following INFO appears in the logs:

```
INFO external shuffle service port = [externalShuffleServicePort]
```

- It registers itself to [BlockManagerMaster](#) (using [BlockManagerMaster.registerBlockManager](#)).
- At the end, if an external shuffle service is used, and it is not a driver, it registers to the external shuffle service.

While registering to the external shuffle service, you should see the following INFO message in the logs:

```
INFO Registering executor with local external shuffle service.
```

Using `shuffleClient` (that is [ExternalShuffleClient](#)) it calls `registerWithShuffleServer` synchronously using `shuffleServerId` and a [ExecutorShuffleInfo](#) (based on [DiskBlockManager](#) for the executor and the short name of [ShuffleManager](#)).

Any issues while connecting to the external shuffle service are reported as ERROR messages in the logs:

```
ERROR Failed to connect to external shuffle server, will retry [attempts] more times after
```

BlockManagerSlaveEndpoint

`BlockManagerSlaveEndpoint` is a RPC endpoint for remote communication between workers and the driver.

When a BlockManager is created, it sets up the RPC endpoint with the name `BlockManagerEndpoint[randomId]` and `BlockManagerSlaveEndpoint` as the class to handle [RPC messages](#).

RPC Messages

	<p>Enable <code>DEBUG</code> logging level for <code>org.apache.spark.storage.BlockManagerSlaveEndpoint</code> logger to see what happens in <code>BlockManagerSlaveEndpoint</code>.</p> <p>Tip Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.storage.BlockManagerSlaveEndpoint=DEBUG</pre>
--	--

`BlockManagerSlaveEndpoint` accepts the following RPC messages. The processing is slow and hence is deliberately done asynchronously (on a separate thread).

- `RemoveBlock(blockId)` to remove a block `blockId`. It calls `BlockManager.removeBlock`.
- `RemoveRdd(rddId)` to remove a RDD `rddId`. It calls `BlockManager.removeRdd`.
- `RemoveShuffle(shuffleId)` to remove a shuffle `shuffleId`. It unregisters it from `MapOutputTracker` if available (using `MapOutputTracker.unregisterShuffle`). It calls `ShuffleManager` to unregister the shuffle (using `ShuffleManager.unregisterShuffle`).
- `RemoveBroadcast(broadcastId, _)` to remove a broadcast `broadcastId`. It calls `BlockManager.removeBroadcast`.
- `GetBlockStatus(blockId, _)` to return the status of a block `blockId` (using `BlockManager.getStatus`).
- `GetMatchingBlockIds(filter, _)` to return the matching block ids for `filter` (using `BlockManager.getMatchingBlockIds`).
- `TriggerThreadDump` to get a thread dump of all threads (using `utils.getThreadDump()`).

BlockTransferService

Caution	FIXME
---------	-----------------------

ExternalShuffleClient

Caution	FIXME
---------	-----------------------

BlockId

BlockId identifies a block of data. It has a globally unique identifier (`name`)

There are the following types of `BlockId` :

- **RDDBlockId** - described by `rddId` and `splitIndex`
- **ShuffleBlockId** - described by `shuffleId`, `mapId` and `reduceId`
- **ShuffleDataBlockId** - described by `shuffleId`, `mapId` and `reduceId`
- **ShuffleIndexBlockId** - described by `shuffleId`, `mapId` and `reduceId`
- **BroadcastBlockId** - described by `broadcastId` and optional `field` - a piece of broadcast value
- **TaskResultBlockId** - described by `taskId`
- **StreamBlockId** - described by `streamId` and `uniqueId`

Broadcast Values

When a new broadcast value is created, `TorrentBroadcast` - the default implementation of `Broadcast` - blocks are put in the block manager. See [TorrentBroadcast](#).

You should see the following `TRACE` message:

```
TRACE Put for block [blockId] took [startTimeMs] to get into synchronized block
```

It puts the data in the memory first and drop to disk if the memory store can't hold it.

```
DEBUG Put block [blockId] locally took [startTimeMs]
```

Stores

A **Store** is the place where blocks are held.

There are the following possible stores:

- `MemoryStore` for memory storage level.
- `DiskStore` for disk storage level.
- `ExternalBlockStore` for OFF_HEAP storage level.

BlockManagerMaster

Caution	FIXME
---------	-------

BlockManagerMaster is the Block Manager that runs on the driver only. It registers itself as `BlockManagerMaster` endpoint in [RPC Environment](#).

BlockManagerMaster.registerBlockManager

Caution	FIXME
---------	-------

BlockManagerMasterEndpoint

Caution	FIXME
---------	-------

BlockManagerMasterEndpoint is the RPC endpoint for [BlockManagerMaster](#) on the master node to track statuses of all slaves' block managers.

The following two-way events are handled:

- RegisterBlockManager
- UpdateBlockInfo
- GetLocations
- GetLocationsMultipleBlockIds
- GetPeers
- GetRpcHostPortForExecutor
- GetMemoryStatus
- GetStorageStatus
- GetBlockStatus
- GetMatchingBlockIds
- RemoveRdd
- RemoveShuffle
- RemoveBroadcast
- RemoveBlock

- RemoveExecutor
- StopBlockManagerMaster
- BlockManagerHeartbeat
- HasCachedBlocks

BlockManagerId

[FIXME](#)

DiskBlockManager

DiskBlockManager creates and maintains the logical mapping between logical blocks and physical on-disk locations.

By default, one block is mapped to one file with a name given by its BlockId. It is however possible to have a block map to only a segment of a file.

Block files are hashed among the directories listed in `spark.local.dir` (or in `SPARK_LOCAL_DIRS` if set).

Caution	FIXME Review me.
---------	----------------------------------

Execution Context

block-manager-future is the execution context for...[FIXME](#)

Metrics

Block Manager uses [Spark Metrics System](#) (via `BlockManagerSource`) to report metrics about internal status.

The name of the source is **BlockManager**.

It emits the following numbers:

- memory / maxMem_MB - the maximum memory configured
- memory / remainingMem_MB - the remaining memory
- memory / memUsed_MB - the memory used
- memory / diskSpaceUsed_MB - the disk used

Misc

The underlying abstraction for blocks in Spark is a `ByteBuffer` that limits the size of a block to 2GB (`Integer.MAX_VALUE` - see [Why does FileChannel.map take up to Integer.MAX_VALUE of data?](#) and [SPARK-1476 2GB limit in spark for blocks](#)). This has implication not just for managed blocks in use, but also for shuffle blocks (memory mapped blocks are limited to 2GB, even though the API allows for `long`), ser-deser via byte array-backed output streams.

When a non-local executor starts, it initializes a Block Manager object for `spark.app.id` `id`.

If a task result is bigger than Akka's message frame size - `spark.akka.frameSize` - executors use the block manager to send the result back. Task results are configured using `spark.driver.maxResultSize` (default: `1g`).

Settings

- `spark.shuffle.service.enabled` (default: `false`) whether an external shuffle service is enabled or not. See [External Shuffle Service](#).
- `spark.broadcast.compress` (default: `true`) whether to compress stored broadcast variables.
- `spark.shuffle.compress` (default: `true`) whether to compress stored shuffle output.
- `spark.rdd.compress` (default: `false`) whether to compress RDD partitions that are stored serialized.
- `spark.shuffle.spill.compress` (default: `true`) whether to compress shuffle output temporarily spilled to disk.

HTTP File Server

It is started on a [driver](#).

Caution	FIXME Review HttpFileServer
---------	---

Settings

- `spark.filesServer.port` (default: `0`) - the port of a file server
- `spark.filesServer.uri` (Spark internal) - the URI of a file server

Broadcast Manager

Broadcast Manager is a Spark service to manage broadcast values in Spark jobs. It is created for a Spark application as part of [SparkContext's initialization](#) and is a simple wrapper around [BroadcastFactory](#).

Broadcast Manager tracks the number of broadcast values (using the internal field `nextBroadcastId`).

The idea is to transfer values used in transformations from a driver to executors in a most effective way so they are copied once and used many times by tasks (rather than being copied every time a task is launched).

When `BroadcastManager` is initialized an instance of `BroadcastFactory` is created based on [spark.broadcast.factory](#) setting.

BroadcastFactory

`BroadcastFactory` is a pluggable interface for broadcast implementations in Spark. It is exclusively used and instantiated inside of `BroadcastManager` to manage broadcast variables.

It comes with 4 methods:

- `def initialize(isDriver: Boolean, conf: SparkConf, securityMgr: SecurityManager): Unit`
- `def newBroadcast[T: ClassTag](value: T, isLocal: Boolean, id: Long): Broadcast[T]` - called after `SparkContext.broadcast()` has been called.
- `def unbroadcast(id: Long, removeFromDriver: Boolean, blocking: Boolean): Unit`
- `def stop(): Unit`

TorrentBroadcast

The `BroadcastManager` implementation used in Spark by default is `org.apache.spark.broadcast.TorrentBroadcast` (see [spark.broadcast.factory](#)). It uses a BitTorrent-like protocol to do the distribution.

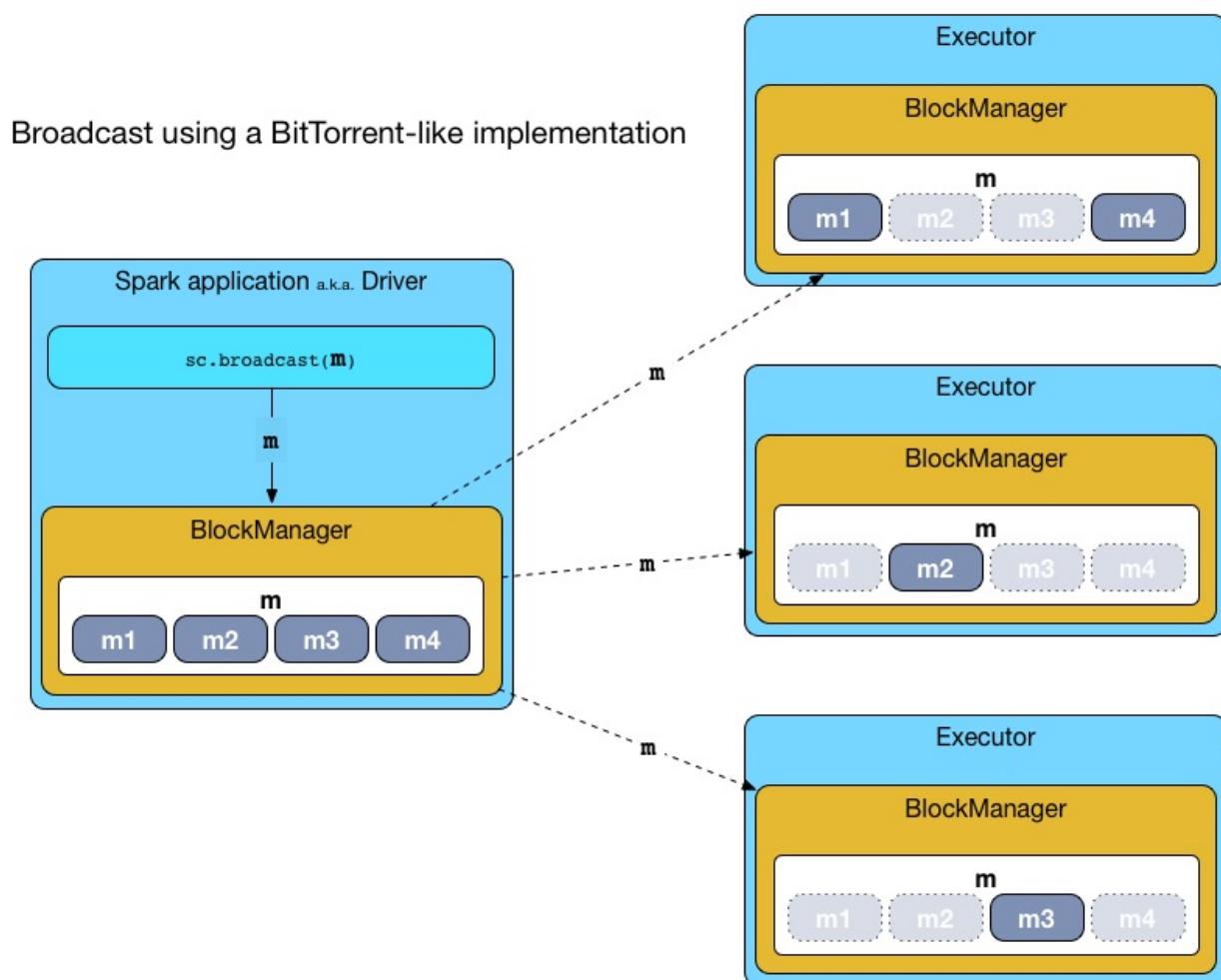


Figure 1. TorrentBroadcast - broadcasting using BitTorrent

`TorrentBroadcastFactory` is the factory of `TorrentBroadcast`-based broadcast values.

When a new broadcast value is created using `SparkContext.broadcast()` method, a new instance of `TorrentBroadcast` is created. It is divided into blocks that are put in [Block Manager](#).

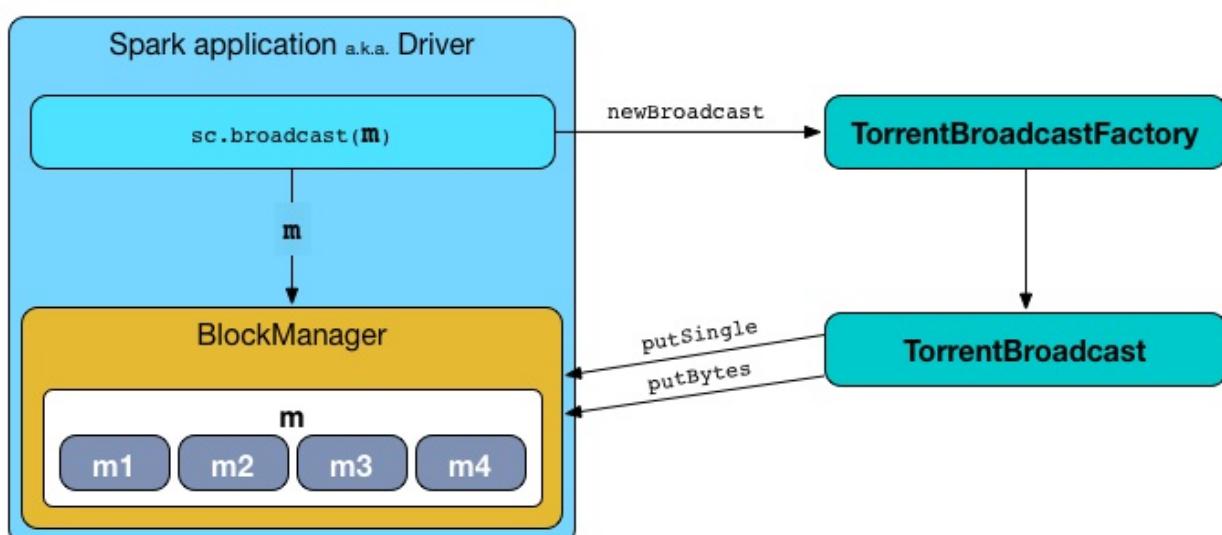


Figure 2. TorrentBroadcast puts broadcast chunks to driver's BlockManager

Compression

When `spark.broadcast.compress` is `true` (default), compression is used.

There are the following compression codec implementations available:

- `lz4` or `org.apache.spark.io.LZ4CompressionCodec`
- `lzf` or `org.apache.spark.io.LZFCompressionCodec` - a fallback when `snappy` is not available.
- `snappy` or `org.apache.spark.io.SnappyCompressionCodec` - the default implementation

An implementation of `CompressionCodec` trait has to offer a constructor that accepts `SparkConf`.

Settings

- `spark.broadcast.factory` (default: `org.apache.spark.broadcast.TorrentBroadcastFactory`) - the fully-qualified class name for the implementation of `BroadcastFactory` interface.
- `spark.broadcast.compress` (default: `true`) - a boolean value whether to use compression or not. See [Compression](#).
- `spark.io.compression.codec` (default: `snappy`) - compression codec to use. See [Compression](#).
- `spark.broadcast.blockSize` (default: `4m`) - the size of a block

Dynamic Allocation

Tip

See the excellent slide deck [Dynamic Allocation in Spark](#) from Databricks.

Dynamic Allocation is an opt-in feature that...[FIXME](#).

It is controlled by `spark.dynamicAllocation.enabled` property. When `--num-executors` or `spark.executor.instances` are set to non-`0` value, they disable dynamic allocation.

- Available since **Spark 1.2.0** with many fixes and extensions up to **1.5.0**.
- Support was first introduced in YARN in 1.2, and then extended to Mesos coarse-grained mode. It is supported in Standalone mode, too.
- In **dynamic allocation** you get as much as needed and no more. It allows to scale the number of executors up and down based on workload, i.e. idle executors are removed, and if you need more executors for pending tasks, you request them.
 - In **static allocation** you reserve resources (CPU, memory) upfront irrespective of how much you really use at a time.
- Scale up / down Policies
 - Exponential increase in number of executors due to slow start and we may need slightly more.
 - Executor removal after N secs

`ExecutorAllocationManager` is the class responsible for the feature. [When enabled](#), it is started when the Spark context is initialized.

Caution

[FIXME](#) Review `ExecutorAllocationManager`

Metrics

Dynamic Allocation feature uses [Spark Metrics System](#) (via `ExecutorAllocationManagerSource`) to report metrics about internal status.

The name of the source is **ExecutorAllocationManager**.

It emits the following numbers:

- `executors / numberExecutorsToAdd`
- `executors / numberExecutorsPendingToRemove`

- executors / numberAllExecutors
- executors / numberTargetExecutors
- executors / numberMaxNeededExecutors

Settings

- `spark.dynamicAllocation.enabled` (default: `false`) - whether dynamic allocation is enabled for the given Spark context. It requires that `spark.executor.instances` (default: `0`) is `0`.

Programmable Dynamic Allocation

- New developer API in `SparkContext`:
 - `def requestExecutors(numAdditionalExecutors: Int): Boolean` to request 5 extra executors
 - `def killExecutors(executorIds: Seq[String]): Boolean` to kill the executors with the IDs.

Future

- SPARK-4922
- SPARK-4751
- SPARK-7955

Data locality / placement

Spark relies on *data locality*, aka *data placement* or *proximity to data source*, that makes Spark jobs sensitive to where the data is located. It is therefore important to have [Spark running on Hadoop YARN cluster](#) if the data comes from HDFS.

With HDFS the Spark driver contacts NameNode about the DataNodes (ideally local) containing the various blocks of a file or directory as well as their locations (represented as `InputSplits`), and then schedules the work to the SparkWorkers.

Spark's compute nodes / workers should be running on storage nodes.

Concept of locality-aware scheduling.

Spark tries to execute tasks as close to the data as possible to minimize data transfer (over the wire).

Tasks										
Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Errors	
0	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/09/11 21:51:04	0 ms			
1	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/09/11 21:51:04	0 ms			
2	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2015/09/11 21:51:04	0 ms			

Figure 1. Locality Level in the Spark UI

There are the following task localities (consult [org.apache.spark.scheduler.TaskLocality](#) object):

- PROCESS_LOCAL
- NODE_LOCAL
- NO_PREF
- RACK_LOCAL
- ANY

Task location can either be a host or a pair of a host and an executor.

Cache Manager

Cache Manager in Spark is responsible for passing RDDs partition contents to [Block Manager](#) and making sure a node doesn't load two copies of [an RDD](#) at once.

It keeps reference to Block Manager.

Caution	FIXME Review the <code>CacheManager</code> class.
---------	---

In the code, the current instance of Cache Manager is available under

`SparkEnv.get.cacheManager`.

Spark, Akka and Netty

From [How does Spark use Netty?](#):

Spark uses Akka Actor for RPC and messaging, which in turn uses Netty.

Also, for moving bulk data, Netty is used.

- For shuffle data, Netty can be optionally used. By default, NIO is directly used to do transfer shuffle data.
- For broadcast data (driver-to-all-worker data transfer), Jetty is used by default.

Tip

Review `org.apache.spark.util.AkkaUtils` to learn about the various utilities using Akka.

- `sparkMaster` is the name of Actor System for the master in Spark Standalone, i.e. `akka://sparkMaster` is the Akka URL.
- Akka configuration is for remote actors (via `akka.actor.provider = "akka.remote.RemoteActorRefProvider"`)
- Enable logging for Akka-related functions in `org.apache.spark.util.Utils` class at `INFO` level.
- Enable logging for RPC messages as `DEBUG` for `org.apache.spark.rpc.akka.AkkaRpcEnv`
- `spark.akka.threads` (default: `4`)
- `spark.akka.batchSize` (default: `15`)
- `spark.akka.timeout` or `spark.network.timeout` (default: `120s`)
- `spark.akka.frameSize` (default: `128`) configures max frame size for Akka messages in bytes
- `spark.akka.logLifecycleEvents` (default: `false`)
- `spark.akka.logAkkaConfig` (default: `true`)
- `spark.akka.heartbeat.pauses` (default: `6000s`)
- `spark.akka.heartbeat.interval` (default: `1000s`)
- Configs starting with `akka.` in properties file are supported.

OutputCommitCoordinator

From the scaladoc (it's a `private[spark]` class so no way to find it [outside the code](#)):

Authority that decides whether tasks can commit output to HDFS. Uses a "first committer wins" policy. OutputCommitCoordinator is instantiated in both the drivers and executors. On executors, it is configured with a reference to the driver's OutputCommitCoordinatorEndpoint, so requests to commit output will be forwarded to the driver's OutputCommitCoordinator.

The most interesting piece is in...

This class was introduced in [SPARK-4879](#); see that JIRA issue (and the associated pull requests) for an extensive design discussion.

RPC Environment (RpcEnv)

FIXME

Caution

- How to know the available endpoints in the environment? See the exercise [Developing RPC Environment](#).

RPC Environment (aka **RpcEnv**) is an environment for RpcEndpoints to process messages. A RPC Environment manages the entire lifecycle of RpcEndpoints:

- registers (sets up) endpoints (by name or uri)
- routes incoming messages to them
- stops them

A RPC Environment is defined by the **name**, **host**, and **port**. It can also be controlled by a **security manager**.

The default implementation of RPC Environment is [Netty-based implementation](#), but Akka-based implementation is still started (perhaps for backward compatibility until no services in Spark use it). Read the section [RpcEnvFactory](#).

RpcEndpoints define how to handle **messages** (what **functions** to execute given a message). RpcEndpoints register (with a name or uri) to RpcEnv to receive messages from **RpcEndpointRefs**.

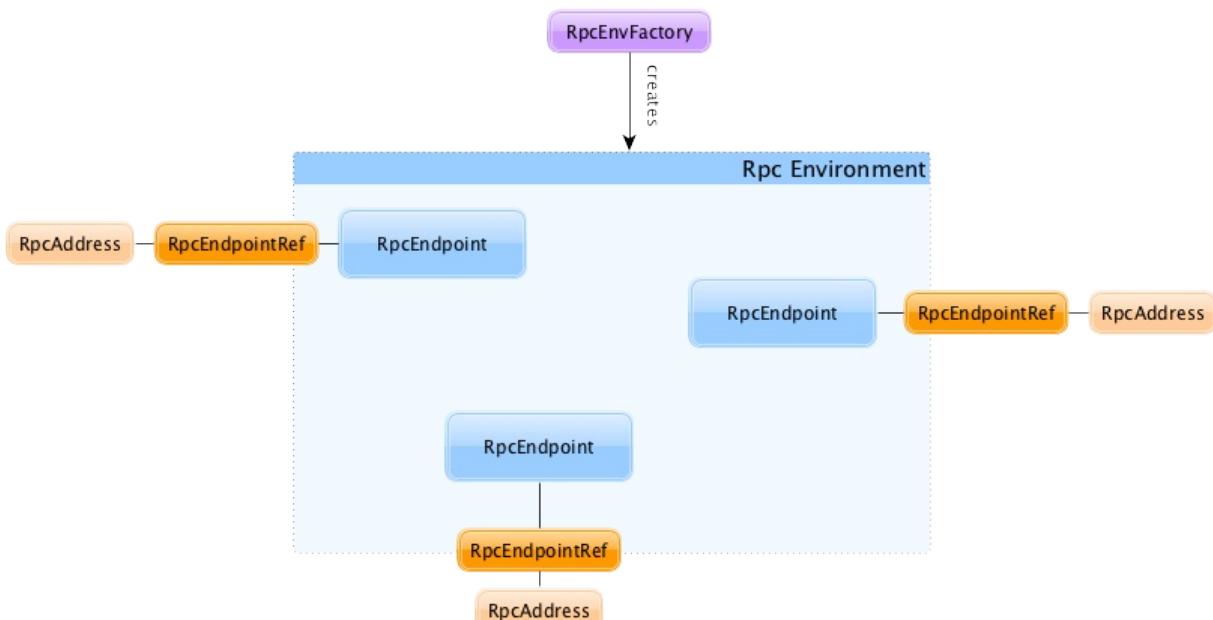


Figure 1. RpcEnvironment with RpcEndpoints and RpcEndpointRefs

RpcEndpointRefs can be looked up by **name** or **uri** (because different RpcEnvs may have different naming schemes).

`org.apache.spark.rpc` package contains the machinery for RPC communication in Spark.

RpcEnvFactory

Spark comes with (`private[spark] trait`) `RpcEnvFactory` which is the factory contract to create a RPC Environment.

An `RpcEnvFactory` implementation has a single method `create(config: RpcEnvConfig): RpcEnv` that returns a `RpcEnv` for a given `RpcEnvConfig`.

There are two `RpcEnvFactory` implementations in Spark:

- `netty` using `org.apache.spark.rpc.netty.NettyRpcEnvFactory`. This is the default factory for `RpcEnv` as of Spark 1.6.0-SNAPSHOT.
- `akka` using `org.apache.spark.rpc.akka.AkkaRpcEnvFactory`

You can choose an RPC implementation to use by `spark.rpc` (default: `netty`). The setting can be one of the two short names for the known `RpcEnvFactories` - `netty` or `akka` - or a fully-qualified class name of your custom factory (including Netty-based and Akka-based implementations).

```
$ ./bin/spark-shell --conf spark.rpc=netty  
$ ./bin/spark-shell --conf spark.rpc=org.apache.spark.rpc.akka.AkkaRpcEnvFactory
```

RpcEndpoint

RpcEndpoints define how to handle **messages** (what **functions** to execute given a message). `RpcEndpoints` live inside `RpcEnv` after being registered by a name.

A `RpcEndpoint` can be registered to one and only one `RpcEnv`.

The lifecycle of a `RpcEndpoint` is `onStart`, `receive` and `onStop` in sequence.

`receive` can be called concurrently. If you want `receive` to be thread-safe, use `ThreadSafeRpcEndpoint`.

`onError` method is called for any exception thrown.

RpcEndpointRef

A **RpcEndpointRef** is a reference for a [RpcEndpoint](#) in a [RpcEnv](#).

It is serializable entity and so you can send it over a network or save it for later use (it can however be deserialized using the owning [RpcEnv](#) only).

A [RpcEndpointRef](#) has [an address](#) (a [Spark URL](#)), and a name.

You can send asynchronous one-way messages to the corresponding [RpcEndpoint](#) using `send` method.

You can send a semi-synchronous message, i.e. "subscribe" to be notified when a response arrives, using `ask` method. You can also block the current calling thread for a response using `askWithRetry` method.

- `spark.rpc.numRetries` (default: `3`) - the number of times to retry connection attempts.
- `spark.rpc.retry.wait` (default: `3s`) - the number of milliseconds to wait on each retry.

It also uses [lookup timeouts](#).

RpcAddress

RpcAddress is the logical address for an RPC Environment, with hostname and port.

[RpcAddress](#) is encoded as a **Spark URL**, i.e. `spark://host:port`.

RpcEndpointAddress

RpcEndpointAddress is the logical address for an endpoint registered to an RPC Environment, with [RpcAddress](#) and **name**.

It is in the format of `spark://[name]@[rpcAddress.host]:[rpcAddress.port]`.

Endpoint Lookup Timeout

When a remote endpoint is resolved, a local RPC environment connects to the remote one. It is called **endpoint lookup**. To configure the time needed for the endpoint lookup you can use the following settings.

It is a prioritized list of **lookup timeout** properties (the higher on the list, the more important):

- `spark.rpc.lookupTimeout`
- `spark.network.timeout`

Their value can be a number alone (seconds) or any number with time suffix, e.g. `50s`, `100ms`, or `250us`. See [Settings](#).

Ask Operation Timeout

Ask operation is when a RPC client expects a response to a message. It is a blocking operation.

You can control the time to wait for a response using the following settings (in that order):

- `spark.rpc.askTimeout`
- `spark.network.timeout`

Their value can be a number alone (seconds) or any number with time suffix, e.g. `50s`, `100ms`, or `250us`. See [Settings](#).

Exceptions

When `RpcEnv` catches uncaught exceptions, it uses `RpcCallContext.sendFailure` to send exceptions back to the sender, or logging them if no such sender or `NotSerializableException`.

If any error is thrown from one of `RpcEndpoint` methods except `onError`, `onError` will be invoked with the cause. If `onError` throws an error, `RpcEnv` will ignore it.

Client Mode = is this an executor or the driver?

When an RPC Environment is initialized [as part of the initialization of the driver or executors](#) (using `RpcEnv.create`), `clientMode` is `false` for the driver and `true` for executors.

```
RpcEnv.create(actorSystemName, hostname, port, conf, securityManager, clientMode = !isDri
```

Refer to [Client Mode](#) in Netty-based `RpcEnv` for the implementation-specific details.

RpcEnvConfig

RpcEnvConfig is a placeholder for an instance of [SparkConf](#), the name of the RPC Environment, host and port, a security manager, and [clientMode](#).

RpcEnv.create

You can create a RPC Environment using the helper method `RpcEnv.create`.

It assumes that you have a [RpcEnvFactory](#) with an empty constructor so that it can be created via Reflection that is available under `spark.rpc` setting.

Settings

- `spark.rpc` (default: `netty` since Spark 1.6.0-SNAPSHOT) - the RPC implementation to use. See [RpcEnvFactory](#).
- `spark.rpc.lookupTimeout` (default: `120s`) - the default timeout to use for RPC remote endpoint lookup. Refer to [Endpoint Lookup Timeout](#).
- `spark.network.timeout` (default: `120s`) - the default network timeout to use for RPC remote endpoint lookup.
- `spark.rpc.numRetries` (default: `3`) - the number of attempts to send a message and receive a response from a remote endpoint.
- `spark.rpc.retry.wait` (default: `3s`) - the time to wait on each retry.
- `spark.rpc.askTimeout` (default: `120s`) - the default timeout to use for RPC ask operations. Refer to [Ask Operation Timeout](#).

Others

The [Worker class](#) calls `startRpcEnvAndEndpoint` with the following configuration options:

- host
- port
- webUiPort
- cores
- memory
- masters
- workDir

It starts `sparkworker[N]` where `N` is the identifier of a worker.

Netty-based RpcEnv

Tip

Read [RPC Environment \(RpcEnv\)](#) about the concept of RPC Environment in Spark.

The class `org.apache.spark.rpc.netty.NettyRpcEnv` is the implementation of `RpcEnv` using `Netty` - *"an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients"*.

Netty-based RPC Environment is created by `NettyRpcEnvFactory` when `spark.rpc` is `netty` or `org.apache.spark.rpc.netty.NettyRpcEnvFactory`.

It uses Java's built-in serialization (the implementation of `JavaSerializerInstance`).

Caution

[FIXME](#) What other choices of `JavaSerializerInstance` are available in Spark?

`NettyRpcEnv` is only started on [the driver](#). See [Client Mode](#).

The default port to listen to is `7077`.

When `NettyRpcEnv` starts, the following INFO message is printed out in the logs:

```
INFO Utils: Successfully started service 'NettyRpcEnv' on port 0.
```

Set `DEBUG` for `org.apache.spark.network.server.TransportServer` logger to know when Shuffle server/`NettyRpcEnv` starts listening to messages.

Tip

`DEBUG` Shuffle server started on port :

[FIXME](#): The message above in `TransportServer` has a space before `:`.

Client Mode

Refer to [Client Mode = is this an executor or the driver?](#) for introduction about **client mode**.

This is only for Netty-based `RpcEnv`.

When created, a Netty-based `RpcEnv` starts the RPC server and register necessary endpoints for non-client mode, i.e. when client mode is `false`.

Caution

[FIXME](#) What endpoints?

It means that the required services for remote communication with **NettyRpcEnv** are only started on the driver (not executors).

Thread Pools

shuffle-server-ID

`EventLoopGroup` uses a daemon thread pool called `shuffle-server-ID`, where `ID` is a unique integer for `NioEventLoopGroup` (`NIO`) or `EpollEventLoopGroup` (`EPOLL`) for the Shuffle server.

Caution

[FIXME](#) Review Netty's `NioEventLoopGroup`.

Caution

[FIXME](#) Where are `SO_BACKLOG`, `SO_RCVBUF`, `SO_SNDBUF` channel options used?

dispatcher-event-loop-ID

NettyRpcEnv's Dispatcher uses the daemon fixed thread pool with `spark.rpc.netty.dispatcher.numThreads` threads.

Thread names are formatted as `dispatcher-event-loop-ID`, where `ID` is a unique, sequentially assigned integer.

It starts the message processing loop on all of the threads.

netty-rpc-env-timeout

NettyRpcEnv uses the daemon single-thread scheduled thread pool `netty-rpc-env-timeout`.

```
"netty-rpc-env-timeout" #87 daemon prio=5 os_prio=31 tid=0x00007f887775a000 nid=0xc503 wa
```

netty-rpc-connection-ID

NettyRpcEnv uses the daemon cached thread pool with up to `spark.rpc.connect.threads` threads.

Thread names are formatted as `netty-rpc-connection-ID`, where `ID` is a unique, sequentially assigned integer.

Settings

The Netty-based implementation uses the following properties:

- `spark.rpc.io.mode` (default: `NIO`) - `NIO` or `EPOLL` for low-level IO. `NIO` is always available, while `EPOLL` is only available on Linux. `NIO` uses `io.netty.channel.nio.NioEventLoopGroup` while `EPOLL` uses `io.netty.channel.epoll.EpollEventLoopGroup`.
- `spark.shuffle.io.numConnectionsPerPeer` always equals `1`
- `spark.rpc.io.threads` (default: `0`; maximum: `8`) - the number of threads to use for the Netty client and server thread pools.
 - `spark.shuffle.io.serverThreads` (default: the value of `spark.rpc.io.threads`)
 - `spark.shuffle.io.clientThreads` (default: the value of `spark.rpc.io.threads`)
- `spark.rpc.netty.dispatcher.numThreads` (default: the number of processors available to JVM)
- `spark.rpc.connect.threads` (default: `64`) - used in cluster mode to communicate with a remote RPC endpoint
- `spark.port.maxRetries` (default: `16` or `100` for testing when `spark.testing` is set) controls the maximum number of binding attempts/retries to a port before giving up.

Endpoints

- `endpoint-verifier` (`RpcEndpointVerifier`) - a `RpcEndpoint` for remote `RpcEnvs` to query whether an `RpcEndpoint` exists or not. It uses `Dispatcher` that keeps track of registered endpoints and responds `true / false` to `CheckExistence` message.

`endpoint-verifier` is used to check out whether a given endpoint exists or not before the endpoint's reference is given back to clients.

One use case is when an [AppClient connects to standalone Masters](#) before it registers the application it acts for.

Caution

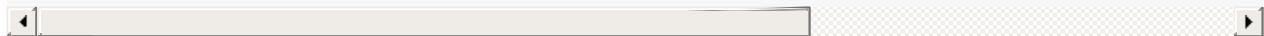
[**FIXME**](#) Who'd like to use `endpoint-verifier` and how?

Message Dispatcher

A message dispatcher is responsible for routing RPC messages to the appropriate endpoint(s).

It uses the daemon fixed thread pool `dispatcher-event-loop` with `spark.rpc.netty.dispatcher.numThreads` threads for dispatching messages.

```
"dispatcher-event-loop-0" #26 daemon prio=5 os_prio=31 tid=0x00007f8877153800 nid=0x7103
```



ContextCleaner

It does cleanup of shuffles, RDDs and broadcasts.

Caution

[FIXME](#) What does the above sentence **really** mean?

It uses a daemon **Spark Context Cleaner** thread that cleans RDD, shuffle, and broadcast states (using `keepCleaning` method).

Caution

[FIXME](#) Review `keepCleaning`

[ShuffleDependencies](#) register themselves for cleanup using

`ContextCleaner.registerShuffleForCleanup` method.

ContextCleaner uses a Spark context.

Settings

- `spark.cleaner.referenceTracking` (default: `true`) controls whether to enable or not ContextCleaner as a [Spark context initializes](#).
- `spark.cleaner.referenceTracking.blocking` (default: `true`) controls whether the cleaning thread will block on cleanup tasks (other than shuffle, which is controlled by the `spark.cleaner.referenceTracking.blocking.shuffle` parameter).

It is `true` as a workaround to [SPARK-3015 Removing broadcast in quick successions causes Akka timeout](#).

- `spark.cleaner.referenceTracking.blocking.shuffle` (default: `false`) controls whether the cleaning thread will block on shuffle cleanup tasks.

It is `false` as a workaround to [SPARK-3139 Akka timeouts from ContextCleaner when cleaning shuffles](#).

MapOutputTracker

A **MapOutputTracker** is a Spark service to track the locations of the (shuffle) map outputs of a stage. It uses an internal MapStatus map with an array of `MapStatus` for every partition for a shuffle id.

There are two versions of `MapOutputTracker` :

- [MapOutputTrackerMaster](#) for a driver
- [MapOutputTrackerWorker](#) for executors

`MapOutputTracker` is available under `SparkEnv.get.mapOutputTracker`. It is also available as `MapOutputTracker` in the driver's RPC Environment.

Tip

Enable `DEBUG` logging level for `org.apache.spark.MapOutputTracker` logger to see what happens in `MapOutputTracker`.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.MapOutputTracker=DEBUG
```

It works with [ShuffledRDD](#) when it asks for **preferred locations for a shuffle** using `tracker.getPreferredLocationsForShuffle`.

It is also used for `mapOutputTracker.containsShuffle` and [MapOutputTrackerMaster.registerShuffle](#) when a new `ShuffleMapStage` is created.

Caution

[FIXME](#) `DAGScheduler.mapOutputTracker`

[MapOutputTrackerMaster.getStatistics\(dependency\)](#) returns `MapOutputStatistics` that becomes the result of `JobWaiter.taskSucceeded` for `ShuffleMapStage` if it's the final stage in a job.

[MapOutputTrackerMaster.registerMapOutputs](#) for a shuffle id and a list of `MapStatus` when a `ShuffleMapStage` is finished.

MapStatus

A **MapStatus** is the result returned by a [ShuffleMapTask](#) to [DAGScheduler](#) that includes:

- the **location** where `ShuffleMapTask` ran (as `def location: BlockManagerId`)

- an **estimated size for the reduce block**, in bytes (as `def getSizeForBlock(reduceId: Int): Long`).

There are two types of MapStatus:

- **CompressedMapStatus** that compresses the estimated map output size to 8 bits (`Byte`) for efficient reporting.
- **HighlyCompressedMapStatus** that stores the average size of non-empty blocks, and a compressed bitmap for tracking which blocks are empty.

When the number of blocks (the size of `uncompressedSizes`) is greater than **2000**, HighlyCompressedMapStatus is chosen.

Caution	FIXME What exactly is 2000? Is this the number of tasks in a job?
---------	---

Caution	FIXME Review ShuffleManager
---------	---

Epoch Number

Caution	FIXME
---------	-----------------------

MapOutputTrackerMaster

A **MapOutputTrackerMaster** is the `MapOutputTracker` for a driver.

A MapOutputTrackerMaster is the source of truth for the collection of `MapStatus` objects (map output locations) per shuffle id (as recorded from `ShuffleMapTasks`).

`MapOutputTrackerMaster` uses Spark's `org.apache.spark.util.TimeStampedHashMap` for `mapStatuses`.

Note	There is currently a hardcoded limit of map and reduce tasks above which Spark does not assign preferred locations aka locality preferences based on map output sizes — <code>1000</code> for map and reduce each.
------	--

It uses `MetadataCleaner` with `MetadataCleanerType.MAP_OUTPUT_TRACKER` as `cleanerType` and `cleanup` function to drop entries in `mapStatuses`.

You should see the following INFO message when the MapOutputTrackerMaster is created ([FIXME](#) it uses `MapOutputTrackerMasterEndpoint`):

INFO SparkEnv: Registering MapOutputTracker

MapOutputTrackerMaster.registerShuffle

Caution

FIXME

MapOutputTrackerMaster.getStatistics

Caution

FIXME

MapOutputTrackerMaster.unregisterMapOutput

Caution

FIXME

MapOutputTrackerMaster.registerMapOutputs

Caution

FIXME

MapOutputTrackerMaster.incrementEpoch

Caution

FIXME

cleanup Function for MetadataCleaner

`cleanup(cleanupTime: Long)` method removes old entries in `mapStatuses` and `cachedSerializedStatuses` that have timestamp earlier than `cleanupTime`.

It uses `org.apache.spark.util.TimeStampedHashMap.clearOldValues` method.

Enable `DEBUG` logging level for `org.apache.spark.util.TimeStampedHashMap` logger to see what happens in `TimeStampedHashMap`.

Tip

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.util.TimeStampedHashMap=DEBUG
```

You should see the following DEBUG message in the logs for entries being removed:

```
DEBUG Removing key [entry.getKey]
```

MapOutputTrackerMaster.getEpoch

Caution	FIXME
---------	-------

Settings

- `spark.shuffle.reduceLocality.enabled` (default: true) - whether to compute locality preferences for reduce tasks.

If `true`, `MapOutputTrackerMaster` computes the preferred hosts on which to run a given map output partition in a given shuffle, i.e. the nodes that the most outputs for that partition are on.

MapOutputTrackerWorker

A **MapOutputTrackerWorker** is the `MapOutputTracker` for executors. The internal `mapStatuses` map serves as a cache and any miss triggers a fetch from the driver's [MapOutputTrackerMaster](#).

Note	The only difference between <code>MapOutputTrackerWorker</code> and the base abstract class <code>MapOutputTracker</code> is that the internal <code>mapStatuses</code> mapping between ints and an array of <code>MapStatus</code> objects is an instance of the thread-safe java.util.concurrent.ConcurrentHashMap .
------	--

ExecutorAllocationManager

[FIXME](#)

Deployment Environments

Spark Deployment Environments:

- [local](#)
- [cluster](#)
 - [Spark Standalone](#)
 - [Spark on Mesos](#)
 - [Spark on YARN](#)

A Spark application can run locally (on a single JVM) or on the cluster which uses a cluster manager and the deploy mode (`--deploy-mode`). See [spark-submit script](#).

Master URLs

Spark supports the following **master URLs** (see [private object SparkMasterRegex](#)):

- **local, local[N] and local[*]** for [Spark local](#)
- **local[N, maxRetries]** for [Spark local-with-retries](#)
- **local-cluster[N, cores, memory]** for simulating a Spark cluster of [N, cores, memory] locally
- **spark://host:port,host1:port1,...** for connecting to [Spark Standalone cluster\(s\)](#)
- **mesos://** or **zk://** for [Spark on Mesos cluster](#)
- **yarn-cluster** (deprecated: **yarn-standalone**) for [Spark on YARN \(cluster mode\)](#)
- **yarn-client** for [Spark on YARN cluster \(client mode\)](#)
- **simr://** for [Spark in MapReduce \(SIMR\) cluster](#)

You use a master URL with [spark-submit](#) as the value of `--master` command-line option or when creating [SparkContext](#) using `setMaster` method.

Spark local

You can run Spark in **local mode**. In this non-distributed single-JVM deployment mode, Spark spawns all the execution components - **driver**, **executor**, **backend**, and **master** - in the same JVM. The default parallelism is the number of threads as specified in the [master URL](#). This is the only mode where a driver is used for execution.

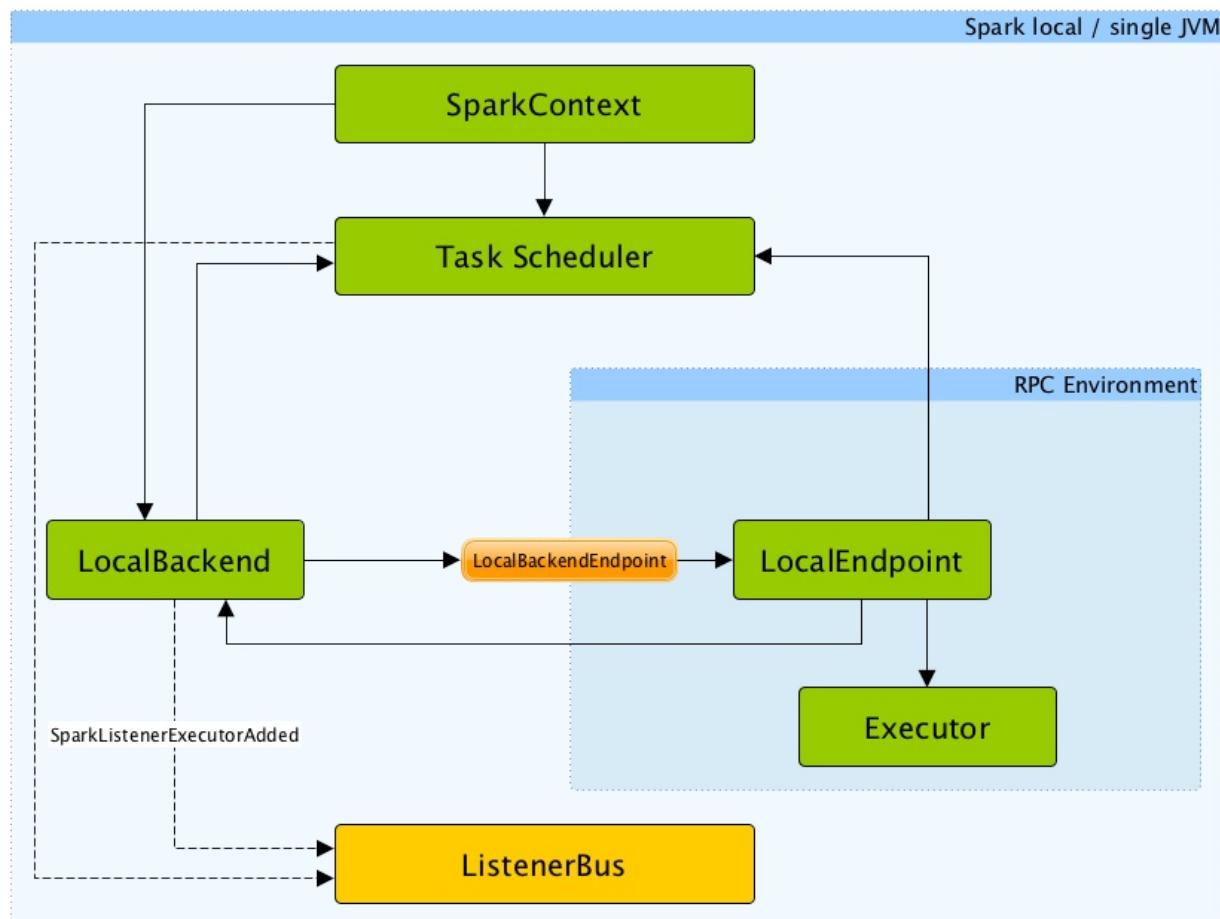


Figure 1. Architecture of Spark local

The local mode is very convenient for testing, debugging or demonstration purposes as it requires no earlier setup to launch Spark applications.

This mode of operation is also called [Spark in-process](#) or (less commonly) **a local version of Spark**.

`SparkContext.isLocal` returns `true` when Spark runs in local mode.

```
scala> sc.isLocal
res0: Boolean = true
```

[Spark shell](#) defaults to local mode with `local[*]` as the [the master URL](#).

```
scala> sc.master  
res0: String = local[*]
```

Tasks are not re-executed on failure in local mode (unless [local-with-retries master URL](#) is used).

The [task scheduler](#) in local mode works with [LocalBackend](#) task scheduler backend.

Master URL

You can run Spark in local mode using `local` , `local[n]` or the most general `local[*]` for [the master URL](#).

The URL says how many threads can be used in total:

- `local` uses 1 thread only.
- `local[n]` uses `n` threads.
- `local[*]` uses as many threads as the number of processors available to the Java virtual machine (it uses [Runtime.getRuntime.availableProcessors\(\)](#) to know the number).

Caution

[FIXME](#) What happens when there's less cores than `n` in the master URL?
It is a question from twitter.

- `local[N, M]` (called **local-with-retries**) with `N` being `*` or the number of threads to use (as explained above) and `M` being the value of [spark.task.maxFailures](#).

Task Submission a.k.a. `reviveOffers`

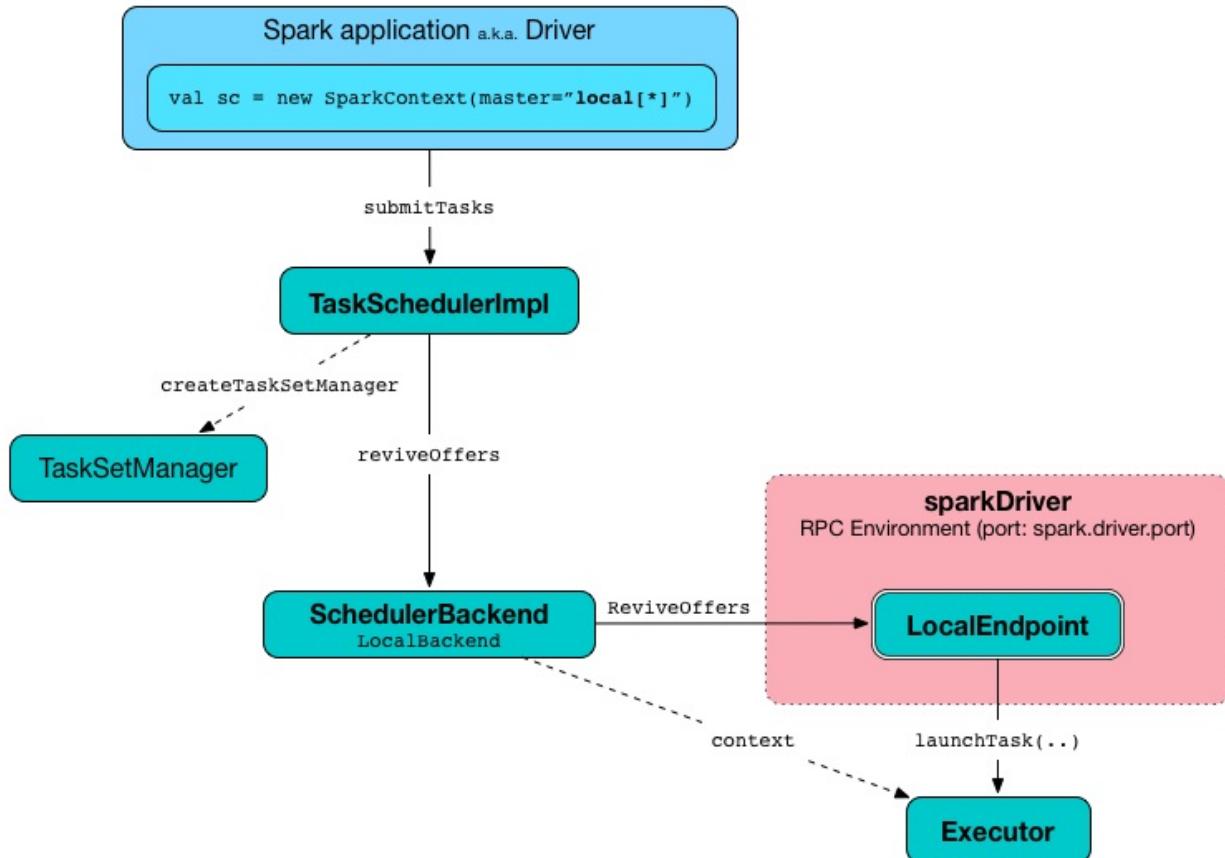


Figure 2. `TaskSchedulerImpl.submitTasks` in local mode

When `ReviveOffers` or `statusUpdate` messages are received, `LocalEndpoint` places an offer to `TaskSchedulerImpl` (using `TaskSchedulerImpl.resourceOffers`).

If there is one or more tasks that match the offer, they are launched (using `executor.launchTask` method).

The number of tasks to be launched is controlled by the number of threads as specified in [master URL](#). The executor uses threads to spawn the tasks.

LocalBackend

`LocalBackend` is a [scheduler backend](#) and a [executor backend](#) for Spark local mode.

It acts as a "cluster manager" for local mode to offer resources on the single `worker` it manages, i.e. it calls `TaskSchedulerImpl.resourceOffers(offers)` with `offers` being a single-element collection with `workerOffer("driver", "localhost", freeCores)`.

Caution	FIXME Review <code>freeCores</code> . It appears you could have many jobs running simultaneously.
---------	---

When an executor sends task status updates (using `ExecutorBackend.statusUpdate`), they are passed along as [StatusUpdate](#) to `LocalEndpoint`.

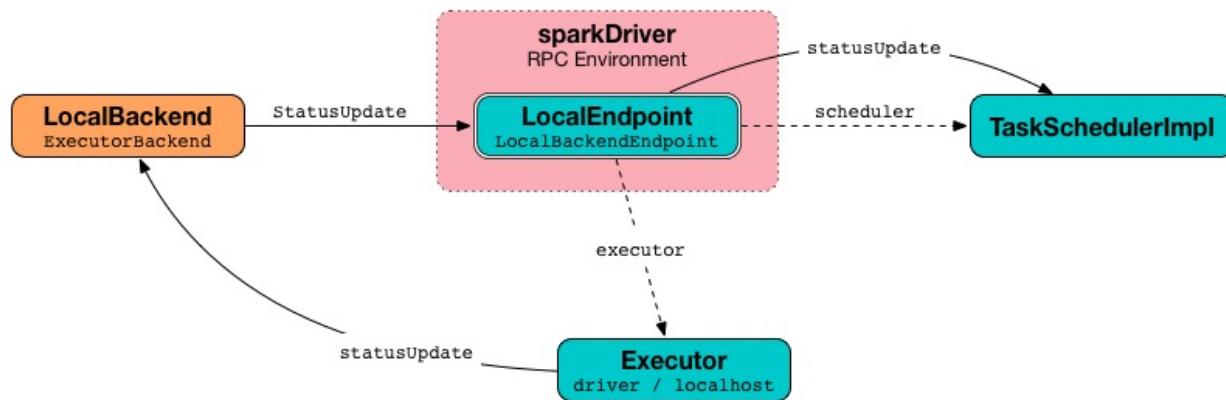


Figure 3. Task status updates flow in local mode

When LocalBackend starts up, it registers a new [RPC Endpoint](#) called **LocalBackendEndpoint** that is backed by [LocalEndpoint](#). This is announced over [Listener Bus](#) as `driver` (using [SparkListenerExecutorAdded](#) event).

The application ids are in the format of `local-[current time millis]`.

It communicates with [LocalEndpoint](#) using [RPC messages](#).

The default parallelism is controlled using [spark.default.parallelism](#).

LocalEndpoint

LocalEndpoint is the communication channel between [Task Scheduler](#) and [LocalBackend](#).

It is a (thread-safe) [RPC Endpoint](#) that hosts an [executor](#) (with id `driver` and hostname `localhost`) for Spark local mode.

When a LocalEndpoint starts up (as part of Spark local's initialization) it prints out the following INFO messages to the logs:

```

INFO Executor: Starting executor ID driver on host localhost
INFO Executor: Using REPL class URI: http://192.168.1.4:56131
  
```

RPC Messages

LocalEndpoint accepts the following RPC message types:

- `ReviveOffers` (receive-only, non-blocking) - read [Task Submission a.k.a. reviveOffers](#).
- `StatusUpdate` (receive-only, non-blocking) that passes the message to TaskScheduler (using `statusUpdate`) and if [the task's status is finished](#), it revives offers (see `ReviveOffers`).
- `KillTask` (receive-only, non-blocking) that kills the task that is currently running on the executor.

- `StopExecutor` (receive-reply, blocking) that stops the executor.

Settings

- `spark.default.parallelism` (default: the number of threads as specified in [master URL](#))
 - the default parallelism for [LocalBackend](#).

Running Spark on cluster

Spark can run on the following (open source) **cluster managers** (also called **schedulers**):

- [Spark's own Standalone cluster manager](#)
- [Hadoop YARN](#)
- [Apache Mesos](#)

It can be on-premise or in cloud.

Running Spark in cluster requires workload and resource management on distributed systems.

Spark driver communicates with a cluster manager for resources, e.g. CPU, memory, disk.
The cluster manager spawns Spark executors in the cluster.

	<p>FIXME</p> <ul style="list-style-type: none">• Spark execution in cluster - Diagram of the communication between driver, cluster manager, workers with executors and tasks. See Cluster Mode Overview.
Caution	<ul style="list-style-type: none">◦ Show Spark's driver with the main code in Scala in the box◦ Nodes with executors with tasks• Hosts drivers• Manages a cluster

The workers are in charge of communicating the cluster manager the availability of their resources.

Communication with a driver is through a RPC interface (at the moment Akka), except [Mesos in fine-grained mode](#).

Executors remain alive after jobs are finished for future ones. This allows for better data utilization as intermediate data is cached in memory.

Spark reuses resources in a cluster for:

- efficient data sharing
- fine-grained partitioning
- low-latency scheduling

Reusing also means the the resources can be hold onto for a long time.

Spark reuses long-running executors for speed (contrary to Hadoop MapReduce using short-lived containers for each task).

Spark Application Submission to Cluster

When you submit a Spark application to the cluster this is what happens (see the answers to [the answer to What are workers, executors, cores in Spark Standalone cluster?](#) on StackOverflow):

- The Spark driver is launched to invoke the `main` method of the Spark application.
- The driver asks the cluster manager for resources to run the application, i.e. to launch executors that run tasks.
- The cluster manager launches executors.
- The driver runs the Spark application and sends tasks to the executors.
- Executors run the tasks and save the results.
- Right after `SparkContext.stop()` is executed from the driver or the `main` method has exited all the executors are terminated and the cluster resources are released by the cluster manager.

Note

"There's not a good reason to run more than one worker per machine." by **Sean Owen** in [What is the relationship between workers, worker instances, and executors?](#)

Caution

One executor per node may not always be ideal, esp. when your nodes have lots of RAM. On the other hand, using fewer executors has benefits like more efficient broadcasts.

Two modes of launching executors

Warning

Review core/src/main/scala/org/apache/spark/deploy/master/Master.scala

Others

Spark application can be split into the part written in Scala, Java, and Python with the cluster itself in which the application is going to run.

Spark application runs on a cluster with the help of **cluster manager**.

A Spark application consists of a single driver process and a set of executor processes scattered across nodes on the cluster.

Both the driver and the executors usually run as long as the application. The concept of **dynamic resource allocation** has changed it.

Caution

[FIXME](#) Figure

A node is a machine, and there's not a good reason to run more than one worker per machine. So two worker nodes typically means two machines, each a Spark worker.

Workers hold many executors for many applications. One application has executors on many workers.

Spark Driver

- A separate Java process running on its own JVM
- Executes `main` of your application
- High-level control flow of work
- Your Spark application runs as long as the Spark driver.
 - Once the driver terminates, so does your Spark application.
- Creates `SparkContext`, `RDD's, and executes transformations and actions
- Spark shell is the driver, too.
 - Creates `SparkContext` that's available as `sc`.
- Launches [tasks](#)

Submit modes

There are two submit modes, i.e. where a driver runs:

- **client-mode** - a driver runs on the machine that submits the job
- **cluster-mode** - a driver runs on a (random) machine in the cluster

Spark Standalone cluster

Introduction

Spark Standalone cluster (aka *Spark deploy cluster* or *standalone cluster* or *standalone cluster deploy mode*) is the Spark built-in cluster. It comes with the default distribution of Apache Spark (available from the project's [Download Spark](#)).

Standalone Master (often written *standalone Master*) is the cluster manager for Spark Standalone cluster. See [Master](#).

Standalone workers (aka *standalone slaves*) are the workers in Spark Standalone cluster. They can be started and stopped using [custom management scripts for standalone Workers](#).

Spark Standalone cluster is one of the three available clustering options in Spark (refer to [Running Spark on cluster](#)).

Caution	<p>FIXME A figure with SparkDeploySchedulerBackend sending messages to AppClient and AppClient RPC Endpoint and later to Master.</p> <p>SparkDeploySchedulerBackend → AppClient → AppClient RPC Endpoint → Master</p> <p>Add SparkDeploySchedulerBackend as AppClientListener in the picture</p>
---------	---

In Standalone cluster mode Spark allocates resources based on cores. By default, an application will grab all the cores in the cluster (refer to [Settings](#)).

Standalone cluster mode is subject to the constraint that only one executor can be allocated on each worker per application.

Once a Spark Standalone cluster has been started, you can access it using `spark://` master URL (refer to [Master URLs](#)).

Caution	<p>FIXME That might be very confusing!</p>
---------	--

You can deploy, i.e. `spark-submit`, your applications to Spark Standalone in `client` or `cluster` deploy mode. Refer to [Deployment modes](#)

Deployment modes

Caution	<p>FIXME</p>
---------	---------------------

Refer to `--deploy-mode` in [spark-submit script](#).

SparkContext initialization in Standalone cluster

When you create a `SparkContext` using `spark:// master URL...`[FIXME](#)

Keeps track of task ids and executor ids, executors per host, hosts per rack

You can give one or many comma-separated masters URLs in `spark:// URL`.

A pair of backend and scheduler is returned.

The result is two have a pair of a backend and a scheduler.

Application Management using spark-submit

Caution

[FIXME](#)

```
→ spark git:(master) ✘ ./bin/spark-submit --help
...
Usage: spark-submit --kill [submission ID] --master [spark://...]
Usage: spark-submit --status [submission ID] --master [spark://...]
...
```

Refer to [Command-line Options](#) in `spark-submit`.

Round-robin Scheduling Across Nodes

[spark.deploy.spreadOut](#) setting controls whether or not to perform round-robin scheduling across the nodes (spreading out each app among all the nodes). It defaults to `true`.

[FIXME](#)

SPARK_WORKER_INSTANCES (and SPARK_WORKER_CORES)

There is really no need to run multiple workers per machine in Spark 1.5 (perhaps in 1.4, too). You can run multiple executors on the same machine with one worker.

Use `SPARK_WORKER_INSTANCES` (default: `1`) in `spark-env.sh` to define the number of worker instances.

If you use `SPARK_WORKER_INSTANCES`, make sure to set `SPARK_WORKER_CORES` explicitly to limit the cores per worker, or else each worker will try to use all the cores.

You can set up the number of cores as an command line argument when you start a worker daemon using `--cores`.

Multiple executors per worker in Standalone mode

Caution	It can be a duplicate of the above section.
---------	---

Since the change [SPARK-1706 Allow multiple executors per worker in Standalone mode](#) in Spark 1.4 it's currently possible to start multiple executors in a single JVM process of a worker.

To launch multiple executors on a machine you start multiple standalone workers, each with its own JVM. It introduces unnecessary overhead due to these JVM processes, provided that there are enough cores on that worker.

If you are running Spark in standalone mode on memory-rich nodes it can be beneficial to have multiple worker instances on the same node as a very large heap size has two disadvantages:

- Garbage collector pauses can hurt throughput of Spark jobs.
- Heap size of >32 GB can't use CompressedOoops. So [35 GB is actually less than 32 GB](#).

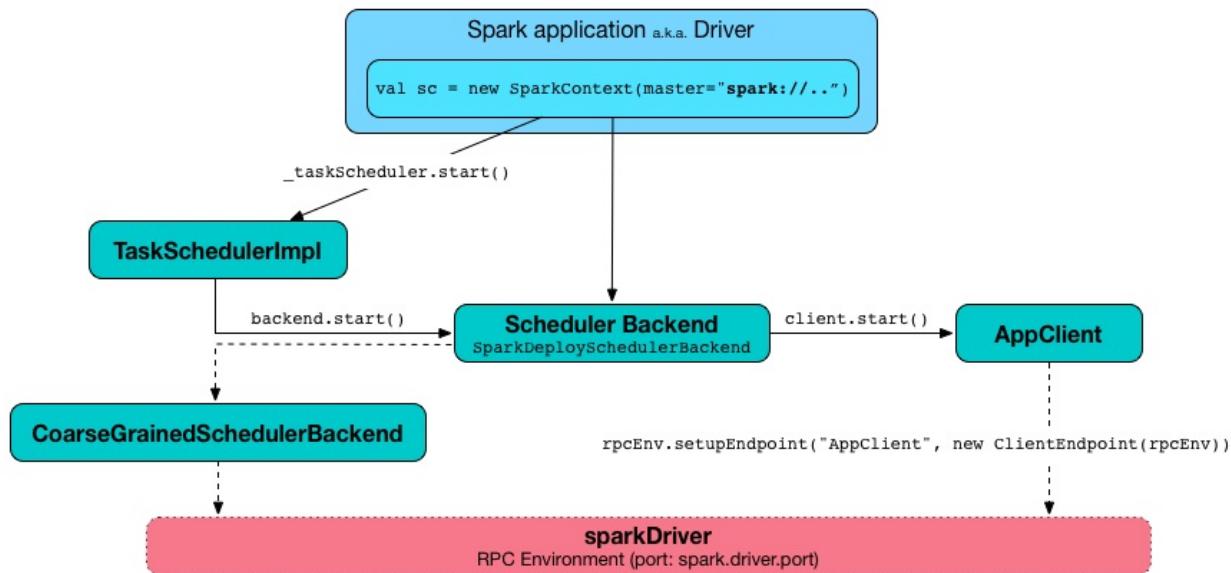
Mesos and YARN can, out of the box, support packing multiple, smaller executors onto the same physical host, so requesting smaller executors doesn't mean your application will have fewer overall resources.

SparkDeploySchedulerBackend

`SparkDeploySchedulerBackend` is the [Scheduler Backend](#) for Spark Standalone, i.e. it is used when you [create a SparkContext](#) using `spark:// master URL`.

It requires a [Task Scheduler](#), a [Spark context](#), and a collection of [master URLs](#).

It is a specialized [CoarseGrainedSchedulerBackend](#) that uses [AppClient](#) and is a [AppClientListener](#).

Figure 1. `SparkDeploySchedulerBackend.start()` (while `SparkContext` starts)

Caution	<code>FIXME</code> <code>AppClientListener</code> & <code>LauncherBackend</code> & <code>ApplicationDescription</code>
---------	--

It uses `AppClient` to talk to executors.

AppClient

`AppClient` is an interface to allow Spark applications to talk to a Standalone cluster (using a RPC Environment). It takes an RPC Environment, a collection of master URLs, a `ApplicationDescription`, and a `AppClientListener`.

It is solely used by `SparkDeploySchedulerBackend`.

`AppClient` registers `AppClient` RPC endpoint (using `ClientEndpoint` class) to a given RPC Environment.

`AppClient` uses a daemon cached thread pool (`askAndReplyThreadPool`) with threads' name in the format of `appclient-receive-and-reply-threadpool-ID`, where `ID` is a unique integer for asynchronous asks and replies. It is used for requesting executors (via `RequestExecutors` message) and kill executors (via `KillExecutors`).

`sendToMaster` sends one-way `ExecutorStateChanged` and `UnregisterApplication` messages to master.

Initialization - `AppClient.start()` method

When `AppClient` starts, `AppClient.start()` method is called that merely registers `AppClient` RPC Endpoint.

Others

- killExecutors
- start
- stop

AppClient RPC Endpoint

AppClient RPC endpoint is started as part of **AppClient**'s initialization (that is in turn part of **SparkDeploySchedulerBackend**'s initialization, i.e. the scheduler backend for **Spark Standalone**).

It is a `ThreadSafeRpcEndpoint` that knows about the RPC endpoint of the primary active standalone Master (there can be a couple of them, but only one can be active and hence primary).

When it starts, it sends [RegisterApplication](#) message to register an application and itself.

RegisterApplication RPC message

An AppClient registers the Spark application to a single master (regardless of [the number of the standalone masters given in the master URL](#)).

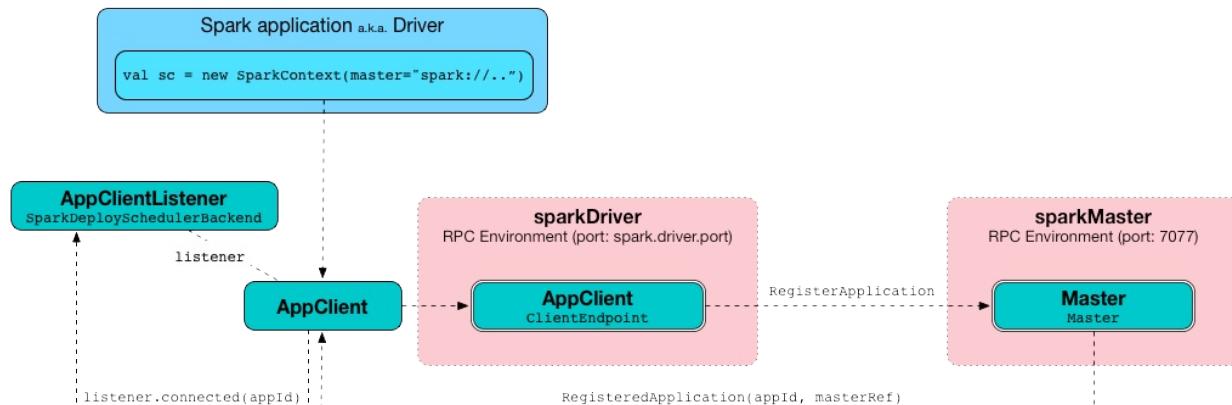


Figure 2. AppClient registers application to standalone Master

It uses a dedicated thread pool **appclient-register-master-threadpool** to asynchronously send `RegisterApplication` messages, one per standalone master.

```
INFO AppClient$ClientEndpoint: Connecting to master spark://localhost:7077...
```

An AppClient tries connecting to a standalone master 3 times every 20 seconds per master before giving up. They are not configurable parameters.

The `appclient-register-master-threadpool` thread pool is used until the registration is finished, i.e. AppClient is connected to the primary standalone Master or the registration fails. It is then `shutdown`.

RegisteredApplication RPC message

`RegisteredApplication` is a one-way message from the primary master to confirm successful application registration. It comes with the application id and the master's RPC endpoint reference.

The `AppClientListener` gets notified about the event via `listener.connected(appId)` with `appId` being an application id.

ApplicationRemoved RPC message

`ApplicationRemoved` is received from the primary master to inform about having removed the application. AppClient RPC endpoint is stopped afterwards.

It can come from the standalone Master after a kill request from Web UI, application has finished properly or the executor where the application was still running on has been killed, failed, lost or exited.

ExecutorAdded RPC message

`ExecutorAdded` is received from the primary master to inform about...[FIXME](#)

Caution	FIXME the message
---------	-----------------------------------

INFO Executor added: %s on %s (%s) with %d cores

ExecutorUpdated RPC message

`ExecutorUpdated` is received from the primary master to inform about...[FIXME](#)

Caution	FIXME the message
---------	-----------------------------------

INFO Executor updated: %s is now %s%

MasterChanged RPC message

`MasterChanged` is received from the primary master to inform about...[FIXME](#)

Caution	FIXME the message
---------	-----------------------------------

INFO Master has changed, new master is at

StopAppClient RPC message

`StopAppClient` is a reply-response message from the `SparkDeploySchedulerBackend` to stop the `AppClient` after the `SparkContext` has been stopped (and so should the running application on the standalone cluster).

It stops the `AppClient` RPC endpoint.

RequestExecutors RPC message

`RequestExecutors` is a reply-response message from the `SparkDeploySchedulerBackend` that is passed on to the master to request executors for the application.

KillExecutors RPC message

`KillExecutors` is a reply-response message from the `SparkDeploySchedulerBackend` that is passed on to the master to kill executors assigned to the application.

Master

Standalone Master (often written *standalone Master*) is the cluster manager for Spark Standalone cluster. It can be started and stopped using [custom management scripts for standalone Master](#).

A standalone Master is pretty much the Master RPC Endpoint that you can access using RPC port (low-level operation communication) or [Web UI](#).

Application ids follows the pattern `app-yyyyMMddHHmmss` .

Master keeps track of the following:

- workers (`workers`)
- mapping between ids and applications (`idToApp`)
- waiting applications (`waitingApps`)
- applications (`apps`)
- mapping between ids and workers (`idToWorker`)
- mapping between RPC address and workers (`addressToWorker`)
- `endpointToApp`
- `addressToApp`
- `completedApps`
- `nextAppNumber`
- mapping between application ids and their Web UIs (`appIdToUI`)
- drivers (`drivers`)
- `completedDrivers`
- drivers currently spooled for scheduling (`waitingDrivers`)
- `nextDriverNumber`

The following INFO shows up when the Master endpoint starts up (`Master#onStart` is called):

```
INFO Master: Starting Spark master at spark://japila.local:7077
INFO Master: Running Spark version 1.6.0-SNAPSHOT
```

Master WebUI

[FIXME](#) MasterWebUI

`MasterWebUI` is the Web UI server for the standalone master. Master starts Web UI to listen to `http://[master's hostname]:webUIPort`, e.g. `http://localhost:8080`.

```
INFO Utils: Successfully started service 'MasterUI' on port 8080.
INFO MasterWebUI: Started MasterWebUI at http://192.168.1.4:8080
```

States

Master can be in the following states:

- `STANDBY` - the initial state while Master is initializing
- `ALIVE` - start scheduling resources among applications.
- `RECOVERING`
- `COMPLETING_RECOVERY`

Caution

[FIXME](#)

RPC Environment

The `org.apache.spark.deploy.master.Master` class starts [sparkMaster](#) RPC environment.

```
INFO Utils: Successfully started service 'sparkMaster' on port 7077.
```

It then registers `Master` endpoint.

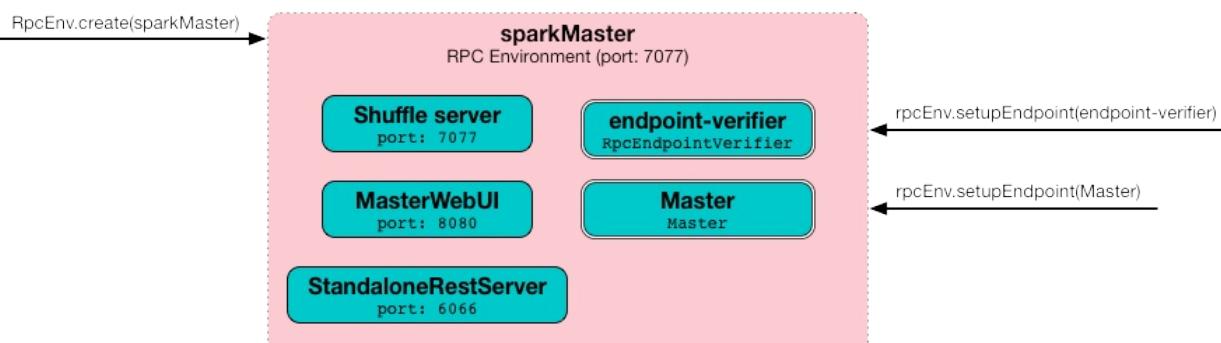


Figure 1. `sparkMaster` - the RPC Environment for Spark Standalone's master
Master endpoint is a `ThreadSafeRpcEndpoint` and `LeaderElectable` (see [Leader Election](#)).

The Master endpoint starts the daemon single-thread scheduler pool `master-forward-message-thread`. It is used for worker management, i.e. removing any timed-out workers.

```
"master-forward-message-thread" #46 daemon prio=5 os_prio=31 tid=0x00007ff322abb000 nid=0
```

Metrics

Master uses [Spark Metrics System](#) (via `MasterSource`) to report metrics about internal status.

The name of the source is **master**.

It emits the following metrics:

- `workers` - the number of all workers (any state)
- `aliveWorkers` - the number of alive workers
- `apps` - the number of applications
- `waitingApps` - the number of waiting applications

The name of the other source is **applications**

	FIXME
Caution	<ul style="list-style-type: none"> • Review <code>org.apache.spark.metrics.MetricsConfig</code> • How to access the metrics for master? See <code>Master#onStart</code> • Review <code>masterMetricsSystem</code> and <code>applicationMetricsSystem</code>

REST Server

The standalone Master starts the REST Server service for alternative application submission that is supposed to work across Spark versions. It is enabled by default (see [spark.master.rest.enabled](#)) and used by [spark-submit](#) for the [standalone cluster mode](#), i.e. `-deploy-mode is cluster`.

`RestSubmissionClient` is the client.

The server includes a JSON representation of `SubmitRestProtocolResponse` in the HTTP body.

The following INFOs show up when the Master Endpoint starts up (`Master#onStart` is called) with REST Server enabled:

```
INFO Utils: Successfully started service on port 6066.
INFO StandaloneRestServer: Started REST server for submitting applications on port 6066
```

Recovery Mode

A standalone Master can run with **recovery mode** enabled and be able to recover state among the available swarm of masters. By default, there is no recovery, i.e. no persistence and no election.

Note	Only a master can schedule tasks so having one always on is important for cases where you want to launch new tasks. Running tasks are unaffected by the state of the master.
------	--

Master uses `spark.deploy.recoveryMode` to set up the recovery mode (see [spark.deploy.recoveryMode](#)).

The Recovery Mode enables [election of the leader master](#) among the masters.

Tip	Check out the exercise Spark Standalone - Using ZooKeeper for High-Availability of Master .
-----	---

Leader Election

Master endpoint is `LeaderElectable`, i.e. [FIXME](#)

Caution	FIXME
---------	-----------------------

RPC Messages

Master communicates with drivers, executors and configures itself using **RPC messages**.

The following message types are accepted by master (see `Master#receive` or `Master#receiveAndReply` methods):

- `ElectedLeader` for [Leader Election](#)
- `CompleteRecovery`
- `RevokedLeadership`
- [RegisterApplication](#)
- `ExecutorStateChanged`
- `DriverStateChanged`

- Heartbeat
- MasterChangeAcknowledged
- WorkerSchedulerStateResponse
- UnregisterApplication
- CheckForWorkerTimeOut
- RegisterWorker
- RequestSubmitDriver
- RequestKillDriver
- RequestDriverStatus
- RequestMasterState
- BoundPortsRequest
- RequestExecutors
- KillExecutors

RegisterApplication event

A **RegisterApplication** event is sent by [AppClient](#) to the standalone Master. The event holds information about the application being deployed (`ApplicationDescription`) and the driver's endpoint reference.

`ApplicationDescription` describes an application by its name, maximum number of cores, executor's memory, command, appUiUrl, and user with optional eventLogDir and eventLogCodec for Event Logs, and the number of cores per executor.

Caution

[FIXME](#) Finish

A standalone Master receives `RegisterApplication` with a `ApplicationDescription` and the driver's `RpcEndpointRef`.

```
INFO Registering app " + description.name
```

Application ids in Spark Standalone are in the format of `app-[yyyyMMddHHmmss]-[4-digit nextAppNumber]`.

Master keeps track of the number of already-scheduled applications (`nextAppNumber`).

ApplicationDescription (AppClient) → ApplicationInfo (Master) - application structure enrichment

```
ApplicationSource metrics + applicationMetricsSystem
```

```
INFO Registered app " + description.name + " with ID " + app.id
```

Caution	FIXME persistenceEngine.addApplication(app)
---------	--

`schedule()` schedules the currently available resources among waiting apps.

FIXME When is `schedule()` method called?

It's only executed when the Master is in `RecoveryState.ALIVE` state.

Worker in `workerState.ALIVE` state can accept applications.

A driver has a state, i.e. `driver.state` and when it's in `DriverState.RUNNING` state the driver has been assigned to a worker for execution.

LaunchDriver RPC message

Warning	It seems a dead message. Disregard it for now.
---------	--

A **LaunchDriver** message is sent by an active standalone Master to a worker to launch a driver.

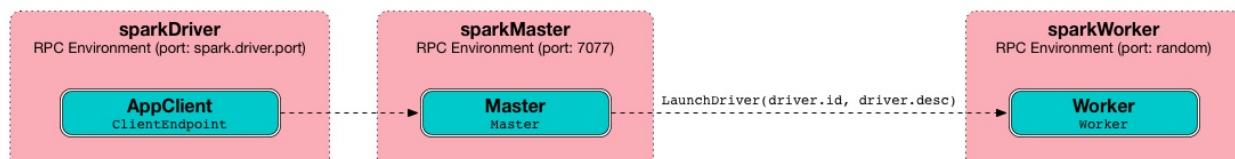


Figure 2. Master finds a place for a driver (posts LaunchDriver)

You should see the following INFO in the logs right before the message is sent out to a worker:

```
INFO Launching driver [driver.id] on worker [worker.id]
```

The message holds information about the id and name of the driver.

A driver can be running on a single worker while a worker can have many drivers running.

When a worker receives a `LaunchDriver` message, it prints out the following INFO:

```
INFO Asked to launch driver [driver.id]
```

It then creates a `DriverRunner` and starts it. It starts a separate JVM process.

Workers' free memory and cores are considered when assigning some to waiting drivers (applications).

Caution	FIXME Go over <code>waitingDrivers</code> ...
---------	---

DriverRunner

Warning	It seems a dead piece of code. Disregard it for now.
---------	--

A `DriverRunner` manages the execution of one driver.

It is a `java.lang.Process`

When started, it spawns a thread `DriverRunner` for `[driver.id]` that:

1. Creates the working directory for this driver.
2. Downloads the user jar [FIXME](#) `downloadUserJar`
3. Substitutes variables like `WORKER_URL` or `USER_JAR` that are set when...[FIXME](#)

Internals of org.apache.spark.deploy.master.Master

Tip	You can debug a Standalone master using the following command: <pre>java -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005 -cp /Use</pre> The above command suspends (<code>suspend=y</code>) the process until a JPDA debugging cli
-----	---

The above command suspends (`suspend=y`) the process until a JPDA debugging cli

When `Master` starts, it first creates the [default SparkConf configuration](#) whose values it then overrides using [environment variables](#) and [command-line options](#).

A fully-configured master instance requires `host`, `port` (default: `7077`), `webUiPort` (default: `8080`) settings defined.

Tip	When in troubles, consult Spark Tips and Tricks document.
-----	---

It starts [RPC Environment](#) with necessary endpoints and lives until the RPC environment terminates.

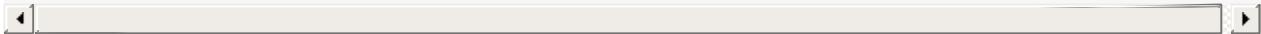
Worker Management

Master uses `master-forward-message-thread` to schedule a thread every `spark.worker.timeout` to check workers' availability and remove timed-out workers.

It is that Master sends `CheckForWorkerTimeOut` message to itself to trigger verification.

When a worker hasn't responded for `spark.worker.timeout`, it is assumed dead and the following WARN message appears in the logs:

```
WARN Removing [worker.id] because we got no heartbeat in [spark.worker.timeout] seconds
```



System Environment Variables

Master uses the following system environment variables (directly or indirectly):

- `SPARK_LOCAL_HOSTNAME` - the custom host name
- `SPARK_LOCAL_IP` - the custom IP to use when `SPARK_LOCAL_HOSTNAME` is not set
- `SPARK_MASTER_HOST` (not `SPARK_MASTER_IP` as used in `start-master.sh` script above!) - the master custom host
- `SPARK_MASTER_PORT` (default: `7077`) - the master custom port
- `SPARK_MASTER_IP` (default: `hostname` command's output)
- `SPARK_MASTER_WEBUI_PORT` (default: `8080`) - the port of the master's WebUI. Overridden by `spark.master.ui.port` if set in the properties file.
- `SPARK_PUBLIC_DNS` (default: `hostname`) - the custom master hostname for WebUI's http URL and master's address.
- `SPARK_CONF_DIR` (default: `$SPARK_HOME/conf`) - the directory of the default properties file `spark-defaults.conf` from which all properties that start with `spark.` prefix are loaded.

Settings

Caution	FIXME
	<ul style="list-style-type: none"> • Where are `RETAINED_`'s properties used?

Master uses the following properties:

- `spark.cores.max` (default: `0`) - total expected number of cores (**FIXME** `totalExpectedCores`). When set, an application could get executors of different sizes (in terms of cores).

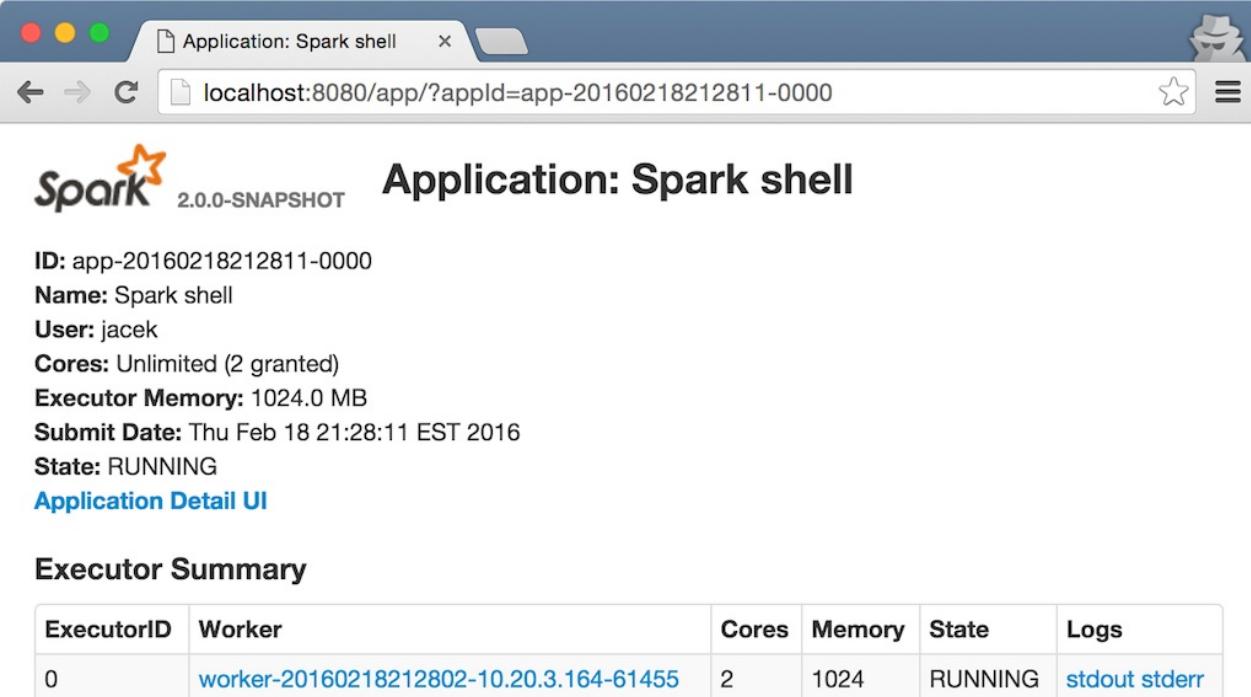
- `spark.worker.timeout` (default: `60`) - time (in seconds) when no heartbeat from a worker means it is lost. See [Worker Management](#).
- `spark.deploy.retainedApplications` (default: `200`)
- `spark.deploy.retainedDrivers` (default: `200`)
- `spark.dead.worker.persistence` (default: `15`)
- `spark.deploy.recoveryMode` (default: `NONE`) - possible modes: `ZOOKEEPER`, `FILESYSTEM`, or `CUSTOM`. Refer to [Recovery Mode](#).
- `spark.deploy.recoveryMode.factory` - the class name of the custom `StandaloneRecoveryModeFactory`.
- `spark.deploy.recoveryDirectory` (default: empty) - the directory to persist recovery state
- `spark.deploy.spreadOut` (default: `true`) - perform round-robin scheduling across the nodes (spreading out each app among all the nodes). Refer to [Round-robin Scheduling Across Nodes](#)
- `spark.deploy.defaultCores` (default: `Int.MaxValue`, i.e. unbounded)- the number of maxCores for applications that don't specify it.
- `spark.master.rest.enabled` (default: `true`) - [master's REST Server](#) for alternative application submission that is supposed to work across Spark versions.
- `spark.master.rest.port` (default: `6066`) - the port of [master's REST Server](#)

master's Administrative web UI

Spark Standalone cluster comes with administrative **web UI**. It is available under <http://localhost:8080> by default.

Executor Summary

Executor Summary page displays information about the executors for the application id given as the `appId` request parameter.



The screenshot shows a web browser window titled "Application: Spark shell". The URL in the address bar is "localhost:8080/app/?appId=app-20160218212811-0000". The page content includes the Spark logo and version "2.0.0-SNAPSHOT". It displays application details: ID: app-20160218212811-0000, Name: Spark shell, User: jacek, Cores: Unlimited (2 granted), Executor Memory: 1024.0 MB, Submit Date: Thu Feb 18 21:28:11 EST 2016, and State: RUNNING. A link "Application Detail UI" is also present. Below this, a section titled "Executor Summary" contains a table with one row, showing ExecutorID 0, Worker worker-20160218212802-10.20.3.164-61455, Cores 2, Memory 1024, State RUNNING, and Logs stdout stderr.

ExecutorID	Worker	Cores	Memory	State	Logs
0	worker-20160218212802-10.20.3.164-61455	2	1024	RUNNING	stdout stderr

Figure 1. Executor Summary Page

The **State** column displays the state of an executor as tracked by the master.

When an executor is added to the pool of available executors, it enters `LAUNCHING` state. It can then enter either `RUNNING` or `FAILED` states.

An executor (as `ExecutorRunner`) sends `ExecutorStateChanged` message to a worker (that it then sends forward to a master) as a means of announcing an executor's state change:

- `ExecutorRunner.fetchAndRunExecutor` sends `EXITED`, `KILLED` OR `FAILED`.
- `ExecutorRunner.killProcess`

A Worker sends `ExecutorStateChanged` messages for the following cases:

- When `LaunchExecutor` is received, an executor (as `ExecutorRunner`) is started and `RUNNING` state is announced.
- When `LaunchExecutor` is received, an executor (as `ExecutorRunner`) fails to start and `FAILED` state is announced.

If no application for the `appId` could be found, **Not Found** page is displayed.

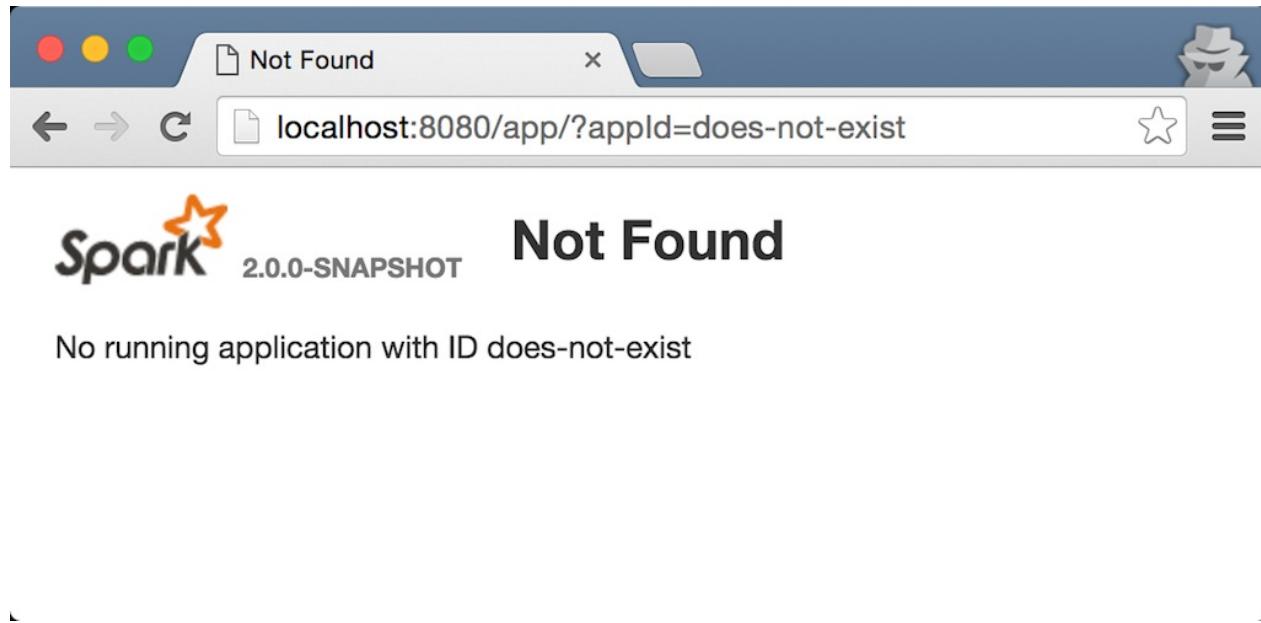


Figure 2. Application Not Found Page

Management Scripts for Standalone Master

You can start a [Spark Standalone master](#) (aka *standalone Master*) using `sbin/start-master.sh` and stop it using `sbin/stop-master.sh`.

`sbin/start-master.sh`

`sbin/start-master.sh` script starts a Spark master on the machine the script is executed on.

```
./sbin/start-master.sh
```

The script prepares the command line to start the class

`org.apache.spark.deploy.master.Master` and by default runs as follows:

```
org.apache.spark.deploy.master.Master \
--ip japila.local --port 7077 --webui-port 8080
```

Note

The command sets `SPARK_PRINT_LAUNCH_COMMAND` environment variable to print out the launch command to standard error output. Refer to [Print Launch Command of Spark Scripts](#).

It has support for starting Tachyon using `--with-tachyon` command line option. It assumes `tachyon/bin/tachyon` command be available in Spark's home directory.

The script uses the following helper scripts:

- `sbin/spark-config.sh`
- `bin/load-spark-env.sh`
- `conf/spark-env.sh` contains environment variables of a Spark executable.

Ultimately, the script calls `sbin/spark-daemon.sh start` to kick off `org.apache.spark.deploy.master.Master` with parameter `1` and `--ip`, `--port`, and `--webui-port` [command-line options](#).

Command-line Options

You can use the following command-line options:

- `--host` or `-h` the hostname to listen on; overrides [SPARK_MASTER_HOST](#).
- `--ip` or `-i` (deprecated) the IP to listen on

- `--port` or `-p` - command-line version of `SPARK_MASTER_PORT` that overrides it.
- `--webui-port` - command-line version of `SPARK_MASTER_WEBUI_PORT` that overrides it.
- `--properties-file` (default: `$SPARK_HOME/conf/spark-defaults.conf`) - the path to a custom Spark properties file
- `--help` - prints out help

sbin/stop-master.sh

You can stop a Spark Standalone master using `sbin/stop-master.sh` script.

```
./sbin/stop-master.sh
```

Caution

FIXME Review the script

It effectively sends SIGTERM to the master's process.

You should see the ERROR in master's logs:

```
ERROR Master: RECEIVED SIGNAL 15: SIGTERM
```

Management Scripts for Standalone Workers

`sbin/start-slave.sh` script starts a Spark worker (aka slave) on the machine the script is executed on. It launches `SPARK_WORKER_INSTANCES` instances.

```
./sbin/start-slave.sh [masterURL]
```

The mandatory `masterURL` parameter is of the form `spark://hostname:port`, e.g. `spark://localhost:7077`. It is also possible to specify a comma-separated master URLs of the form `spark://hostname1:port1,hostname2:port2,...` with each element to be `hostname:port`.

Internally, the script starts [sparkWorker RPC environment](#).

The order of importance of Spark configuration settings is as follows (from least to the most important):

- [System environment variables](#)
- [Command-line options](#)
- [Spark properties](#)

System environment variables

The script uses the following system environment variables (directly or indirectly):

- `SPARK_WORKER_INSTANCES` (default: `1`) - the number of worker instances to run on this slave.
- `SPARK_WORKER_PORT` - the base port number to listen on for the first worker. If set, subsequent workers will increment this number. If unset, Spark will pick a random port.
- `SPARK_WORKER_WEBUI_PORT` (default: `8081`) - the base port for the web UI of the first worker. Subsequent workers will increment this number. If the port is used, the successive ports are tried until a free one is found.
- `SPARK_WORKER_CORES` - the number of cores to use by a single executor
- `SPARK_WORKER_MEMORY` (default: `1G`) - the amount of memory to use, e.g. `1000M`, `2G`
- `SPARK_WORKER_DIR` (default: `$SPARK_HOME/work`) - the directory to run apps in

The script uses the following helper scripts:

- `sbin/spark-config.sh`
- `bin/load-spark-env.sh`

Command-line Options

You can use the following command-line options:

- `--host` or `-h` - sets the hostname to be available under.
- `--port` or `-p` - command-line version of `SPARK_WORKER_PORT` environment variable.
- `--cores` or `-c` (default: the number of processors available to the JVM) - command-line version of `SPARK_WORKER_CORES` environment variable.
- `--memory` or `-m` - command-line version of `SPARK_WORKER_MEMORY` environment variable.
- `--work-dir` or `-d` - command-line version of `SPARK_WORKER_DIR` environment variable.
- `--webui-port` - command-line version of `SPARK_WORKER_WEBUI_PORT` environment variable.
- `--properties-file` (default: `conf/spark-defaults.conf`) - the path to a custom Spark properties file
- `--help`

Spark properties

After loading the `default SparkConf`, if `--properties-file` or `SPARK_WORKER_OPTS` define `spark.worker.ui.port`, the value of the property is used as the port of the worker's web UI.

```
SPARK_WORKER_OPTS=-Dspark.worker.ui.port=21212 ./sbin/start-slave.sh spark://localhost:70
```

or

```
$ cat worker.properties  
spark.worker.ui.port=33333  
  
$ ./sbin/start-slave.sh spark://localhost:7077 --properties-file worker.properties
```

sbin/spark-daemon.sh

Ultimately, the script calls `sbin/spark-daemon.sh start` to kick off `org.apache.spark.deploy.worker.Worker` with `--webui-port`, `--port` and the master URL.

Internals of org.apache.spark.deploy.worker.Worker

Upon starting, a Spark worker creates the [default SparkConf](#).

It parses command-line arguments for the worker using `WorkerArguments` class.

- `SPARK_LOCAL_HOSTNAME` - custom host name
- `SPARK_LOCAL_IP` - custom IP to use (when `SPARK_LOCAL_HOSTNAME` is not set or hostname resolves to incorrect IP)

It starts [sparkWorker RPC Environment](#) and waits until the RpcEnv terminates.

RPC environment

The `org.apache.spark.deploy.worker.Worker` class starts its own [sparkWorker RPC environment](#) with `worker` endpoint.

sbin/start-slaves.sh script starts slave instances

The `./sbin/start-slaves.sh` script starts slave instances on each machine specified in the `conf/slaves` file.

It has support for starting Tachyon using `--with-tachyon` command line option. It assumes `tachyon/bin/tachyon` command be available in Spark's home directory.

The script uses the following helper scripts:

- `sbin/spark-config.sh`
- `bin/load-spark-env.sh`
- `conf/spark-env.sh`

The script uses the following environment variables (and sets them when unavailable):

- `SPARK_PREFIX`
- `SPARK_HOME`
- `SPARK_CONF_DIR`

- SPARK_MASTER_PORT
- SPARK_MASTER_IP

The following command will launch 3 worker instances on each node. Each worker instance will use two cores.

```
SPARK_WORKER_INSTANCES=3 SPARK_WORKER_CORES=2 ./sbin/start-slaves.sh
```

Checking Status of Spark Standalone

jps

Since you're using Java tools to run Spark, use `jps -lm` as the tool to get status of any JVMs on a box, Spark's ones including. Consult [jps documentation](#) for more details beside `-lm` command-line options.

If you however want to filter out the JVM processes that really belong to Spark you should pipe the command's output to OS-specific tools like `grep`.

```
$ jps -lm
999 org.apache.spark.deploy.master.Master --ip japila.local --port 7077 --webui-port 8080
397
669 org.jetbrains.idea.maven.server.RemoteMavenServer
1198 sun.tools.jps.Jps -lm

$ jps -lm | grep -i spark
999 org.apache.spark.deploy.master.Master --ip japila.local --port 7077 --webui-port 8080
```

spark-daemon.sh status

You can also check out `./sbin/spark-daemon.sh status`.

When you start Spark Standalone using scripts under `sbin`, PIDs are stored in `/tmp` directory by default. `./sbin/spark-daemon.sh status` can read them and do the "boilerplate" for you, i.e. status a PID.

```
$ jps -lm | grep -i spark
999 org.apache.spark.deploy.master.Master --ip japila.local --port 7077 --webui-port 8080

$ ls /tmp/spark-*.pid
/tmp/spark-jacek-org.apache.spark.deploy.master.Master-1.pid

$ ./sbin/spark-daemon.sh status org.apache.spark.deploy.master.Master 1
org.apache.spark.deploy.master.Master is running.
```

Example 2-workers-on-1-node Standalone Cluster (one executor per worker)

The following steps are a recipe for a Spark Standalone cluster with 2 workers on a single machine.

The aim is to have a complete Spark-clustered environment at your laptop.

	Consult the following documents:
Tip	<ul style="list-style-type: none"> • Operating Spark master • Starting Spark workers on node using sbin/start-slave.sh
Important	<p>You can use the Spark Standalone cluster in the following ways:</p> <ul style="list-style-type: none"> • Use <code>spark-shell</code> with <code>--master MASTER_URL</code> • Use <code>SparkConf.setMaster(MASTER_URL)</code> in your Spark application <p>For our learning purposes, <code>MASTER_URL</code> is <code>spark://localhost:7077</code> .</p>

1. Start a standalone master server.

```
./sbin/start-master.sh
```

Notes:

- Read [Operating Spark Standalone master](#)
- Use `SPARK_CONF_DIR` for the configuration directory (defaults to `$SPARK_HOME/conf`).
- Use `spark.deploy.retainedApplications` (`default: 200`)
- Use `spark.deploy.retainedDrivers` (`default: 200`)
- Use `spark.deploy.recoveryMode` (`default: NONE`)
- Use `spark.deploy.spreadout` (`default: true`) to allow users to set a flag that will perform round-robin scheduling across the nodes (spreading out each app among all the nodes) instead of trying to consolidate each app onto a small # of nodes.
- Use `spark.deploy.defaultCores` (`default: Int.MaxValue`)

2. Open master's web UI at <http://localhost:8080> to know the current setup - no workers and applications.

Figure 1. Master's web UI with no workers and applications

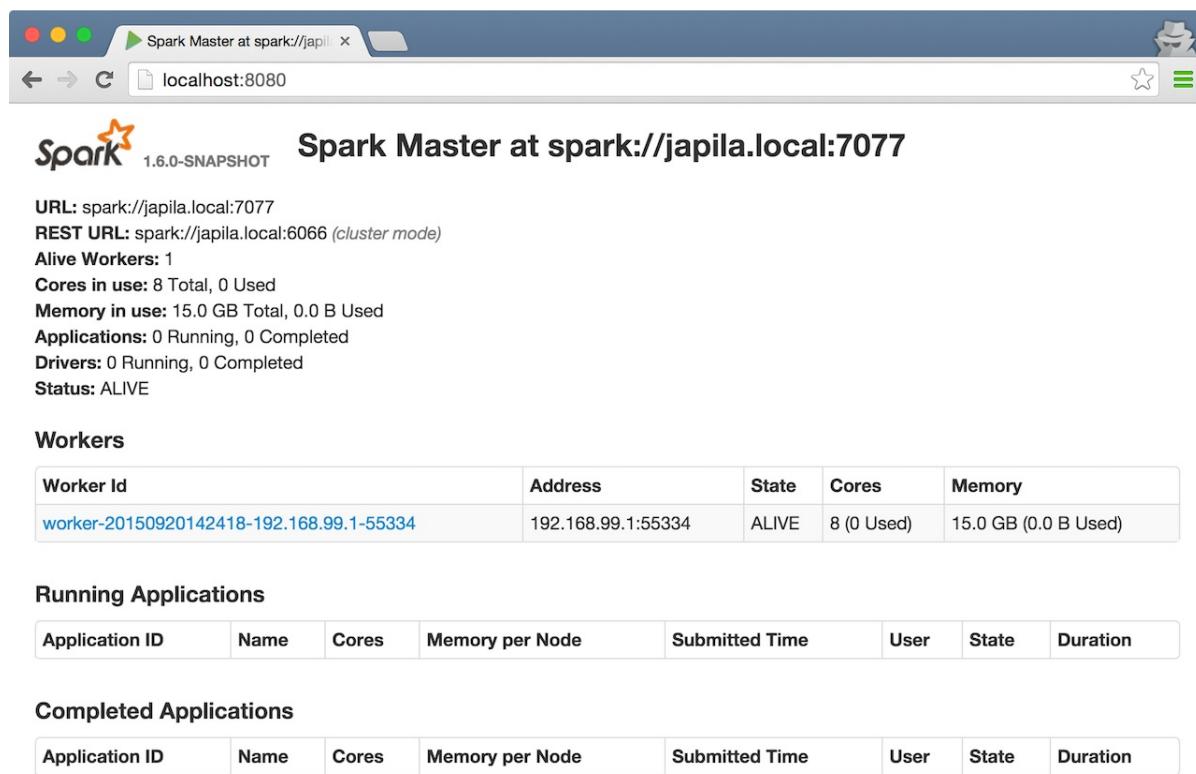
3. Start the first worker.

```
./sbin/start-slave.sh spark://japila.local:7077
```

Note

The command above in turn executes
`org.apache.spark.deploy.worker.Worker --webui-port 8081
 spark://japila.local:7077`

4. Check out master's web UI at <http://localhost:8080> to know the current setup - one worker.



The screenshot shows the Spark Master web interface at `spark://japila.local:7077`. The UI includes the following sections:

- Spark Logo** and **Version: 1.6.0-SNAPSHOT**
- Spark Master at spark://japila.local:7077**
- URL: spark://japila.local:7077**
- REST URL: spark://japila.local:6066 (cluster mode)**
- Alive Workers: 1**
- Cores in use: 8 Total, 0 Used**
- Memory in use: 15.0 GB Total, 0.0 B Used**
- Applications: 0 Running, 0 Completed**
- Drivers: 0 Running, 0 Completed**
- Status: ALIVE**
- Workers** table:

Worker Id	Address	State	Cores	Memory
worker-20150920142418-192.168.99.1-55334	192.168.99.1:55334	ALIVE	8 (0 Used)	15.0 GB (0.0 B Used)
- Running Applications** table header:

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
----------------	------	-------	-----------------	----------------	------	-------	----------
- Completed Applications** table header:

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
----------------	------	-------	-----------------	----------------	------	-------	----------

Figure 2. Master's web UI with one worker ALIVE

Note the number of CPUs and memory, 8 and 15 GBs, respectively (one gigabyte left for the OS — *oh, how generous, my dear Spark!*!).

- Let's stop the worker to start over with custom configuration. You use `./sbin/stop-slave.sh` to stop the worker.

```
./sbin/stop-slave.sh
```

- Check out master's web UI at <http://localhost:8080> to know the current setup - one worker in **DEAD** state.

Worker Id	Address	State	Cores	Memory
worker-20150920142418-192.168.99.1-55334	192.168.99.1:55334	DEAD	8 (0 Used)	15.0 GB (0.0 B Used)

Figure 3. Master's web UI with one worker DEAD

- Start a worker using `--cores 2` and `--memory 4g` for two CPU cores and 4 GB of RAM.

```
./sbin/start-slave.sh spark://japila.local:7077 --cores 2 --mem
```

Note

The command translates to `org.apache.spark.deploy.worker.Worker --webui-port 8081 spark://japila.local:7077 --cores 2 --memory 4g`

- Check out master's web UI at <http://localhost:8080> to know the current setup - one worker **ALIVE** and another **DEAD**.

The screenshot shows the Spark Master's web interface at `spark://japila.local:7077`. It displays system statistics and a table of workers.

System Statistics:

- URL: `spark://japila.local:7077`
- REST URL: `spark://japila.local:6066 (cluster mode)`
- Alive Workers: 1
- Cores in use: 2 Total, 0 Used
- Memory in use: 4.0 GB Total, 0.0 B Used
- Applications: 0 Running, 0 Completed
- Drivers: 0 Running, 0 Completed
- Status: ALIVE

Workers Table:

Worker Id	Address	State	Cores	Memory
worker-20150920142418-192.168.99.1-55334	192.168.99.1:55334	DEAD	8 (0 Used)	15.0 GB (0.0 B Used)
worker-20150920144742-192.168.99.1-55538	192.168.99.1:55538	ALIVE	2 (0 Used)	4.0 GB (0.0 B Used)

Running Applications Table:

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
----------------	------	-------	-----------------	----------------	------	-------	----------

Completed Applications Table:

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
----------------	------	-------	-----------------	----------------	------	-------	----------

Figure 4. Master's web UI with one worker ALIVE and one DEAD

9. Configuring cluster using `conf/spark-env.sh`

There's the `conf/spark-env.sh.template` template to start from.

We're going to use the following `conf/spark-env.sh`:

`conf/spark-env.sh`

```
SPARK_WORKER_CORES=2 (1)
SPARK_WORKER_INSTANCES=2 (2)
SPARK_WORKER_MEMORY=2g
```

- the number of cores per worker
- the number of workers per node (a machine)

10. Start the workers.

```
./sbin/start-slave.sh spark://japila.local:7077
```

As the command progresses, it prints out *starting org.apache.spark.deploy.worker.Worker*, logging to for each worker. You defined two workers in `conf/spark-env.sh` using `SPARK_WORKER_INSTANCES`, so you should see two lines.

```
$ ./sbin/start-slave.sh spark://japila.local:7077
starting org.apache.spark.deploy.worker.Worker, logging to ../log/japila.local.out
starting org.apache.spark.deploy.worker.Worker, logging to ../log/japila.local.out
```

11. Check out master's web UI at <http://localhost:8080> to know the current setup - at least two workers should be **ALIVE**.

The screenshot shows the Spark Master web UI at localhost:8080. The title bar says "Spark Master at spark://japila.local:7077". The main content area displays the following information:

- URL:** spark://japila.local:7077
- REST URL:** spark://japila.local:6066 (cluster mode)
- Alive Workers:** 2
- Cores in use:** 4 Total, 0 Used
- Memory in use:** 4.0 GB Total, 0.0 B Used
- Applications:** 0 Running, 0 Completed
- Drivers:** 0 Running, 0 Completed
- Status:** ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20150920144742-192.168.99.1-55538	192.168.99.1:55538	DEAD	2 (0 Used)	4.0 GB (0.0 B Used)
worker-20150920150853-192.168.99.1-55669	192.168.99.1:55669	ALIVE	2 (0 Used)	2.0 GB (0.0 B Used)
worker-20150920150855-192.168.99.1-55671	192.168.99.1:55671	ALIVE	2 (0 Used)	2.0 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Figure 5. Master's web UI with two workers ALIVE

Use `jps` on master to see the instances given they all run on the same machine, e.g. `localhost`).

Note

```
$ jps
6580 Worker
4872 Master
6874 Jps
6539 Worker
```

12. Stop all instances - the driver and the workers.

```
./sbin/stop-all.sh
```

Spark on Mesos

Running Spark on Mesos

A Mesos cluster needs at least one Mesos Master to coordinate and dispatch tasks onto Mesos Slaves.

```
$ mesos-master --registry=in_memory --ip=127.0.0.1
I1119 12:30:22.153122 2107527936 main.cpp:229] Build: 2015-11-15 11:52:29 by brew
I1119 12:30:22.153700 2107527936 main.cpp:231] Version: 0.25.0
I1119 12:30:22.153736 2107527936 main.cpp:252] Using 'HierarchicalDRF' allocator
I1119 12:30:22.155131 2107527936 main.cpp:465] Starting Mesos master
I1119 12:30:22.161689 2107527936 master.cpp:376] Master 91979713-a32d-4e08-aaea-5dffbfd44
...

```

Visit the management console at <http://localhost:5050>.

The screenshot shows the Mesos Management Console interface. At the top, there is a navigation bar with tabs: Mesos, Frameworks, Slaves, and Offers. The 'Mesos' tab is selected, indicated by a dark background. Below the navigation bar, there is a header bar with the text "Master 91979713-a32d-4e08-aaea-5dffbfd44e5d".

Active Tasks:

ID	Name	State	Started ▾	Host
No active tasks.				

Completed Tasks:

ID	Name	State	Started ▾	Stopped	Host
No completed tasks.					

Logs:

- Slaves:**

Activated	0
Deactivated	0
- Tasks:**

Staged	0
Started	0
Finished	0
Killed	0
Failed	0
Lost	0
- Resources:**

	CPUs	Mem
Total	0	0 B
Used	0	0 B
Offered	0	0 B
Idle	0	0 B

Figure 1. Mesos Management Console

Run Mesos Slave onto which Master will dispatch jobs.

```
$ mesos-slave --master=127.0.0.1:5050
I1119 12:35:18.750463 2107527936 main.cpp:185] Build: 2015-11-15 11:52:29 by brew
I1119 12:35:18.750747 2107527936 main.cpp:187] Version: 0.25.0
I1119 12:35:18.753484 2107527936 containerizer.cpp:143] Using isolation: posix/cpu,posix/
I1119 12:35:18.763685 2107527936 main.cpp:272] Starting Mesos slave
I1119 12:35:18.766768 219140096 slave.cpp:190] Slave started on 1)@192.168.1.4:5051
...
I1119 12:35:18.787820 217530368 slave.cpp:741] Detecting new master
I1119 12:35:19.639030 218066944 slave.cpp:880] Registered with master master@127.0.0.1:5050
...
```

Switch to the management console at <http://localhost:5050/#/slaves> to see the slaves available.

ID ▾	Host	CPUs	Mem	Disk	Registered	Re-Registered
...aaea-5dffbfd44e5d-S0	192.168.1.4	8	15.0 GB	459.8 GB	a minute ago	

Figure 2. Mesos Management Console (Slaves tab) with one slave running

Important	<p>You have to export <code>MESOS_NATIVE_JAVA_LIBRARY</code> environment variable before connecting to the Mesos cluster.</p> <pre>\$ export MESOS_NATIVE_JAVA_LIBRARY=/usr/local/lib/libmesos.dylib</pre>
Note	<p>The preferred approach to launch Spark on Mesos and to give the location of Spark binaries is through <code>spark.executor.uri</code> setting.</p> <pre>--conf spark.executor.uri=/Users/jacek/Downloads/spark-1.5.2-bin-hadoop2.6.tgz</pre> <p>For us, on a bleeding edge of Spark development, it is very convenient to use <code>spark.mesos.executor.home</code> setting, instead.</p> <pre>-c spark.mesos.executor.home=`pwd`</pre>

```
$ ./bin/spark-shell --master mesos://127.0.0.1:5050 -c spark.mesos.executor.home=`pwd` ...
...
I1119 13:02:55.468960 747294720 sched.cpp:164] Version: 0.25.0
I1119 13:02:55.470788 698224640 sched.cpp:262] New master detected at master@127.0.0.1:50
I1119 13:02:55.470947 698224640 sched.cpp:272] No credentials provided. Attempting to reg
I1119 13:02:55.471592 734216192 sched.cpp:641] Framework registered with 91979713-a32d-4e
...

```

In [Frameworks tab](#) you should see a single active framework for `spark-shell`.

The screenshot shows the Mesos Management Console interface. At the top, there are tabs: 'Mesos', 'Frameworks' (which is selected and highlighted in blue), 'Slaves', and 'Offers'. Below the tabs, a breadcrumb navigation shows 'Master / Frameworks'. The main content area is titled 'Active Frameworks' and contains a table with one row. The table columns are: ID, Host, User, Name, Active Tasks, CPUs, Mem, Max Share, Registered, and Re-Registered. The single row shows an ID of '...aaea-5dffbd44e5d-0002', host 'japila.local', user 'jacek', name 'Spark shell', 0 active tasks, 0 CPUs, 0 B memory, 0% max share, registered 3 minutes ago, and last re-registered at the same time.

Figure 3. Mesos Management Console (Frameworks tab) with Spark shell active

Tip	Consult slave logs under <code>/tmp/mesos/slaves</code> when facing troubles.
-----	---

Important	Ensure that the versions of Spark of <code>spark-shell</code> and as pointed out by <code>spark.executor.uri</code> are the same or compatible.
-----------	---

```
scala> sc.parallelize(0 to 10, 8).count
res0: Long = 11
```

The screenshot shows the Mesos Management Console interface with the 'Slaves' tab selected. On the left, there's a sidebar with 'Executor Name:' (empty), 'Executor Source:' (empty), 'Cluster: (Unnamed)'), 'Master: localhost', 'Active Tasks: 0', and a 'Resources' table showing CPU usage (0.011 used, 1 allocated), Memory (402 MB used, 1.4 GB allocated), and Disk (0 B used, 0 B allocated). The main content area has three sections: 'Queued Tasks' (empty), 'Tasks' (empty), and 'Completed Tasks'. The 'Completed Tasks' section contains a table with 8 rows, each representing a task finished in stage 0.0. The columns are: ID, Name, State, CPUs (allocated), Mem (allocated), and Executor. The tasks are: task 7.0 in stage 0.0, task 6.0 in stage 0.0, task 5.0 in stage 0.0, task 4.0 in stage 0.0, task 3.0 in stage 0.0, task 2.0 in stage 0.0, task 1.0 in stage 0.0, and task 0.0 in stage 0.0. All tasks have a state of 'TASK_FINISHED', 1 CPU allocated, 0 B memory allocated, and are associated with the 'Sandbox' executor.

Figure 4. Completed tasks in Mesos Management Console

Stop Spark shell.

```
scala> Stopping spark context.
I1119 16:01:37.831179 206073856 sched.cpp:1771] Asked to stop the driver
I1119 16:01:37.831310 698224640 sched.cpp:1040] Stopping framework '91979713-a32d-4e08-aa
```

CoarseMesosSchedulerBackend

`CoarseMesosSchedulerBackend` is the [scheduler backend](#) for Spark on Mesos.

It requires a [Task Scheduler](#), [Spark context](#), `mesos://` master URL, and [Security Manager](#).

It is a specialized [CoarseGrainedSchedulerBackend](#) and implements Mesos's [org.apache.mesos.Scheduler](#) interface.

It accepts only two failures before blacklisting a Mesos slave (it is hardcoded and not configurable).

It tracks:

- the number of tasks already submitted (`nextMesosTaskId`)
- the number of cores per task (`coresByTaskId`)
- the total number of cores acquired (`totalCoresAcquired`)
- slave ids with executors (`slaveIdsWithExecutors`)
- slave ids per host (`slaveIdToHost`)
- task ids per slave (`taskIdToSlaveId`)
- How many times tasks on each slave failed (`failuresBySlaveId`)

Tip

`createSchedulerDriver` instantiates Mesos's [org.apache.mesos.MesosSchedulerDriver](#)

`CoarseMesosSchedulerBackend` starts the **MesosSchedulerUtils-mesos-driver** daemon thread with Mesos's [org.apache.mesos.MesosSchedulerDriver](#).

Default Level of Parallelism

The default parallelism is controlled by [spark.default.parallelism](#).

Settings

- `spark.default.parallelism` (default: 8) - the number of cores to use as [Default Level of Parallelism](#).
- `spark.cores.max` (default: Int.MaxValue) - maximum number of cores to acquire
- `spark.mesos.extra.cores` (default: 0) - extra cores per slave (`extraCoresPerSlave`)
[FIXME](#)
- `spark.mesos.constraints` (default: (empty)) - offer constraints [FIXME](#)
`slaveOfferConstraints`
- `spark.mesos.rejectOfferDurationForUnmetConstraints` (default: 120s) - reject offers with mismatched constraints in seconds
- `spark.mesos.executor.home` (default: SPARK_HOME) - the home directory of Spark for executors. It is only required when no `spark.executor.uri` is set.

MesosExternalShuffleClient

[FIXME](#)

(Fine)MesosSchedulerBackend

When `spark.mesos.coarse` is false , Spark on Mesos uses `MesosSchedulerBackend`

reviveOffers

It calls `mesosDriver.reviveOffers()` .

Caution	FIXME
---------	-----------------------

Settings

- `spark.mesos.coarse` (default: true) controls whether the scheduler backend for Mesos works in coarse- (`CoarseMesosSchedulerBackend`) or fine-grained mode (`MesosSchedulerBackend`).

	FIXME Review
Caution	<ul style="list-style-type: none"> • MesosClusterScheduler.scala • MesosExternalShuffleService

Introduction to Mesos

- Mesos is a *distributed system kernel*
- Mesos essentially uses a container architecture but is abstracted enough to allow seamless execution of multiple, sometimes identical, distributed systems on the same architecture, minus the resource overhead of virtualization systems. This includes appropriate resource isolation while still allowing for data locality needed for frameworks like MapReduce.
- Mesos' computation-agnostic approach makes it suitable for
 - | Program against your datacenter as a single pool of resources.
- Concepts in Mesos:
 - **(Resource) Offers**, i.e. CPU cores, memory, ports, disk
 - Mesos *offers resources* to frameworks
 - **Frameworks**
 - Frameworks *accept or reject offers*
 - (Mesos-specific) Chronos, Marathon
 - Spark, HDFS, YARN (Myriad), Jenkins, Cassandra
 - Mesos master
 - Mesos API
 - Mesos agent
- Mesos is a *scheduler of schedulers*
 - Mesos is really an additional layer of (resource) scheduling on top of application frameworks that each bring their own brand of scheduling. Application schedulers interface with a Mesos master setup in a familiar Zookeeper-coordinated active-passive architecture, which passes jobs down to compute slaves to run the application of choice.
- Mesos assigns jobs
- Mesos typically runs with an agent on every virtual machine or bare metal server under management (<https://www.joyent.com/blog/mesos-by-the-pound>)
- Mesos uses Zookeeper for master election and discovery. Apache Auroa is a scheduler that runs on Mesos.
- Mesos slaves, masters, schedulers, executors, tasks

- Mesos makes use of event-driven message passing.
- Mesos is written in C++, not Java, and includes support for Docker along with other frameworks. Mesos, then, is the core of the Mesos Data Center Operating System, or DCOS, as it was coined by Mesosphere.
- This Operating System includes other handy components such as Marathon and Chronos. Marathon provides cluster-wide “init” capabilities for application in containers like Docker or cgroups. This allows one to programmatically automate the launching of large cluster-based applications. Chronos acts as a Mesos API for longer-running batch type jobs while the core Mesos SDK provides an entry point for other applications like Hadoop and Spark.
- The true goal is a full shared, generic and reusable on demand distributed architecture.
- [Infinity](#) to package and integrate the deployment of clusters
 - Out of the box it will include Cassandra, Kafka, Spark, and Akka.
 - an early access project
- Apache Myriad = Integrate YARN with Mesos
 - making the execution of YARN work on Mesos scheduled systems transparent, multi-tenant, and smoothly managed
 - to allow Mesos to centrally schedule YARN work via a Mesos based framework, including a REST API for scaling up or down
 - includes a Mesos executor for launching the node manager

Schedulers in Mesos

Available scheduler modes:

- **fine-grained mode**
- **coarse-grained mode** - `spark.mesos.coarse=true`

The main difference between these two scheduler modes is the number of tasks per Spark executor per single Mesos executor. In fine-grained mode, there is a single task in a single Spark executor that shares a single Mesos executor with the other Spark executors. In coarse-grained mode, there is a single Spark executor per Mesos executor with many Spark tasks.

Coarse-grained mode pre-starts all the executor backends, e.g. [Executor Backends](#), so it has the least overhead comparing to **fine-grain mode**. Since the executors are up before tasks get launched, it is better for interactive sessions. It also means that the resources are locked up in a task.

Spark on Mesos supports [dynamic allocation](#) in the Mesos coarse-grained scheduler since Spark 1.5. It can add/remove executors based on load, i.e. kills idle executors and adds executors when tasks queue up. It needs an [external shuffle service](#) on each node.

Mesos Fine-Grained Mode offers a better resource utilization. It has a slower startup for tasks and hence it is fine for batch and relatively static streaming.

Commands

The following command is how you could execute a Spark application on Mesos:

```
./bin/spark-submit --master mesos://iq-cluster-master:5050 --total-executor-cores 2 --exe
```

Other Findings

From [Four reasons to pay attention to Apache Mesos](#):

Spark workloads can also be sensitive to the physical characteristics of the infrastructure, such as memory size of the node, access to fast solid state disk, or proximity to the data source.

to run Spark workloads well you need a resource manager that not only can handle the rapid swings in load inherent in analytics processing, but one that can do so smartly. Matching of the task to the RIGHT resources is crucial and awareness of the physical environment is a must. Mesos is designed to manage this problem on behalf of workloads like Spark.

Spark on YARN

Spark on YARN supports [multiple application attempts](#).

Introduction to YARN

[Introduction to YARN](#) by **Adam Kawa** is an excellent introduction to YARN. Here are the most important facts to get you going.

- Hadoop 2.0 comes with **Yet Another Resource Negotiator (YARN)** that is a *generic cluster resource management framework that can run applications on a Hadoop cluster*. (see [Apache Twill](#))
- YARN model of computation (aka YARN components):
 - **ResourceManager** runs as a master daemon and manages ApplicationMasters and NodeManagers.
 - **ApplicationMaster** is a lightweight process that coordinates the execution of tasks of an application and asks the ResourceManager for resource containers for tasks. It monitors tasks, restarts failed ones, etc. It can run any type of tasks, be them MapReduce tasks or Giraph tasks, or Spark tasks.
 - **NodeManager** offers resources (memory and CPU) as resource containers.
 - **NameNode**
 - **Container** can run tasks, including ApplicationMasters.
- YARN manages distributed applications.
- YARN offers (macro-level) container allocation.
- Hadoop for storing and processing large amount of data on a cluster of commodity hardware.
- The Pre-YARN MapReduce engine - **MRv1** - was rewritten for YARN. It became yet another YARN distributed application called **MRv2**.

There's another article that covers the fundamentals of YARN - [Untangling Apache Hadoop YARN, Part 1](#). Notes follow:

- A **host** is the Hadoop term for a computer (also called a **node**, in YARN terminology).
- A **cluster** is two or more hosts connected by a high-speed local network.

- It can technically also be a single host used for debugging and simple testing.
 - Master hosts are a small number of hosts reserved to control the rest of the cluster.
Worker hosts are the non-master hosts in the cluster.
 - A **master** host is the communication point for a client program. A master host sends the work to the rest of the cluster, which consists of **worker** hosts.
- In a YARN cluster, there are two types of hosts:
 - The **ResourceManager** is the master daemon that communicates with the client, tracks resources on the cluster, and orchestrates work by assigning tasks to NodeManagers.
 - In a Hadoop cluster with YARN running, the master process is called the ResourceManager and the worker processes are called NodeManagers.
 - A **NodeManager** is a worker daemon that launches and tracks processes spawned on worker hosts.
 - The NodeManager on each host keeps track of the local host's resources, and the ResourceManager keeps track of the cluster's total.
 - The YARN configuration file is an XML file that contains properties. This file is placed in a well-known location on each host in the cluster and is used to configure the ResourceManager and NodeManager. By default, this file is named `yarn-site.xml`.
 - YARN currently defines two resources, vcores and memory.
 - vcore = usage share of a CPU core.
 - Each NodeManager tracks its own local resources and communicates its resource configuration to the ResourceManager, which keeps a running total of the cluster's available resources.
 - By keeping track of the total, the ResourceManager knows how to allocate resources as they are requested.
 - A **container** in YARN holds resources on the YARN cluster.
 - A container hold request consists of vcore and memory.
 - Once a hold has been granted on a host, the NodeManager launches a process called a **task**.
 - An application is a YARN client program that is made up of one or more tasks.

- For each running application, a special piece of code called an ApplicationMaster helps coordinate tasks on the YARN cluster. The ApplicationMaster is the first process run after the application starts.
- An application in YARN comprises three parts:
 - The application client, which is how a program is run on the cluster.
 - An ApplicationMaster which provides YARN with the ability to perform allocation on behalf of the application.
 - One or more tasks that do the actual work (runs in a process) in the container allocated by YARN.
- An application running tasks on a YARN cluster consists of the following steps:
 - The application starts and talks to the ResourceManager (running on the master) for the cluster.
 - The ResourceManager makes a single container request on behalf of the application.
 - The ApplicationMaster starts running within that container.
 - The ApplicationMaster requests subsequent containers from the ResourceManager that are allocated to run tasks for the application. Those tasks do most of the status communication with the ApplicationMaster.
 - Once all tasks are finished, the ApplicationMaster exits. The last container is deallocated from the cluster.
 - The application client exits. (The ApplicationMaster launched in a container is more specifically called a managed AM).
- The ResourceManager, NodeManager, and ApplicationMaster work together to manage the cluster's resources and ensure that the tasks, as well as the corresponding application, finish cleanly.

Caution	<p>FIXME: Where is <code>ApplicationMaster.registerAM</code> used?</p> <ul style="list-style-type: none">• Registering the ApplicationMaster with the RM.• Contains a map with hints about where to allocate containers.
---------	---

Hadoop YARN

From [Apache Hadoop's web site](#):

Hadoop YARN: A framework for job scheduling and cluster resource management.

- YARN could be considered a cornerstone of Hadoop OS (operating system) for big distributed data with HDFS as the storage along with YARN as a process scheduler.
- YARN is essentially a container system and scheduler designed primarily for use with a Hadoop-based cluster.
- The containers in YARN are capable of running various types of tasks.
- Resource manager, node manager, container, application master, jobs
- focused on data storage and offline batch analysis
- Hadoop is storage and compute platform:
 - MapReduce is the computing part.
 - HDFS is the storage.
- Hadoop is a resource and cluster manager (YARN)
- Spark runs on YARN clusters, and can read from and save data to HDFS.
 - leverages [data locality](#)
- Spark needs distributed file system and HDFS (or Amazon S3, but slower) is a great choice.
- HDFS allows for [data locality](#).
- Excellent throughput when Spark and Hadoop are both distributed and co-located on the same (YARN or Mesos) cluster nodes.
- HDFS offers (important for initial loading of data):
 - high data locality
 - high throughput when co-located with Spark
 - low latency because of data locality
 - very reliable because of replication
- When reading data from HDFS, each `InputSplit` maps to exactly one Spark partition.
- HDFS is distributing files on data-nodes and storing a file on the filesystem, it will be split into partitions.

How it works

The Spark driver in Spark on YARN launches a number of executors. Each executor processes a partition of HDFS-based data.

YarnAllocator

`YarnAllocator` requests containers from the YARN ResourceManager and decides what to do with containers when YARN fulfills these requests. It uses YARN's AMRMClient APIs.

ExecutorAllocationClient

ExecutorAllocationClient is a client class that communicates with the cluster manager to request or kill executors.

This is currently supported only in YARN mode.

Caution

FIXME See the code and deduce its use.

Misc

- `SPARK_YARN_MODE` property and environment variable
 - `true` when `yarn-client` used for master URL
 - It's set by Spark internally for YARN mode
- `yarn-cluster` and `yarn-client` modes
- `spark-submit --deploy-mode cluster`
- `org.apache.spark.deploy.yarn.YarnSparkHadoopUtil`
- YARN integration has some advantages, like [dynamic allocation](#). If you enable dynamic allocation, after the stage including InputSplits gets submitted, Spark will try to request an appropriate number of executors.
- On YARN, a Spark executor maps to a single YARN container.
- The memory in the YARN resource requests is `--executor-memory` + what's set for `spark.yarn.executor.memoryOverhead`, which defaults to 10% of `--executor-memory`.
- if YARN has enough resources it will deploy the executors distributed across the cluster, then each of them will try to process the data locally (`NODE_LOCAL` in Spark Web UI), with as many splits in parallel as you defined in `spark.executor.cores`.
- "*YarnClusterScheduler: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources*"

- Mandatory settings (`spark-defaults.conf`) for dynamic allocation:

<code>spark.dynamicAllocation.enabled</code>	<code>true</code>
<code>spark.shuffle.service.enabled</code>	<code>true</code>

- Optional settings for dynamic allocation (to tune it):

<code>spark.dynamicAllocation.minExecutors</code>	<code>0</code>
<code>spark.dynamicAllocation.maxExecutors</code>	<code>N</code>
<code>spark.dynamicAllocation.initialExecutors</code>	<code>0</code>

- `spark.dynamicAllocation.minExecutors` requires `spark.dynamicAllocation.initialExecutors`
- Review `spark.dynamicAllocation.*` settings
- YARN UI under scheduler - pools where Spark operates

Cluster Mode

Spark on YARN supports submitting Spark applications in [cluster deploy mode](#).

In cluster deploy mode Spark on YARN uses [YarnClusterSchedulerBackend](#).

YarnClusterSchedulerBackend

`YarnClusterSchedulerBackend` is a [scheduler backend](#) for Spark on YARN in [cluster deploy mode](#).

This is the only scheduler backend that supports [multiple application attempts](#) and [URLs for driver's logs](#) to display as links in the web UI in the Executors tab for the driver.

It uses `spark.yarn.app.attemptId` under the covers (that the YARN resource manager sets?).

Multiple Application Attempts

Spark on YARN supports **multiple application attempts** in [Cluster Mode](#).

Caution	FIXME
---------	-----------------------

YarnScheduler

Caution	FIXME Review
---------	------------------------------

It appears that this is a custom implementation to keep track of racks per host that is used in [TaskSetManager.resourceOffer](#) to find a task with `RACK_LOCAL` locality preferences.

Execution Model

Caution

FIXME This is the **single** place for explaining jobs, stages, tasks. Move relevant parts from the other places.

Broadcast Variables

From [the official documentation about Broadcast Variables](#):

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

And later in the document:

Explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

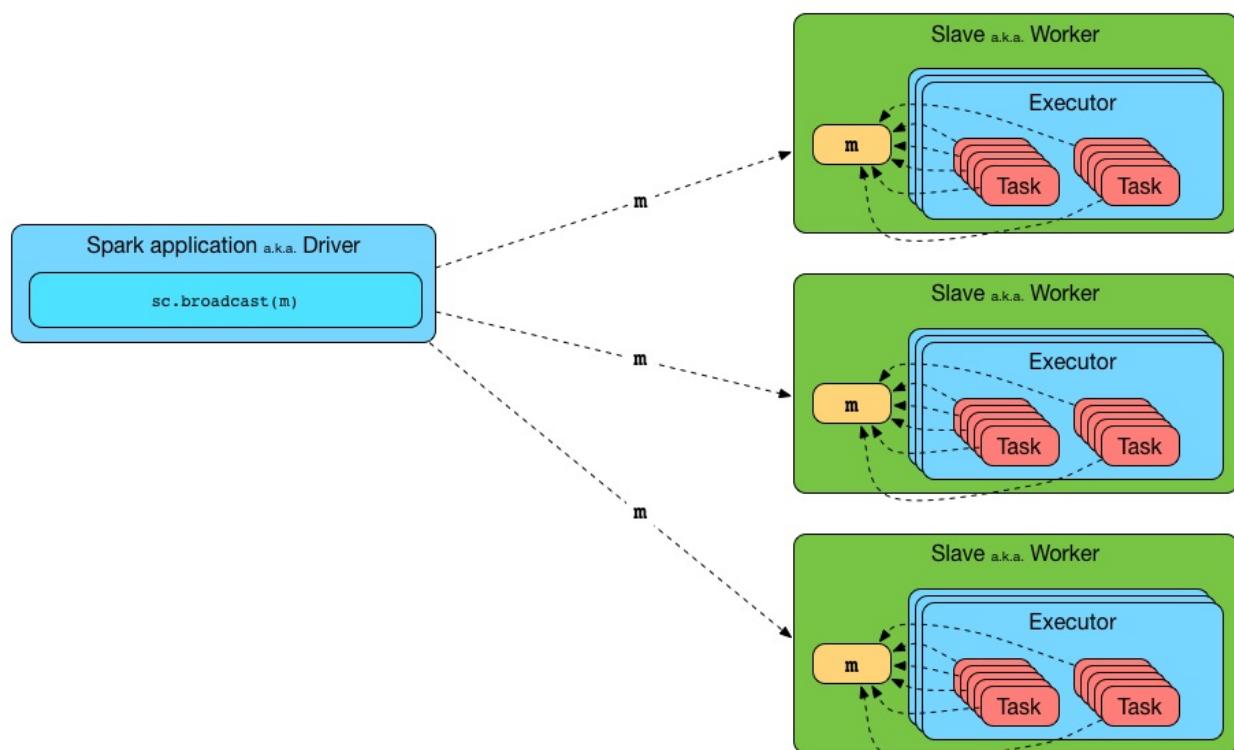


Figure 1. Broadcasting a value to executors

To use a broadcast value in a transformation you have to create it first using `SparkContext.broadcast()` and then use `value` method to access the shared value. Learn it in [Introductory Example](#).

The Broadcast feature in Spark uses `SparkContext` to create broadcast values and `BroadcastManager` and `ContextCleaner` to manage their lifecycle.

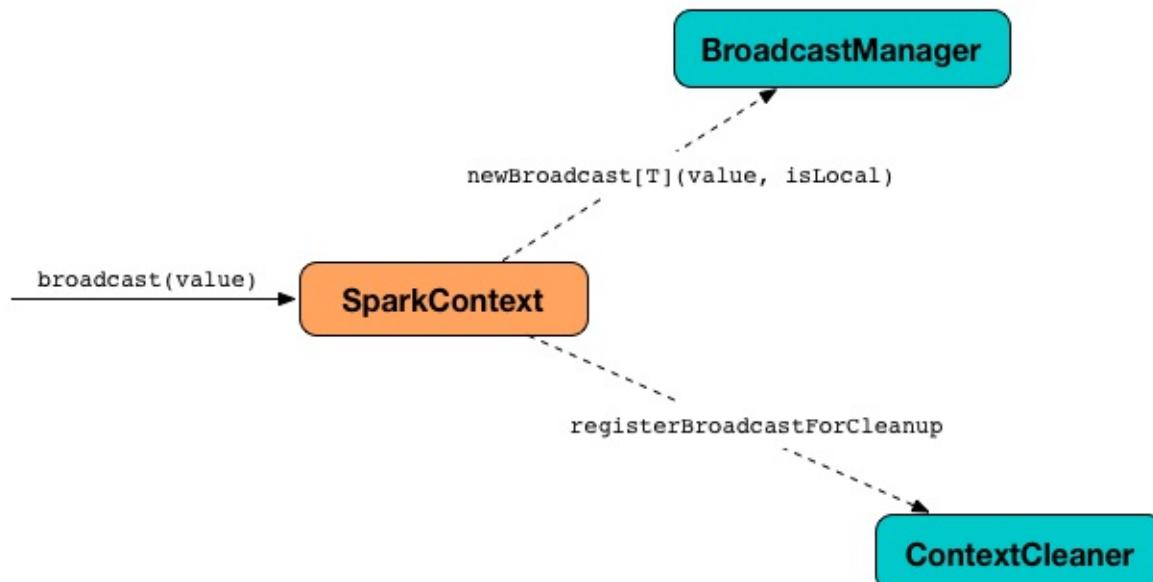


Figure 2. SparkContext to broadcast using BroadcastManager and ContextCleaner

Introductory Example

Let's start with an introductory example to check out how to use broadcast variables and build your initial understanding.

You're going to use a static mapping of interesting projects with their websites, i.e.

`Map[String, String]` that the tasks, i.e. closures (anonymous functions) in transformations, use.

```

scala> val pws = Map("Apache Spark" -> "http://spark.apache.org/", "Scala" -> "http://www
pws: scala.collection.immutable.Map[String,String] = Map(Apache Spark -> http://spark.apa

scala> val websites = sc.parallelize(Seq("Apache Spark", "Scala")).map(pws).collect
...
websites: Array[String] = Array(http://spark.apache.org/, http://www.scala-lang.org/)
  
```

It works, but is very ineffective as the `pws` map is sent over the wire to executors while it could have been there already. If there were more tasks that need the `pws` map, you could improve their performance by minimizing the number of bytes that are going to be sent over the network for task execution.

Enter broadcast variables.

```
scala> val pwsB = sc.broadcast(pws)
INFO MemoryStore: Ensuring 1048576 bytes of free space for block broadcast_8(free: 771881
INFO MemoryStore: Ensuring 360 bytes of free space for block broadcast_8(free: 771881313,
INFO MemoryStore: Block broadcast_8 stored as values in memory (estimated size 360.0 B, f
INFO MemoryStore: Ensuring 205 bytes of free space for block broadcast_8_piece0(free: 771
INFO MemoryStore: Block broadcast_8_piece0 stored as bytes in memory (estimated size 205.
INFO BlockManagerInfo: Added broadcast_8_piece0 in memory on localhost:54142 (size: 205.0
INFO SparkContext: Created broadcast 8 from broadcast at <console>:26
pwsB: org.apache.spark.broadcast.Broadcast[scala.collection.immutable.Map[String, String]]

scala> val websites = sc.parallelize(Seq("Apache Spark", "Scala")).map(pwsB.value).collect
...
websites: Array[String] = Array(http://spark.apache.org/, http://www.scala-lang.org/)
```

Semantically, the two computations - with and without the broadcast value - are exactly the same, but the broadcast-based one wins performance-wise when there are more executors spawned to execute many tasks that use `pws` map.

Introduction

Broadcast is part of Spark that is responsible for broadcasting information across nodes in a cluster.

When you broadcast a value, it is copied to executors only once (while it is copied multiple times for tasks otherwise). It means that broadcast can help to get your Spark application faster if you have a large value to use in tasks or there are more tasks than executors.

Spark comes with a BitTorrent implementation.

It is not enabled by default.

- Use large broadcasted HashMaps over RDDs whenever possible and leave RDDs with a key to lookup necessary data — [FIXME](#): elaborate why

SparkContext.broadcast

Read about `SparkContext.broadcast` method in [Creating broadcast variables](#).

Accumulators in Spark

Tip

Read [the latest documentation about accumulators](#) before looking for anything useful here.

Each task creates its own local accumulator.

Noticed on the user@spark mailing list that using an external key-value store (like HBase, Redis, Cassandra) and performing lookups/updates inside of your mappers (creating a connection within a `mapPartitions` code block to avoid the connection setup/teardown overhead) might be a better solution.

If hbase is used as the external key value store, atomicity is guaranteed

[Performance and Scalability of Broadcast in Spark](#)

Spark Security

- Enable security via `spark.authenticate` property (defaults to `false`).
- See `org.apache.spark.SecurityManager`
- Enable `INFO` for `org.apache.spark.SecurityManager` to see messages regarding security in Spark.
- Enable `DEBUG` for `org.apache.spark.SecurityManager` to see messages regarding SSL in Spark, namely file server and Akka.

Securing Web UI

Tip	Read the official document Web UI .
-----	---

To secure Web UI you implement a security filter and use `spark.ui.filters` setting to refer to the class.

Examples of filters implementing basic authentication:

- [Servlet filter for HTTP basic auth](#)
- [neolitec/BasicAuthenticationFilter.java](#)

Data Sources in Spark

Spark can access data from many data sources, including [Hadoop Distributed File System \(HDFS\)](#), [Cassandra](#), [HBase](#), [S3](#) and many more.

Spark offers different APIs to read data based upon the content and the storage.

There are two groups of data based upon the content:

- binary
- text

You can also group data by the storage:

- [files](#)
- databases, e.g. [Cassandra](#)

Using Input and Output (I/O)

Caution

FIXME What are the differences between `textFile` and the rest methods in `SparkContext` like `newAPIHadoopRDD`, `newAPIHadoopFile`, `hadoopFile`, `hadoopRDD` ?

From [SPARK AND MERGED CSV FILES](#):

Spark is like Hadoop - uses Hadoop, in fact - for performing actions like outputting data to HDFS. You'll know what I mean the first time you try to save "all-the-data.csv" and are surprised to find a directory named all-the-data.csv/ containing a 0 byte _SUCCESS file and then several part-0000n files for each partition that took part in the job.

The read operation is lazy - it is [a transformation](#).

Methods:

- [SparkContext.textFile\(path: String, minPartitions: Int = defaultMinPartitions\): RDD\[String\]](#) reads a text data from a file from a remote HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI (e.g. sources in HBase or S3) at `path`, and automatically distributes the data across a Spark cluster as an RDD of Strings.
 - Uses Hadoop's [org.apache.hadoop.mapred.InputFormat](#) interface and file-based [org.apache.hadoop.mapred.FileInputFormat](#) class to read.
 - Uses the global Hadoop's `configuration` with all `spark.hadoop.xxx=yyy` properties mapped to `xxx=yyy` in the configuration.
 - `io.file.buffer.size` is the value of `spark.buffer.size` (default: `65536`).
 - Returns [HadoopRDD](#)
 - When using `textFile` to read an HDFS folder with multiple files inside, the number of partitions are equal to the number of HDFS blocks.
- What does `sc.binaryFiles` ?

URLs supported:

- `s3://...` or `s3n://...`
- `hdfs://...`
- `file://...;`

The general rule seems to be to use HDFS to read files multiple times with S3 as a storage for a one-time access.

Creating RDDs from Input

[FIXME](#)

```
sc.newAPIHadoopFile("filepath1, filepath2", classOf[NewTextInputFormat], classOf[LongWrit
```



Saving RDDs to files - saveAs* actions

An RDD can be saved to a file using the following actions:

- `saveAsTextFile`
- `saveAsObjectFile`
- `saveAsSequenceFile`
- `saveAsHadoopFile`

Since an RDD is actually a set of partitions that make for it, saving an RDD to a file saves the content of each partition to a file (per partition).

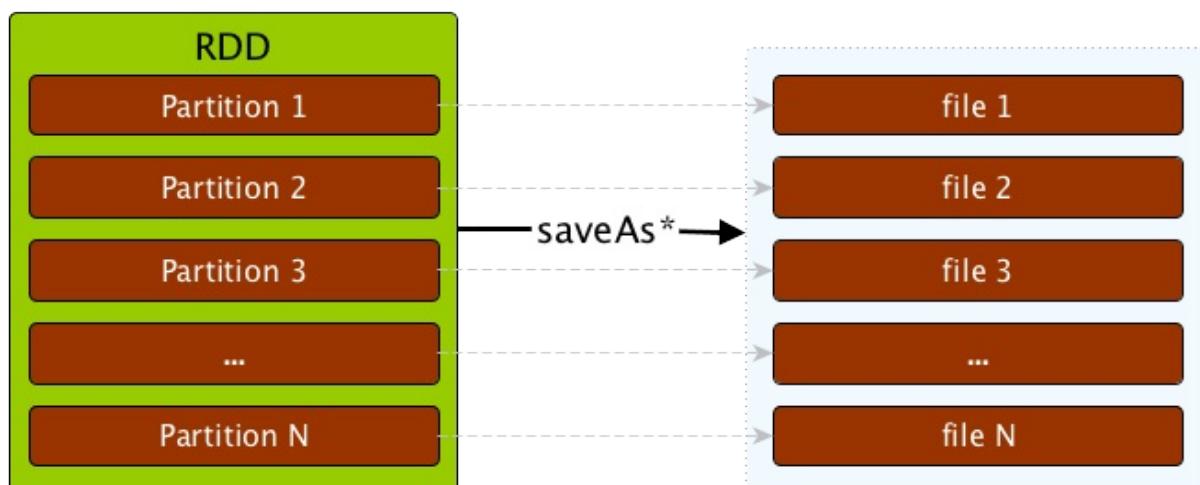


Figure 1. `saveAs` on RDD

If you want to reduce the number of files, you will need to [repartition](#) the RDD you are saving to the number of files you want, say 1.

```
scala> sc.parallelize(0 to 10, 4).saveAsTextFile("numbers") (1)
...
INFO FileOutputCommitter: Saved output of task 'attempt_201511050904_0000_m_000001_1' to
INFO FileOutputCommitter: Saved output of task 'attempt_201511050904_0000_m_000002_2' to
INFO FileOutputCommitter: Saved output of task 'attempt_201511050904_0000_m_000000_0' to
INFO FileOutputCommitter: Saved output of task 'attempt_201511050904_0000_m_000003_3' to
...
scala> sc.parallelize(0 to 10, 4).repartition(1).saveAsTextFile("numbers1") (2)
...
INFO FileOutputCommitter: Saved output of task 'attempt_201511050907_0002_m_000000_8' to
```

1. `parallelize` uses `4` to denote the number of partitions so there are going to be 4 files saved.
2. `repartition(1)` to reduce the number of the files saved to 1.

S3

`s3://...` or `s3n://...` URL are supported.

Upon executing `sc.textFile`, it checks for `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. They both have to be set to have the keys `fs.s3.awsAccessKeyId`, `fs.s3n.awsAccessKeyId`, `fs.s3.awsSecretAccessKey`, and `fs.s3n.awsSecretAccessKey` set up (in the Hadoop configuration).

textFile reads compressed files

```
scala> val f = sc.textFile("f.txt.gz")
f: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[5] at textFile at <console>:24

scala> f.foreach(println)
...
15/09/13 19:06:52 INFO HadoopRDD: Input split: file:/Users/jacek/dev/oss/spark/f.txt.gz:0
15/09/13 19:06:52 INFO CodecPool: Got brand-new decompressor [.gz]
Ala ma kota
```

Reading Sequence Files

- `sc.sequenceFile`
 - if the directory contains multiple `SequenceFiles` all of them will be added to RDD
- `SequenceFile RDD`

Changing log levels

Create `conf/log4j.properties` out of the Spark template:

```
cp conf/log4j.properties.template conf/log4j.properties
```

Edit `conf/log4j.properties` so the line `log4j.rootCategory` uses appropriate log level, e.g.

```
log4j.rootCategory=ERROR, console
```

If you want to do it from the code instead, do as follows:

```
import org.apache.log4j.Logger  
import org.apache.log4j.Level  
  
Logger.getLogger("org").setLevel(Level.OFF)  
Logger.getLogger("akka").setLevel(Level.OFF)
```

FIXME

Describe the other computing models using Spark SQL, MLlib, Spark Streaming, and GraphX.

```
$ ./bin/spark-shell
...
Spark context available as sc.
...
SQL context available as sqlContext.
Welcome to

   __
  / _/_
  _\ \V_ _\V_ _`/ _/_/ '_/
 /__/ .__/\_,/_/_/ /_\`/    version 1.5.0-SNAPSHOT
  /_/

Using Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60)
Type in expressions to have them evaluated.
Type :help for more information.

scala> sc.addFile("/Users/jacek/dev/sandbox/hello.json")

scala> import org.apache.spark.SparkFiles
import org.apache.spark.SparkFiles

scala> SparkFiles.get("/Users/jacek/dev/sandbox/hello.json")
```

See [org.apache.spark.SparkFiles](#).

Caution

Review the classes in the following stacktrace.

```
scala> sc.textFile("http://japila.pl").foreach(println)
java.io.IOException: No FileSystem for scheme: http
  at org.apache.hadoop.fs.FileSystem.getFileSystemClass(FileSystem.java:2644)
  at org.apache.hadoop.fs.FileSystem.createFileSystem(FileSystem.java:2651)
  at org.apache.hadoop.fs.FileSystem.access$200(FileSystem.java:92)
  at org.apache.hadoop.fs.FileSystem$Cache.getInternal(FileSystem.java:2687)
  at org.apache.hadoop.fs.FileSystem$Cache.get(FileSystem.java:2669)
  at org.apache.hadoop.fs.FileSystem.get(FileSystem.java:371)
  at org.apache.hadoop.fs.Path.getFileSystem(Path.java:295)
  at org.apache.hadoop.mapred.FileInputFormat.singleThreadedListStatus(FileInputFormat.java:229)
  at org.apache.hadoop.mapred.FileInputFormat.listStatus(FileInputFormat.java:229)
  at org.apache.hadoop.mapred.FileInputFormat.getSplits(FileInputFormat.java:315)
  at org.apache.spark.rdd.HadoopRDD.getPartitions(HadoopRDD.scala:207)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:239)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:237)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.rdd.RDD.partitions(RDD.scala:237)
  at org.apache.spark.rdd.MapPartitionsRDD.getPartitions(MapPartitionsRDD.scala:35)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:239)
  at org.apache.spark.rdd.RDD$$anonfun$partitions$2.apply(RDD.scala:237)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.rdd.RDD.partitions(RDD.scala:237)
...

```

Spark and Parquet

Apache Parquet is a **columnar storage** format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.

Spark 1.5 uses Parquet 1.7.

- excellent for local file storage on HDFS (instead of external databases).
- writing very large datasets to disk
- supports **schema**
 - **schema evolution**
- faster than json/gzip
- Used in Spark SQL
- `DataFrame.write.parquet(destination)`

Serialization

Serialization systems:

- Java serialization
- Kryo
- Avro
- Thrift
- Protobuf

Using Cassandra in Spark

[Excellent write-up about how to run Cassandra inside Docker from DataStax. Read it first.](#)

Kafka and Spark

Caution	<p>FIXME:</p> <ol style="list-style-type: none">1. Kafka Direct API in Spark Streaming2. Getting information on the current topic being consumed by each executor
---------	---

Spark Application Frameworks

Spark's application frameworks are libraries (components) that aim at simplifying development of distributed applications on top of Spark.

They further abstract the concept of [RDD](#) and leverage [the resiliency and distribution of the computing platform](#).

There are the following built-in libraries that all constitute **the Spark platform**:

- [Spark SQL](#)
- [Spark GraphX](#)
- [Spark Streaming](#)
- [Spark MLlib](#)

Spark Streaming

Spark Streaming runs [streaming jobs per batch](#) (time) to pull and process data (often called *records*) from one or many [input streams](#) periodically at fixed [batch interval](#). Each batch [computes](#) ([generates](#)) RDDs based on data received from input streams and [submits a](#) Spark job to compute the final result for a batch. It does this over and over again until [the streaming context is stopped](#) (and the owning streaming application terminated).

To avoid losing records in case of failure, Spark Streaming supports [checkpointing that writes received records to a highly-available HDFS-compatible storage](#) and allows to recover from temporary downtimes.

Checkpointing is also the foundation of [stateful](#) and [windowed](#) operations.

[About Spark Streaming from the official documentation](#) (that pretty much nails what it offers):

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams.

Essential concepts in Spark Streaming:

- [StreamingContext](#)
- [Stream Operators](#)
- [Batch](#), [Batch time](#), and [JobSet](#)
- [Streaming Job](#)
- [Discretized Streams \(DStreams\)](#)
- [Receivers](#)

Other concepts often used in Spark Streaming:

- **ingestion** = the act of processing streaming data.

Micro Batch

Micro Batch is a collection of input records as collected by Spark Streaming that is later represented as an RDD.

A **batch** is internally represented as a [JobSet](#).

Batch Interval (aka batchDuration)

Batch Interval is a property of a Streaming application that describes how often an RDD of input records is generated. It is the time to collect input records before they become a [micro-batch](#).

Streaming Job

A streaming `Job` represents a Spark computation with one or many Spark jobs.

It is identified (in the logs) as `streaming job [time].[outputOpId]` with `outputOpId` being the position in the sequence of jobs in a [JobSet](#).

When executed, it runs the computation (the input `func` function).

Note

A collection of streaming jobs is generated for a batch using [DStreamGraph.generateJobs\(time: Time\)](#).

Internal Registries

- `nextInputStreamId` - the current InputStream id

StreamingSource

Caution

[FIXME](#)

StreamingContext

`StreamingContext` is the main entry point for all Spark Streaming functionality. Whatever you do in Spark Streaming has to start from [creating an instance of StreamingContext](#).

Note	<code>StreamingContext</code> belongs to <code>org.apache.spark.streaming</code> package.
------	---

With an instance of `StreamingContext` in your hands, you can [create ReceiverInputDStreams](#) or [set the checkpoint directory](#).

Once streaming pipelines are developed, you [start StreamingContext](#) to set the stream transformations in motion. You [stop](#) the instance when you are done.

Creating Instance

You can create a new instance of `StreamingContext` using the following constructors. You can group them by whether a `StreamingContext` constructor creates it from scratch or it is recreated from checkpoint directory (follow the links for their extensive coverage).

- [Creating StreamingContext from scratch:](#)
 - `StreamingContext(conf: SparkConf, batchDuration: Duration)`
 - `StreamingContext(master: String, appName: String, batchDuration: Duration, sparkHome: String, jars: Seq[String], environment: Map[String, String])`
 - `StreamingContext(sparkContext: SparkContext, batchDuration: Duration)`
- [Recreating StreamingContext from a checkpoint file](#) (where `path` is the [checkpoint directory](#)):
 - `StreamingContext(path: String)`
 - `StreamingContext(path: String, hadoopConf: Configuration)`
 - `StreamingContext(path: String, sparkContext: SparkContext)`

Note	<code>StreamingContext(path: String)</code> uses SparkHadoopUtil.get.conf .
------	---

Note	When a <code>StreamingContext</code> is created and <code>spark.streaming.checkpoint.directory</code> setting is set, the value gets passed on to <code>checkpoint</code> method.
------	---

Creating StreamingContext from Scratch

When you create a new instance of `StreamingContext`, it first checks whether a `SparkContext` or the `checkpoint directory` are given (but not both!)

	<code>StreamingContext</code> will warn you when you use <code>local</code> or <code>local[1]</code> master URLs:
Tip	<code>WARN StreamingContext: spark.master should be set as local[n]</code>

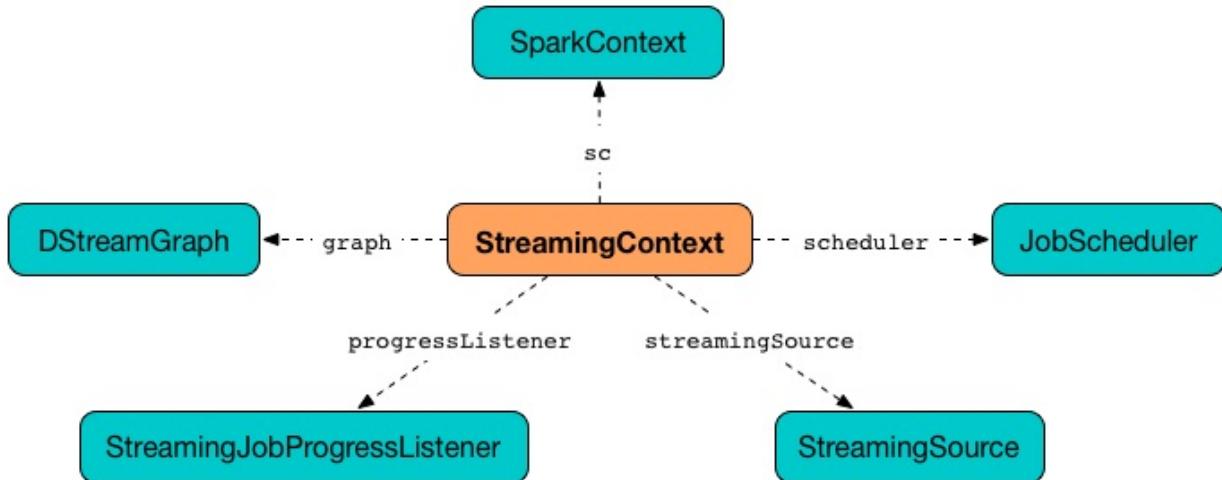


Figure 1. `StreamingContext` and Dependencies

A `DStreamGraph` is created.

A `JobScheduler` is created.

A `StreamingJobProgressListener` is created.

`Streaming tab` in web UI is created (when `spark.ui.enabled` is set).

A `StreamingSource` is instantiated.

At this point, `StreamingContext` enters `INITIALIZED` state.

Creating ReceiverInputDStreams

`StreamingContext` offers the following methods to create `ReceiverInputDStreams` (that you apply transformations to in order to build streaming pipelines):

- `receiverStream[T](receiver: Receiver[T]): ReceiverInputDStream[T]`
- `actorStream[T](props: Props, name: String, storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2, supervisorStrategy: SupervisorStrategy = ActorSupervisorStrategy.defaultStrategy): ReceiverInputDStream[T]`
- `socketTextStream(hostname: String, port: Int, storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2): ReceiverInputDStream[String]`

- `socketStream[T](hostname: String, port: Int, converter: (InputStream) ⇒ Iterator[T], storageLevel: StorageLevel): ReceiverInputDStream[T]`
- `rawSocketStream[T](hostname: String, port: Int, storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2): ReceiverInputDStream[T]`

`StreamingContext` offers the following methods to create [InputDStreams](#):

- `queueStream[T](queue: Queue[RDD[T]], oneAtATime: Boolean = true): InputDStream[T]`
- `queueStream[T](queue: Queue[RDD[T]], oneAtATime: Boolean, defaultRDD: RDD[T]): InputDStream[T]`

You can also use two additional methods in `StreamingContext` to build (or better called *compose*) a custom [DStream](#):

- `union[T](streams: Seq[DStream[T]]): DStream[T]`
- `transform(dstreams, transformFunc): DStream[T]`

transform method

```
transform[T](dstreams: Seq[DStream[_]], transformFunc: (Seq[RDD[_]], Time) => RDD[T]): DS
```

transform Example

```
import org.apache.spark.rdd.RDD
def union(rdds: Seq[RDD[_]], time: Time) = {
  rdds.head.context.union(rdds.map(_.asInstanceOf[RDD[Int]]))
}
ssc.transform(Seq(cis), union)
```

remember method

```
remember(duration: Duration): Unit
```

`remember` method sets the [remember interval](#) (for the graph of output dstreams). It simply calls [DStreamGraph.remember](#) method and exits.

Caution	FIXME figure
---------	------------------------------

Checkpoint Interval

The **checkpoint interval** is an internal property of `StreamingContext` and corresponds to **batch interval** or **checkpoint interval of the checkpoint** (when `checkpoint was present`).

Note	The checkpoint interval property is also called graph checkpointing interval .
------	---

`checkpoint interval` is mandatory when `checkpoint directory` is defined (i.e. not `null`).

Checkpoint Directory

A **checkpoint directory** is a HDFS-compatible directory where `checkpoints` are written to.

Note	<i>"A HDFS-compatible directory"</i> means that it is Hadoop's Path class to handle all file system-related operations.
------	---

Its initial value depends on whether the `StreamingContext` was (re)created from a checkpoint or not, and is the checkpoint directory if so. Otherwise, it is not set (i.e. `null`).

You can set the checkpoint directory when a `StreamingContext` is created or later using `checkpoint` method.

Internally, a checkpoint directory is tracked as `checkpointDir` .

Tip	Refer to Checkpointing for more detailed coverage.
-----	--

Initial Checkpoint

Initial checkpoint is the `checkpoint (file)` this `StreamingContext` has been recreated from.

The initial checkpoint is specified when a `StreamingContext` is created.

```
val ssc = new StreamingContext("_checkpoint")
```

Marking StreamingContext As Recreated from Checkpoint (`isCheckpointPresent` method)

`isCheckpointPresent` internal method behaves like a flag that remembers whether the `StreamingContext` instance was created from a `checkpoint` or not so the other internal parts of a streaming application can make decisions how to initialize themselves (or just be initialized).

`isCheckpointPresent` checks the existence of the `initial checkpoint` that gave birth to the `StreamingContext`.

Setting Checkpoint Directory (checkpoint method)

```
checkpoint(directory: String): Unit
```

You use `checkpoint` method to set `directory` as the current [checkpoint directory](#).

Note	Spark creates the directory unless it exists already.
------	---

`checkpoint` uses [SparkContext.hadoopConfiguration](#) to get the file system and create `directory` on. The full path of the directory is passed on to [SparkContext.setCheckpointDir](#) method.

Note	Calling <code>checkpoint</code> with <code>null</code> as <code>directory</code> clears the checkpoint directory that effectively disables checkpointing.
------	---

Note	When StreamingContext is created and <code>spark.streaming.checkpoint.directory</code> setting is set, the value gets passed on to <code>checkpoint</code> method.
------	--

Starting StreamingContext (using start method)

```
start(): Unit
```

You start stream processing by calling `start()` method. It acts differently per state of [StreamingContext](#) and only [INITIALIZED](#) state makes for a proper startup.

Note	Consult States section in this document to learn about the states of StreamingContext .
------	---

Starting in INITIALIZED state

Right after [StreamingContext](#) has been instantiated, it enters `INITIALIZED` state in which `start` first checks whether another `StreamingContext` instance has already been started in the JVM. It throws `IllegalStateException` exception if it was and exits.

```
java.lang.IllegalStateException: Only one StreamingContext may be s
```

If no other [StreamingContext](#) exists, it performs [setup validation](#) and starts [JobScheduler](#) (in a separate dedicated daemon thread called **streaming-start**).



Figure 2. When started, StreamingContext starts JobScheduler

It enters **ACTIVE** state.

It then register the `shutdown hook stopOnShutdown` and registers streaming metrics source. If web UI is enabled (by `spark.ui.enabled`), it attaches the `Streaming tab`.

Given all the above has have finished properly, it is assumed that the StreamingContext started fine and so you should see the following INFO message in the logs:

```
INFO StreamingContext: StreamingContext started
```

Starting in ACTIVE state

When in **ACTIVE** state, i.e. `after it has been started`, executing `start` merely leads to the following WARN message in the logs:

```
WARN StreamingContext: StreamingContext has already been started
```

Starting in STOPPED state

Attempting to start `StreamingContext` in **STOPPED** state, i.e. `after it has been stopped`, leads to the `IllegalStateException` exception:

```
java.lang.IllegalStateException: StreamingContext has already been stopped
```

Stopping StreamingContext (using stop methods)

You stop `StreamingContext` using one of the three variants of `stop` method:

- `stop(stopSparkContext: Boolean = true)`
- `stop(stopSparkContext: Boolean, stopGracefully: Boolean)`

Note	The first <code>stop</code> method uses <code>spark.streaming.stopSparkContextByDefault</code> configuration setting that controls <code>stopSparkContext</code> input parameter.
------	---

The first `stop` method uses `spark.streaming.stopSparkContextByDefault` configuration setting that controls `stopSparkContext` input parameter.

`stop` methods stop the execution of the streams immediately (`stopGracefully` is `false`) or wait for the processing of all received data to be completed (`stopGracefully` is `true`).

`stop` reacts appropriately per the state of `StreamingContext`, but the end state is always `STOPPED` state with shutdown hook removed.

If a user requested to stop the underlying `SparkContext` (when `stopSparkContext` flag is enabled, i.e. `true`), it is now attempted to be stopped.

Stopping in ACTIVE state

It is only in `ACTIVE` state when `stop` does more than printing out `WARN` messages to the logs.

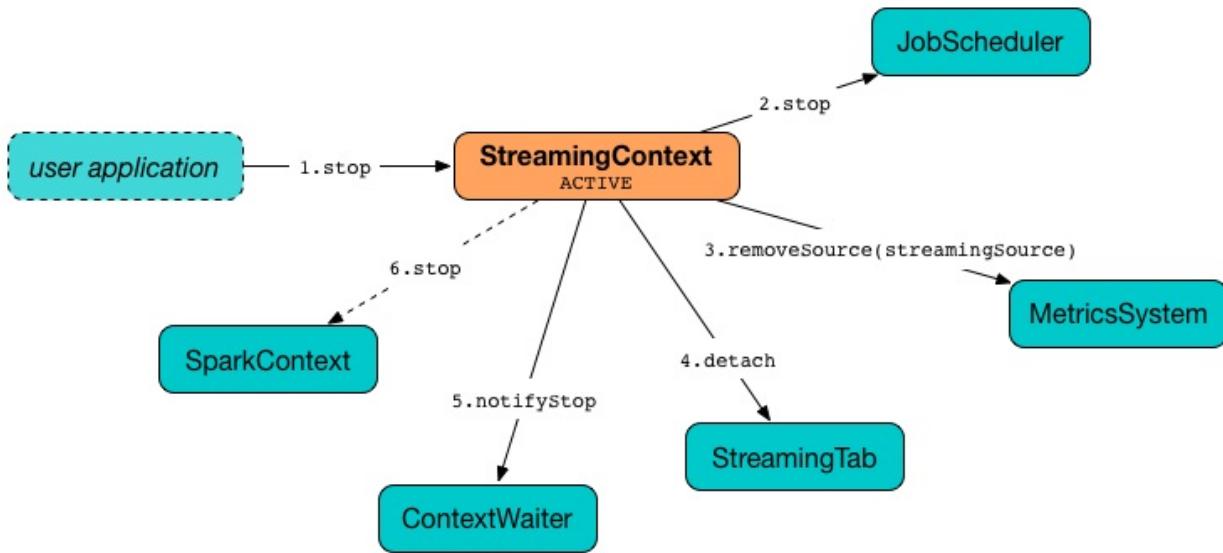


Figure 3. StreamingContext Stop Procedure

It does the following (in order):

1. `JobScheduler` is stopped.
2. `StreamingSource` is removed from `MetricsSystem` (using `MetricsSystem.removeSource`).
3. `Streaming tab` is detached (using `streamingTab.detach`).
4. `ContextWaiter` is `notifyStop()`
5. `shutdownHookRef` is cleared.

At that point, you should see the following `INFO` message in the logs:

```
INFO StreamingContext: StreamingContext stopped successfully
```

`StreamingContext` enters `STOPPED` state.

Stopping in INITIALIZED state

When in `INITIALIZED` state, you should see the following `WARN` message in the logs:

```
WARN StreamingContext: StreamingContext has not been started yet
```

`StreamingContext` enters `STOPPED` state.

Stopping in `STOPPED` state

When in `STOPPED` state, it prints the `WARN` message to the logs:

```
WARN StreamingContext: StreamingContext has already been stopped
```

`StreamingContext` enters `STOPPED` state.

stopOnShutdown Shutdown Hook

`stopOnShutdown` is a [JVM shutdown hook](#) to clean up after `StreamingContext` when the JVM shuts down, e.g. all non-daemon thread exited, `System.exit` was called or `^C` was typed.

Note	It is registered to <code>ShutdownHookManager</code> when StreamingContext starts .
------	---

Note	<code>ShutdownHookManager</code> uses <code>org.apache.hadoop.util.ShutdownHookManager</code> for its work.
------	---

When executed, it first reads `spark.streaming.stopGracefullyOnShutdown` setting that controls [whether to stop `StreamingContext` gracefully or not](#). You should see the following `INFO` message in the logs:

```
INFO Invoking stop(stopGracefully=[stopGracefully]) from shutdown hook
```

With the setting it [stops `StreamingContext`](#) without stopping the accompanying `SparkContext` (i.e. `stopSparkContext` parameter is disabled).

Setup Validation

```
validate(): Unit
```

`validate()` method validates configuration of `StreamingContext`.

Note	The method is executed when <code>StreamingContext</code> is started .
------	--

It first asserts that `dstreamGraph` has been assigned (i.e. `graph` field is not `null`) and triggers [validation of DStreamGraph](#).

Caution

It appears that `graph` could never be `null`, though.

If [checkpointing is enabled](#), it ensures that [checkpoint interval](#) is set and checks whether the current streaming runtime environment can be safely serialized by [serializing a checkpoint for fictitious batch time 0 \(not zero time\)](#).

If [dynamic allocation](#) is enabled, it prints the following WARN message to the logs:

```
WARN StreamingContext: Dynamic Allocation is enabled for this appli
```

Registering Streaming Listeners

Caution**FIXME**

Streaming Metrics Source

Caution**FIXME**

States

`StreamingContext` can be in three states:

- `INITIALIZED`, i.e. after [it was instantiated](#).
- `ACTIVE`, i.e. after [it was started](#).
- `STOPPED`, i.e. after [it has been stopped](#)

Stream Operators

You use **stream operators** to apply **transformations** to the elements received (often called **records**) from input streams and ultimately trigger computations using **output operators**.

Transformations are **stateless**, but Spark Streaming comes with an *experimental* support for **stateful operators** (e.g. `mapWithState` or `updateStateByKey`). It also offers **windowed operators** that can work across batches.

Note

You may use RDDs from other (non-streaming) data sources to build more advanced pipelines.

There are two main types of operators:

- **transformations** that transform elements in input data RDDs
- **output operators** that register input streams as output streams so the execution can start.

Every [Discretized Stream \(DStream\)](#) offers the following operators:

- (output operator) `print` to print 10 elements only or the more general version `print(num: Int)` to print up to `num` elements. See [print operation](#) in this document.
- `slice`
- `window`
- `reduceByWindow`
- `reduce`
- `map`
- (output operator) `foreachRDD`
- `glom`
- (output operator) `saveAsObjectFiles`
- (output operator) `saveAsTextFiles`
- `transform`
- `transformWith`
- `flatMap`

- `filter`
- `repartition`
- `mapPartitions`
- `count`
- `countByValue`
- `countByWindow`
- `countByValueAndWindow`
- `union`

Note `DStream` companion object offers a Scala implicit to convert `DStream[(K, V)]` to `PairDStreamFunctions` with methods on DStreams of key-value pairs, e.g. [mapWithState](#) or [updateStateByKey](#).

Most streaming operators come with their own custom `DStream` to offer the service. It however very often boils down to overriding the `compute` method and applying corresponding [RDD operator](#) on a generated RDD.

print Operator

`print(num: Int)` operator prints `num` first elements of each RDD in the input stream.

`print` uses `print(num: Int)` with `num` being `10`.

It is a **output operator** (that returns `Unit`).

For each batch, `print` operator prints the following header to the standard output (regardless of the number of elements to be printed out):

```
-----  
Time: [time] ms  
-----
```

Internally, it calls `RDD.take(num + 1)` (see [take action](#)) on each RDD in the stream to print `num` elements. It then prints `...` if there are more elements in the RDD (that would otherwise exceed `num` elements being requested to print).

It creates a [ForEachDStream](#) stream and [registers it as an output stream](#).

foreachRDD Operators

```
foreachRDD(foreachFunc: RDD[T] => Unit): Unit
foreachRDD(foreachFunc: (RDD[T], Time) => Unit): Unit
```

`foreachRDD` operator applies `foreachFunc` function to every RDD in the stream.

It creates a [ForEachDStream](#) stream and [registers it as an output stream](#).

foreachRDD Example

```
val clicks: InputDStream[(String, String)] = messages
// println every single data received in clicks input stream
clicks.foreachRDD(rdd => rdd.foreach(println))
```

glom Operator

```
glom(): DStream[Array[T]]
```

`glom` operator creates a new stream in which RDDs in the source stream are [RDD.glom](#) over, i.e. it [coalesces](#) all elements in RDDs within each partition into an array.

reduce Operator

```
reduce(reduceFunc: (T, T) => T): DStream[T]
```

`reduce` operator creates a new stream of RDDs of a single element that is a result of applying `reduceFunc` to the data received.

Internally, it uses [map](#) and [reduceByKey](#) operators.

reduce Example

```
val clicks: InputDStream[(String, String)] = messages
type T = (String, String)
val reduceFunc: (T, T) => T = {
  case in @ ((k1, v1), (k2, v2)) =>
    println(s">>> input: $in")
    (k2, s"$v1 + $v2")
}
val reduceClicks: DStream[(String, String)] = clicks.reduce(reduceFunc)
reduceClicks.print
```

map Operator

```
map[U](mapFunc: T => U): DStream[U]
```

`map` operator creates a new stream with the source elements being mapped over using `mapFunc` function.

It creates `MappedDStream` stream that, when requested to compute a RDD, uses [RDD.map](#) operator.

map Example

```
val clicks: DStream[...] = ...
val mappedClicks: ... = clicks.map(...)
```

reduceByKey Operator

```
reduceByKey(reduceFunc: (V, V) => V): DStream[(K, V)]
reduceByKey(reduceFunc: (V, V) => V, numPartitions: Int): DStream[(K, V)]
reduceByKey(reduceFunc: (V, V) => V, partitioner: Partitioner): DStream[(K, V)]
```

transform Operators

```
transform(transformFunc: RDD[T] => RDD[U]): DStream[U]
transform(transformFunc: (RDD[T], Time) => RDD[U]): DStream[U]
```

`transform` operator applies `transformFunc` function to the generated RDD for a batch.

It creates a [TransformedDStream](#) stream.

Note	It asserts that one and exactly one RDD has been generated for a batch before calling the <code>transformFunc</code> .
Note	It is not allowed to return <code>null</code> from <code>transformFunc</code> or a <code>sparkException</code> is reported. See TransformedDStream .

transform Example

```

import org.apache.spark.streaming.{ StreamingContext, Seconds }
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

val rdd = sc.parallelize(0 to 9)
import org.apache.spark.streaming.dstream.ConstantInputDStream
val clicks = new ConstantInputDStream(ssc, rdd)

import org.apache.spark.rdd.RDD
val transformFunc: RDD[Int] => RDD[Int] = { inputRDD =>
  println(s">>> inputRDD: $inputRDD")

  // Use SparkSQL's DataFrame to manipulate the input records
  import sqlContext.implicits._
  inputRDD.toDF("num").show

  inputRDD
}

clicks.transform(transformFunc).print

```

transformWith Operators

```

transformWith(other: DStream[U], transformFunc: (RDD[T], RDD[U]) => RDD[V]): DStream[V]
transformWith(other: DStream[U], transformFunc: (RDD[T], RDD[U], Time) => RDD[V]): DStream[V]

```

`transformWith` operators apply the `transformFunc` function to two generated RDD for a batch.

It creates a [TransformedDStream](#) stream.

Note	It asserts that two and exactly two RDDs have been generated for a batch before calling the <code>transformFunc</code> .
------	--

Note	It is not allowed to return <code>null</code> from <code>transformFunc</code> or a <code>SparkException</code> is reported. See TransformedDStream .
------	--

transformWith Example

```
import org.apache.spark.streaming.{ StreamingContext, Seconds }
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

val ns = sc.parallelize(0 to 2)
import org.apache.spark.streaming.ConstantInputDStream
val nums = new ConstantInputDStream(ssc, ns)

val ws = sc.parallelize(Seq("zero", "one", "two"))
import org.apache.spark.streaming.ConstantInputDStream
val words = new ConstantInputDStream(ssc, ws)

import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.Time
val transformFunc: (RDD[Int], RDD[String], Time) => RDD[(Int, String)] = { case (ns, ws,
  println(s">>> ns: $ns")
  println(s">>> ws: $ws")
  println(s">>> batch: $time")

  ns.zip(ws)
}
nums.transformWith(words, transformFunc).print
```



Windowed Operators

Note	Go to Window Operations to read the official documentation. This document aims at presenting the <i>internals</i> of window operators with examples.
------	---

In short, **windowed operators** allow you to apply transformations over a **sliding window** of data, i.e. build a *stateful computation* across multiple batches.

Note	Windowed operators, windowed operations, and window-based operations are all the same concept.
------	--

By default, you apply transformations using different [stream operators](#) to a single RDD that represents a dataset that has been built out of data received from one or many [input streams](#). The transformations know nothing about the past (datasets received and already processed). The computations are hence *stateless*.

You can however build datasets based upon the past ones, and that is when windowed operators enter the stage. Using them allows you to cross the boundary of a single dataset (per batch) and have a series of datasets in your hands (as if the data they hold arrived in a single batch interval).

slice Operators

```
slice(interval: Interval): Seq[RDD[T]]
slice(fromTime: Time, toTime: Time): Seq[RDD[T]]
```

`slice` operators return a collection of RDDs that were generated during time interval inclusive, given as `Interval` or a pair of `Time` ends.

Both `Time` ends have to be a multiple of this stream's slide duration. Otherwise, they are aligned using `Time.floor` method.

When used, you should see the following INFO message in the logs:

```
INFO Slicing from [fromTime] to [toTime] (aligned to [alignedFromTime] and [alignedToTime])
```

For every batch in the slicing interval, a [RDD is computed](#).

window Operators

```
window(windowDuration: Duration): DStream[T]
window(windowDuration: Duration, slideDuration: Duration): DStream[T]
```

`window` operator creates a new stream that generates RDDs containing all the elements received during `windowDuration` with `slideDuration` [slide duration](#).

Note	<code>windowDuration</code> must be a multiple of the slide duration of the source stream.
------	--

`window(windowDuration: Duration): DStream[T]` operator uses `window(windowDuration: Duration, slideDuration: Duration)` with the source stream's [slide duration](#).

```
messages.window(Seconds(10))
```

It creates [WindowedDStream](#) stream and register it as an output stream.

Note	<code>window</code> Operator is used by <code>reduceByWindow</code> , <code>reduceByKeyAndWindow</code> and <code>groupByKeyAndWindow</code> operators.
------	---

reduceByWindow Operator

```
reduceByWindow(reduceFunc: (T, T) => T, windowDuration: Duration, slideDuration: Duration)
reduceByWindow(reduceFunc: (T, T) => T, invReduceFunc: (T, T) => T, windowDuration: Durat
```

`reduceByWindow` operator create a new stream of RDDs of one element only that was computed using `reduceFunc` function over the data received during batch duration that later was *again* applied to a collection of the reduced elements from the past being window duration `windowDuration` sliding `slideDuration` forward.

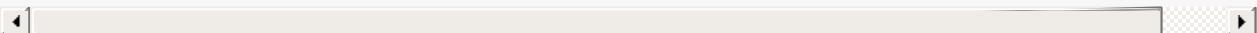
Note	<code>reduceByWindow</code> is window and reduce operators applied to the collection of RDDs collected over window duration.
------	--

reduceByWindow Example

```
// batchDuration = Seconds(5)

val clicks: InputDStream[(String, String)] = messages
type T = (String, String)
val reduceFn: (T, T) => T = {
  case in @ ((k1, v1), (k2, v2)) =>
    println(s">>> input: $in")
    (k2, s"$v1 + $v2")
}
val windowedClicks: DStream[(String, String)] =
  clicks.reduceByWindow(reduceFn, windowDuration = Seconds(10), slideDuration = Seconds(5))

windowedClicks.print
```



SaveAs Operators

There are two **saveAs operators** in **DStream**:

- `saveAsObjectFiles`
- `saveAsTextFiles`

They are **output operators** that return nothing as they save each RDD in a batch to a storage.

Their full signature is as follows:

```
saveAsObjectFiles(prefix: String, suffix: String = ""): Unit  
saveAsTextFiles(prefix: String, suffix: String = ""): Unit
```

Note

SaveAs operators use `foreachRDD` output operator.

`saveAsObjectFiles` uses [RDD.saveAsObjectFile](#) while `saveAsTextFiles` uses [RDD.saveAsTextFile](#).

The file name is based on mandatory `prefix` and batch `time` with optional `suffix`. It is in the format of `[prefix]-[time in milliseconds].[suffix]`.

Example

```
val clicks: InputDStream[(String, String)] = messages  
clicks.saveAsTextFiles("clicks", "txt")
```

Working with State using Stateful Operators

Building Stateful Stream Processing Pipelines using Spark (Streaming)

Stateful operators (like `mapWithState` or `updateStateByKey`) are part of the set of additional operators available on `DStreams` of key-value pairs, i.e. instances of `DStream[(K, V)]`. They allow you to build **stateful stream processing pipelines** and are also called **cumulative calculations**.

The motivation for the stateful operators is that by design streaming operators are stateless and know nothing about the previous records and hence a state. If you'd like to react to new records appropriately given the previous records you would have to resort to using persistent storages outside Spark Streaming.

Note

These additional operators are available automatically on pair DStreams through the Scala implicit conversion `DStream.toPairDStreamFunctions`.

mapWithState Operator

```
mapWithState(spec: StateSpec[K, V, ST, MT]): MapWithStateDStream[K, V, ST, MT]
```

You create `StateSpec` instances for `mapWithState` operator using the factory methods `StateSpec.function`.

`mapWithState` creates a `MapWithStateDStream` dstream.

mapWithState Example

```

import org.apache.spark.streaming.{ StreamingContext, Seconds }
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

// checkpointing is mandatory
ssc.checkpoint("_checkpoints")

val rdd = sc.parallelize(0 to 9).map(n => (n, n % 2 toString))
import org.apache.spark.streaming.dstream.ConstantInputDStream
val sessions = new ConstantInputDStream(ssc, rdd)

import org.apache.spark.streaming.{State, StateSpec, Time}
val updateState = (batchTime: Time, key: Int, value: Option[String], state: State[Int]) =
  println(s">>> batchTime = $batchTime")
  println(s">>> key      = $key")
  println(s">>> value     = $value")
  println(s">>> state     = $state")
  val sum = value.getOrElse("").size + state.getOption.getOrElse(0)
  state.update(sum)
  Some((key, value, sum)) // mapped value
}
val spec = StateSpec.function(updateState)
val mappedStatefulStream = sessions.mapWithState(spec)

mappedStatefulStream.print()

```

StateSpec - Specification of mapWithState

`StateSpec` is a state specification of `mapWithState` and describes how the corresponding state RDD should work (RDD-wise) and maintain a state (streaming-wise).

Note

`StateSpec` is a Scala `sealed abstract class` and hence all the implementations are in the same compilation unit, i.e. source file.

It requires the following:

- `initialState` which is the initial state of the transformation, i.e. paired `RDD[(KeyType, StateType)]`.
- `numPartitions` which is the number of partitions of the state RDD. It uses `HashPartitioner` with the given number of partitions.
- `partitioner` which is the partitioner of the state RDD.
- `timeout` that sets the idle duration after which the state of an *idle* key will be removed. A key and its state is considered *idle* if it has not received any data for at least the given idle duration.

StateSpec.function Factory Methods

You create `StateSpec` instances using the factory methods `stateSpec.function` (that differ in whether or not you want to access a batch time and return an optional mapped value):

```
// batch time and optional mapped return value
StateSpec.function(f: (Time, K, Option[V], State[S]) => Option[M]): StateSpec[K, V, S, M]

// no batch time and mandatory mapped value
StateSpec.function(f: (K, Option[V], State[S]) => M): StateSpec[K, V, S, M]
```

Internally, the `StateSpec.function` executes `ClosureCleaner.clean` to clean up the input function `f` and makes sure that `f` can be serialized and sent over the wire (cf. [Closure Cleaning \(clean method\)](#)). It will throw an exception when the input function cannot be serialized.

updateStateByKey Operator

```
updateStateByKey(updateFn: (Seq[V], Option[S]) => Option[S]): DStream[(K, S)] (1)
updateStateByKey(updateFn: (Seq[V], Option[S]) => Option[S],
    numPartitions: Int): DStream[(K, S)] (2)
updateStateByKey(updateFn: (Seq[V], Option[S]) => Option[S],
    partitioner: Partitioner): DStream[(K, S)] (3)
updateStateByKey(updateFn: (Iterator[(K, Seq[V], Option[S])]) => Iterator[(K, S)],
    partitioner: Partitioner,
    rememberPartitioner: Boolean): DStream[(K, S)] (4)
updateStateByKey(updateFn: (Seq[V], Option[S]) => Option[S],
    partitioner: Partitioner,
    initialRDD: RDD[(K, S)]): DStream[(K, S)]
updateStateByKey(updateFn: (Iterator[(K, Seq[V], Option[S])]) => Iterator[(K, S)],
    partitioner: Partitioner,
    rememberPartitioner: Boolean,
    initialRDD: RDD[(K, S)]): DStream[(K, S)]
```

1. When not specified explicitly, the partitioner used is [HashPartitioner](#) with the number of partitions being the default level of parallelism of a [Task Scheduler](#).
2. You may however specify the number of partitions explicitly for [HashPartitioner](#) to use.
3. This is the "canonical" `updateStateByKey` the other two variants (without a partitioner or the number of partitions) use that allows specifying a partitioner explicitly. It then executes the "last" `updateStateByKey` with `rememberPartitioner` enabled.
4. The "last" `updateStateByKey`

`updateStateByKey` stateful operator allows for maintaining per-key state and updating it using `updateFn`. The `updateFn` is called for each key, and uses new data and existing state of the key, to generate an updated state.

Tip

You should use [mapWithState operator](#) instead as a much performance effective alternative.

Note

Please consult [SPARK-2629 Improved state management for Spark Streaming](#) for performance-related changes to the operator.

The state update function `updateFn` scans every key and generates a new state for every key given a collection of values per key in a batch and the current state for the key (if exists).

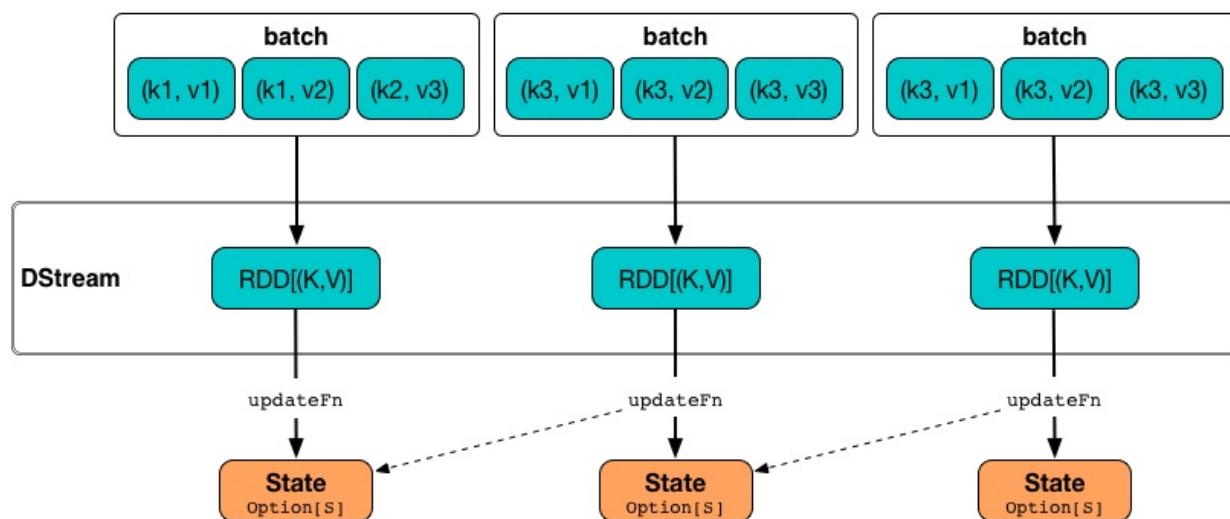


Figure 1. `updateStateByKey` in motion

Internally, `updateStateByKey` executes [SparkContext.clean](#) on the input function `updateFn`.

Note

The operator does not offer any timeout of idle data.

`updateStateByKey` creates a [StateDStream](#) stream.

updateStateByKey Example

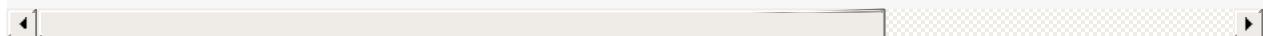
```
import org.apache.spark.streaming.{ StreamingContext, Seconds }
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

// checkpointing is mandatory
ssc.checkpoint("_checkpoints")

val rdd = sc.parallelize(0 to 9).map(n => (n, n % 2 toString))
import org.apache.spark.streaming.dstream.ConstantInputDStream
val clicks = new ConstantInputDStream(ssc, rdd)

// helper functions
val inc = (n: Int) => n + 1
def buildState: Option[Int] = {
    println(s">>> >>> Initial execution to build state or state is deliberately uninitialized")
    println(s">>> >>> Building the state being the number of calls to update state function")
    Some(1)
}

// the state update function
val updateFn: (Seq[String], Option[Int]) => Option[Int] = { case (vs, state) =>
    println(s">>> update state function with values only, i.e. no keys")
    println(s">>> vs      = $vs")
    println(s">>> state   = $state")
    state.map(inc).orElse(buildState)
}
val statefulStream = clicks.updateStateByKey(updateFn)
statefulStream.print()
```



web UI and Streaming Statistics Page

When you [start a Spark Streaming application](#), you can use [web UI](#) to monitor streaming statistics in **Streaming tab** (aka *page*).

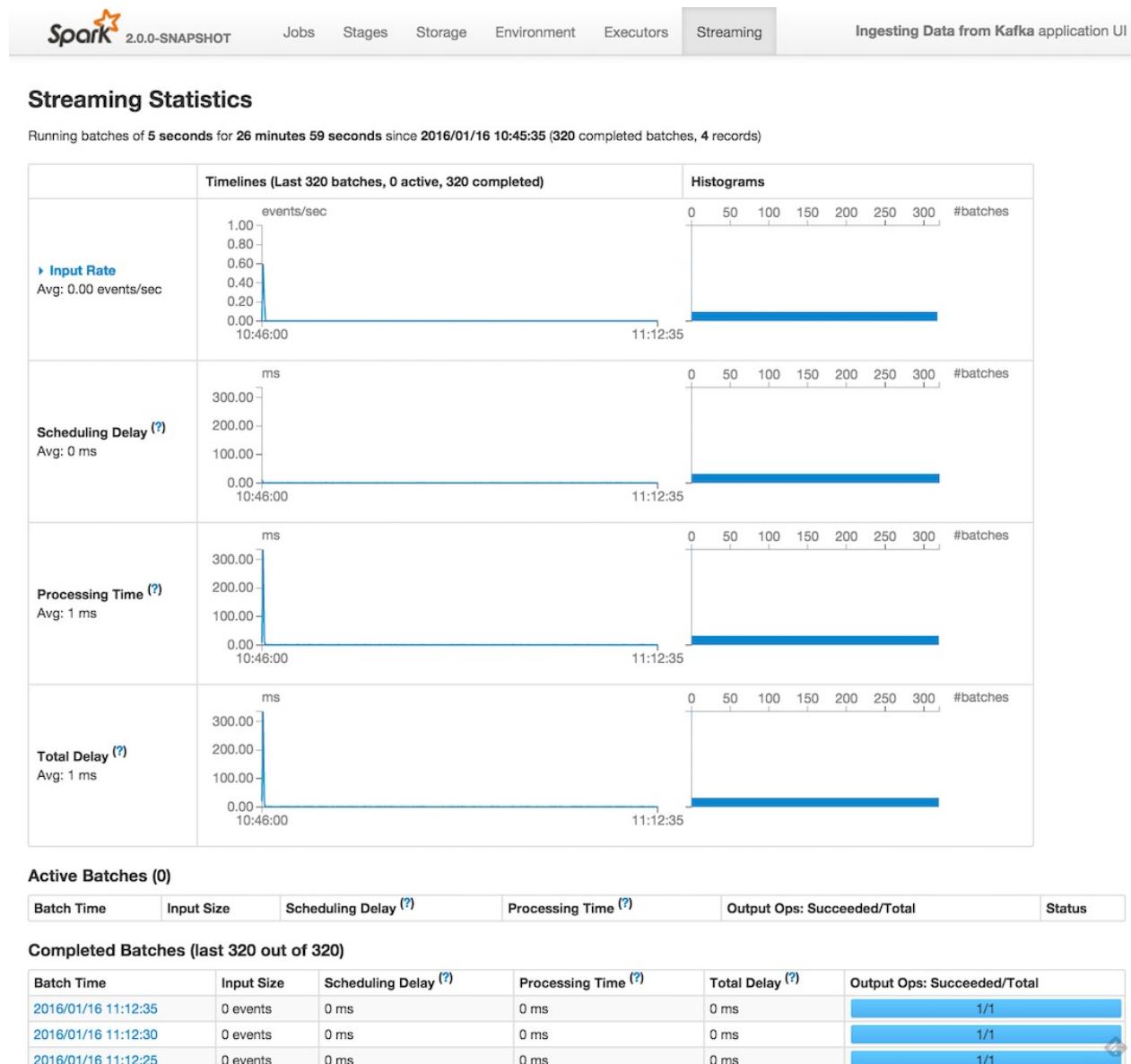


Figure 1. Streaming Tab in web UI

Note	The number of completed batches to retain to compute statistics upon is controlled by <code>spark.streaming.ui.retainedBatches</code> (and defaults to 1000).
------	--

The page is made up of three sections (aka *tables*) - the unnamed, top-level one with [basic information](#) about the streaming application (right below the title **Streaming Statistics**), [Active Batches](#) and [Completed Batches](#).

Note	The Streaming page uses <code>StreamingJobProgressListener</code> for most of the information displayed.
------	--

Basic Information

Basic Information section is the top-level section in the Streaming page that offers basic information about the streaming application.

Streaming Statistics

Running batches of 10 seconds for 7 seconds 231 ms since 2016/01/16 19:11:52 (1 completed batches, 0 records)

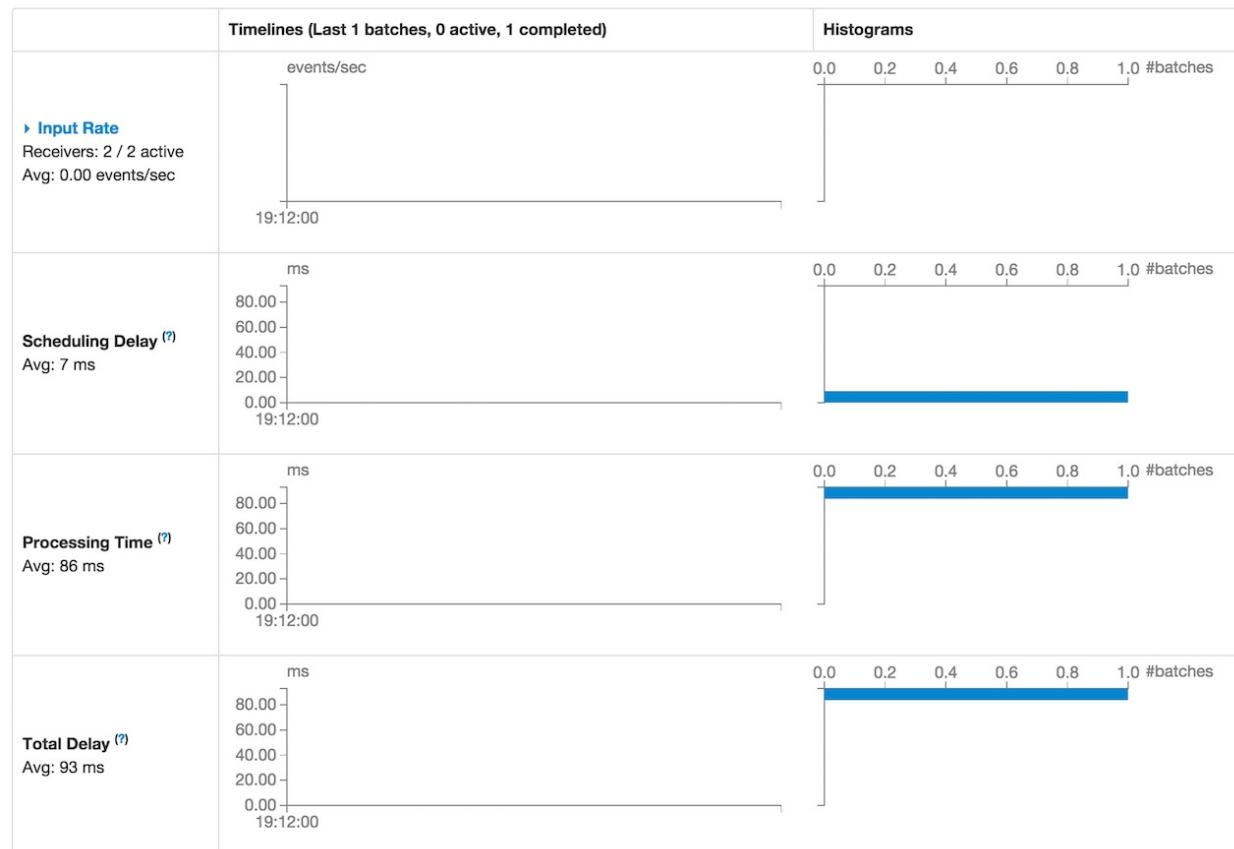


Figure 2. Basic Information section in Streaming Page (with Receivers)

The section shows the [batch duration](#) (in *Running batches of [batch duration]*), and the time it runs for and since [StreamingContext was created](#) (*not when this streaming application has been started!*).

It shows the number of all **completed batches** (for the entire period since the StreamingContext was started) and **received records** (in parenthesis). These information are later displayed in detail in [Active Batches](#) and [Completed Batches](#) sections.

Below is the table for [retained batches](#) (i.e. waiting, running, and completed batches).

In **Input Rate** row, you can show and hide details of each input stream.

If there are [input streams with receivers](#), the numbers of all the receivers and active ones are displayed (as depicted in the Figure 2 above).

The average event rate for all registered streams is displayed (as *Avg: [avg] events/sec*).

Scheduling Delay

Scheduling Delay is the time spent from [when the collection of streaming jobs for a batch was submitted](#) to [when the first streaming job \(out of possibly many streaming jobs in the collection\) was started](#).

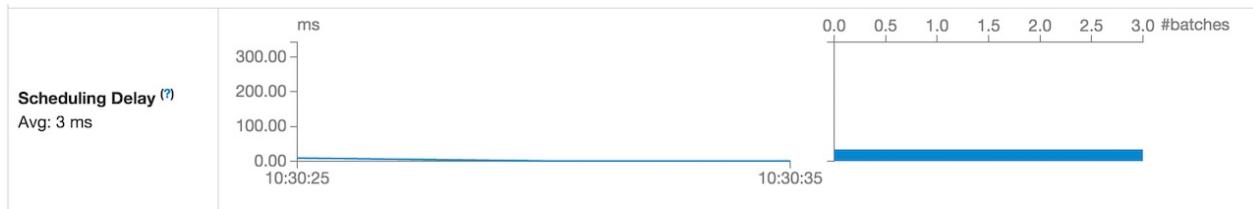


Figure 3. Scheduling Delay in Streaming Page

It should be as low as possible meaning that the streaming jobs in batches are scheduled almost instantly.

Note

The values in the timeline (the first column) depict the time between the events [StreamingListenerBatchSubmitted](#) and [StreamingListenerBatchStarted](#) (with minor yet additional delays to deliver the events).

You may see increase in scheduling delay in the timeline when streaming jobs are queued up as in the following example:

```
// batch duration = 5 seconds
val messages: InputDStream[(String, String)] = ...
messages.foreachRDD { rdd =>
    println(">>> Taking a 15-second sleep")
    rdd.foreach(println)
    java.util.concurrent.TimeUnit.SECONDS.sleep(15)
}
```

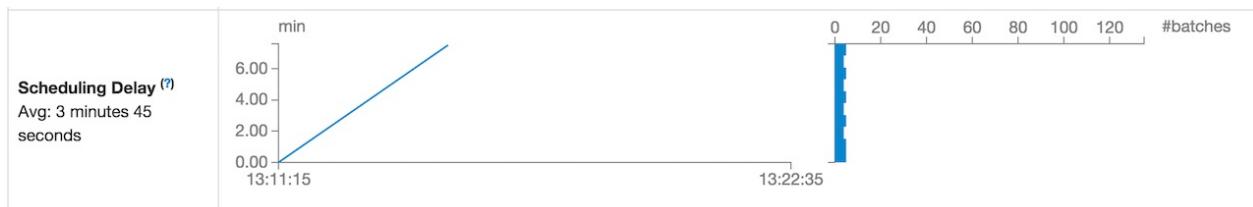


Figure 4. Scheduling Delay Increased in Streaming Page

Processing Time

Processing Time is the time spent to complete all the streaming jobs of a batch.

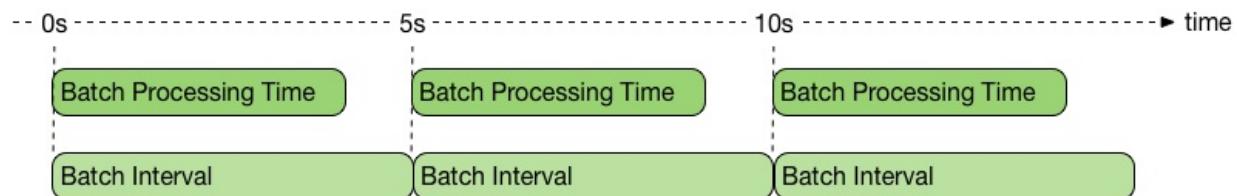


Figure 5. Batch Processing Time and Batch Intervals

Total Delay

Total Delay is the time spent from submitting to complete all jobs of a batch.

Active Batches

Active Batches section presents `waitingBatches` and `runningBatches` together.

Completed Batches

Completed Batches section presents retained completed batches (using `completedBatchUIData`).

Note	The number of retained batches is controlled by <code>spark.streaming.ui.retainedBatches</code> .
------	---

Completed Batches (last 5 out of 42)

Batch Time	Input Size	Scheduling Delay <small>(?)</small>	Processing Time <small>(?)</small>	Total Delay <small>(?)</small>	Output Ops: Succeeded/Total
2016/01/19 21:34:00	0 events	1 ms	0 ms	1 ms	1/1
2016/01/19 21:33:55	0 events	0 ms	1 ms	1 ms	1/1
2016/01/19 21:33:50	0 events	0 ms	0 ms	0 ms	1/1
2016/01/19 21:33:45	0 events	1 ms	0 ms	1 ms	1/1
2016/01/19 21:33:40	0 events	1 ms	0 ms	1 ms	1/1

Figure 6. Completed Batches (limited to 5 elements only)

Example - Kafka Direct Stream in web UI

2016/01/16 10:46:10	1 events	0 ms	13 ms	13 ms	1/1
2016/01/16 10:46:05	3 events	0 ms	0.3 s	0.3 s	1/1
2016/01/16 10:46:00	0 events	12 ms	7 ms	19 ms	1/1

Figure 7. Two Batches with Incoming Data inside for Kafka Direct Stream in web UI
(Streaming tab)

Spark 2.0.0-SNAPSHOT Jobs Stages Storage Environment Executors Streaming Ingesting Data from Kafka application UI

Spark Jobs (?) Total Uptime: 5.2 min Scheduling Mode: FIFO Completed Jobs: 2 ▶ Event Timeline

Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	Streaming job from [output operation 0, batch time 10:46:10] print at <console>:35	2016/01/16 10:46:10	10 ms	1/1	1/1
0	Streaming job from [output operation 0, batch time 10:46:05] print at <console>:35	2016/01/16 10:46:05	0.3 s	1/1	1/1

Figure 8. Two Jobs for Kafka Direct Stream in web UI (Jobs tab)

Streaming Listeners

Streaming listeners are listeners interested in streaming events like batch submitted, started or completed.

Streaming listeners implement [org.apache.spark.streaming.scheduler.StreamingListener](#) listener interface and process [StreamingListenerEvent](#) events.

The following streaming listeners are available in Spark Streaming:

- [StreamingJobProgressListener](#)
- [RateController](#)

StreamingListenerEvent Events

- `StreamingListenerBatchSubmitted` is posted when [streaming jobs](#) are submitted for [execution](#) and triggers `StreamingListener.onBatchSubmitted` (see [StreamingJobProgressListener.onBatchSubmitted](#)).
- `StreamingListenerBatchStarted` triggers `StreamingListener.onBatchStarted`
- `StreamingListenerBatchCompleted` is posted to inform that a [collection of streaming jobs has completed](#), i.e. all the streaming jobs in [JobSet](#) have stopped their execution.

StreamingJobProgressListener

`StreamingJobProgressListener` is a streaming listener that collects information for [StreamingSource](#) and [Streaming](#) page in [web UI](#).

Note

It is created while [StreamingContext is created](#) and later registered as a `StreamingListener` and `sparkListener` when [Streaming tab](#) is created.

onBatchSubmitted

For `StreamingListenerBatchSubmitted(batchInfo: BatchInfo)` events, it stores `batchInfo` batch information in the internal `waitingBatchUIData` registry per batch time.

The number of entries in `waitingBatchUIData` registry contributes to `numUnprocessedBatches` (together with `runningBatchUIData`), `waitingBatches`, and `retainedBatches`. It is also used to look up the batch data for a batch time (in `getBatchUIData`).

`numUnprocessedBatches`, `waitingBatches` are used in [StreamingSource](#).

Note

`waitingBatches` and `runningBatches` are displayed together in [Active Batches in Streaming tab in web UI](#).

onBatchStarted

Caution

[FIXME](#)

onBatchCompleted

Caution

[FIXME](#)

Retained Batches

`retainedBatches` are waiting, running, and completed batches that [web UI uses to display streaming statistics](#).

The number of retained batches is controlled by [spark.streaming.ui.retainedBatches](#).

Checkpointing

Checkpointing is a process of [writing received records](#) (by means of [input dstreams](#)) at [checkpoint intervals](#) to a [highly-available HDFS-compatible storage](#). It allows creating **fault-tolerant stream processing pipelines** so when a failure occurs input dstreams can restore the before-failure streaming state and continue stream processing (as if nothing had happened).

DStreams can checkpoint [input data](#) at specified [time intervals](#).

Marking StreamingContext as Checkpointed

You use [StreamingContext.checkpoint](#) method to set up a HDFS-compatible **checkpoint directory** where [checkpoint data](#) will be persisted, as follows:

```
ssc.checkpoint("_checkpoint")
```

Checkpoint Interval and Checkpointing DStreams

You can set up periodic checkpointing of a dstream every **checkpoint interval** using [DStream.checkpoint](#) method.

```
val ssc: StreamingContext = ...
// set the checkpoint directory
ssc.checkpoint("_checkpoint")
val ds: DStream[Int] = ...
val cds: DStream[Int] = ds.checkpoint(Seconds(5))
// do something with the input dstream
cds.print
```

Recreating StreamingContext from Checkpoint

You can create a StreamingContext from a [checkpoint directory](#), i.e. recreate a fully-working StreamingContext as recorded in the [last valid checkpoint file that was written to the checkpoint directory](#).

Note

You can also [create a brand new StreamingContext](#) (and putting checkpoints aside).

Warning

You must not create input dstreams using a StreamingContext that has been recreated from checkpoint. Otherwise, you will not start the StreamingContext at all.

When you use `StreamingContext(path: String)` constructor (or [the variants thereof](#)), it uses [Hadoop configuration](#) to access `path` directory on a Hadoop-supported file system.

Effectively, the two variants use `StreamingContext(path: String, hadoopConf: Configuration)` constructor that [reads the latest valid checkpoint file](#) (and hence enables)

Note

`SparkContext` and batch interval are set to their corresponding values using the checkpoint file.

Example: Recreating StreamingContext from Checkpoint

The following Scala code demonstrates how to use the checkpoint directory `_checkpoint` to (re)create the StreamingContext or create one from scratch.

```
val appName = "Recreating StreamingContext from Checkpoint"
val sc = new SparkContext("local[*]", appName, new SparkConf())

val checkpointDir = "_checkpoint"

def createsc(): StreamingContext = {
    val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

    // NOTE: You have to create dstreams inside the method
    // See http://stackoverflow.com/q/35090180/1305344

    // Create constant input dstream with the RDD
    val rdd = sc.parallelize(0 to 9)
    import org.apache.spark.streaming.dstream.ConstantInputDStream
    val cis = new ConstantInputDStream(ssc, rdd)

    // Sample stream computation
    cis.print

    ssc.checkpoint(checkpointDir)
    ssc
}

val ssc = StreamingContext.getOrCreate(checkpointDir, createsc)

// Start streaming processing
ssc.start
```

DStreamCheckpointData

`DStreamCheckpointData` works with a single dstream. An instance of `DStreamCheckpointData` is created when a dstream is.

It tracks checkpoint data in the internal `data` registry that records batch time and the checkpoint data at that time. The internal checkpoint data can be anything that a dstream wants to checkpoint. `DStreamCheckpointData` returns the registry when `currentCheckpointFiles` method is called.

Note	By default, <code>DStreamCheckpointData</code> records the checkpoint files to which the generated RDDs of the DStream has been saved.
------	--

Tip	Enable <code>DEBUG</code> logging level for <code>org.apache.spark.streaming.dstream.DStreamCheckpointData</code> logger to see what happens inside.
-----	---

Tip	Add the following line to <code>conf/log4j.properties</code> :
-----	--

```
log4j.logger.org.apache.spark.streaming.dstream.DStreamCheckpointData=DEBUG
```

Tip	Refer to Logging .
-----	------------------------------------

Updating Collection of Batches and Checkpoint Directories (update method)

```
update(time: Time): Unit
```

`update` collects batches and the directory names where the corresponding RDDs were checkpointed (filtering [the dstream's internal generatedRDDs mapping](#)).

You should see the following DEBUG message in the logs:

```
DEBUG Current checkpoint files:  
[checkpointFile per line]
```

The collection of the batches and their checkpointed RDDs is recorded in an internal field for serialization (i.e. it becomes the current value of the internal field `currentCheckpointFiles` that is serialized when requested).

The collection is also added to an internal *transient* (non-serializable) mapping `timeToCheckpointFile` and the oldest checkpoint (given batch times) is recorded in an internal *transient* mapping for the current `time`.

Note	It is called by DStream.updateCheckpointData(currentTime: Time) .
------	---

Deleting Old Checkpoint Files (cleanup method)

```
cleanup(time: Time): Unit
```

`cleanup` deletes checkpoint files older than the oldest batch for the input `time`.

It first gets the oldest batch time for the input `time` (see [Updating Collection of Batches and Checkpoint Directories \(update method\)](#)).

If the (batch) time has been found, all the checkpoint files older are deleted (as tracked in the internal `timeToCheckpointFile` mapping).

You should see the following DEBUG message in the logs:

```
DEBUG Files to delete:  
[comma-separated files to delete]
```

For each checkpoint file successfully deleted, you should see the following INFO message in the logs:

```
INFO Deleted checkpoint file '[file]' for time [time]
```

Errors in checkpoint deletion are reported as WARN messages in the logs:

```
WARN Error deleting old checkpoint file '[file]' for time [time]
```

Otherwise, when no (batch) time has been found for the given input `time`, you should see the following DEBUG message in the logs:

```
DEBUG Nothing to delete
```

Note

It is called by [DStream.clearCheckpointData\(time: Time\)](#).

Restoring Generated RDDs from Checkpoint Files (restore method)

```
restore(): Unit
```

`restore` restores the dstream's [generatedRDDs](#) given persistent internal `data` mapping with batch times and corresponding checkpoint files.

`restore` takes the current checkpoint files and restores checkpointed RDDs from each checkpoint file (using `SparkContext.checkpointFile`).

You should see the following INFO message in the logs per checkpoint file:

```
INFO Restoring checkpointed RDD for time [time] from file '[file]'
```

Note	It is called by <code>DStream.restoreCheckpointData()</code> .
------	--

Checkpoint

`Checkpoint` class requires a `StreamingContext` and `checkpointTime` time to be instantiated. The internal property `checkpointTime` corresponds to the batch time it represents.

Note	<code>Checkpoint</code> class is written to a persistent storage (aka <i>serialized</i>) using <code>CheckpointWriter.write</code> method and read back (aka <i>deserialize</i>) using <code>Checkpoint.deserialize</code> .
------	--

Note	<code>Initial checkpoint</code> is the checkpoint a <code>StreamingContext</code> was started with.
------	---

It is merely a collection of the settings of the current streaming runtime environment that is supposed to recreate the environment after it goes down due to a failure or when the [streaming context is stopped immediately](#).

It collects the settings from the input `StreamingContext` (and indirectly from the corresponding `JobScheduler` and `SparkContext`):

- The [master URL from `SparkContext`](#) as `master` .
- The [mandatory application name from `SparkContext`](#) as `framework` .
- The [jars to distribute to workers from `SparkContext`](#) as `jars` .
- The [DStreamGraph](#) as `graph`
- The [checkpoint directory](#) as `checkpointDir`
- The [checkpoint interval](#) as `checkpointDuration`
- The [collection of pending batches to process](#) as `pendingTimes`
- The [Spark configuration \(aka `SparkConf`\)](#) as `sparkConfPairs`

Enable `INFO` logging level for `org.apache.spark.streaming.Checkpoint` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

Tip

```
log4j.logger.org.apache.spark.streaming.Checkpoint=INFO
```

Refer to [Logging](#).

Serializing Checkpoint (serialize method)

```
serialize(checkpoint: Checkpoint, conf: SparkConf): Array[Byte]
```

`serialize` serializes the `checkpoint` object. It does so by creating a compression codec to write the input `checkpoint` object with and returns the result as a collection of bytes.

Caution

[FIXME](#) Describe compression codecs in Spark.

Deserializing Checkpoint (deserialize method)

```
deserialize(inputStream: InputStream, conf: SparkConf): Checkpoint
```

`deserialize` reconstructs a `Checkpoint` object from the input `inputStream`. It uses a compression codec and once read [the just-built Checkpoint object is validated](#) and returned back.

Note

`deserialize` is called when [reading the latest valid checkpoint file](#).

Validating Checkpoint (validate method)

```
validate(): Unit
```

`validate` validates the `Checkpoint`. It ensures that `master`, `framework`, `graph`, and `checkpointTime` are defined, i.e. not `null`.

Note

`validate` is called when a [checkpoint is deserialized from an input stream](#).

You should see the following INFO message in the logs when the object passes the validation:

```
INFO Checkpoint: Checkpoint for time [checkpointTime] ms validated
```

Get Collection of Checkpoint Files from Directory (`getCheckpointFiles` method)

```
getCheckpointFiles(checkpointDir: String, fsOption: Option[FileSystem] = None): Seq[Path]
```

`getCheckpointFiles` method returns a collection of checkpoint files from the given checkpoint directory `checkpointDir`.

The method sorts the checkpoint files by time with a temporary `.bk` checkpoint file first (given a pair of a checkpoint file and its backup file).

CheckpointWriter

An instance of `CheckpointWriter` is created (lazily) when `JobGenerator` is, but only when `JobGenerator` is configured for checkpointing.

It uses the internal single-thread thread pool executor to execute checkpoint writes asynchronously and does so until it is stopped.

Writing Checkpoint for Batch Time (write method)

```
write(checkpoint: Checkpoint, clearCheckpointDataLater: Boolean): Unit
```

`write` method serializes the checkpoint object and passes the serialized form to `CheckpointWriteHandler` to write asynchronously (i.e. on a separate thread) using single-thread thread pool executor.

Note

It is called when `JobGenerator` receives `DoCheckpoint` event and the batch time is eligible for checkpointing.

You should see the following INFO message in the logs:

```
INFO CheckpointWriter: Submitted checkpoint of time [checkpoint.checkpointTime] ms writer
```

If the asynchronous checkpoint write fails, you should see the following ERROR in the logs:

```
ERROR Could not submit checkpoint task to the thread pool executor
```

Stopping CheckpointWriter (using stop method)

```
stop(): Unit
```

`CheckpointWriter` uses the internal `stopped` flag to mark whether it is stopped or not.

Note	<code>stopped</code> flag is disabled, i.e. <code>false</code> , when <code>CheckpointWriter</code> is created.
------	---

`stop` method checks the internal `stopped` flag and returns if it says it is stopped already.

If not, it orderly shuts down the [internal single-thread thread pool executor](#) and awaits termination for 10 seconds. During that time, any asynchronous checkpoint writes can be safely finished, but no new tasks will be accepted.

Note	The wait time before <code>executor</code> stops is fixed, i.e. not configurable, and is set to 10 seconds.
------	---

After 10 seconds, when the thread pool did not terminate, `stop` stops it forcefully.

You should see the following INFO message in the logs:

```
INFO CheckpointWriter: CheckpointWriter executor terminated? [terminated], waited for [ti
```

`CheckpointWriter` is marked as stopped, i.e. `stopped` flag is set to `true`.

Single-Thread Thread Pool Executor

`executor` is an internal single-thread thread pool executor for executing [asynchronous checkpoint writes using CheckpointWriteHandler](#).

It shuts down when [CheckpointWriter is stopped](#) (with a 10-second graceful period before it terminated forcefully).

CheckpointWriteHandler — Asynchronous Checkpoint Writes

`CheckpointWriteHandler` is an (internal) thread of execution that does checkpoint writes. It is instantiated with `checkpointTime`, the serialized form of the checkpoint, and whether or not to clean checkpoint data later flag (as `clearCheckpointDataLater`).

Note	It is only used by CheckpointWriter to queue a checkpoint write for a batch time .
------	--

It records the current checkpoint time (in `latestCheckpointTime`) and calculates the name of the checkpoint file.

Note	The name of the checkpoint file is <code>checkpoint-[checkpointTime.milliseconds]</code> .
------	--

It uses a backup file to do atomic write, i.e. it writes to the checkpoint backup file first and renames the result file to the final checkpoint file name.

Note	The name of the checkpoint backup file is <code>checkpoint-[checkpointTime.milliseconds].bk</code> .
------	--

Note	<code>CheckpointWriteHandler</code> does 3 write attempts at the maximum. The value is not configurable.
------	--

When attempting to write, you should see the following INFO message in the logs:

```
INFO CheckpointWriter: Saving checkpoint for time [checkpointTime] ms to file '[checkpointTime].bk'
```

Note	It deletes any checkpoint backup files that may exist from the previous attempts.
------	---

It then deletes checkpoint files when there are more than 10.

Note	The number of checkpoint files when the deletion happens, i.e. 10 , is fixed and not configurable.
------	---

You should see the following INFO message in the logs:

```
INFO CheckpointWriter: Deleting [file]
```

If all went fine, you should see the following INFO message in the logs:

```
INFO CheckpointWriter: Checkpoint for time [checkpointTime] ms saved to file '[checkpointTime].bk'
```

[JobGenerator](#) is informed that the checkpoint write completed (with `checkpointTime` and `clearCheckpointDataLater` flag).

In case of write failures, you can see the following WARN message in the logs:

```
WARN CheckpointWriter: Error in attempt [attempts] of writing checkpoint to [checkpointFi
```

If the number of write attempts exceeded (the fixed) 10 or [CheckpointWriter was stopped](#) before any successful checkpoint write, you should see the following WARN message in the logs:

```
WARN CheckpointWriter: Could not write checkpoint for time [checkpointTime] to file [che
```

CheckpointReader

`CheckpointReader` is a `private[streaming]` helper class to [read the latest valid checkpoint file to recreate StreamingContext from \(given the checkpoint directory\)](#).

Reading Latest Valid Checkpoint File

```
read(checkpointDir: String): Option[Checkpoint]
read(checkpointDir: String, conf: SparkConf,
     hadoopConf: Configuration, ignoreReadError: Boolean = false): Option[Checkpoint]
```

`read` methods read the latest valid checkpoint file from the [checkpoint directory](#) `checkpointDir`. They differ in whether Spark configuration `conf` and Hadoop configuration `hadoopConf` are given or created in place.

Note	The 4-parameter <code>read</code> method is used by StreamingContext to recreate itself from a checkpoint file .
------	--

The first `read` throws no `SparkException` when no checkpoint file could be read.

Note	It appears that no part of Spark Streaming uses the simplified version of <code>read</code> .
------	---

`read` uses Apache Hadoop's [Path](#) and [Configuration](#) to get the checkpoint files (using [Checkpoint.getCheckpointFiles](#)) in reverse order.

If there is no checkpoint file in the checkpoint directory, it returns None.

You should see the following INFO message in the logs:

```
INFO CheckpointReader: Checkpoint files found: [checkpointFiles]
```

The method reads all the checkpoints (from the youngest to the oldest) until one is successfully loaded, i.e. [deserialized](#).

You should see the following INFO message in the logs just before deserializing a `checkpoint file`:

```
INFO CheckpointReader: Attempting to load checkpoint from file [file]
```

If the checkpoint file was loaded, you should see the following INFO messages in the logs:

```
INFO CheckpointReader: Checkpoint successfully loaded from file [file]
INFO CheckpointReader: Checkpoint was generated at time [checkpointTime]
```

In case of any issues while loading a checkpoint file, you should see the following WARN in the logs and the corresponding exception:

```
WARN CheckpointReader: Error reading checkpoint from file [file]
```

Unless `ignoreReadError` flag is disabled, when no checkpoint file could be read, `SparkException` is thrown with the following message:

```
Failed to read checkpoint from directory [checkpointPath]
```

`None` is returned at this point and the method finishes.

JobScheduler

Streaming scheduler (`JobScheduler`) schedules streaming jobs to be run as Spark jobs. It is created as part of [creating a StreamingContext](#) and starts with it.

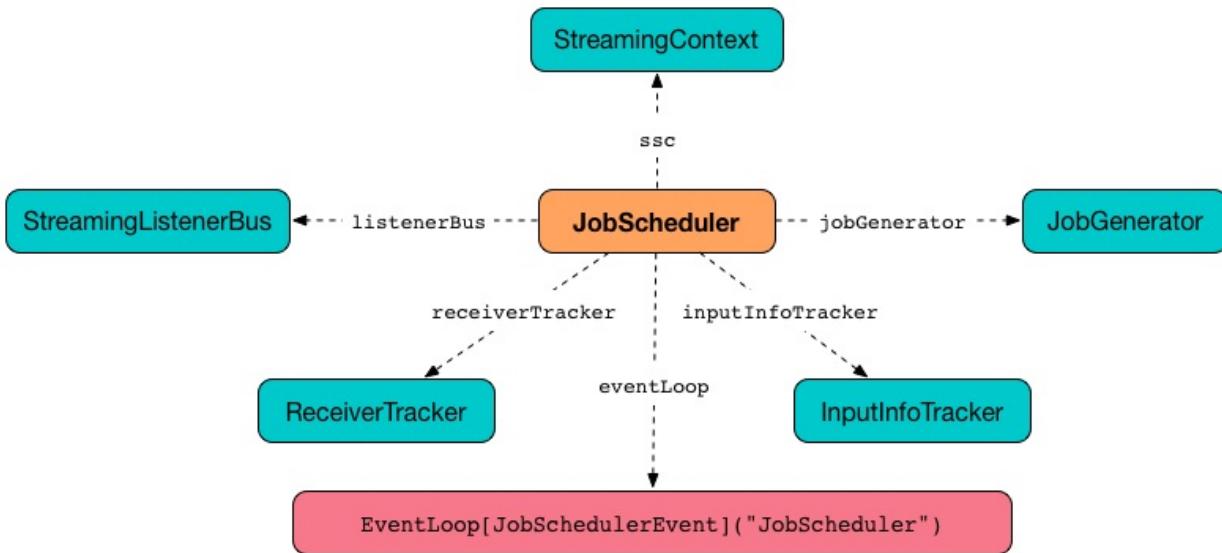


Figure 1. JobScheduler and Dependencies

It tracks jobs submitted for execution (as `JobSets` via `submitJobSet` method) in `jobSets` internal map.

Note

JobSets are submitted by [JobGenerator](#).

It uses a **streaming scheduler queue** for streaming jobs to be executed.

Tip

Enable `DEBUG` logging level for `org.apache.spark.streaming.scheduler.JobScheduler` logger to see what happens in JobScheduler.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.streaming.scheduler.JobScheduler=DEBUG
```

Refer to [Logging](#).

Starting JobScheduler (start method)

```
start(): Unit
```

When `JobScheduler` starts (i.e. when `start` is called), you should see the following DEBUG message in the logs:

```
DEBUG JobScheduler: Starting JobScheduler
```

It then goes over all the dependent services and starts them one by one as depicted in the figure.

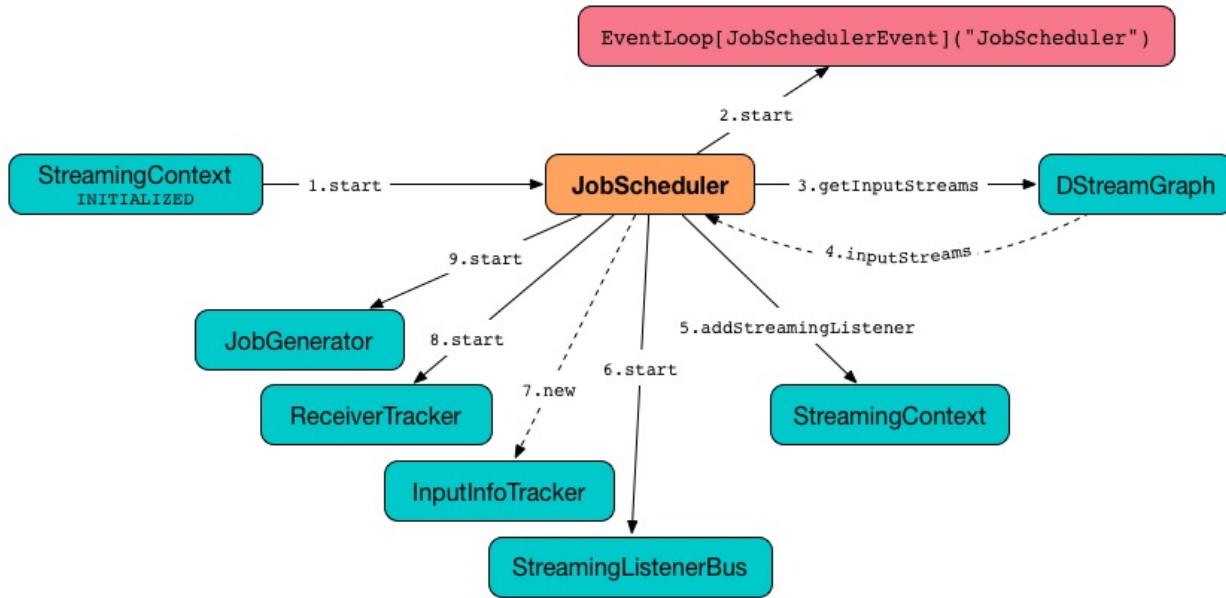


Figure 2. JobScheduler Start procedure

It first starts [JobSchedulerEvent Handler](#).

It asks [DStreamGraph](#) for input [dstreams](#) and registers [their RateControllers](#) (if defined) as [streaming listeners](#). It starts [StreamingListenerBus](#) afterwards.

It instantiates [ReceiverTracker](#) and [InputInfoTracker](#). It then starts the `ReceiverTracker`.

It starts [JobGenerator](#).

Just before `start` finishes, you should see the following INFO message in the logs:

```
INFO JobScheduler: Started JobScheduler
```

Pending Batches to Process (getPendingTimes method)

Caution	FIXME
---------	-----------------------

Stopping JobScheduler (stop method)

```
stop(processAllReceivedData: Boolean): Unit
```

`stop` stops `JobScheduler`.

Note	It is called when StreamingContext is being stopped.
------	--

You should see the following DEBUG message in the logs:

```
DEBUG JobScheduler: Stopping JobScheduler
```

[ReceiverTracker](#) is stopped.

Note	ReceiverTracker is only assigned (and started) while <code>JobScheduler</code> is starting.
------	---

It stops generating jobs.

You should see the following DEBUG message in the logs:

```
DEBUG JobScheduler: Stopping job executor
```

[jobExecutor Thread Pool](#) is shut down (using `jobExecutor.shutdown()`).

If the stop should wait for all received data to be processed (the input parameter `processAllReceivedData` is `true`), `stop` awaits termination of [jobExecutor Thread Pool](#) for **1 hour** (it is assumed that it is enough and is not configurable). Otherwise, it waits for **2 seconds**.

[jobExecutor Thread Pool](#) is forcefully shut down (using `jobExecutor.shutdownNow()`) unless it has terminated already.

You should see the following DEBUG message in the logs:

```
DEBUG JobScheduler: Stopped job executor
```

[StreamingListenerBus](#) and [eventLoop - JobSchedulerEvent Handler](#) are stopped.

You should see the following INFO message in the logs:

```
INFO JobScheduler: Stopped JobScheduler
```

Submitting Collection of Jobs for Execution (`submitJobSet` method)

When `submitJobSet(jobSet: JobSet)` is called, it reacts appropriately per `jobSet` [JobSet](#) given.

Note

The method is called by [JobGenerator](#) only (as part of [JobGenerator.generateJobs](#) and [JobGenerator.restart](#)).

When no streaming jobs are inside the `jobSet`, you should see the following INFO in the logs:

```
INFO JobScheduler: No jobs added for time [jobSet.time]
```

Otherwise, when there is at least one streaming job inside the `jobSet`, [StreamingListenerBatchSubmitted](#) (with data statistics of every registered input stream for which the streaming jobs were generated) is posted to [StreamingListenerBus](#).

The JobSet is added to the internal [jobSets](#) registry.

It then goes over every streaming job in the `jobset` and executes a [JobHandler](#) (on [jobExecutor Thread Pool](#)).

At the end, you should see the following INFO message in the logs:

```
INFO JobScheduler: Added jobs for time [jobSet.time] ms
```

JobHandler

`JobHandler` is a thread of execution for a [streaming job](#) (that simply calls `Job.run`).

Note

It is called when a new [JobSet](#) is submitted (see [submitJobSet](#) in this document).

When started, it prepares the environment (so the streaming job can be nicely displayed in the web UI under `/streaming/batch/?id=[milliseconds]`) and posts `JobStarted` event to [JobSchedulerEvent](#) event loop.

It runs the [streaming job](#) that executes the job function as defined while [generating a streaming job for an output stream](#).

Note

This is the moment when a [Spark \(core\) job is run](#).

You may see similar-looking INFO messages in the logs (it depends on the [operators](#) you use):

```

INFO SparkContext: Starting job: print at <console>:39
INFO DAGScheduler: Got job 0 (print at <console>:39) with 1 output partitions
...
INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 0 (KafkaRDD[2] at createDi
...
INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 987 bytes result sent to driver
...
INFO DAGScheduler: Job 0 finished: print at <console>:39, took 0.178689 s

```

It posts `JobCompleted` event to `JobSchedulerEvent` event loop.

jobExecutor Thread Pool

While `JobScheduler` is instantiated, the daemon thread pool `streaming-job-executor-ID` with `spark.streaming.concurrentJobs` threads is created.

It is used to execute `JobHandler` for jobs in `JobSet` (see `submitJobSet` in this document).

It shuts down when `StreamingContext` stops.

eventLoop - JobSchedulerEvent Handler

`JobScheduler` uses `EventLoop` for `JobSchedulerEvent` events. It accepts `JobStarted` and `JobCompleted` events. It also processes `ErrorReported` events.

JobStarted and JobScheduler.handleJobStart

When `JobStarted` event is received, `JobScheduler.handleJobStart` is called.

Note	It is <code>JobHandler</code> to post <code>JobStarted</code> .
------	---

`handleJobStart(job: Job, startTime: Long)` takes a `JobSet` (from `jobSets`) and checks whether it has already been started.

It posts `StreamingListenerBatchStarted` to `StreamingListenerBus` when the `JobSet` is about to start.

It posts `StreamingListenerOutputOperationStarted` to `StreamingListenerBus`.

You should see the following INFO message in the logs:

```

INFO JobScheduler: Starting job [job.id] from job set of time [jobSet.time] ms

```

JobCompleted and JobScheduler.handleJobCompletion

When `JobCompleted` event is received, it triggers `JobScheduler.handleJobCompletion(job: Job, completedTime: Long)`.

Note

[JobHandler](#) posts `JobCompleted` events when it finishes running a streaming job.

`handleJobCompletion` looks the [JobSet](#) up (from the [jobSets](#) internal registry) and calls `JobSet.handleJobCompletion(job)` (that marks the `JobSet` as completed when no more streaming jobs are incomplete). It also calls `Job.setEndTime(completedTime)`.

It posts `StreamingListenerOutputOperationCompleted` to [StreamingListenerBus](#).

You should see the following INFO message in the logs:

```
INFO JobScheduler: Finished job [job.id] from job set of time [jobSet.time] ms
```

If the entire JobSet is completed, it removes it from [jobSets](#), and calls [JobGenerator.onBatchCompletion](#).

You should see the following INFO message in the logs:

```
INFO JobScheduler: Total delay: [totalDelay] s for time [time] ms (execution: [processing
```

It posts `StreamingListenerBatchCompleted` to [StreamingListenerBus](#).

It reports an error if the job's result is a failure.

StreamingListenerBus and StreamingListenerEvents

[StreamingListenerBus](#) is a asynchronous listener bus to post `StreamingListenerEvent` events to [streaming listeners](#).

Internal Registries

`JobScheduler` maintains the following information in internal registries:

- `jobSets` - a mapping between time and JobSets. See [JobSet](#).

JobSet

A `JobSet` represents a collection of [streaming jobs](#) that were created at (batch) `time` for [output streams](#) (that have ultimately produced a streaming job as they may opt out).

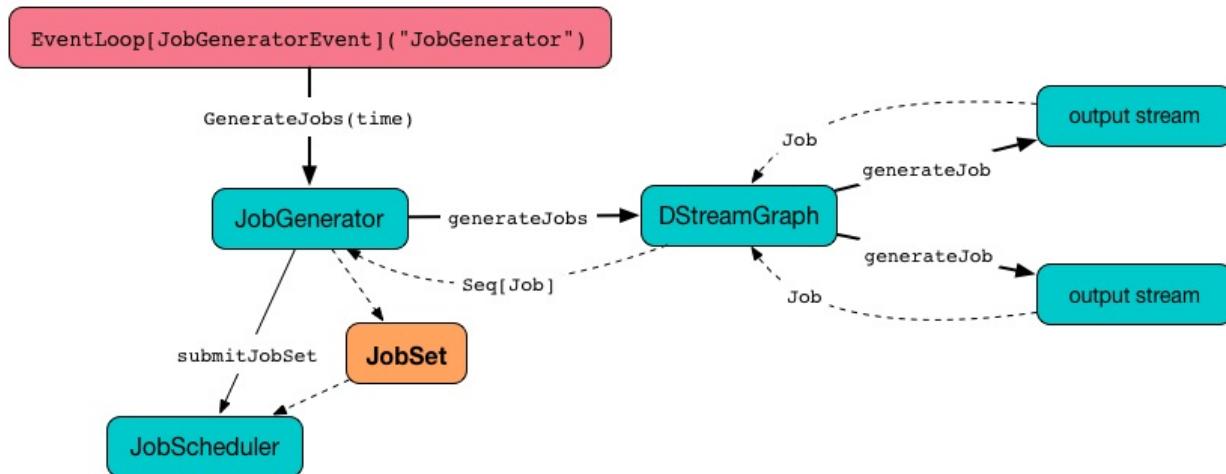


Figure 3. JobSet Created and Submitted to JobScheduler

`JobSet` tracks what streaming jobs are in incomplete state (in `incompleteJobs` internal registry).

Note	At the beginning (when <code>JobSet</code> is created) all streaming jobs are incomplete.
------	---

Caution	FIXME There is a duplication in how streaming jobs are tracked as completed since a <code>Job</code> knows about its <code>_endTime</code> . Is this a optimization? How much time does it buy us?
---------	---

A `JobSet` tracks the following moments in its lifecycle:

- `submissionTime` being the time when the instance was created.
- `processingStartTime` being the time when the first streaming job in the collection was started.
- `processingEndTime` being the time when the last streaming job in the collection finished processing.

A `JobSet` changes state over time. It can be in the following states:

- **Created** after a `JobSet` was created. `submissionTime` is set.
- **Started** after `JobSet.handleJobStart` was called. `processingStartTime` is set.
- **Completed** after `JobSet.handleJobCompletion` and no more jobs are incomplete (in `incompleteJobs` internal registry). `processingEndTime` is set.

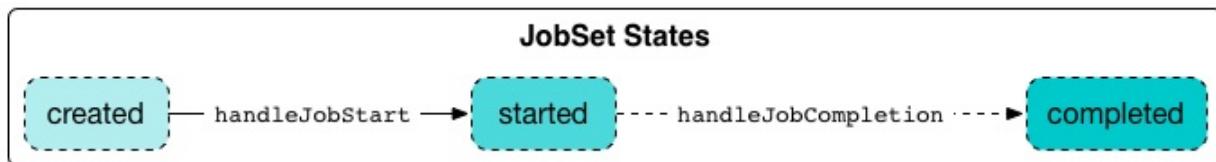


Figure 4. JobSet States

Given the states a `JobSet` has **delays**:

- **Processing delay** is the time spent for processing all the streaming jobs in a `JobSet` from the time the very first job was started, i.e. the time between started and completed states.
- **Total delay** is the time from the batch time until the `JobSet` was completed.

Note	Total delay is always longer than processing delay.
------	---

You can map a `JobSet` to a `BatchInfo` using `toBatchInfo` method.

Note	<code>BatchInfo</code> is used to create and post <code>StreamingListenerBatchSubmitted</code> , <code>StreamingListenerBatchStarted</code> , and <code>StreamingListenerBatchCompleted</code> events.
------	--

`JobSet` is used (created or processed) in:

- `JobGenerator.generateJobs`
- `JobScheduler.submitJobSet(jobSet: JobSet)`
- `JobGenerator.restart`
- `JobScheduler.handleJobStart(job: Job, startTime: Long)`
- `JobScheduler.handleJobCompletion(job: Job, completedTime: Long)`

InputInfoTracker

`InputInfoTracker` tracks batch times and batch statistics for `input streams` (per input stream id with `StreamInputInfo`). It is later used when `JobGenerator` submits streaming jobs for a `batch time` (and propagated to interested listeners as `StreamingListenerBatchSubmitted` event).

Note	<code>InputInfoTracker</code> is managed by <code>JobScheduler</code> , i.e. it is created when <code>JobScheduler</code> starts and is stopped alongside.
------	--

`InputInfoTracker` uses internal registry `batchTimeToInputInfos` to maintain the mapping of batch times and `input streams` (i.e. another mapping between input stream ids and `StreamInputInfo`).

It accumulates batch statistics at every batch time when [input streams are computing RDDs](#) (and explicitly call `InputInfoTracker.reportInfo` method).

	<p>It is up to input streams to have these batch statistics collected (and requires calling <code>InputInfoTracker.reportInfo</code> method explicitly).</p> <p>The following input streams report information:</p> <ul style="list-style-type: none">• DirectKafkaInputDStream• ReceiverInputDStreams - Input Streams with Receivers• FileInputDStream
--	---

Cleaning up

```
cleanup(batchThreshTime: Time): Unit
```

You should see the following INFO message when cleanup of old batch times is requested (akin to *garbage collection*):

```
INFO InputInfoTracker: remove old batch metadata: [timesToCleanup]
```

Caution	FIXME When is this called?
---------	--

JobGenerator

`JobGenerator` asynchronously generates streaming jobs every `batch interval` (using `recurring timer`) that may or may not be checkpointed afterwards. It also periodically requests `clearing up metadata` and `checkpoint data` for each input `dstream`.

Note

`JobGenerator` is completely owned and managed by `JobScheduler`, i.e. `JobScheduler` creates an instance of `JobGenerator` and starts it (while being started itself).

Tip

Enable `INFO` or `DEBUG` logging level for `org.apache.spark.streaming.scheduler.JobGenerator` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.streaming.scheduler.JobGenerator=DEBUG
```

Refer to [Logging](#).

Starting JobGenerator (start method)

```
start(): Unit
```

`start` method creates and starts the internal `JobGeneratorEvent` handler.

Note

`start` is called when `JobScheduler` starts.

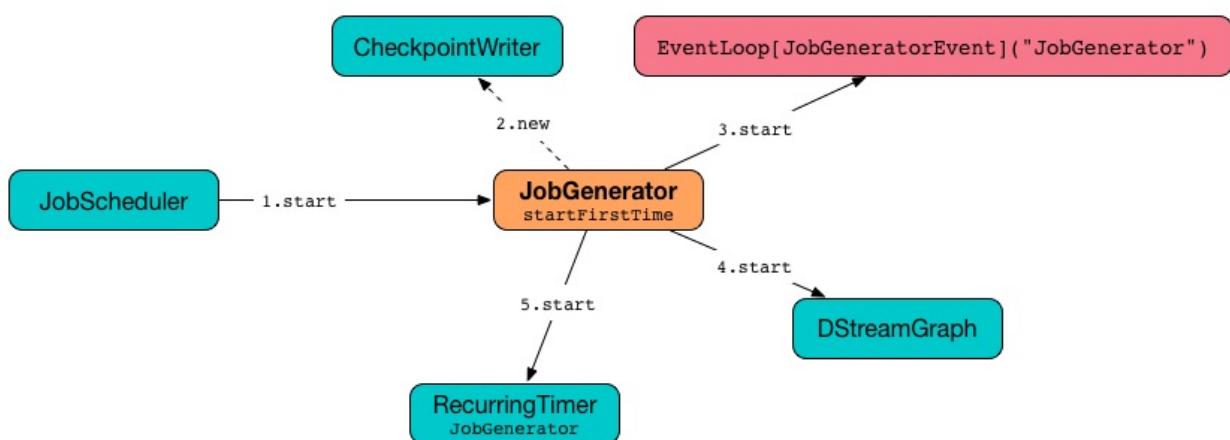


Figure 1. JobGenerator Start (First Time) procedure (tip: follow the numbers)

It first checks whether or not the internal event loop has already been created which is the way to know that the `JobScheduler` was started. If so, it does nothing and exits.

Only if [checkpointing is enabled](#), it creates [CheckpointWriter](#).

It then creates and starts the internal [JobGeneratorEvent handler](#).

Depending on whether [checkpoint directory](#) is available or not it [restarts itself](#) or [starts](#), respectively.

Start Time and startFirstTime Method

```
startFirstTime(): Unit
```

`startFirstTime` starts [DStreamGraph](#) and the [timer](#).

Note

`startFirstTime` is called when [JobGenerator](#) starts (and no [checkpoint directory](#) is available).

It first requests [timer](#) for the [start time](#) and passes the start time along to [DStreamGraph.start](#) and [RecurringTimer.start](#).

Note

The start time has the property of being a multiple of [batch interval](#) and after the current system time. It is in the hands of [recurring timer](#) to calculate a time with the property given a batch interval.

Note

Because of the property of the start time, [DStreamGraph.start](#) is passed the time of one batch interval before the calculated start time.

Note

When [recurring timer](#) starts for [JobGenerator](#), you should see the following INFO message in the logs:

```
INFO RecurringTimer: Started timer for JobGenerator at time [nextTime]
```

Right before the method finishes, you should see the following INFO message in the logs:

```
INFO JobGenerator: Started JobGenerator at [startTime] ms
```

Stopping JobGenerator (stop method)

```
stop(processReceivedData: Boolean): Unit
```

`stop` stops a [JobGenerator](#). The [processReceivedData](#) flag tells whether to stop [JobGenerator](#) gracefully, i.e. after having processed all received data and pending streaming jobs, or not.

Note	<code>JobGenerator</code> is stopped as JobScheduler stops .
	<code>processReceivedData</code> flag in <code>JobGenerator</code> corresponds to the value of <code>processAllReceivedData</code> in <code>JobScheduler</code> .

It first checks whether `eventLoop` internal event loop was ever started (through checking `null`).

Warning	It doesn't set <code>eventLoop</code> to <code>null</code> (but it is assumed to be the marker).
---------	--

When `JobGenerator` should stop immediately, i.e. ignoring unprocessed data and pending streaming jobs (`processReceivedData` flag is disabled), you should see the following INFO message in the logs:

```
INFO JobGenerator: Stopping JobGenerator immediately
```

It requests [the timer to stop forcefully](#) (`interruptTimer` is enabled) and [stops the graph](#).

Otherwise, when `JobGenerator` should stop gracefully, i.e. `processReceivedData` flag is enabled, you should see the following INFO message in the logs:

```
INFO JobGenerator: Stopping JobGenerator gracefully
```

You should immediately see the following INFO message in the logs:

```
INFO JobGenerator: Waiting for all received blocks to be consumed for job generation
```

`JobGenerator` waits [spark.streaming.gracefulStopTimeout](#) milliseconds or until [ReceiverTracker has any blocks left to be processed](#) (whatever is shorter) before continuing.

Note	Poll (sleeping) time is <code>100</code> milliseconds and is not configurable.
------	--

When a timeout occurs, you should see the WARN message in the logs:

```
WARN JobGenerator: Timed out while stopping the job generator (timeout = [stopTimeoutMs])
```

After the waiting is over, you should see the following INFO message in the logs:

```
INFO JobGenerator: Waited for all received blocks to be consumed for job generation
```

It requests `timer` to stop generating streaming jobs (`interruptTimer` flag is disabled) and stops the graph.

You should see the following INFO message in the logs:

```
INFO JobGenerator: Stopped generation timer
```

You should immediately see the following INFO message in the logs:

```
INFO JobGenerator: Waiting for jobs to be processed and checkpoints to be written
```

`JobGenerator` waits `spark.streaming.gracefulStopTimeout` milliseconds or until all the batches have been processed (whatever is shorter) before continuing. It waits for batches to complete using `last processed batch` internal property that should eventually be exactly the time when the `timer was stopped` (it returns the last time for which the streaming job was generated).

Note `spark.streaming.gracefulStopTimeout` is ten times the `batch interval` by default.

After the waiting is over, you should see the following INFO message in the logs:

```
INFO JobGenerator: Waited for jobs to be processed and checkpoints to be written
```

Regardless of `processReceivedData` flag, if `checkpointing was enabled`, it stops `CheckpointWriter`.

It then stops the `event loop`.

As the last step, when `JobGenerator` is assumed to be stopped completely, you should see the following INFO message in the logs:

```
INFO JobGenerator: Stopped JobGenerator
```

Starting from Checkpoint (restart method)

```
restart(): Unit
```

`restart` starts `JobGenerator` from `checkpoint`. It basically reconstructs the runtime environment of the past execution that may have stopped immediately, i.e. without waiting for all the streaming jobs to complete when checkpoint was enabled, or due to a abrupt shutdown (a unrecoverable failure or similar).

Note	<code>restart</code> is called when JobGenerator starts and checkpoint is present .
------	---

`restart` first calculates the batches that may have been missed while `JobGenerator` was down, i.e. batch times between the current restart time and the time of [initial checkpoint](#).

Warning	<code>restart</code> doesn't check whether the initial checkpoint exists or not that may lead to NPE.
---------	---

You should see the following INFO message in the logs:

```
INFO JobGenerator: Batches during down time ([size] batches): [downTimes]
```

It then ask the initial checkpoint for pending batches, i.e. the times of streaming job sets.

Caution	FIXME What are the pending batches? Why would they ever exist?
---------	--

You should see the following INFO message in the logs:

```
INFO JobGenerator: Batches pending processing ([size] batches): [pendingTimes]
```

It then computes the batches to reschedule, i.e. pending and down time batches that are before restart time.

You should see the following INFO message in the logs:

```
INFO JobGenerator: Batches to reschedule ([size] batches): [timesToReschedule]
```

For each batch to reschedule, `restart` requests [ReceiverTracker to allocate blocks to batch](#) and [submits streaming job sets for execution](#).

Note	<code>restart</code> mimics generateJobs method.
------	--

It [restarts the timer](#) (by using `restartTime` as `startTime`).

You should see the following INFO message in the logs:

```
INFO JobGenerator: Restarted JobGenerator at [restartTime]
```

Last Processed Batch (aka lastProcessedBatch)

JobGenerator tracks the last batch time for which the batch was completed and cleanups performed as `lastProcessedBatch` internal property.

The only purpose of the `lastProcessedBatch` property is to allow for [stopping the streaming context gracefully](#), i.e. to wait until all generated streaming jobs are completed.

Note

It is set to the batch time after [ClearMetadata Event](#) is processed (when [checkpointing is disabled](#)).

JobGenerator eventLoop and JobGeneratorEvent Handler

`JobGenerator` uses the internal `EventLoop` event loop to process `JobGeneratorEvent` events asynchronously (one event at a time) on a separate dedicated *single* thread.

Note

`EventLoop` uses unbounded [java.util.concurrent.LinkedBlockingDeque](#).

For every `JobGeneratorEvent` event, you should see the following DEBUG message in the logs:

```
DEBUG JobGenerator: Got event [event]
```

There are 4 `JobGeneratorEvent` event types:

- [GenerateJobs](#)
- [DoCheckpoint](#)
- [ClearMetadata](#)
- [ClearCheckpointData](#)

See below in the document for the extensive coverage of the supported `JobGeneratorEvent` event types.

GenerateJobs Event and generateJobs method

Note

`GenerateJobs` events are posted regularly by the internal `timer RecurringTimer` every [batch interval](#). The `time` parameter is exactly the current batch time.

When `GenerateJobs(time: Time)` event is received the internal `generateJobs` method is called that [submits a collection of streaming jobs for execution](#).

```
generateJobs(time: Time)
```

It first calls `ReceiverTracker.allocateBlocksToBatch` (it does nothing when there are no receiver input streams in use), and then requests `DStreamGraph` for streaming jobs for a given batch time.

If the above two calls have finished successfully, `InputInfoTracker` is requested for data statistics of every registered input stream for the given batch time that together with the collection of streaming jobs (from `DStreamGraph`) is passed on to `JobScheduler.submitJobSet` (as a `JobSet`).

In case of failure, `JobScheduler.reportError` is called.

Ultimately, `DoCheckpoint` event is posted (with `clearCheckpointDataLater` being disabled, i.e. `false`).

DoCheckpoint Event and doCheckpoint method

Note	<code>DoCheckpoint</code> events are posted by <code>JobGenerator</code> itself as part of generating streaming jobs (with <code>clearCheckpointDataLater</code> being disabled, i.e. <code>false</code>) and clearing metadata (with <code>clearCheckpointDataLater</code> being enabled, i.e. <code>true</code>).
------	---

`DoCheckpoint` events trigger execution of `doCheckpoint` method.

```
doCheckpoint(time: Time, clearCheckpointDataLater: Boolean)
```

If `checkpointing is disabled` or the current batch `time` is not eligible for checkpointing, the method does nothing and exits.

Note	A current batch is eligible for checkpointing when the time interval between current batch <code>time</code> and zero time is a multiple of <code>checkpoint interval</code> .
------	---

Caution	FIXME Who checks and when whether checkpoint interval is greater than batch interval or not? What about checking whether a checkpoint interval is a multiple of batch time?
---------	--

Caution	FIXME What happens when you start a <code>StreamingContext</code> with a checkpoint directory that was used before?
---------	--

Otherwise, when checkpointing should be performed, you should see the following INFO message in the logs:

```
INFO JobGenerator: Checkpointing graph for time [time] ms
```

It requests `DStreamGraph` for updating checkpoint data and `CheckpointWriter` for writing a new checkpoint. Both are given the current batch `time` .

ClearMetadata Event and clearMetadata method

Note	<code>clearMetadata</code> are posted after a micro-batch for a batch time has completed.
------	---

It removes old RDDs that have been generated and collected so far by output streams (managed by [DStreamGraph](#)). It is a sort of *garbage collector*.

When `ClearMetadata(time)` arrives, it first asks [DStreamGraph](#) to clear metadata for the given time.

If [checkpointing is enabled](#), it posts a [DoCheckpoint](#) event (with `clearCheckpointDataLater` being enabled, i.e. `true`) and exits.

Otherwise, when checkpointing is disabled, it asks [DStreamGraph](#) for the maximum remember duration across all the input streams and requests [ReceiverTracker](#) and [InputInfoTracker](#) to do their cleanups.

Caution	<code>FIXME</code> Describe cleanups of ReceiverTracker and InputInfoTracker .
---------	--

Eventually, it marks the batch as fully processed, i.e. that the batch completed as well as checkpointing or metadata cleanups, using the [internal lastProcessedBatch marker](#).

ClearCheckpointData Event and clearCheckpointData method

Note	<code>clearCheckpointData</code> event is posted after checkpoint is saved and checkpoint cleanup is requested .
------	--

`ClearCheckpointData` events trigger execution of `clearCheckpointData` method.

```
clearCheckpointData(time: Time)
```

In short, `clearCheckpointData` requests [DStreamGraph](#), [ReceiverTracker](#), and [InputInfoTracker](#) to do the cleaning and marks the current batch `time` as [fully processed](#).

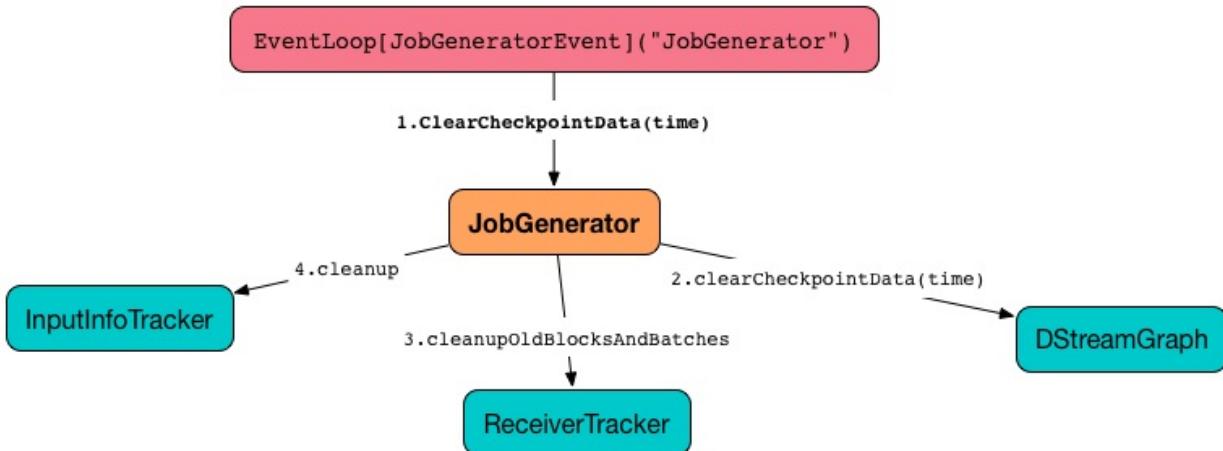


Figure 2. JobGenerator and ClearCheckpointData event

When executed, `clearCheckpointData` first requests `DStreamGraph` to clear checkpoint data for the given batch time.

It then asks `DStreamGraph` for the maximum remember interval. Given the maximum remember interval `JobGenerator` requests `ReceiverTracker` to cleanup old blocks and batches and `InputInfoTracker` to do cleanup for data accumulated before the maximum remember interval (from `time`).

Having done that, the current batch `time` is marked as [fully processed](#).

Whether or Not to Checkpoint (aka `shouldCheckpoint`)

`shouldCheckpoint` flag is used to control a `CheckpointWriter` as well as whether to post `DoCheckpoint` in `clearMetadata` or not.

`shouldCheckpoint` flag is enabled (i.e. `true`) when `checkpoint interval` and `checkpoint directory` are defined (i.e. not `null`) in `StreamingContext`.

Note	However the flag is completely based on the properties of <code>StreamingContext</code> , these dependent properties are used by <code>JobScheduler</code> only. <i>Really?</i>
------	---

[FIXME Report an issue](#)

Caution	When and what for are they set? Can one of <code>ssc.checkpointDuration</code> and <code>ssc.checkpointDir</code> be <code>null</code> ? Do they all have to be set and is this checked somewhere?
---------	--

Answer: See [Setup Validation](#).

Caution	Potential bug: Can <code>streamingContext</code> have no checkpoint duration set? At least, the batch interval must be set. In other words, it's <code>StreamingContext</code> to say whether to checkpoint or not and there should be a method in <code>StreamingContext</code> <i>not</i> <code>JobGenerator</code> .
---------	--

onCheckpointCompletion

Caution	FIXME
---------	-------

timer RecurringTimer

`timer RecurringTimer` (with the name being `JobGenerator`) is used to posts `GenerateJobs` events to the internal `JobGeneratorEvent handler` every `batch interval`.

Note	<code>timer</code> is created when <code>JobGenerator</code> is. It starts when <code>JobGenerator</code> starts (for the first time only).
------	---

DStreamGraph

`DStreamGraph` (is a final helper class that) manages **input** and **output dstreams**. It also holds **zero time** for the other components that marks the time when **it was started**.

`DStreamGraph` maintains the collections of `InputDStream` instances (as `inputStreams`) and output `DStream` instances (as `outputStreams`), but, more importantly, **it generates streaming jobs for output streams for a batch (time)**.

`DStreamGraph` holds the **batch interval** for the other parts of a Streaming application.

Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.streaming.DStreamGraph</code> logger to see what happens in <code>DStreamGraph</code>.</p> <p>Add the following line to <code>conf/log4j.properties</code>:</p> <pre>log4j.logger.org.apache.spark.streaming.DStreamGraph=DEBUG</pre> <p>Refer to Logging.</p>
------------	---

Zero Time (aka zeroTime)

Zero time (internally `zeroTime`) is the time when `DStreamGraph` has been started.

It is passed on down the output dstream graph so **output dstreams can initialize themselves**.

Start Time (aka startTime)

Start time (internally `startTime`) is the time when `DStreamGraph` has been started or restarted.

Note	At regular start start time is exactly zero time .
-------------	---

Batch Interval (aka batchDuration)

`DStreamGraph` holds the **batch interval** (as `batchDuration`) for the other parts of a Streaming application.

`setBatchDuration(duration: Duration)` is the method to set the batch interval.

It appears that it is *the* place for the value since it must be set before `JobGenerator` can be instantiated.

It *is* set while `StreamingContext` is being instantiated and is validated (using `validate()` method of `streamingContext` and `DStreamGraph`) before `StreamingContext` is started.

Maximum Remember Interval (`getMaxInputStreamRememberDuration` method)

```
getMaxInputStreamRememberDuration(): Duration
```

Maximum Remember Interval is the maximum `remember interval` across all the input `dstreams`. It is calculated using `getMaxInputStreamRememberDuration` method.

Note

It is called when `JobGenerator` is requested to `clear metadata` and `checkpoint data`.

Input DStreams Registry

Caution

[FIXME](#)

Output DStreams Registry

`DStream` by design has no notion of being an output `dstream`. To mark a `dstream` as output you need to register a `dstream` (using `DStream.register` method) which happens for...[FIXME](#)

Starting DStreamGraph

```
start(time: Time): Unit
```

When `DStreamGraph` is started (using `start` method), it sets zero time and `start time`.

Note

`start` method is called when `JobGenerator` starts for the first time (not from a `checkpoint`).

Note

You can start `DStreamGraph` as many times until `time` is not `null` and `zero time` has been set.

(*output dstreams*) `start` then walks over the collection of output `dstreams` and for each output `dstream`, one at a time, calls their `initialize(zeroTime)`, `remember` (with the current `remember interval`), and `validateAtStart` methods.

(*input dstreams*) When all the output streams are processed, it starts the input dstreams (in parallel) using `start` method.

Stopping DStreamGraph

```
stop(): Unit
```

Caution	FIXME
---------	-----------------------

Restarting DStreamGraph

```
restart(time: Time): Unit
```

`restart` sets `start time` to be `time` input parameter.

Note	This is the only moment when <code>zero time</code> can be different than <code>start time</code> .
------	---

Generating Streaming Jobs for Output Streams for Batch Time

```
generateJobs(time: Time): Seq[Job]
```

`generateJobs` method generates a collection of streaming jobs for output streams for a given batch `time`. It walks over each `registered output stream` (in `outputStreams` internal registry) and `requests each stream for a streaming job`

Note	<code>generateJobs</code> is called by <code>JobGenerator</code> to generate jobs for a given batch time or when restarted from checkpoint.
------	---

When `generateJobs` method executes, you should see the following DEBUG message in the logs:

```
DEBUG DStreamGraph: Generating jobs for time [time] ms
```

`generateJobs` then walks over each `registered output stream` (in `outputStreams` internal registry) and `requests the streams for a streaming job`.

Right before the method finishes, you should see the following DEBUG message with the number of streaming jobs generated (as `jobs.length`):

```
DEBUG DStreamGraph: Generated [jobs.length] jobs for time [time] ms
```

Validation Check

`validate()` method checks whether batch duration and at least one output stream have been set. It will throw `java.lang.IllegalArgumentException` when either is not.

Note	It is called when StreamingContext starts .
------	---

Metadata Cleanup

Note	It is called when JobGenerator clears metadata .
------	--

When `clearMetadata(time: Time)` is called, you should see the following DEBUG message in the logs:

```
DEBUG DStreamGraph: Clearing metadata for time [time] ms
```

It merely walks over the collection of output streams and (synchronously, one by one) asks to do [its own metadata cleaning](#).

When finishes, you should see the following DEBUG message in the logs:

```
DEBUG DStreamGraph: Cleared old metadata for time [time] ms
```

Restoring State for Output DStreams (`restoreCheckpointData` method)

```
restoreCheckpointData(): Unit
```

When `restoreCheckpointData()` is executed, you should see the following INFO message in the logs:

```
INFO DStreamGraph: Restoring checkpoint data
```

Then, every [output dstream](#) is requested to [restoreCheckpointData](#).

At the end, you should see the following INFO message in the logs:

```
INFO DStreamGraph: Restored checkpoint data
```

Note

`restoreCheckpointData` is executed when [StreamingContext is recreated from checkpoint](#).

Updating Checkpoint Data

```
updateCheckpointData(time: Time): Unit
```

Note

`It is called when JobGenerator processes DoCheckpoint events.`

When `updateCheckpointData` is called, you should see the following INFO message in the logs:

```
INFO DStreamGraph: Updating checkpoint data for time [time] ms
```

It then walks over every output dstream and calls its [updateCheckpointData\(time\)](#).

When `updateCheckpointData` finishes it prints out the following INFO message to the logs:

```
INFO DStreamGraph: Updated checkpoint data for time [time] ms
```

Checkpoint Cleanup

```
clearCheckpointData(time: Time)
```

Note

`clearCheckpointData` is called when [JobGenerator clears checkpoint data](#).

When `clearCheckpointData` is called, you should see the following INFO message in the logs:

```
INFO DStreamGraph: Clearing checkpoint data for time [time] ms
```

It merely walks through the collection of output streams and (synchronously, one by one) asks to do [their own checkpoint data cleaning](#).

When finished, you should see the following INFO message in the logs:

```
INFO DStreamGraph: Cleared checkpoint data for time [time] ms
```

Remember Interval

Remember interval is the time to remember (aka *cache*) the RDDs that have been generated by (output) dstreams in the context (before they are released and garbage collected).

It can be set using `remember` method.

remember method

```
remember(duration: Duration): Unit
```

`remember` method simply sets `remember interval` and exits.

Note	It is called by <code>StreamingContext.remember</code> method.
------	--

It first checks whether or not it has been set already and if so, throws

`java.lang.IllegalArgumentException` as follows:

```
java.lang.IllegalArgumentException: requirement failed: Remember du
at scala.Predef$.require(Predef.scala:219)
at org.apache.spark.streaming.DStreamGraph.remember(DStreamGraph.
at org.apache.spark.streaming.StreamingContext.remember(StreamingC
... 43 elided
```

Note	It only makes sense to call <code>remember</code> method before <code>DStreamGraph is started</code> , i.e. before <code>StreamingContext is started</code> , since the output dstreams are only given the remember interval when DStreamGraph starts.
------	--

Discretized Streams (DStreams)

Discretized Stream (DStream) is the fundamental concept of Spark Streaming. It is basically a stream of [RDDs](#) with elements being the data received from input streams over [batch duration](#) (possibly extended in scope by [windowed](#) or [stateful](#) operators).

There is no notion of input and output dstreams. DStreams are all instances of `DStream` abstract class (see [DStream Contract](#) in this document). You may however *correctly* assume that all dstreams are input. And it happens to be so until you [register a dstream](#) that marks it as output.

It is represented as [org.apache.spark.streaming.dstream.DStream](#) abstract class.

Tip Enable `INFO` or `DEBUG` logging level for `org.apache.spark.streaming.dstream.DStream` logger to see what happens inside a DStream .

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.streaming.dstream.DStream=DEBUG
```

Refer to [Logging](#).

DStream Contract

A `DStream` is defined by the following properties (with the names of the corresponding methods that subclasses have to implement):

- **dstream dependencies**, i.e. a collection of `DStreams` that this `DStream` depends on. They are often referred to as **parent dstreams**.

```
def dependencies: List[DStream[_]]
```

- **slide duration** (aka *slide interval*), i.e. a time interval after which the stream is requested to generate a RDD out of input data it consumes.

```
def slideDuration: Duration
```

- How to **compute (generate)** an optional RDD for the given batch if any. `validTime` is a point in time that marks the end boundary of slide duration.

```
def compute(validTime: Time): Option[RDD[T]]
```

Creating DStreams

You can create dstreams through [the built-in input stream constructors using streaming context](#) or more specialized add-ons for external input data sources, e.g. [Apache Kafka](#).

Note

DStreams can only be created before [StreamingContext is started](#).

Zero Time (aka zeroTime)

Zero time (internally `zeroTime`) is the time when a [dstream was initialized](#).

It serves as the initialization marker (via `isInitialized` method) and helps calculating intervals for RDD checkpointing (when [checkpoint interval](#) is set and the current batch time is a multiple thereof), [slicing](#), and the time validation for a batch (when a [dstream generates a RDD](#)).

Remember Interval (aka rememberDuration)

Remember interval (internally `rememberDuration`) is the time interval for how long a dstream is supposed to remember (aka [cache](#)) RDDs created. This is a mandatory attribute of every dstream which is [validated at startup](#).

Note

It is used for [metadata cleanup](#) of a dstream.

Initially, when a [dstream is created](#), the remember interval is not set (i.e. `null`), but is set when the [dstream is initialized](#).

It can be set to a custom value using [remember](#) method.

Note

You may see the current value of remember interval when a dstream is [validated at startup](#) and the log level is INFO.

generatedRDDs - Internal Cache of Batch Times and Corresponding RDDs

`generatedRDDs` is an internal collection of pairs of batch times and the corresponding RDDs that were generated for the batch. It acts as a cache when [a dstream is requested to compute a RDD for batch](#) (i.e. `generatedRDDs` may already have the RDD or gets a new RDD added).

`generatedRDDs` is empty initially, i.e. when a dstream is created.

It is a *transient* data structure so it is not serialized when a dstream is. It is initialized to an empty collection when deserialized. You should see the following DEBUG message in the logs when it happens:

```
DEBUG [the simple class name of dstream].readObject used
```

As new RDDs are added, dstreams offer a way [to clear the old metadata](#) during which the old RDDs are removed from `generatedRDDs` collection.

If [checkpointing is used](#), `generatedRDDs` collection can be [recreated from a storage](#).

Initializing DStreams (initialize method)

```
initialize(time: Time): Unit
```

`initialize` method sets [zero time](#) and optionally [checkpoint interval](#) (if the dstream [must checkpoint](#) and the interval was not set already) and [remember duration](#).

Note	<code>initialize</code> method is called for output dstreams only when DStreamGraph is started .
------	--

The zero time of a dstream can only be set once or be set again to the same zero time.

Otherwise, it throws `SparkException` as follows:

```
zeroTime is already initialized to [zeroTime], cannot initialize it again to [time]
```

It verifies that [checkpoint interval](#) is defined when [mustCheckpoint](#) was enabled.

Note	The internal <code>mustCheckpoint</code> flag is disabled by default. It is set by custom dstreams like StateDStreams .
------	---

If `mustCheckpoint` is enabled and the checkpoint interval was not set, it is automatically set to the [slide interval](#) or 10 seconds, whichever is longer. You should see the following INFO message in the logs when the checkpoint interval was set automatically:

```
INFO [DStreamType]: Checkpoint interval automatically set to [checkpointDuration]
```

It then ensures that [remember interval](#) is at least twice the checkpoint interval (only if defined) or the slide duration.

At the very end, it initializes the parent dstreams (available as [dependencies](#)) that recursively initializes the entire graph of dstreams.

remember Method

```
remember(duration: Duration): Unit
```

`remember` sets [remember interval](#) for the current dstream and the dstreams it depends on (see [dependencies](#)).

If the input `duration` is specified (i.e. not `null`), `remember` allows setting the remember interval (only when the current value was not set already) or extend it (when the current value is shorter).

You should see the following INFO message in the logs when the remember interval changes:

```
INFO Duration for remembering RDDs set to [rememberDuration] for [dstream]
```

At the end, `remember` always sets the current [remember interval](#) (whether it was set, extended or did not change).

Checkpointing DStreams (checkpoint method)

```
checkpoint(interval: Duration): DStream[T]
```

You use `checkpoint(interval: Duration)` method to set up a periodic checkpointing every (`checkpoint`) `interval`.

You can only enable checkpointing and set the checkpoint interval before [StreamingContext is started](#) or [UnsupportedOperationException](#) is thrown as follows:

```
java.lang.UnsupportedOperationException: Cannot change checkpoint interval of an DStream  
at org.apache.spark.streaming.dstream.DStream.checkpoint(DStream.scala:177)  
... 43 elided
```

Internally, `checkpoint` method calls `persist` (that sets the default `MEMORY_ONLY_SER` storage level).

If checkpoint interval is set, the [checkpoint directory](#) is mandatory. Spark validates it when [StreamingContext starts](#) and throws a `IllegalArgumentException` exception if not set.

```
java.lang.IllegalArgumentException: requirement failed: The checkpoint directory has not
```

You can see the value of the checkpoint interval for a dstream in the logs when [it is validated](#):

```
INFO Checkpoint interval = [checkpointDuration]
```

Checkpointing

DStreams can [checkpoint](#) input data at specified time intervals.

The following settings are internal to a dstream and define how it checkpoints the input data if any.

- `mustCheckpoint` (default: `false`) is an internal private flag that marks a dstream as being checkpointed (`true`) or not (`false`). It is an implementation detail and the author of a `DStream` implementation sets it.

Refer to [Initializing DStreams \(initialize method\)](#) to learn how it is used to set the checkpoint interval, i.e. `checkpointDuration`.

- `checkpointDuration` is a configurable property that says how often a dstream checkpoints data. It is often called **checkpoint interval**. If not set explicitly, but the dstream is checkpointed, it will be while [initializing dstreams](#).
- `checkpointData` is an instance of [DStreamCheckpointData](#).
- `restoredFromCheckpointData` (default: `false`) is an internal flag to describe the initial state of a dstream, i.e.. whether (`true`) or not (`false`) it was started by restoring state from checkpoint.

Validating Setup at Startup (`validateAtStart` method)

Caution	FIXME Describe me!
---------	------------------------------------

Registering Output Streams (`register` method)

```
register(): DStream[T]
```

`DStream` by design has no notion of being an output stream. It is [DStreamGraph](#) to know and be able to differentiate between input and output streams.

`DStream` comes with internal `register` method that registers a `DStream` as an output stream.

The internal private `foreachRDD` method uses `register` to register output streams to `DStreamGraph`. Whenever called, it creates `ForEachDStream` and calls `register` upon it. That is how streams become output streams.

Generating Streaming Jobs (generateJob method)

```
generateJob(time: Time): Option[Job]
```

The internal `generateJob` method generates a streaming job for a batch `time` for a (output) `dstream`. It may or may not generate a streaming job for the requested batch `time`.

Note	It is called when <code>DStreamGraph</code> generates jobs for a batch time.
------	--

It computes an `RDD` for the batch and, if there is one, returns a `streaming job` for the batch `time` and a job function that will run a `Spark job` (with the generated `RDD` and the job function) when executed.

Note	The Spark job uses an empty function to calculate partitions of a <code>RDD</code> .
------	--

Caution	<code>FIXME</code> What happens when <code>SparkContext.runJob(rdd, emptyFunc)</code> is called with the empty function, i.e. <code>(iterator: Iterator[T]) => {}</code> ?
---------	---

Computing RDD for Batch (getOrCompute method)

The internal (`private final`) `getOrCompute(time: Time)` method returns an optional `RDD` for a batch (`time`).

It uses `generatedRDDs` to return the `RDD` if it has already been generated for the `time`. If not, it generates one by computing the input stream (using `compute(validTime: Time)` method).

If there was anything to process in the input stream, i.e. computing the input stream returned a `RDD`, the `RDD` is first `persisted` (only if `storageLevel` for the input stream is different from `StorageLevel.NONE`).

You should see the following DEBUG message in the logs:

```
DEBUG Persisting RDD [id] for time [time] to [storageLevel]
```

The generated RDD is [checkpointed](#) if `checkpointDuration` is defined and the time interval between current and `zero` times is a multiple of `checkpointDuration`.

You should see the following DEBUG message in the logs:

```
DEBUG Marking RDD [id] for time [time] for checkpointing
```

The generated RDD is saved in the [internal generatedRDDs registry](#).

Caching and Persisting

Caution	FIXME
---------	-----------------------

Checkpoint Cleanup

Caution	FIXME
---------	-----------------------

`restoreCheckpointData`

```
restoreCheckpointData(): Unit
```

`restoreCheckpointData` does its work only when the internal `transient restoredFromCheckpointData` flag is disabled (i.e. `false`) and is so initially.

Note	restoreCheckpointData method is called when DStreamGraph is requested to restore state of output dstreams .
------	---

If `restoredFromCheckpointData` is disabled, you should see the following INFO message in the logs:

```
INFO ...DStream: Restoring checkpoint data
```

[DStreamCheckpointData.restore\(\)](#) is executed. And then `restoreCheckpointData` method is executed for every dstream the current dstream depends on (see [DStream Contract](#)).

Once completed, the internal `restoredFromCheckpointData` flag is enabled (i.e. `true`) and you should see the following INFO message in the logs:

```
INFO Restored checkpoint data
```

Metadata Cleanup

Note It is called when [DStreamGraph](#) clears metadata for every output stream.

`clearMetadata(time: Time)` is called to remove old RDDs that have been generated so far (and collected in [generatedRDDs](#)). It is a sort of *garbage collector*.

When `clearMetadata(time: Time)` is called, it checks [spark.streaming.unpersist](#) flag (default enabled).

It collects generated RDDs (from [generatedRDDs](#)) that are older than [rememberDuration](#).

You should see the following DEBUG message in the logs:

```
DEBUG Clearing references to old RDDs: [[time] -> [rddId], ...]
```

Regardless of [spark.streaming.unpersist](#) flag, all the collected RDDs are removed from [generatedRDDs](#).

When [spark.streaming.unpersist](#) flag is set (it is by default), you should see the following DEBUG message in the logs:

```
DEBUG Unpersisting old RDDs: [id1, id2, ...]
```

For every RDD in the list, it [unpersist them \(without blocking\)](#) one by one and explicitly [removes blocks for BlockRDDs](#). You should see the following INFO message in the logs:

```
INFO Removing blocks of RDD [blockRDD] of time [time]
```

After RDDs have been removed from [generatedRDDs](#) (and perhaps unpersisted), you should see the following DEBUG message in the logs:

```
DEBUG Cleared [size] RDDs that were older than [time]: [time1, time2, ...]
```

The stream passes the call to clear metadata to its [dependencies](#).

updateCheckpointData

```
updateCheckpointData(currentTime: Time): Unit
```

Note

It is called when [DStreamGraph](#) is requested to do `updateCheckpointData` itself.

When `updateCheckpointData` is called, you should see the following DEBUG message in the logs:

```
DEBUG Updating checkpoint data for time [currentTime] ms
```

It then executes [DStreamCheckpointData.update\(currentTime\)](#) and calls `updateCheckpointData` method on each dstream the dstream depends on.

When `updateCheckpointData` finishes, you should see the following DEBUG message in the logs:

```
DEBUG Updated checkpoint data for time [currentTime]: [checkpointData]
```

Internal Registries

`DStream` implementations maintain the following internal properties:

- `storageLevel` (default: `NONE`) is the [StorageLevel](#) of the RDDs in the `DStream`.
- `restoredFromCheckpointData` is a flag to inform whether it was restored from checkpoint.
- `graph` is the reference to [DStreamGraph](#).

Input DStreams

Input DStreams in Spark Streaming are the way to ingest data from external data sources. They are represented as `InputDStream` abstract class.

`InputDStream` is the abstract base class for all input **DStreams**. It provides two abstract methods `start()` and `stop()` to start and stop ingesting data, respectively.

When instantiated, an `InputDStream` registers itself as an input stream (using `DStreamGraph.addInputStream`) and, while doing so, is told about its owning `DStreamGraph`.

It asks for its own unique identifier using `StreamingContext.getNewInputStreamId()`.

Note	It is <code>StreamingContext</code> to maintain the identifiers and how many input streams have already been created.
------	---

`InputDStream` has a human-readable `name` that is made up from a nicely-formatted part based on the class name and the unique identifier.

Tip	Name your custom <code>InputDStream</code> using the CamelCase notation with the suffix <code>InputDStream</code> , e.g. <code>MyCustomInputDStream</code> .
-----	--

- `slideDuration` calls `DStreamGraph.batchDuration`.
- `dependencies` method returns an empty collection.

Note	<code>compute(validTime: Time): Option[RDD[T]]</code> abstract method from <code>DStream</code> abstract class is not defined.
------	--

Custom implementations of `InputDStream` can override (and actually provide!) the optional `RateController`. It is undefined by default.

Custom Input DStream

Here is an example of a custom input dstream that produces an RDD out of the input collection of elements (of type `T`).

Note	It is similar to <code>ConstantInputDStreams</code> , but this custom implementation does not use an external RDD, but generates its own.
------	---

```

package pl.japila.spark.streaming

import org.apache.spark.rdd.RDD
import org.apache.spark.streaming.{ Time, StreamingContext }
import org.apache.spark.streaming.dstream.InputDStream

import scala.reflect.ClassTag

class CustomInputDStream[T: ClassTag](ssc: StreamingContext, seq: Seq[T])
  extends InputDStream[T](ssc) {
  override def compute(validTime: Time): Option[RDD[T]] = {
    Some(ssc.sparkContext.parallelize(seq))
  }
  override def start(): Unit = {}
  override def stop(): Unit = {}
}

```

Its use could be as simple as follows (compare it to the [example of ConstantInputDStreams](#)):

```

// sc is the SparkContext instance
import org.apache.spark.streaming.Seconds
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

// Create the collection of numbers
val nums = 0 to 9

// Create constant input dstream with the RDD
import pl.japila.spark.streaming.CustomInputDStream
val cis = new CustomInputDStream(ssc, nums)

// Sample stream computation
cis.print

```

Tip

Copy and paste it to `spark-shell` to run it.

ReceiverInputDStreams - Input Streams with Receivers

Receiver Input Streams (`ReceiverInputDStreams`) are specialized [input streams](#) that use [receivers](#) to receive data (and hence the name which stands for an `InputDStream` with a receiver).

Note

Receiver input streams run receivers as long-running tasks that occupy a core per stream.

`ReceiverInputDStream` abstract class defines the following abstract method that custom implementations use to create receivers:

```
def getReceiver(): Receiver[T]
```

The receiver is then sent to and run on workers (when [ReceiverTracker is started](#)).

Note

A fine example of a very minimalistic yet still useful implementation of `ReceiverInputDStream` class is the pluggable input stream `org.apache.spark.streaming.dstream.PluggableInputDStream` ([the sources on GitHub](#)). It requires a `Receiver` to be given (by a developer) and simply returns it in `getReceiver`.
`PluggableInputDStream` is used by [StreamingContext.receiverStream\(\)](#) method.

`ReceiverInputDStream` uses `ReceiverRateController` when `spark.streaming.backpressure.enabled` is enabled.

Note

Both, `start()` and `stop` methods are implemented in `ReceiverInputDStream`, but do nothing. `ReceiverInputDStream` management is left to [ReceiverTracker](#). Read [ReceiverTrackerEndpoint.startReceiver](#) for more details.

The source code of `ReceiverInputDStream` is [here at GitHub](#).

Generate RDDs (using compute method)

The abstract `compute(validTime: Time): Option[RDD[T]]` method (from [DStream](#)) uses [start time of DStreamGraph](#), i.e. the start time of `StreamingContext`, to check whether `validTime` input parameter is really valid.

If the time to generate RDDs (`validTime`) is earlier than the start time of StreamingContext, an empty `BlockRDD` is generated.

Otherwise, `ReceiverTracker` is requested for all the blocks that have been allocated to this stream for this batch (using `ReceiverTracker.getBlocksOfBatch`).

The number of records received for the batch for the input stream (as `StreamInputInfo` aka **input blocks information**) is registered to `InputInfoTracker` (using `InputInfoTracker.reportInfo`).

If all `BlockIds` have `WriteAheadLogRecordHandle`, a `WriteAheadLogBackedBlockRDD` is generated. Otherwise, a `BlockRDD` is.

Back Pressure

Caution	FIXME
---------	-----------------------

[Back pressure](#) for input streams with receivers can be configured using `spark.streaming.backpressure.enabled` setting.

Note	Back pressure is disabled by default.
------	---------------------------------------

ConstantInputDStreams

`ConstantInputDStream` is an [input stream](#) that always returns the same mandatory input `RDD` at every batch `time`.

```
ConstantInputDStream[T](_ssc: StreamingContext, rdd: RDD[T])
```

`ConstantInputDStream` `dstream` belongs to `org.apache.spark.streaming.dstream` package.

The `compute` method returns the input `rdd`.

Note	<code>rdd</code> input parameter is mandatory.
------	--

The mandatory `start` and `stop` methods do nothing.

Example

```
val sc = new SparkContext("local[*]", "Constant Input DStream Demo", new SparkConf())
import org.apache.spark.streaming.{ StreamingContext, Seconds }
val ssc = new StreamingContext(sc, batchDuration = Seconds(5))

// Create the RDD
val rdd = sc.parallelize(0 to 9)

// Create constant input dstream with the RDD
import org.apache.spark.streaming.dstream.ConstantInputDStream
val cis = new ConstantInputDStream(ssc, rdd)

// Sample stream computation
cis.print
```

ForEachDStreams

`ForEachDStream` is an internal `DStream` with dependency on the `parent` stream with the exact same `slideDuration`.

The `compute` method returns no RDD.

When `generateJob` is called, it returns a streaming job for a batch when `parent` stream does. And if so, it uses the "foreach" function (given as `foreachFunc`) to work on the RDDs generated.

Note

Although it may seem that `ForEachDStreams` are by design output streams they are not. You have to use `DStreamGraph.addOutputStream` to register a stream as output.

You use `stream operators` that do the registration as part of their operation, like `print`.

WindowedDStreams

`WindowedDStream` (aka **windowed stream**) is an internal `DStream` with dependency on the parent `stream`.

Note	It is the result of window operators .
------	--

`windowDuration` has to be a multiple of the parent stream's slide duration.

`slideDuration` has to be a multiple of the parent stream's slide duration.

Note	When <code>windowDuration</code> or <code>slideDuration</code> are <i>not</i> multiples of the parent stream's slide duration, <code>Exception</code> is thrown.
------	--

The parent's RDDs are automatically changed to be [persisted](#) at `StorageLevel.MEMORY_ONLY_SER` level (since they need to last longer than the parent's slide duration for this stream to generate its own RDDs).

Obviously, slide duration of the stream is given explicitly (and must be a multiple of the parent's slide duration).

`parentRememberDuration` is extended to cover the parent's `rememberDuration` and the window duration.

`compute` method always returns a RDD, either `PartitionerAwareUnionRDD` or `UnionRDD`, depending on the number of the [partitioners](#) defined by the RDDs in the window. It uses `slice` operator on the parent stream (using the slice window of `[now - windowDuration + parent.slideDuration, now]`).

If only one partitioner is used across the RDDs in window, `PartitionerAwareUnionRDD` is created and you should see the following DEBUG message in the logs:

```
DEBUG WindowedDStream: Using partition aware union for windowing at [time]
```

Otherwise, when there are multiple different partitioners in use, `UnionRDD` is created and you should see the following DEBUG message in the logs:

```
DEBUG WindowedDStream: Using normal union for windowing at [time]
```

Enable `DEBUG` logging level for
`org.apache.spark.streaming.dstream.WindowedDStream` logger to see what happens
inside `WindowedDStream`.

Tip Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.streaming.dstream.WindowedDStream=DEBUG
```

MapWithStateDStream

`MapWithStateDStream` is the result of `mapWithState` stateful operator.

It extends [DStream Contract](#) with the following additional method:

```
def stateSchemas(): DStream[(KeyType, StateType)]
```

Note	<code>MapWithStateDStream</code> is a Scala <code>sealed abstract class</code> (and hence all the available implementations are in the source file).
------	--

Note	<code>MapWithStateDStreamImpl</code> is the only implementation of <code>MapWithStateDStream</code> (see below in this document for more coverage).
------	---

MapWithStateDStreamImpl

`MapWithStateDStreamImpl` is an internal [DStream](#) with dependency on the parent `dataStream` key-value dstream. It uses a custom internal dstream called `internalStream` (of type [InternalMapWithStateDStream](#)).

`slideDuration` is exactly the slide duration of the internal stream `internalStream`.

`dependencies` returns a single-element collection with the internal stream `internalStream`.

The `compute` method may or may not return a `RDD[MappedType]` by `getOrCompute` on the internal stream and...TK

Caution	FIXME
---------	-----------------------

InternalMapWithStateDStream

`InternalMapWithStateDStream` is an internal dstream to support [MapWithStateDStreamImpl](#) and uses `dataStream` (as parent of type `DStream[(K, V)]`) as well as `StateSpecImpl[K, V, S, E]` (as spec).

It is a `DStream[MapWithStateRDDRecord[K, S, E]]`.

It uses `StorageLevel.MEMORY_ONLY` storage level by default.

It uses the `StateSpec`'s partitioner or [HashPartitioner](#) (with `SparkContext`'s `defaultParallelism`).

`slideDuration` is the slide duration of `parent`.

`dependencies` is a single-element collection with the `parent` stream.

It forces [checkpointing](#) (i.e. `mustCheckpoint` flag is enabled).

When initialized, if [checkpoint interval](#) is *not* set, it sets it as ten times longer than the slide duration of the `parent` stream (the multiplier is not configurable and always `10`).

Computing a `RDD[MapWithStateRDDRecord[K, S, E]]` (i.e. `compute` method) first looks up a previous RDD for the last `slideDuration`.

If the RDD is found, it is returned as is given the partitioners of the RDD and the stream are equal. Otherwise, when the partitioners are different, the RDD is "repartitioned" using

`MapWithStateRDD.createFromRDD`.

Caution

[FIXME](#) `MapWithStateRDD.createFromRDD`

StateDStream

`StateDStream` is the specialized `DStream` that is the result of `updateStateByKey` stateful operator. It is a wrapper around a `parent` key-value pair `dstream` to build stateful pipeline (by means of `updateStateByKey` operator) and as a stateful `dstream` enables `checkpointing` (and hence requires some additional setup).

It uses a `parent` key-value pair `dstream`, `updateFunc` update state function, a `partitioner`, a flag whether or not to `preservePartitioning` and an optional key-value pair `initialRDD`.

It works with `MEMORY_ONLY_SER` storage level enabled.

The only `dependency` of `StateDStream` is the input `parent` key-value pair `dstream`.

The `slide duration` is exactly the same as that in `parent`.

It forces `checkpointing` regardless of the current `dstream` configuration, i.e. the internal `mustCheckpoint` is enabled.

When requested to `compute a RDD` it first attempts to get the **state RDD** for the previous batch (using `DStream.getOrCompute`). If there is one, `parent` stream is requested for a `RDD` for the current batch (using `DStream.getOrCompute`). If `parent` has computed one, `computeUsingPreviousRDD(parentRDD, prevStateRDD)` is called.

Caution	FIXME When could <code>getOrCompute</code> not return an <code>RDD</code> ? How does this apply to the <code>StateDStream</code> ? What about the parent's <code>getOrCompute</code> ?
---------	--

If however `parent` has not generated a `RDD` for the current batch but the state `RDD` existed, `updateFn` is called for every key of the state `RDD` to generate a new state per partition (using `RDD.mapPartitions`)

Note	No input data for already-running input stream triggers (re)computation of the state <code>RDD</code> (per partition).
------	--

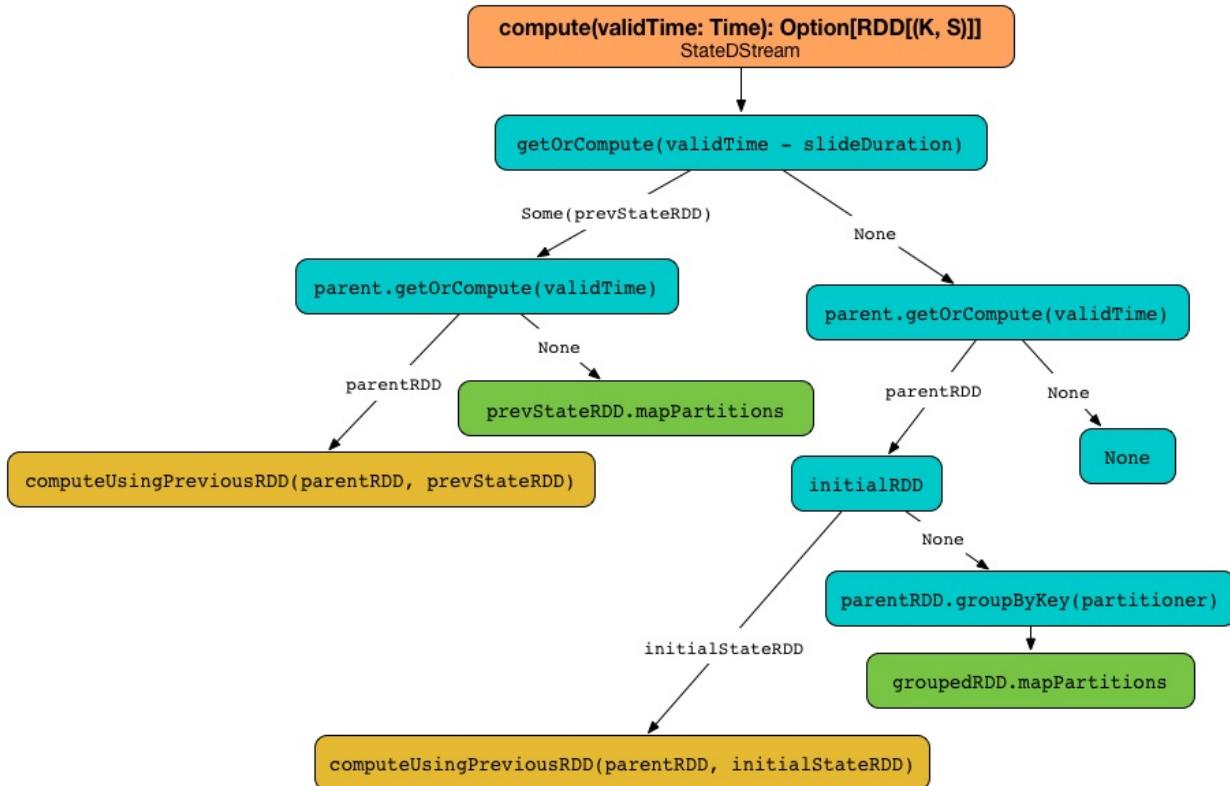


Figure 1. Computing stateful RDDs (StateDStream.compute)

If the state RDD has been found, which means that this is the first input data batch, `parent` stream is requested to `getOrCompute` the RDD for the current batch.

Otherwise, when no state RDD exists, `parent` stream is requested for a RDD for the current batch (using `DStream.getOrCompute`) and when no RDD was generated for the batch, no computation is triggered.

Note

When the stream processing starts, i.e. no state RDD exists, and there is no input data received, no computation is triggered.

Given no state RDD and with `parent` RDD computed, when `initialRDD` is `NONE`, the input data batch (as `parent` RDD) is grouped by key (using `groupByKey` with `partitioner`) and then the update state function `updateFunc` is applied to the partitioned input data (using `mapPartitions`) with `None` state. Otherwise, `computeUsingPreviousRDD(parentRDD, initialStateRDD)` is called.

updateFunc - State Update Function

The signature of `updateFunc` is as follows:

```
updateFunc: (Iterator[(K, Seq[V], Option[S])]) => Iterator[(K, S)]
```

It should be read as given a collection of triples of a key, new records for the key, and the current state for the key, generate a collection of keys and their state.

computeUsingPreviousRDD

```
computeUsingPreviousRDD(parentRDD: RDD[(K, V)], prevStateRDD: RDD[(K, S)]): Option[RDD[(K,
```

The `computeUsingPreviousRDD` method uses `cogroup` and `mapPartitions` to build the final state RDD.

Note

Regardless of the return type `Option[RDD[(K, S)]]` that really allows no state, it will always return *some* state.

It first performs `cogroup` of `parentRDD` and `prevStateRDD` using the constructor's `partitioner` so it has a pair of iterators of elements of each RDDs per every key.

Note

It is acceptable to end up with keys that have no new records per batch, but these keys do have a state (since they were received previously when no state might have been built yet).

Note

The signature of `cogroup` is as follows and applies to key-value pair RDDs, i.e. `RDD[(K, V)]`.

```
cogroup[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Iterable[V],
```

It defines an internal update function `finalFunc` that maps over the collection of all the keys, new records per key, and at-most-one-element state per key to build new iterator that ensures that:

1. a state per key exists (it is `None` or the state built so far)
2. the *lazy* iterable of new records is transformed into an *eager* sequence.

Caution

FIXME Why is the transformation from an Iterable into a Seq so important? Why could not the constructor's `updateFunc` accept the former?

With every triple per every key, the internal update function calls the constructor's `updateFunc`.

The state RDD is a cogrouped RDD (on `parentRDD` and `prevStateRDD` using the constructor's `partitioner`) with every element per partition mapped over using the internal update function `finalFunc` and the constructor's `preservePartitioning` (through `mapPartitions`).

Caution

FIXME Why is `preservePartitioning` important? What happens when `mapPartitions` does not preserve partitioning (which by default it does **not!**)

TransformedDStream

`TransformedDStream` is the specialized `DStream` that is the result of `transform` operator.

It is constructed with a collection of `parents` `dstreams` and `transformFunc` `transform` function.

Note	When created, it asserts that the input collection of <code>dstreams</code> use the same <code>StreamingContext</code> and slide interval.
------	--

Note	It is acceptable to have more than one dependent <code>dstream</code> .
------	---

The `dependencies` is the input collection of `dstreams`.

The `slide interval` is exactly the same as that in the first `dstream` in `parents`.

When requested to `compute a RDD`, it goes over every `dstream` in `parents` and asks to `getOrCompute` a `RDD`.

Note	It may throw a <code>SparkException</code> when a <code>dstream</code> does not compute a <code>RDD</code> for a batch.
------	---

Caution	<code>FIXME</code> Prepare an example to face the exception.
---------	--

It then calls `transformFunc` with the collection of `RDDs`.

If the transform function returns `null` a `SparkException` is thrown:

```
org.apache.spark.SparkException: Transform function must not return null
at org.apache.spark.streaming.dstream.TransformedDStream.co
```

The result of `transformFunc` is returned.

Receivers

Receivers run on [workers](#) to receive external data. They are created and belong to [ReceiverInputDStreams](#).

Note

[ReceiverTracker](#) launches a receiver on a worker.

It is represented by [abstract class Receiver](#) that is parameterized by [StorageLevel](#).

The abstract `Receiver` class requires the following methods to be implemented:

- `onStart()` that starts the receiver when the application starts.
- `onStop()` that stops the receiver.

A receiver is identified by the unique identifier `Receiver.streamId` (that corresponds to the unique identifier of the receiver input stream it is associated with).

Note

[StorageLevel](#) of a receiver is used to instantiate [ReceivedBlockHandler](#) in [ReceiverSupervisorImpl](#).

A receiver uses `store` methods to store received data as data blocks into Spark's memory.

Note

Receivers must have [ReceiverSupervisors](#) attached before they can be started since `store` and management methods simply pass calls on to the respective methods in the [ReceiverSupervisor](#).

A receiver can be in one of the three states: `Initialized`, `Started`, and `Stopped`.

ReceiverTracker

Introduction

`ReceiverTracker` manages execution of all [Receivers](#).

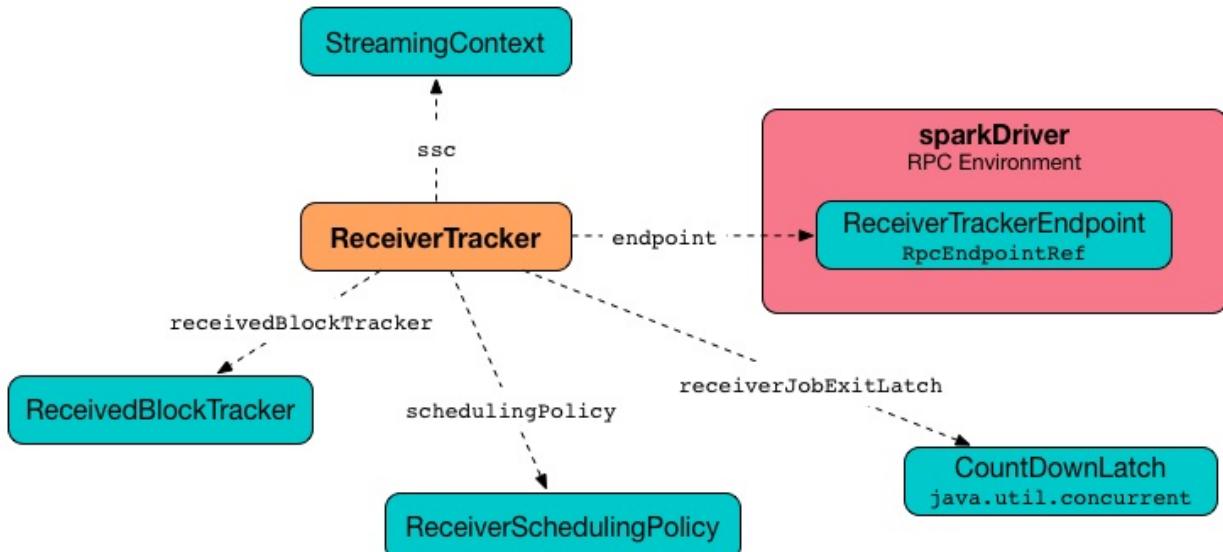


Figure 1. ReceiverTracker and Dependencies

It uses [RPC environment](#) for communication with [ReceiverSupervisors](#).

Note

`ReceiverTracker` is started when [JobScheduler](#) starts.

It can only be started once and only when at least one input receiver has been registered.

`ReceiverTracker` can be in one of the following states:

- `Initialized` - it is in the state after having been instantiated.
- `Started` -
- `Stopping`
- `Stopped`

Starting ReceiverTracker (start method)

Note

You can only start `ReceiverTracker` once and multiple attempts lead to throwing `SparkException` exception.

Note

Starting `ReceiverTracker` when no [ReceiverInputDStream](#) has registered does nothing.

When `ReceiverTracker` starts, it first sets [ReceiverTracker RPC endpoint](#) up.

It then launches receivers, i.e. it collects receivers for all registered `ReceiverDStream` and posts them as [StartAllReceivers](#) to [ReceiverTracker RPC endpoint](#).

In the meantime, receivers have their ids assigned that correspond to the unique identifier of their `ReceiverDStream`.

You should see the following INFO message in the logs:

```
INFO ReceiverTracker: Starting [receivers.length] receivers
```

A successful startup of `ReceiverTracker` finishes with the following INFO message in the logs:

```
INFO ReceiverTracker: ReceiverTracker started
```

`ReceiverTracker` enters `Started` state.

Cleanup Old Blocks And Batches (`cleanupOldBlocksAndBatches` method)

Caution	FIXME
---------	-----------------------

hasUnallocatedBlocks

Caution	FIXME
---------	-----------------------

ReceiverTracker RPC endpoint

Caution	FIXME
---------	-----------------------

StartAllReceivers

`StartAllReceivers(receivers)` is a local message sent by `ReceiverTracker` when [it starts](#) (using `ReceiverTracker.launchReceivers()`).

It schedules receivers (using `ReceiverSchedulingPolicy.scheduleReceivers(receivers, getExecutors)`).

Caution	FIXME What does <code>ReceiverSchedulingPolicy.scheduleReceivers(receivers, getExecutors)</code> do?
---------	--

It does *some* bookkeeping.

Caution	FIXME What is <i>the</i> bookkeeping?
---------	---

It finally starts every receiver (using the helper method [ReceiverTrackerEndpoint.startReceiver](#)).

ReceiverTrackerEndpoint.startReceiver

Caution	FIXME When is the method called?
---------	--

`ReceiverTrackerEndpoint.startReceiver(receiver: Receiver[_], scheduledLocations: Seq[TaskLocation])` starts a receiver [Receiver](#) at the given `Seq[TaskLocation]` locations.

Caution	FIXME When the scaladoc says " <i>along with the scheduled executors</i> ", does it mean that the executors are already started and waiting for the receiver?!
---------	--

It defines an internal function (`startReceiverFunc`) to start `receiver` on a worker (in Spark cluster).

Namely, the internal `startReceiverFunc` function checks that the task attempt is `0`.

Caution	FIXME When could <code>TaskContext.get().attemptNumber()</code> be different than <code>0</code> ?
---------	--

It then starts a [ReceiverSupervisor](#) for `receiver` and keeps awaiting termination, i.e. once the task is run it does so until a *termination message* comes from *some* other external source). The task is a long-running task for `receiver`.

Caution	FIXME When does <code>supervisor.awaitTermination()</code> finish?
---------	--

Having the internal function, it creates `receiverRDD` - an instance of `RDD[Receiver[_]]` - that uses [SparkContext.makeRDD](#) with a one-element collection with the only element being `receiver`. When the collection of `TaskLocation` is empty, it uses exactly one partition. Otherwise, it distributes the one-element collection across the nodes (and potentially even executors) for `receiver`. The RDD has the name `Receiver [receiverId]`.

The Spark job's description is set to `Streaming job running receiver [receiverId]`.

Caution	FIXME What does <code>sparkContext.setJobDescription</code> actually do and how does this influence Spark jobs? It uses <code>ThreadLocal</code> so it assumes that a single thread will do a job?
---------	--

Having done so, it submits a job (using [SparkContext.submitJob](#)) on the instance of `RDD[Receiver[_]]` with the function `startReceiverFunc` that runs `receiver`. It has [SimpleFutureAction](#) to monitor `receiver`.

Note	The method demonstrates how you could use Spark Core as the distributed computation platform to launch <code>any</code> process on clusters and let Spark handle the distribution. <i>Very clever indeed!</i>
------	--

When it completes (successfully or not), `onReceiverJobFinish(receiverId)` is called, but only for cases when the tracker is fully up and running, i.e. started. When the tracker is being stopped or has already stopped, the following INFO message appears in the logs:

```
INFO Restarting Receiver [receiverId]
```

And a `RestartReceiver(receiver)` message is sent.

When there was a failure submitting the job, you should also see the ERROR message in the logs:

```
ERROR Receiver has been stopped. Try to restart it.
```

Ultimately, right before the method exits, the following INFO message appears in the logs:

```
INFO Receiver [receiver.streamId] started
```

StopAllReceivers

Caution	FIXME
---------	-----------------------

AllReceiverIds

Caution	FIXME
---------	-----------------------

Stopping ReceiverTracker (stop method)

`ReceiverTracker.stop(graceful: Boolean)` stops `ReceiverTracker` only when it is in `started` state. Otherwise, it does nothing and simply exits.

Note	The <code>stop</code> method is called while JobScheduler is being stopped.
------	---

The state of `ReceiverTracker` is marked `Stopping`.

It then sends the stop signal to all the receivers (i.e. posts [StopAllReceivers](#) to [ReceiverTracker RPC endpoint](#)) and waits **10 seconds** for all the receivers to quit gracefully (unless `graceful` flag is set).

Note	The 10-second wait time for graceful quit is not configurable.
------	--

You should see the following INFO messages if the `graceful` flag is enabled which means that the receivers quit in a graceful manner:

```
INFO ReceiverTracker: Waiting for receiver job to terminate gracefully
INFO ReceiverTracker: Waited for receiver job to terminate gracefully
```

It then checks whether all the receivers have been deregistered or not by posting [AllReceiverIds](#) to [ReceiverTracker RPC endpoint](#).

You should see the following INFO message in the logs if they have:

```
INFO ReceiverTracker: All of the receivers have deregistered successfully
```

Otherwise, when there were receivers not having been deregistered properly, the following WARN message appears in the logs:

```
WARN ReceiverTracker: Not all of the receivers have deregistered, [receivers]
```

It stops [ReceiverTracker RPC endpoint](#) as well as [ReceivedBlockTracker](#).

You should see the following INFO message in the logs:

```
INFO ReceiverTracker: ReceiverTracker stopped
```

The state of `ReceiverTracker` is marked `Stopped`.

Allocating Blocks To Batch (`allocateBlocksToBatch` method)

```
allocateBlocksToBatch(batchTime: Time): Unit
```

`allocateBlocksToBatch` simply passes all the calls on to [ReceivedBlockTracker.allocateBlocksToBatch](#), but only when there [are receiver input streams](#) registered (in `receiverInputStreams` internal registry).

Note	When there are no <code>receiver input streams</code> in use, the method does nothing.
------	--

ReceivedBlockTracker

Caution	FIXME
---------	-----------------------

You should see the following INFO message in the logs when `cleanupOldBatches` is called:

```
INFO ReceivedBlockTracker: Deleting batches [timesToCleanup]
```

allocateBlocksToBatch Method

```
allocateBlocksToBatch(batchTime: Time): Unit
```

`allocateBlocksToBatch` starts by checking whether the internal `lastAllocatedBatchTime` is younger than (after) the current batch time `batchTime`.

If so, it grabs all unallocated blocks per stream (using `getReceivedBlockQueue` method) and creates a map of stream ids and sequences of their `ReceivedBlockInfo`. It then writes the received blocks to **write-ahead log (WAL)** (using `writeToLog` method).

`allocateBlocksToBatch` stores the allocated blocks with the current batch time in `timeToAllocatedBlocks` internal registry. It also sets `lastAllocatedBatchTime` to the current batch time `batchTime`.

If there has been an error while writing to WAL or the batch time is older than `lastAllocatedBatchTime`, you should see the following INFO message in the logs:

```
INFO Possibly processed batch [batchTime] needs to be processed again in WAL recovery
```

ReceiverSupervisors

`ReceiverSupervisor` is an (abstract) handler object that is responsible for supervising a `receiver` (that runs on the worker). It assumes that implementations offer concrete methods to push received data to Spark.

Note

`Receiver`'s `store` methods pass calls to respective `push` methods of `ReceiverSupervisors`.

Note

`ReceiverTracker` starts a `ReceiverSupervisor` per receiver.

`ReceiverSupervisor` can be started and stopped. When a supervisor is started, it calls (empty by default) `onStart()` and `startReceiver()` afterwards.

It attaches itself to the receiver it is a supervisor of (using `Receiver.attachSupervisor`). That is how a receiver knows about its supervisor (and can hence offer the `store` and management methods).

ReceiverSupervisor Contract

`ReceiverSupervisor` is a `private[streaming]` abstract class that assumes that concrete implementations offer the following **push methods**:

- `pushBytes`
- `pushIterator`
- `pushArrayBuffer`

There are the other methods required:

- `createBlockGenerator`
- `reportError`
- `onReceiverStart`

Starting Receivers

`startReceiver()` calls (abstract) `onReceiverStart()`. When `true` (it is unknown at this point to know when it is `true` or `false` since it is an abstract method - see `ReceiverSupervisorImpl.onReceiverStart` for the default implementation), it prints the following INFO message to the logs:

```
INFO Starting receiver
```

The receiver's `onStart()` is called and another INFO message appears in the logs:

```
INFO Called receiver onStart
```

If however `onReceiverStart()` returns `false`, the supervisor stops (using `stop`).

Stopping Receivers

`stop` method is called with a message and an optional cause of the stop (called `error`). It calls `stopReceiver` method that prints the INFO message and checks the state of the receiver to react appropriately.

When the receiver is in `started` state, `stopReceiver` calls `Receiver.onStop()`, prints the following INFO message, and `onReceiverStop(message, error)`.

```
INFO Called receiver onStop
```

Restarting Receivers

A `ReceiverSupervisor` uses `spark.streaming.receiverRestartDelay` to restart the receiver with delay.

Note	Receivers can request to be restarted using <code>restart</code> methods.
------	---

When requested to restart a receiver, it uses a separate thread to perform it asynchronously. It prints the WARNING message to the logs:

```
WARNING Restarting receiver with delay [delay] ms: [message]
```

It then stops the receiver, sleeps for `delay` milliseconds and starts the receiver (using `startReceiver()`).

You should see the following messages in the logs:

```
DEBUG Sleeping for [delay]
INFO Starting receiver again
INFO Receiver started again
```

Caution	FIXME What is a backend data store?
---------	---

Awaiting Termination

`awaitTermination` method blocks the current thread to wait for the receiver to be stopped.

Note

ReceiverTracker uses `awaitTermination` to wait for receivers to stop (see [StartAllReceivers](#)).

When called, you should see the following INFO message in the logs:

```
INFO Waiting for receiver to be stopped
```

If a receiver has terminated successfully, you should see the following INFO message in the logs:

```
INFO Stopped receiver without error
```

Otherwise, you should see the ERROR message in the logs:

```
ERROR Stopped receiver with error: [stoppingError]
```

`stoppingError` is the exception associated with the stopping of the receiver and is rethrown.

Note

Internally, ReceiverSupervisor uses [java.util.concurrent.CountDownLatch](#) with count `1` to await the termination.

Internals - How to count stopLatch down

`stopLatch` is decremented when ReceiverSupervisor's `stop` is called which is in the following cases:

- When a receiver itself calls `stop(message: String)` or `stop(message: String, error: Throwable)`
- When [ReceiverSupervisor.onReceiverStart\(\)](#) returns `false` or `NonFatal` (less severe) exception is thrown in `ReceiverSupervisor.startReceiver`.
- When [ReceiverTracker.stop](#) is called that posts `StopAllReceivers` message to `ReceiverTrackerEndpoint`. It in turn sends `StopReceiver` to the `ReceiverSupervisorImpl` for every `ReceiverSupervisor` that calls `ReceiverSupervisorImpl.stop`.

	FIXME Prepare exercises
Caution	<ul style="list-style-type: none"> for a receiver to call <code>stop(message: String)</code> when a custom "TERMINATE" message arrives send <code>StopReceiver</code> to a <code>ReceiverTracker</code>

ReceiverSupervisorImpl

`ReceiverSupervisorImpl` is the implementation of [ReceiverSupervisor contract](#).

Note	A dedicated <code>ReceiverSupervisorImpl</code> is started for every receiver when ReceiverTracker starts . See ReceiverTrackerEndpoint.startReceiver .
------	---

It communicates with [ReceiverTracker](#) that runs on the driver (by posting messages using the [ReceiverTracker RPC endpoint](#)).

	Enable <code>DEBUG</code> logging level for <code>org.apache.spark.streaming.receiver.ReceiverSupervisorImpl</code> logger to see what happens in <code>ReceiverSupervisorImpl</code> .
Tip	Add the following line to <code>conf/log4j.properties</code> : <pre>log4j.logger.org.apache.spark.streaming.receiver.ReceiverSupervisorImpl=DEBUG</pre>

push Methods

[push methods](#), i.e. `pushArrayBuffer` , `pushIterator` , and `pushBytes` solely pass calls on to `ReceiverSupervisorImpl.pushAndReportBlock`.

ReceiverSupervisorImpl.onReceiverStart

`ReceiverSupervisorImpl.onReceiverStart` sends a blocking `RegisterReceiver` message to [ReceiverTracker](#) that responds with a boolean value.

Current Rate Limit

`getCurrentRateLimit` controls the current rate limit. It asks the `BlockGenerator` for the value (using `getCurrentLimit`).

ReceivedBlockHandler

`ReceiverSupervisorImpl` uses the internal field `receivedBlockHandler` for `ReceivedBlockHandler` to use.

It defaults to `BlockManagerBasedBlockHandler`, but could use `WriteAheadLogBasedBlockHandler` instead when `spark.streaming.receiver.writeAheadLog.enable` is `true`.

It uses `ReceivedBlockHandler` to `storeBlock` (see `ReceivedBlockHandler Contract` for more coverage and `ReceiverSupervisorImpl.pushAndReportBlock` in this document).

ReceiverSupervisorImpl.pushAndReportBlock

`ReceiverSupervisorImpl.pushAndReportBlock(receivedBlock: ReceivedBlock, metadataOption: Option[Any], blockIdOption: Option[StreamBlockId])` stores `receivedBlock` using `ReceivedBlockHandler.storeBlock` and reports it to the driver.

Note

`ReceiverSupervisorImpl.pushAndReportBlock` is only used by the `push methods`, i.e. `pushArrayBuffer`, `pushIterator`, and `pushBytes`. Calling the method is actually all they do.

When it calls `ReceivedBlockHandler.storeBlock`, you should see the following DEBUG message in the logs:

```
DEBUG Pushed block [blockId] in [time] ms
```

It then sends `AddBlock` (with `ReceivedBlockInfo` for `streamId`, `BlockStoreResult.numRecords`, `metadataOption`, and the result of `ReceivedBlockHandler.storeBlock`) to `ReceiverTracker RPC endpoint` (that runs on the driver).

When a response comes, you should see the following DEBUG message in the logs:

```
DEBUG Reported block [blockId]
```

ReceivedBlockHandlers

`ReceivedBlockHandler` represents how to handle the storage of blocks received by [receivers](#).

Note	It is used by ReceiverSupervisorImpl (as the internal <code>receivedBlockHandler</code>).
------	--

ReceivedBlockHandler Contract

`ReceivedBlockHandler` is a `private[streaming] trait`. It comes with two methods:

- `storeBlock(blockId: StreamBlockId, receivedBlock: ReceivedBlock): ReceivedBlockStoreResult` to store a received block as `blockId`.
- `cleanupOldBlocks(threshTime: Long)` to clean up blocks older than `threshTime`.

Note	<code>cleanupOldBlocks</code> implies that there is a relation between blocks and the time they arrived.
------	--

Implementations of ReceivedBlockHandler Contract

There are two implementations of `ReceivedBlockHandler` contract:

- `BlockManagerBasedBlockHandler` that stores received blocks in Spark's [BlockManager](#) with the specified [StorageLevel](#).

Read [BlockManagerBasedBlockHandler](#) in this document.

- `WriteAheadLogBasedBlockHandler` that stores received blocks in a write ahead log and Spark's [BlockManager](#). It is a more advanced option comparing to a simpler [BlockManagerBasedBlockHandler](#).

Read [WriteAheadLogBasedBlockHandler](#) in this document.

BlockManagerBasedBlockHandler

`BlockManagerBasedBlockHandler` is the default `ReceivedBlockHandler` in Spark Streaming.

It uses [BlockManager](#) and a receiver's [StorageLevel](#).

`cleanupOldBlocks` is not used as blocks are cleared by *some other means* ([FIXME](#))

`putResult` returns `BlockManagerBasedStoreResult`. It uses `BlockManager.putIterator` to store `ReceivedBlock`.

WriteAheadLogBasedBlockHandler

`WriteAheadLogBasedBlockHandler` is used when
`spark.streaming.receiver.writeAheadLog.enable` is `true`.

It uses [BlockManager](#), a receiver's `streamId` and [StorageLevel](#), [SparkConf](#) for additional configuration settings, Hadoop Configuration, the checkpoint directory.

Ingesting Data from Apache Kafka

Spark Streaming comes with two ways of ingesting data from [Apache Kafka](#):

- Using receivers
- [With no receivers](#)

There is yet another "middle-ground" approach (so-called unofficial since it is not available by default in Spark Streaming):

- ...

Data Ingestion with no Receivers

In this approach, **with no receivers**, you find two modes of ingesting data from Kafka:

- **Streaming mode** using `KafkaUtils.createDirectStream` that creates an [input stream](#) that directly pulls messages from Kafka brokers (with no receivers). See [Streaming mode](#) section.
- **Non-streaming mode** using `KafkaUtils.createRDD` that just creates a [KafkaRDD](#) of key-value pairs, i.e. `RDD[(K, V)]`.

Streaming mode

You create [DirectKafkaInputDStream](#) using `KafkaUtils.createDirectStream`.

Note	Define the types of keys and values in <code>KafkaUtils.createDirectStream</code> , e.g. <code>KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder]</code> , so proper decoders are used to decode messages from Kafka.
------	--

You have to specify `metadata.broker.list` or `bootstrap.servers` (in that order of precedence) for your Kafka environment. `metadata.broker.list` is a comma-separated list of Kafka's (seed) brokers in the format of `<host>:<port>`.

Note	Kafka brokers have to be up and running <i>before</i> you can create a direct stream.
------	---

```

val conf = new SparkConf().setMaster("local[*]").setAppName("Ingesting Data from Kafka")
conf.set("spark.streaming.ui.retainedBatches", "5")

// Enable Back Pressure
conf.set("spark.streaming.backpressure.enabled", "true")

val ssc = new StreamingContext(conf, batchDuration = Seconds(5))

// Enable checkpointing
ssc.checkpoint("_checkpoint")

// You may or may not want to enable some additional DEBUG logging
import org.apache.log4j._

Logger.getLogger("org.apache.spark.streaming.DStream").setLevel(Level.DEBUG)
Logger.getLogger("org.apache.spark.streaming.WindowedDStream").setLevel(Level.DEB
Logger.getLogger("org.apache.spark.streaming.DStreamGraph").setLevel(Level.DEBUG)
Logger.getLogger("org.apache.spark.streaming.scheduler.JobGenerator").setLevel(Level.DEBU

// Connect to Kafka
import org.apache.spark.streaming.kafka.KafkaUtils
import _root_.kafka.serializer.StringDecoder
val kafkaParams = Map("metadata.broker.list" -> "localhost:9092")
val kafkaTopics = Set("spark-topic")
val messages = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder]

// print 10 last messages
messages.print()

// start streaming computation
ssc.start

```

If `zookeeper.connect` or `group.id` parameters are not set, they are added with their values being empty strings.

In this mode, you will only see jobs submitted (in the **Jobs** tab in [web UI](#)) when a message comes in.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	Streaming job from [output operation 0, batch time 22:17:15] print at <console>:32	2016/01/09 22:17:15	18 ms	1/1	1/1
1	Streaming job from [output operation 0, batch time 22:11:45] print at <console>:32	2016/01/09 22:11:45	16 ms	1/1	1/1
0	Streaming job from [output operation 0, batch time 22:09:15] print at <console>:32	2016/01/09 22:09:15	0.2 s	1/1	1/1

Figure 1. Complete Jobs in web UI for batch time 22:17:15

It corresponds to **Input size** larger than 0 in the **Streaming** tab in the web UI.



Figure 2. Completed Batch in web UI for batch time 22:17:15

Click the link in Completed Jobs for a batch and you see the details.

Figure 3. Details of batch in web UI for batch time 22:17:15

spark-streaming-kafka Library Dependency

The new API for both Kafka RDD and DStream is in the `spark-streaming-kafka` artifact. Add the following dependency to sbt project to use the streaming integration:

```
libraryDependencies += "org.apache.spark" %% "spark-streaming-kafka" % "2.0.0-SNAPSHOT"
```

Note

Replace "2.0.0-SNAPSHOT" with available version as found at [The Central Repository's search](#).

DirectKafkaInputDStream

`DirectKafkaInputDStream` is an [input stream](#) of [KafkaRDD](#) batches.

As an input stream, it implements the *five* mandatory abstract methods - three from `DStream` and two from `InputDStream`:

- `dependencies: List[DStream[_]]` returns an empty collection, i.e. it has no dependencies on other streams (other than Kafka brokers to read data from).
- `slideDuration: Duration` passes all calls on to `DStreamGraph.batchDuration`.

- `compute(validTime: Time): Option[RDD[T]]` - consult [Computing RDDs \(using compute Method\)](#) section.
- `start()` does nothing.
- `stop()` does nothing.

The `name` of the input stream is **Kafka direct stream [id]**. You can find the name in the [Streaming tab](#) in web UI (in the details of a batch in **Input Metadata** section).

It uses `spark.streaming.kafka.maxRetries` setting while computing `latestLeaderOffsets` (i.e. a mapping of `kafka.common.TopicAndPartition` and `LeaderOffset`).

Computing RDDs (using compute Method)

`DirectKafkaInputDStream.compute` always computes a **KafkaRDD** instance (despite the [DStream contract](#) that says it may or may not generate one).

Note	It is DStreamGraph to request generating streaming jobs for batches.
------	--

Every time the method is called, `latestLeaderOffsets` calculates the latest offsets (as `Map[TopicAndPartition, LeaderOffset]`).

Note	Every call to <code>compute</code> does call Kafka brokers for the offsets.
------	---

The *moving* parts of generated `KafkaRDD` instances are offsets. Others are taken directly from `DirectKafkaInputDStream` (given at the time of instantiation).

It then filters out empty offset ranges to build `StreamInputInfo` for [InputInfoTracker.reportInfo](#).

It sets the just-calculated offsets as current (using `currentOffsets`) and returns a new **KafkaRDD** instance.

Back Pressure

Caution	FIXME
---------	-----------------------

[Back pressure](#) for Direct Kafka input dstream can be configured using `spark.streaming.backpressure.enabled` setting.

Note	Back pressure is disabled by default.
------	---------------------------------------

Kafka Concepts

- broker
- leader
- topic
- partition
- offset
- exactly-once semantics
- Kafka high-level consumer

LeaderOffset

`LeaderOffset` is an internal class to represent an offset on the topic partition on the broker that works on a host and a port.

Recommended Reading

- [Exactly-once Spark Streaming from Apache Kafka](#)

KafkaRDD

`KafkaRDD` class represents a [RDD dataset](#) from Apache Kafka. It uses `KafkaRDDPartition` for partitions that know their preferred locations as the host of the topic (not port however!). It then nicely maps a RDD partition to a Kafka partition.

Tip

Studying `KafkaRDD` class can greatly improve understanding of Spark (core) in general, i.e. how RDDs are used for distributed computations.

`KafkaRDD` overrides methods of `RDD` class to base them on `offsetRanges`, i.e. partitions.

You can create `KafkaRDD` using `KafkaUtils.createRDD(sc: SparkContext, kafkaParams: Map[String, String], offsetRanges: Array[OffsetRange])`.

Enable `INFO` logging level for `org.apache.spark.streaming.kafka.KafkaRDD` logger to see what happens in `KafkaRDD`.

Tip

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.streaming.kafka.KafkaRDD=INFO
```

Computing Partitions

To compute a partition, `KafkaRDD`, checks for validity of beginning and ending offsets (so they range over at least one element) and returns an (internal) `KafkaRDDIterator`.

You should see the following INFO message in the logs:

```
INFO KafkaRDD: Computing topic [topic], partition [partition] offsets [fromOffset] -> [to
```

It creates a new `KafkaCluster` every time it is called as well as `kafka.serializer.Decoder` for the key and the value (that come with a constructor that accepts `kafka.utils.VerifiableProperties`).

It fetches batches of `kc.config.fetchMessageMaxBytes` size per topic, partition, and offset (it uses `kafka.consumer.SimpleConsumer.fetch(kafka.api.FetchRequest)` method).

Caution

[FIXME](#) Review

RecurringTimer

```
class RecurringTimer(clock: Clock, period: Long, callback: (Long) => Unit, name: String)
```

`RecurringTimer` (aka **timer**) is a `private[streaming]` class that uses a single daemon thread prefixed `RecurringTimer - [name]` that, once `started`, executes `callback` in a loop every `period` time (until it is `stopped`).

The wait time is achieved by `Clock.waitTillTime` (that makes testing easier).

Tip Enable `INFO` or `DEBUG` logging level for `org.apache.spark.streaming.util.RecurringTimer` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.streaming.util.RecurringTimer=DEBUG
```

Refer to [Logging](#).

When `RecurringTimer` triggers an action for a `period`, you should see the following `DEBUG` message in the logs:

```
DEBUG RecurringTimer: Callback for [name] called at time [prevTime]
```

Start and Restart Times

```
getStartTime(): Long
getRestartTime(originalStartTime: Long): Long
```

`getStartTime` and `getRestartTime` are helper methods that calculate time.

`getStartTime` calculates a time that is a multiple of the timer's `period` and is right after the current system time.

Note `getStartTime` is used when [JobGenerator is started](#).

`getRestartTime` is similar to `getStartTime` but includes `originalStartTime` input parameter, i.e. it calculates a time as `getStartTime` but shifts the result to accommodate the time gap since `originalStartTime`.

Note	<code>getRestartTime</code> is used when JobGenerator is restarted.
------	---

Starting Timer

```
start(startTime: Long): Long
start(): Long (1)
```

1. Uses the internal [getStartTime](#) method to calculate `startTime` and calls `start(startTime: Long)`.

You can start a `RecurringTimer` using `start` methods.

Note	<code>start()</code> method uses the internal getStartTime method to calculate <code>startTime</code> and calls <code>start(startTime: Long)</code> .
------	---

When `start` is called, it sets the internal `nextTime` to the given input parameter `startTime` and starts the internal daemon thread. This is the moment when the clock starts ticking...

You should see the following INFO message in the logs:

```
INFO RecurringTimer: Started timer for [name] at time [nextTime]
```

Stopping Timer

```
stop(interruptTimer: Boolean): Long
```

A timer is stopped using `stop` method.

Note	It is called when JobGenerator stops.
------	---

When called, you should see the following INFO message in the logs:

```
INFO RecurringTimer: Stopped timer for [name] after time [prevTime]
```

`stop` method uses the internal `stopped` flag to mark the stopped state and returns the last period for which it was successfully executed (tracked as `prevTime` internally).

Note	Before it fully terminates, it triggers <code>callback</code> one more/last time, i.e. <code>callback</code> is executed for a period after <code>RecurringTimer</code> has been (marked) stopped.
------	--

Fun Fact

You can execute `org.apache.spark.streaming.util.RecurringTimer` as a command-line standalone application.

```
$ ./bin/spark-class org.apache.spark.streaming.util.RecurringTimer
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
INFO RecurringTimer: Started timer for Test at time 1453787444000
INFO RecurringTimer: 1453787444000: 1453787444000
DEBUG RecurringTimer: Callback for Test called at time 1453787444000
INFO RecurringTimer: 1453787445005: 1005
DEBUG RecurringTimer: Callback for Test called at time 1453787445000
INFO RecurringTimer: 1453787446004: 999
DEBUG RecurringTimer: Callback for Test called at time 1453787446000
INFO RecurringTimer: 1453787447005: 1001
DEBUG RecurringTimer: Callback for Test called at time 1453787447000
INFO RecurringTimer: 1453787448000: 995
DEBUG RecurringTimer: Callback for Test called at time 1453787448000
^C
INFO ShutdownHookManager: Shutdown hook called
INFO ShutdownHookManager: Deleting directory /private/var/folders/0w/kb0d3rqn4zb9fcc91pxh
INFO ShutdownHookManager: Deleting directory /private/var/folders/0w/kb0d3rqn4zb9fcc91pxh
```

Streaming DataFrames

Note

Watch the video [The Future of Real Time in Spark](#) from Spark Summit East 2016 in which Reynold Xin first presents the concept of **Streaming DataFrames** to the public.

Streaming DataFrames are *infinite dataframes* to be the foundation of **Structured Streaming** for a high-level streaming API over Spark SQL engine. It is slated to be available in Spark 2.0.

It is supposed to unify streaming, interactive, and batch queries.

Every **trigger** (time) a computation on DataFrames should be performed.

Why did Reynold speak about DataFrames (not Datasets)?

Backpressure (Back Pressure)

Quoting TD from his talk about Spark Streaming:

Backpressure is to make applications robust against data surges.

With backpressure you can guarantee that your Spark Streaming application is **stable**, i.e. receives data only as fast as it can process it.

Note

Backpressure shifts the trouble of buffering input records to the sender so it keeps records until they could be processed by a streaming application. You could alternatively use [dynamic allocation](#) feature in Spark Streaming to increase the capacity of streaming infrastructure without slowing down the senders.

Backpressure is disabled by default and can be turned on using [spark.streaming.backpressure.enabled](#) setting.

You can monitor a streaming application using [web UI](#). It is important to ensure that the [batch processing time](#) is shorter than the [batch interval](#). Backpressure introduces a **feedback loop** so the streaming system can adapt to longer processing times and avoid instability.

Note

Backpressure is available since Spark 1.5.

RateController

Tip

Read up on [back pressure](#) in Wikipedia.

`RateController` is a contract for single-dstream [StreamingListeners](#) that listens to [batch completed updates](#) for a dstream and maintains a **rate limit**, i.e. an estimate of the speed at which this stream should ingest messages. With every batch completed update event it calculates the current processing rate and estimates the correct receiving rate.

Note

`RateController` works for a single dstream and requires a [RateEstimator](#).

The contract says that RateControllers offer the following method:

```
protected def publish(rate: Long): Unit
```

When created, it creates a daemon single-thread executor service called **stream-rate-update** and initializes the internal `rateLimit` counter which is the current message-ingestion speed.

When a batch completed update happens, a `RateController` grabs `processingEndTime`, `processingDelay`, `schedulingDelay`, and `numRecords` processed for the batch, computes a rate limit and publishes the current value. The computed value is set as the present rate limit, and published (using the sole abstract `publish` method).

Computing a rate limit happens using the `RateEstimator`'s `compute` method.

Caution

FIXME Where is this used? What are the use cases?

`InputDStreams` can define a `RateController` that is registered to `JobScheduler`'s `listenerBus` (using `ssc.addStreamingListener`) when `JobScheduler` starts.

RateEstimator

`RateEstimator` computes the rate given the input `time`, `elements`, `processingDelay`, and `schedulingDelay`.

It is an abstract class with the following abstract method:

```
def compute(
    time: Long,
    elements: Long,
    processingDelay: Long,
    schedulingDelay: Long): Option[Double]
```

You can control what `RateEstimator` to use through `spark.streaming.backpressure.rateEstimator` setting.

The only possible `RateEstimator` to use is the [pid rate estimator](#).

PID Rate Estimator

PID Rate Estimator (represented as `PIDRateEstimator`) implements a [proportional-integral-derivative \(PID\) controller](#) which acts on the speed of ingestion of records into an input `dstream`.

Warning

The **PID rate estimator** is the only possible estimator. All other rate estimators lead to `IllegalArgumentException` being thrown.

It uses the following settings:

- `spark.streaming.backpressure.pid.proportional` (default: 1.0) can be 0 or greater.
- `spark.streaming.backpressure.pid.integral` (default: 0.2) can be 0 or greater.
- `spark.streaming.backpressure.pid.derived` (default: 0.0) can be 0 or greater.
- `spark.streaming.backpressure.pid.minRate` (default: 100) must be greater than 0.

Note

The PID rate estimator is used by [DirectKafkaInputDStream](#) and [input streams with receivers \(aka ReceiverInputDStreams\)](#).

Tip Enable `INFO` or `TRACE` logging level for `org.apache.spark.streaming.scheduler.rate.PIDRateEstimator` logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.streaming.scheduler.rate.PIDRateEstimator=TRACE
```

Refer to [Logging](#).

When the PID rate estimator is created you should see the following INFO message in the logs:

```
INFO PIDRateEstimator: Created PIDRateEstimator with proportional = [proportional], integ
```

When the pid rate estimator computes the rate limit for the current time, you should see the following TRACE message in the logs:

```
TRACE PIDRateEstimator:  
time = [time], # records = [numElements], processing time = [processingDelay], scheduling
```

If the time to compute the current rate limit for is before the latest time or the number of records is 0 or less, or processing delay is 0 or less, the rate estimation is skipped. You should see the following TRACE message in the logs:

```
TRACE PIDRateEstimator: Rate estimation skipped
```

And no rate limit is returned.

Otherwise, when this is to compute the rate estimation for next time and there are records processed as well as the processing delay is positive, it computes the rate estimate.

Once the new rate has already been computed, you should see the following TRACE message in the logs:

```
TRACE PIDRateEstimator:  
latestRate = [latestRate], error = [error]  
latestError = [latestError], historicalError = [historicalError]  
delaySinceUpdate = [delaySinceUpdate], dError = [dError]
```

If it was the first computation of the limit rate, you should see the following TRACE message in the logs:

```
TRACE PIDRateEstimator: First run, rate estimation skipped
```

No rate limit is returned.

Otherwise, when it is another limit rate, you should see the following TRACE message in the logs:

```
TRACE PIDRateEstimator: New rate = [newRate]
```

And the current rate limit is returned.

Elastic Scaling (Dynamic Allocation)

Dynamic Allocation in Spark Streaming makes for **adaptive streaming applications** by scaling them up and down to adapt to load variations. It actively controls resources (as executors) and prevents resources from being wasted when the processing time is short (comparing to a batch interval) - **scale down** - or adds new executors to decrease the processing time - **scale up**.

Note

It is a work in progress in Spark Streaming and should be available in Spark 2.0.

The motivation is to control the number of executors required to process input records when their number increases to the point when the [processing time](#) could become longer than the [batch interval](#).

Configuration

- `spark.streaming.dynamicAllocation.enabled` controls whether to enable dynamic allocation (`true`) or not (`false`).

Settings

The following list are the settings used to configure Spark Streaming applications.

Caution

FIXME Describe how to set them in streaming applications.

- `spark.streaming.kafka.maxRetries` (default: `1`) sets up the number of connection attempts to Kafka brokers.
- `spark.streaming.receiver.writeAheadLog.enable` (default: `false`) controls what [ReceivedBlockHandler](#) to use: `WriteAheadLogBasedBlockHandler` or `BlockManagerBasedBlockHandler`.
- `spark.streaming.receiver.blockStoreTimeout` (default: `30`) time in seconds to wait until both writes to a write-ahead log and BlockManager complete successfully.
- `spark.streaming.clock` (default: `org.apache.spark.util.SystemClock`) specifies a fully-qualified class name that extends `org.apache.spark.util.Clock` to represent time. It is used in [JobGenerator](#).
- `spark.streaming.ui.retainedBatches` (default: `1000`) controls the number of `BatchUIData` elements about completed batches in a first-in-first-out (FIFO) queue that are used to [display statistics in Streaming page in web UI](#).
- `spark.streaming.receiverRestartDelay` (default: `2000`) - the time interval between a receiver is stopped and started again.
- `spark.streaming.concurrentJobs` (default: `1`) is the number of concurrent jobs, i.e. threads in [streaming-job-executor thread pool](#).
- `spark.streaming.stopSparkContextByDefault` (default: `true`) controls whether (`true`) or not (`false`) to stop the underlying `SparkContext` (regardless of whether this `StreamingContext` has been started).
- `spark.streaming.kafka.maxRatePerPartition` (default: `0`) if non-`0` sets maximum number of messages per partition.
- `spark.streaming.manualClock.jump` (default: `0`) offsets (aka *jumps*) the system time, i.e. adds its value to checkpoint time, when used with the clock being a subclass of `org.apache.spark.util.ManualClock`. It is used when [JobGenerator](#) is restarted from checkpoint.
- `spark.streaming.unpersist` (default: `true`) is a flag to control whether [output streams](#) should unpersist old RDDs.

- `spark.streaming.gracefulStopTimeout` (default: `10 * batch interval`)
- `spark.streaming.stopGracefullyOnShutdown` (default: `false`) controls whether to stop `StreamingContext` gracefully or not and is used by `stopOnShutdown` Shutdown Hook.

Checkpointing

- `spark.streaming.checkpoint.directory` - when set and `StreamingContext` is created, the value of the setting gets passed on to `StreamingContext.checkpoint` method.

Back Pressure

- `spark.streaming.backpressure.enabled` (default: `false`) - enables (`true`) or disables (`false`) back pressure in input streams with receivers or DirectKafkaInputDStream.
- `spark.streaming.backpressure.rateEstimator` (default: `pid`) is the RateEstimator to use.

Spark SQL

From [Spark SQL home page](#):

Spark SQL is Spark's module for working with structured data (rows and columns) in Spark.

From [Spark's Role in the Big Data Ecosystem - Matei Zaharia video](#):

Spark SQL enables loading & querying structured data in Spark.

It comes with a uniform interface for data access that can live in distributed storage systems like Cassandra or HDFS (Hive, Parquet, JSON).

DataFrame

Spark SQL introduces a tabular data abstraction called [DataFrame](#). It is designed to ease processing large amount of structured tabular data on Spark infrastructure.

	Found the following note about Apache Drill, but appears to apply to Spark SQL perfectly:
Note	A SQL query engine for relational and NoSQL databases with direct queries on self-describing and semi-structured data in files, e.g. JSON or Parquet, and HBase tables without needing to specify metadata definitions in a centralized store.

	FIXME What does the following do?
Caution	<code>df.selectExpr("rand()", "randn()", "rand(5)", "randn(50)')</code>

From user@spark:

If you already loaded csv data into a [dataframe](#), why not register it as a table, and use Spark SQL to find max/min or any other aggregates? `SELECT MAX(column_name) FROM dftable_name ...` seems natural.

you're more comfortable with SQL, it might worth registering this [DataFrame](#) as a table and generating SQL query to it (generate a string with a series of min-max calls)

Caution	FIXME Transform the quotes into working examples.
---------	--

Creating DataFrames

From <http://stackoverflow.com/a/32514683/1305344>:

```
val df = sc.parallelize(Seq(
    Tuple1("08/11/2015"), Tuple1("09/11/2015"), Tuple1("09/12/2015")
)).toDF("date_string")

df.registerTempTable("df")

sqlContext.sql(
    """SELECT date_string,
       from_unixtime(unix_timestamp(date_string, 'MM/dd/yyyy'), 'EEEEEE') AS dow
      FROM df"""
).show
```

The result:

date_string	dow
08/11/2015	Tuesday
09/11/2015	Friday
09/12/2015	Saturday

- Where do `from_unixtime` and `unix_timestamp` come from? `HiveContext` perhaps?
How are they registered
- What other UDFs are available?

Reading JSON file

Execute the following using `./bin/spark-shell` (it provides `sc` for `SparkContext` and `sqlContext` for `Spark SQL context`):

```
val hello = sc.textFile("/Users/jacek/dev/sandbox/hello.json")
val helloDF = sqlContext.read.json(hello)
```

Depending on the content of `hello.json` you may see different schema. The point, however, is that you can parse JSON files and let the *schema inferencer* to deduct the schema.

```
scala> helloDF.printSchema
root
```

Register temp table to use for queries:

```
helloDF.registerTempTable("helloT")
```

Reading data via JDBC

```
scala> sqlContext.read.format("jdbc")
res0: org.apache.spark.sql.DataFrameReader = org.apache.spark.sql.DataFrameReader@46baac0
```

Handling data in Avro format

Use custom serializer using [spark-avro](#).

Run Spark shell with `--packages com.databricks:spark-avro_2.11:2.0.0` ([see 2.0.0 artifact is not in any public maven repo why --repositories is required](#)).

```
./bin/spark-shell --packages com.databricks:spark-avro_2.11:2.0.0 --repositories "http://
```

And then...

```
val fileRdd = sc.textFile("README.md")
val df = fileRdd.toDF

import org.apache.spark.sql.SaveMode

val outputF = "test.avro"
df.write.mode(SaveMode.Append).format("com.databricks.spark.avro").save(outputF)
```

See [org.apache.spark.sql.SaveMode](#) (and perhaps [org.apache.spark.sql.SaveMode](#) from Scala's perspective).

```
val df = sqlContext.read.format("com.databricks.spark.avro").load("test.avro")
```

Show the result:

```
df.show
```

Group and aggregate

```
val df = sc.parallelize(Seq(
  (1441637160, 10.0),
  (1441637170, 20.0),
  (1441637180, 30.0),
  (1441637210, 40.0),
  (1441637220, 10.0),
  (1441637230, 0.0))).toDF("timestamp", "value")

import org.apache.spark.sql.types._

val tsGroup = (floor($"timestamp" / lit(60)) * lit(60)).cast(IntegerType).alias("timestamp")

df.groupBy(tsGroup).agg(mean($"value").alias("value")).show
```



The above example yields the following result:

```
+-----+-----+
| timestamp|value|
+-----+-----+
|1441637160| 25.0|
|1441637220|  5.0|
+-----+-----+
```

[See the answer on StackOverflow.](#)

More examples

Another example:

```
val df = Seq(1 -> 2).toDF("i", "j")
val query = df.groupBy('i)
  .agg(max('j).as("aggOrdering"))
  .orderBy(sum('j))
query == Row(1, 2) // should return true
```

What does it do?

```
val df = Seq((1, 1), (-1, 1)).toDF("key", "value")
df.registerTempTable("src")
sql("SELECT IF(a > 0, a, 0) FROM (SELECT key a FROM src) temp")
```

SQLContext

SQLContext is the entry point for Spark SQL. Whatever you do in Spark SQL it has to start from [creating an instance of SQLContext](#).

A `SQLContext` object requires a `SparkContext`, a `CacheManager`, and a `SQLListener`. They are all `transient` and do not participate in serializing a `SQLContext`.

You should use `SQLContext` for the following:

- [Creating Datasets](#)
- [Creating DataFrames](#)
- [Creating DataFrames from Range \(range method\)](#)
- [Creating DataFrames for Table](#)
- [Accessing DataFrameReader](#)
- [Accessing ContinuousQueryManager](#)
- [Registering User-Defined Functions \(UDF\)](#)
- [Caching DataFrames in In-Memory Cache](#)
- [Setting Configuration Properties](#)
- [Bringing Converter Objects into Scope](#)
- [Creating External Tables](#)
- [Dropping Temporary Tables](#)
- [Listing Existing Tables](#)
- [Managing Active SQLContext for JVM](#)

Creating SQLContext Instance

You can create a `SQLContext` using the following constructors:

- `SQLContext(sc: SparkContext)`
- `SQLContext.getOrCreate(sc: SparkContext)`
- `SQLContext.newSession()` allows for creating a new instance of `SQLContext` with a separate SQL configuration (through a shared `SparkContext`).

Setting Configuration Properties

You can set Spark SQL configuration properties using:

- `setConf(props: Properties): Unit`
- `setConf(key: String, value: String): Unit`

You can get the current value of a configuration property by key using:

- `getConf(key: String): String`
- `getConf(key: String, defaultValue: String): String`
- `getAllConfs: immutable.Map[String, String]`

Note

Properties that start with `spark.sql` are reserved for Spark SQL.

Creating DataFrames

emptyDataFrame

```
emptyDataFrame: DataFrame
```

`emptyDataFrame` creates an empty `DataFrame`. It calls `createDataFrame` with an empty `RDD[Row]` and an empty schema `StructType(Nil)`.

createDataFrame for RDD and Seq

```
createDataFrame[A <: Product](rdd: RDD[A]): DataFrame  
createDataFrame[A <: Product](data: Seq[A]): DataFrame
```

`createDataFrame` family of methods can create a `DataFrame` from an `RDD` of Scala's Product types like case classes or tuples or `Seq` thereof.

createDataFrame for RDD of Row with Explicit Schema

```
createDataFrame(rowRDD: RDD[Row], schema: StructType): DataFrame
```

This variant of `createDataFrame` creates a `DataFrame` from `RDD` of `Row` and explicit schema.

Registering User-Defined Functions (UDF)

```
udf: UDFRegistration
```

`udf` method gives access to `UDFRegistration` to manipulate user-defined functions.

Caching DataFrames in In-Memory Cache

```
isCached(tableName: String): Boolean
```

`isCached` method asks `CacheManager` whether `tableName` table is cached in memory or not. It simply requests `CacheManager` for `CachedData` and when exists, it assumes the table is cached.

```
cacheTable(tableName: String): Unit
```

You can cache a table in memory using `cacheTable`.

Caution	Why would I want to cache a table?
---------	------------------------------------

```
uncacheTable(tableName: String)
clearCache(): Unit
```

`uncacheTable` and `clearCache` remove one or all in-memory cached tables.

Implicits - SQLContext.implicitly

The `implicitly` object is a helper class with methods to convert objects into [Datasets](#) and [DataFrames](#), and also comes with many [Encoders](#) for "primitive" types as well as the collections thereof.

Note	Import the implicits by <code>import sqlContext.implicitly._</code> as follows:
------	---

Import the implicits by `import sqlContext.implicitly._` as follows:

```
val sqlContext = new SQLContext(sc)
import sqlContext.implicitly._
```

It holds [Encoders](#) for Scala "primitive" types like `Int`, `Double`, `String`, and their collections.

It offers support for creating `Dataset` from `RDD` of any types (for which an Encoder exists in scope), or case classes or tuples, and `Seq`.

It also offers conversions from Scala's `Symbol` or `$` to `Column`.

It also offers conversions from `RDD` or `Seq` of `Product` types (e.g. case classes or tuples) to `DataFrame`. It has direct conversions from `RDD` of `Int`, `Long` and `String` to `DataFrame` with a single column name `_1`.

Note

It is not possible to call `toDF` methods on `RDD` objects of other "primitive" types except `Int`, `Long`, and `String`.

Creating Datasets

```
createDataset[T: Encoder](data: Seq[T]): Dataset[T]
createDataset[T: Encoder](data: RDD[T]): Dataset[T]
```

`createDataset` family of methods creates a `Dataset` from a collection of elements of type `T`, be it a regular Scala `Seq` or Spark's `RDD`.

It requires that there is an `Encoder` in scope.

Note

[Importing SQLContext.implicits](#) brings many Encoders available in scope.

Accessing DataFrameReader

```
read: DataFrameReader
```

The experimental `read` method returns a `DataFrameReader` that is used to read data from external storage systems and load it into a `DataFrame`.

Creating External Tables

```
createExternalTable(tableName: String, path: String): DataFrame
createExternalTable(tableName: String, path: String, source: String): DataFrame
createExternalTable(tableName: String, source: String, options: Map[String, String]): DataFrame
createExternalTable(tableName: String, source: String, schema: StructType, options: Map[S
```

The experimental `createExternalTable` family of methods is used to create an external table `tableName` and return a corresponding `DataFrame`.

Caution

FIXME What is an external table?

Dropping Temporary Tables

```
dropTempTable(tableName: String): Unit
```

`dropTempTable` method drops a temporary table `tableName`.

Caution

FIXME What is a temporary table?

Creating DataFrames from Range (range method)

```
range(end: Long): DataFrame
range(start: Long, end: Long): DataFrame
range(start: Long, end: Long, step: Long): DataFrame
range(start: Long, end: Long, step: Long, numPartitions: Int): DataFrame
```

The `range` family of methods creates a `DataFrame` with the sole `id` column of `LongType` for given `start`, `end`, and `step`.

Note

The three first variants use `SparkContext.defaultParallelism` for the number of partitions `numPartitions`.

Creating DataFrames for Table

```
table(tableName: String): DataFrame
```

`table` method creates a `tableName` table and returns a corresponding `DataFrame`.

Listing Existing Tables

```
tables(): DataFrame
tables(databaseName: String): DataFrame
```

`table` methods return a `DataFrame` that holds names of existing tables in a database. The schema consists of two columns - `tableName` of `StringType` and `isTemporary` of `BooleanType`.

Note

`tables` is a result of `SHOW TABLES [IN databaseName]`.

```
tableNames(): Array[String]
tableNames(databaseName: String): Array[String]
```

`tableNames` are similar to `tables` with the only difference that they return `Array[String]` which is a collection of table names.

Accessing ContinuousQueryManager

```
streams: ContinuousQueryManager
```

The `streams` method returns a [ContinuousQueryManager](#) that is used to...TK

Caution	FIXME
---------	-------

Managing Active SQLContext for JVM

```
SQLContext.getOrCreate(sparkContext: SparkContext): SQLContext
```

`SQLContext.getOrCreate` method returns an active `SQLContext` object for the JVM or creates a new one using a given `sparkContext`.

Note	It is a factory-like method that works on <code>SQLContext</code> class.
------	--

Interestingly, there are two helper methods to set and clear the active `SQLContext` object - `setActive` and `clearActive` respectively.

```
setActive(sqlContext: SQLContext): Unit
clearActive(): Unit
```

Dataset

Dataset is an experimental feature of Spark SQL that has first been introduced in Apache Spark 1.6.0. It aims at expanding on [DataFrames](#) to offer strong type-safety (of Scala the programming language) at compile time and a functional programming interface to work with structured data.

A `Dataset` object requires a [SQLContext](#), a [QueryExecution](#), and an [Encoder](#). In some cases, a `Dataset` can also be seen as a pair of [LogicalPlan](#) in a given [SQLContext](#).

Note

`SQLContext` and `QueryExecution` are transient and hence do not participate when a dataset is serialized. The only *firmly-tied* feature of a Dataset is the Encoder.

A `dataset` is [Queryable](#) and [Serializable](#), i.e. can be saved to a persistent storage.

A Dataset has a [schema](#).

You can convert a Dataset to a [DataFrame](#) (using `toDF()` method) or an RDD (using `rdd` method).

```

import sqlContext.implicits._

case class Token(name: String, productId: Int, score: Double)
val data = Token("aaa", 100, 0.12) :: 
  Token("aaa", 200, 0.29) :: 
  Token("bbb", 200, 0.53) :: 
  Token("bbb", 300, 0.42) :: Nil

org.apache.spark.sql.catalyst.encoders.OuterScopes.addOuterScope(this) (1)

val ds = data.toDS

scala> ds.show
+---+-----+-----+
|name|productId|score|
+---+-----+-----+
| aaa|      100|  0.12|
| aaa|      200|  0.29|
| bbb|      200|  0.53|
| bbb|      300|  0.42|
+---+-----+-----+

scala> ds.printSchema
root
 |-- name: string (nullable = true)
 |-- productId: integer (nullable = false)
 |-- score: double (nullable = false)

```

1. Alas, you need to use the line to make the conversion work.
2. Create a Dataset of Token instances.

Tip

In spark-shell, you have to use the following line to successfully register custom case classes:

```
org.apache.spark.sql.catalyst.encoders.OuterScopes.addOuterScope(this)
```

Converting Datasets into RDDs (using rdd method)

Whenever in need to convert a Dataset into a RDD, executing `rdd` method gives you a RDD of the proper input type (not `Row` as in DataFrames).

```

scala> val rdd = ds.rdd
rdd: org.apache.spark.rdd.RDD[Token] = MapPartitionsRDD[11] at rdd at <console>:30

```

Schema

A Dataset has a **schema** that is available as `schema`.

You may also use the following methods to learn about the schema:

- `printSchema(): Unit`
- `explain(): Unit`
- `explain(extended: Boolean): Unit`

Logical and Physical Plans

Caution	FIXME
---------	-------

Encoder

Caution	FIXME
---------	-------

It works with the type of the accompanying Dataset.

An `Encoder` object is used to convert a JVM object into a Dataset row. It is designed for fast serialization and deserialization.

Note	<code>SQLContext.implicits</code> object comes with Encoders for many types in Scala.
------	---

QueryExecution

Caution	FIXME
---------	-------

Note	It is a transient feature of a Dataset, i.e. it is not preserved across serializations.
------	---

Queryable

Caution	FIXME
---------	-------

LogicalPlan

Caution	FIXME
---------	-------

DataFrame

A **DataFrame** is a data abstraction or a domain-specific language (DSL) for working with **structured and semi-structured data**.

A **DataFrame** is a distributed collection of tabular data organized into **rows** and **named columns**. It is conceptually equivalent to a table in a relational database and provides operations to project (`select`), `filter` , `intersect` , `join` , `group` , `sort` , `join` , `aggregate` , or `convert` to a RDD (consult [DataFrame API](#))

```
data.groupBy("Product_ID").sum("Score")
```

Spark SQL borrowed the concept of **DataFrame** (from pandas' **DataFrame**) and made it **immutable**, **parallel** (one machine, perhaps with many processors and cores) and **distributed** (many machines, perhaps with many processors and cores).

Note

Hey, big data consultants, time to help teams migrate the code from pandas' **DataFrame** into Spark's DataFrames (at least to PySpark's **DataFrame**) and offer services to set up large clusters!

DataFrames in Spark SQL strongly rely on [the features of RDD](#) - it's basically a RDD exposed as **DataFrame** by appropriate operations to handle very big data from the day one. So, petabytes of data should *not* scare you (unless you're an administrator to create such clustered Spark environment - [contact me when you feel alone with the task](#)).

You can create DataFrames by loading data from structured files (JSON, Parquet, CSV), RDDs, tables in Hive, or external databases (JDBC). You can also create DataFrames from scratch and build upon them. See [DataFrame API](#). You can read any format given you have appropriate Spark SQL extension of [DataFrameReader](#).

[FIXME](#) Diagram of reading data from sources to create **DataFrame**

You can execute queries over DataFrames using two approaches:

- [the good ol' SQL](#) - helps migrating from SQL databases into the world of **DataFrame** in Spark SQL
- [Query DSL](#) - an API that helps ensuring proper syntax at compile time.

`DataFrame` also allows you to do the following tasks:

- [Filtering](#)

Traits of DataFrame

A `DataFrame` is a collection of `Row` instances (as `RDD[Row]`) and a schema (as `StructType`).

Note

However you create a `DataFrame`, it ends up as a pair of `RDD[Row]` and `StructType`.

A schema describes the columns and for each column it defines the name, the type and whether or not it accepts empty values.

StructType

Caution**FIXME**

Adding Column using withColumn

```
withColumn(colName: String, col: Column): DataFrame
```

`withColumn` method returns a new `DataFrame` with the new column `col` with `colName` name added.

Note

`withColumn` can replace an existing `colName` column.

```
scala> val df = Seq((1, "jeden"), (2, "dwa")).toDF("number", "polish")
df: org.apache.spark.sql.DataFrame = [number: int, polish: string]
```

```
scala> df.show
+----+-----+
|number|polish|
+----+-----+
|     1| jeden|
|     2|   dwa|
+----+-----+
```

```
scala> df.withColumn("polish", lit(1)).show
+----+-----+
|number|polish|
+----+-----+
|     1|      1|
|     2|      1|
+----+-----+
```

SQLContext, sqlContext, and Spark shell

You use `org.apache.spark.sql.SQLContext` to build DataFrames and execute SQL queries.

The quickest and easiest way to work with Spark SQL is to use [Spark shell](#) and `sqlContext` object.

```
scala> sqlContext
res1: org.apache.spark.sql.SQLContext = org.apache.spark.sql.hive.HiveContext@60ae950f
```

As you may have noticed, `sqlContext` in Spark shell is actually a [org.apache.spark.sql.hive.HiveContext](#) that integrates the **Spark SQL execution engine** with data stored in [Apache Hive](#).

The Apache Hive™ data warehouse software facilitates querying and managing large datasets residing in distributed storage.

Creating DataFrames from Scratch

Use Spark shell as described in [Spark shell](#).

Creating DataFrame using Case Classes in Scala

This method assumes the data comes from a Scala case class that will describe the schema.

```
scala> case class Person(name: String, age: Int)
defined class Person

scala> val people = Seq(Person("Jacek", 42), Person("Patryk", 19), Person("Maksym", 5))
people: Seq[Person] = List(Person(Jacek,42), Person(Patryk,19), Person(Maksym,5))

scala> val df = sqlContext.createDataFrame(people)
df: org.apache.spark.sql.DataFrame = [name: string, age: int]

scala> df.show
+-----+---+
| name | age |
+-----+---+
| Jacek | 42 |
| Patryk | 19 |
| Maksym | 5 |
+-----+---+
```

SQLContext.emptyDataFrame

Spark SQL 1.3 offers [SQLContext.emptyDataFrame](#) operation to create an empty [DataFrame](#).

```

scala> val df = sqlContext.emptyDataFrame
df: org.apache.spark.sql.DataFrame = []

scala> val adf = df.withColumn("a", lit(1))
adf: org.apache.spark.sql.DataFrame = [a: int]

scala> adf.registerTempTable("t")

scala> sqlContext.sql("select a from t")
res32: org.apache.spark.sql.DataFrame = [a: int]

```

Custom DataFrame Creation using `createDataFrame`

`SQLContext` offers a family of `createDataFrame` operations.

```

scala> val lines = sc.textFile("Cartier+for+WinnersCurse.csv")
lines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at textFile at <console>:24

scala> val headers = lines.first
headers: String = auctionid,bid,bidtime,bidder,bidderrate,openbid,price

scala> import org.apache.spark.sql.types.{StructField, StringType}
import org.apache.spark.sql.types.{StructField, StringType}

scala> val fs = headers.split(",").map(f => StructField(f, StringType))
fs: Array[org.apache.spark.sql.types.StructField] = Array(StructField(auctionid, StringType)

scala> import org.apache.spark.sql.types.StructType
import org.apache.spark.sql.types.StructType

scala> val schema = StructType(fs)
schema: org.apache.spark.sql.types.StructType = StructType(StructField(auctionid, StringType

scala> val noheaders = lines.filter(_ != header)
noheaders: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[10] at filter at <console>

scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row

scala> val rows = noheaders.map(_.split(",")).map(a => Row.fromSeq(a))
rows: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[12] at map at

scala> val auctions = sqlContext.createDataFrame(rows, schema)
auctions: org.apache.spark.sql.DataFrame = [auctionid: string, bid: string, bidtime: string, bidderrate: double, bidder: string, openbid: string]

scala> auctions.printSchema
root
|-- auctionid: string (nullable = true)
|-- bid: string (nullable = true)
|-- bidtime: string (nullable = true)

```

```

| -- bidder: string (nullable = true)
| -- bidderrate: string (nullable = true)
| -- openbid: string (nullable = true)
| -- price: string (nullable = true)

scala> auctions.dtypes
res28: Array[(String, String)] = Array((auctionid,StringType), (bid,StringType), (bidtime

scala> auctions.show(5)
+-----+-----+-----+-----+-----+
| auctionid| bid|   bidtime|   bidder|bidderrate|openbid|price|
+-----+-----+-----+-----+-----+
|1638843936| 500|0.478368056| kona-java|      181|    500| 1625|
|1638843936| 800|0.826388889| doc213|       60|    500| 1625|
|1638843936| 600|3.761122685|      zmxu|        7|    500| 1625|
|1638843936|1500|5.226377315|carloss8055|        5|    500| 1625|
|1638843936|1600| 6.570625|     jdrinaz|       6|    500| 1625|
+-----+-----+-----+-----+-----+
only showing top 5 rows

```

Loading data from structured files

Creating DataFrame from CSV file

Let's start with an example in which **schema inference** relies on a custom case class in Scala.

```

scala> val lines = sc.textFile("Cartier+for+WinnersCurse.csv")
lines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at textFile at <console>:24

scala> val header = lines.first
header: String = auctionid,bid,bidtime,bidder,bidderrate,openbid,price

scala> lines.count
res3: Long = 1349

scala> case class Auction(auctionid: String, bid: Float, bidtime: Float, bidder: String,
defined class Auction

scala> val noheader = lines.filter(_ != header)
noheader: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[53] at filter at <console>:

scala> val auctions = noheader.map(_.split(",")).map(r => Auction(r(0), r(1).toFloat, r(2
auctions: org.apache.spark.rdd.RDD[Auction] = MapPartitionsRDD[59] at map at <console>:35

scala> val df = auctions.toDF
df: org.apache.spark.sql.DataFrame = [auctionid: string, bid: float, bidtime: float, bidd

scala> df.printSchema

```

```

root
|-- auctionid: string (nullable = true)
|-- bid: float (nullable = false)
|-- bidtime: float (nullable = false)
|-- bidder: string (nullable = true)
|-- bidderrate: integer (nullable = false)
|-- openbid: float (nullable = false)
|-- price: float (nullable = false)

scala> df.show
+-----+-----+-----+-----+-----+-----+
| auctionid|    bid|   bidtime|      bidder|bidderrate|openbid|   price|
+-----+-----+-----+-----+-----+-----+
|1638843936| 500.0| 0.47836804|  kona-java|        181| 500.0|1625.0|
|1638843936| 800.0| 0.8263889| doc213|         60| 500.0|1625.0|
|1638843936| 600.0| 3.7611227|      zmxu|          7| 500.0|1625.0|
|1638843936|1500.0| 5.2263775| carloss8055|         5| 500.0|1625.0|
|1638843936|1600.0| 6.570625| jdrinaz|         6| 500.0|1625.0|
|1638843936|1550.0| 6.8929167| carloss8055|         5| 500.0|1625.0|
|1638843936|1625.0| 6.8931136| carloss8055|         5| 500.0|1625.0|
|1638844284| 225.0| 1.237419|dre_313@yahoo.com|        0| 200.0| 500.0|
|1638844284| 500.0| 1.2524074| njbirdmom|        33| 200.0| 500.0|
|1638844464| 300.0| 1.8111342|  aprefer|        58| 300.0| 740.0|
|1638844464| 305.0| 3.2126737| 197509260|         3| 300.0| 740.0|
|1638844464| 450.0| 4.1657987| coharley|        30| 300.0| 740.0|
|1638844464| 450.0| 6.7363195| adammurry|         5| 300.0| 740.0|
|1638844464| 500.0| 6.7364697| adammurry|         5| 300.0| 740.0|
|1638844464|505.78| 6.9881945| 197509260|         3| 300.0| 740.0|
|1638844464| 551.0| 6.9896526| 197509260|         3| 300.0| 740.0|
|1638844464| 570.0| 6.9931483| 197509260|         3| 300.0| 740.0|
|1638844464| 601.0| 6.9939003| 197509260|         3| 300.0| 740.0|
|1638844464| 610.0| 6.994965| 197509260|         3| 300.0| 740.0|
|1638844464| 560.0| 6.9953704| ps138|         5| 300.0| 740.0|
+-----+-----+-----+-----+-----+-----+
only showing top 20 rows

```

Creating DataFrame from CSV files using spark-csv module

You're going to use [spark-csv](#) module to load data from a CSV data source that handles proper parsing and loading.

Note

Support for CSV data sources is available by default in Spark 2.0.0. No need for an external module.

Start the Spark shell using `--packages` option as follows:

```

→ spark git:(master) ✘ ./bin/spark-shell --packages com.databricks:spark-csv_2.11:1.2.0
Ivy Default Cache set to: /Users/jacek/.ivy2/cache
The jars for the packages stored in: /Users/jacek/.ivy2/jars
:: loading settings :: url = jar:file:/Users/jacek/dev/oss/spark/assembly/target/scala-2.
com.databricks#spark-csv_2.11 added as a dependency

scala> val df = sqlContext.read.format("com.databricks.spark.csv").option("header", "true")
df: org.apache.spark.sql.DataFrame = [auctionid: string, bid: string, bidtime: string, bi

scala> df.printSchema
root
|-- auctionid: string (nullable = true)
|-- bid: string (nullable = true)
|-- bidtime: string (nullable = true)
|-- bidder: string (nullable = true)
|-- bidderrate: string (nullable = true)
|-- openbid: string (nullable = true)
|-- price: string (nullable = true)

scala> df.show
+-----+-----+-----+-----+-----+-----+
| auctionid|    bid|   bidtime|      bidder|bidderrate|openbid|price|
+-----+-----+-----+-----+-----+-----+
|1638843936|  500|0.478368056|  kona-java|        181|     500| 1625|
|1638843936|  800|0.826388889| doc213|         60|     500| 1625|
|1638843936|  600|3.761122685|      zmxu|          7|     500| 1625|
|1638843936| 1500|5.226377315| carloss8055|         5|     500| 1625|
|1638843936| 1600|  6.570625| jdrinaz|         6|     500| 1625|
|1638843936| 1550|6.892916667| carloss8055|         5|     500| 1625|
|1638843936| 1625|6.893113426| carloss8055|         5|     500| 1625|
|1638844284| 225|1.237418982|dre_313@yahoo.com|        0|     200| 500|
|1638844284|  500|1.252407407| njbirdmom|        33|     200| 500|
|1638844464|  300|1.811134259|  aprefer|        58|     300| 740|
|1638844464|  305|3.212673611| 197509260|         3|     300| 740|
|1638844464|  450|4.165798611| coharley|        30|     300| 740|
|1638844464|  450|6.736319444| adammurry|         5|     300| 740|
|1638844464|  500|6.736469907| adammurry|         5|     300| 740|
|1638844464| 505.78|6.988194444| 197509260|         3|     300| 740|
|1638844464|  551|6.989652778| 197509260|         3|     300| 740|
|1638844464|  570|6.993148148| 197509260|         3|     300| 740|
|1638844464|  601|6.993900463| 197509260|         3|     300| 740|
|1638844464|  610|6.994965278| 197509260|         3|     300| 740|
|1638844464|  560| 6.99537037| ps138|         5|     300| 740|
+-----+-----+-----+-----+-----+-----+
only showing top 20 rows

```

Loading DataFrames using read

Spark SQL 1.4 offers `SQLContext.read` operation to build a `dataframe` from external storage systems, e.g. file systems, key-value stores, etc.

The supported structured data (file) formats are:

- JSON
- Parquet
- JDBC
- ORC
- Hive tables
- libsvm (using `sqlContext.read.format("libsvm")`)

```
scala> val r = sqlContext.read
r: org.apache.spark.sql.DataFrameReader = org.apache.spark.sql.DataFrameReader@59e67a18

scala> r.parquet("/path/to/file.parquet")
scala> r.schema(schema).json("/path/to/file.json")
scala> r.format("libsvm").load("data/mllib/sample_libsvm_data.txt")
```

Querying DataFrame

Note	Spark SQL offers a Pandas-like Query DSL .
------	--

Using Query DSL

You can select specific columns using `select` method.

Note	This variant (in which you use stringified column names) can only select existing columns, i.e. you cannot create new ones using select expressions.
------	--

```

scala> predictions.printSchema
root
|-- id: long (nullable = false)
|-- topic: string (nullable = true)
|-- text: string (nullable = true)
|-- label: double (nullable = true)
|-- words: array (nullable = true)
|   |-- element: string (containsNull = true)
|-- features: vector (nullable = true)
|-- rawPrediction: vector (nullable = true)
|-- probability: vector (nullable = true)
|-- prediction: double (nullable = true)

scala> predictions.select("label", "words").show
+-----+-----+
|label|      words|
+-----+-----+
| 1.0|[hello, math!]|
| 0.0|[hello, religion!]|
| 1.0|[hello, phy, ic, !]|
+-----+-----+

```

```

scala> auctions.groupBy("bidder").count().show(5)
+-----+-----+
|      bidder|count|
+-----+-----+
|dennisthemenace1|    1|
|amskymom|      5|
|nguyenat@san.rr.com|    4|
|millyjohn|    1|
|ykelectro@hotmail...|    2|
+-----+-----+
only showing top 5 rows

```

In the following example you query for the top 5 of the most active bidders.

Note the `tiny` `$` and `desc` together with the column name to sort the rows by.

```
scala> auctions.groupBy("bidder").count().sort($"count".desc).show(5)
+-----+----+
|    bidder|count|
+-----+----+
|  lass1004|   22|
| pascal1666|   19|
|  freembd|   17|
|restdynamics|   17|
| happyrova|   17|
+-----+----+
only showing top 5 rows

scala> import org.apache.spark.sql.functions._  
import org.apache.spark.sql.functions._

scala> auctions.groupBy("bidder").count().sort(desc("count")).show(5)
+-----+----+
|    bidder|count|
+-----+----+
|  lass1004|   22|
| pascal1666|   19|
|  freembd|   17|
|restdynamics|   17|
| happyrova|   17|
+-----+----+
only showing top 5 rows
```

```

scala> df.select("auctionid").distinct.count
res88: Long = 97

scala> df.groupBy("bidder").count.show
+-----+----+
|      bidder | count |
+-----+----+
| dennisthemenace1 | 1 |
| amskymom | 5 |
| nguyenat@san.rr.com | 4 |
| millyjohn | 1 |
| ykelectro@hotmail... | 2 |
| shetellia@aol.com | 1 |
| rrolex | 1 |
| bupper99 | 2 |
| cheddaboy | 2 |
| adcc007 | 1 |
| varvara_b | 1 |
| yokarine | 4 |
| steven1328 | 1 |
| anjara | 2 |
| roysco | 1 |
| lennonjasonmia@ne... | 2 |
| northwestportland... | 4 |
| bosspad | 10 |
| 31strawberry | 6 |
| nana-tyler | 11 |
+-----+----+
only showing top 20 rows

```

Using SQL

Register a [DataFrame](#) as a named temporary table to run SQL.

```

scala> df.registerTempTable("auctions") (1)

scala> val sql = sqlContext.sql("SELECT count(*) AS count FROM auctions")
sql: org.apache.spark.sql.DataFrame = [count: bigint]

```

1. Register a temporary table so SQL queries make sense

You can execute a SQL query on a [DataFrame](#) using `sql` operation, but before the query is executed it is optimized by **Catalyst query optimizer**. You can print the physical plan for a [DataFrame](#) using the `explain` operation.

```

scala> sql.explain
== Physical Plan ==
TungstenAggregate(key=[], functions=[(count(1),mode=Final,isDistinct=false)], output=[cou
TungstenExchange SinglePartition
TungstenAggregate(key=[], functions=[(count(1),mode=Partial,isDistinct=false)], output=
TungstenProject
Scan PhysicalRDD[auctionid#49,bid#50,bidtime#51,bidder#52,bidderrate#53,openbid#54,pr

scala> sql.show
+---+
|count|
+---+
| 1348|
+---+

scala> val count = sql.collect()(0).getLong(0)
count: Long = 1348

```

Filtering

```

scala> df.show
+---+-----+---+
|name|productId|score|
+---+-----+---+
| aaa|     100| 0.12|
| aaa|     200| 0.29|
| bbb|     200| 0.53|
| bbb|     300| 0.42|
+---+-----+---+

scala> df.filter($"name".like("a%")).show
+---+-----+---+
|name|productId|score|
+---+-----+---+
| aaa|     100| 0.12|
| aaa|     200| 0.29|
+---+-----+---+

```

DataFrame.explain

When performance is the issue you should use `DataFrame.explain(true)` .

Caution	What does it do exactly?
---------	--------------------------

Example Datasets

- eBay online auctions
- SFPD Crime Incident Reporting system

DataFrameReaders

Caution	FIXME
---------	-----------------------

DataFrameWriter

`DataFrameWriter` is used to write `DataFrame` to external storage systems or `data streams`.

Data Streams

`DataFrameWriter` comes with `stream` methods to return a `ContinuousQuery` object.

```
stream(): ContinuousQuery (1)
stream(path: String): ContinuousQuery
```

1. Uses `queryName` option or generates a random name.

```
scala> val df = sqlContext.read.option("inferSchema", "true").format("csv").stream("input")
df: org.apache.spark.sql.DataFrame = [C0: string]

scala> val handle = df.write.format("text").stream("output")
java.lang.UnsupportedOperationException: Data source text does not support streamed writing
  at org.apache.spark.sql.execution.datasources.ResolvedDataSource$.createSink(ResolvedDataSour
  at org.apache.spark.sql.DataFrameWriter.stream(DataFrameWriter.scala:238)
  at org.apache.spark.sql.DataFrameWriter.stream(DataFrameWriter.scala:227)
... 49 elided
```

Standard Functions for DataFrames (functions object)

`org.apache.spark.sql.functions` object comes with many functions for column manipulation in DataFrames.

Note The `functions` object is an experimental feature of Spark since version 1.3.0.

You can access the functions using the following import statement:

```
import org.apache.spark.sql.functions._
```

There are nearly 50 or more functions in the `functions` object. They are grouped by functional areas:

- [Defining UDFs](#)
- Aggregate functions
- Date time functions
- *...and others*

Tip You should read the [official documentation of the functions object](#).

Defining UDFs (udf factories)

```
udf(f: FunctionN[...]): UserDefinedFunction
```

The `udf` family of functions allows you to create [user-defined functions \(UDFs\)](#) based on a user-defined function in Scala. It accepts `f` function of 0 to 10 arguments and the input and output types are automatically inferred (given the types of the respective input and output types of the function `f`).

```

import org.apache.spark.sql.functions._
val _length: String => Int = _.length
val _lengthUDF = udf(_length)

// define a dataframe
val df = sc.parallelize(0 to 3).toDF("num")

// apply the user-defined function to "num" column
scala> df.withColumn("len", _lengthUDF($"num")).show
+---+---+
|num|len|
+---+---+
|  0|  1|
|  1|  1|
|  2|  1|
|  3|  1|
+---+---+

```

Since Spark 2.0.0, there is another variant of `udf` function:

```
udf(f: AnyRef, dataType: DataType): UserDefinedFunction
```

`udf(f: AnyRef, dataType: DataType)` allows you to use a Scala closure for the function argument (as `f`) and explicitly declaring the output data type (as `dataType`).

```

// given the dataframe above

import org.apache.spark.sql.types.IntegerType
val byTwo = udf((n: Int) => n * 2, IntegerType)

scala> df.withColumn("len", byTwo($"num")).show
+---+---+
|num|len|
+---+---+
|  0|  0|
|  1|  2|
|  2|  4|
|  3|  6|
+---+---+

```

Streaming DataFrames (aka Continuous Queries)

Tip

Watch [SPARK-8360 Streaming DataFrames](#) to track progress of the feature that is slated for Spark 2.0.0.

Note

The feature is also called **Streaming Spark SQL Query**.

It works with Sources and Sinks.

ContinuousQueryManager

Note

`ContinuousQueryManager` is an experimental feature of Spark 2.0.0.

A `continuousQueryManager` is the Management API for Continuous Queries to manage `ContinuousQuery` instances per single `SQLContext`.

You can access `ContinuousQueryManager` for the current session using `SQLContext.streams` method. It is lazily created when a `SQLContext` instance starts.

```
scala> val queries = sqlContext.streams
queries: org.apache.spark.sql.ContinuousQueryManager = org.apache.spark.sql.ContinuousQue
```

Note

There is a single `ContinuousQueryManager` instance per `SQLContext` session.

Return All Active Continuous Queries per SQLContext

```
active: Array[ContinuousQuery]
```

`active` method returns a collection of `ContinuousQuery` instances for the current `SQLContext`.

Getting Active Continuous Query By Name

```
get(name: String): ContinuousQuery
```

`get` method returns a `ContinuousQuery` by `name`.

It may throw an `IllegalArgumentException` when no `ContinuousQuery` exists for the `name`.

```
java.lang.IllegalArgumentException: There is no active query with name hello
  at org.apache.spark.sql.ContinuousQueryManager$$anonfun$get$1.apply(ContinuousQueryMana
  at org.apache.spark.sql.ContinuousQueryManager$$anonfun$get$1.apply(ContinuousQueryMana
  at scala.collection.MapLike$class.getOrElse(MapLike.scala:128)
  at scala.collection.AbstractMap.getOrElse(Map.scala:59)
  at org.apache.spark.sql.ContinuousQueryManager.get(ContinuousQueryManager.scala:58)
  ... 49 elided
```

ContinuousQueryListener

Caution	FIXME
---------	-----------------------

`ContinuousQueryListener` is an interface for listening to query life cycle events, i.e. a query start, progress and termination events.

ContinuousQuery

A `ContinuousQuery` has a name. It belongs to a single `SQLContext`.

Note

`ContinuousQuery` is a Scala trait with the only implementation being `StreamExecution`

It can be in two states: active (started) or inactive (stopped). If inactive, it may have transitioned into the state due to an `ContinuousQueryException` (that is available under `exception`).

It tracks current state of all the sources, i.e. `SourceStatus`, as `sourceStatuses`.

There could only be a single Sink for a `ContinuousQuery` with many `Source's.

`ContinuousQuery` can be stopped by `stop` or an exception.

StreamExecution

`StreamExecution` manages execution of a streaming query for a `SQLContext` and a `Sink`. It requires a `LogicalPlan` to know the `Source` objects from which records are periodically pulled down.

It starts an internal thread (`microBatchThread`) to periodically (every 10 milliseconds) poll for new records in the sources and create a batch.

Note

The time between batches - 10 milliseconds - is fixed (i.e. not configurable).

`StreamExecution` can be in three states:

- `INITIALIZED` when the instance was created.
- `ACTIVE` when batches are pulled from the sources.
- `TERMINATED` when batches were successfully processed or the query stopped.

Aggregation (GroupedData)

Note

Executing aggregation on DataFrames by means of `groupBy` is *still* an experimental feature. It is available since Apache Spark 1.3.0.

You can use [DataFrame](#) to compute aggregates over a collection of (grouped) rows.

`DataFrame` offers the following operators:

- `groupBy`
- `rollup`
- `cube`

Each method returns [GroupedData](#).

groupBy Operator

Note

The following session uses the data setup as described in [Test Setup](#) section below.

```

scala> df.show
+---+-----+-----+
|name|productId|score|
+---+-----+-----+
| aaa|     100| 0.12|
| aaa|     200| 0.29|
| bbb|     200| 0.53|
| bbb|     300| 0.42|
+---+-----+-----+

scala> df.groupBy("name").count.show
+---+-----+
|name|count|
+---+-----+
| aaa|    2|
| bbb|    2|
+---+-----+

scala> df.groupBy("name").max("score").show
+---+-----+
|name|max(score)|
+---+-----+
| aaa|    0.29|
| bbb|    0.53|
+---+-----+

scala> df.groupBy("name").sum("score").show
+---+-----+
|name|sum(score)|
+---+-----+
| aaa|    0.41|
| bbb|    0.95|
+---+-----+

scala> df.groupBy("productId").sum("score").show
+-----+-----+
|productId|      sum(score)|
+-----+-----+
|      300|        0.42|
|      100|        0.12|
|      200| 0.8200000000000001|
+-----+-----+

```

GroupedData

`GroupedData` is a result of executing

It offers the following operators to work on group of rows:

- `agg`

- `count`
- `mean`
- `max`
- `avg`
- `min`
- `sum`
- `pivot`

Test Setup

This is a setup for learning `GroupedData`. Paste it into Spark Shell using `:paste`.

```
import sqlContext.implicits._

case class Token(name: String, productId: Int, score: Double)
val data = Token("aaa", 100, 0.12) ::  
  Token("aaa", 200, 0.29) ::  
  Token("bbb", 200, 0.53) ::  
  Token("bbb", 300, 0.42) :: Nil
val df = data.toDF.cache (1)
```

1. Cache the `DataFrame` so following queries won't load data over and over again.

Joins

Caution	FIXME
---------	-----------------------

UDFs — User-Defined Functions

Caution

[FIXME](#)

udf Function (in functions object)

`org.apache.spark.sql.functions` object comes with `udf` function that defines a user-defined function using a Scala function.

```
udf(f: AnyRef, dataType: DataType): UserDefinedFunction
```

Note

`udf` function is a feature of Spark 2.0.0.

Windows in DataFrames

- Window-based framework since 1.4
- Window function support in Spark SQL
- Define a (row) window to execute aggregations on
- Operate on a group of rows to return a single value for every input row
- Before 1.4, merely two kinds of functions in Spark SQL to calculate a single return value:
 - **Built-in functions** or **UDFs** - `substr` or `round` - take values from a single row as input, and they generate a single return value for every input row
 - **Aggregate functions**, such as `SUM` or `MAX`, operate on a group of rows and calculate a single return value for every group.

A window specification defines the **partitioning**, **ordering**, and **frame boundaries** (see [org.apache.spark.sql.expressions.WindowSpec API](#)).

Spark SQL supports three kinds of window functions: **ranking** functions, **analytic** functions, and **aggregate** functions.

Important	Where are they defined in the code?
-----------	-------------------------------------

Table 1. Window functions in Spark SQL (see [Introducing Window Functions in Spark SQL](#))

	SQL	DataFrame API
Ranking functions	rank	rank
	dense_rank	denseRank
	percent_rank	percentRank
	ntile	ntile
	row_number	rowNumber
Analytic functions	cume_dist	cumeDist
	first_value	firstValue
	last_value	lastValue
	lag	lag
	lead	lead

For aggregate functions, users can use any existing aggregate function as a window function.

Window functions

You can mark a function *window* by the `OVER` clause after a supported function in SQL, e.g. `avg(revenue) OVER (...)`; or the `over` method on a supported function in the [DataFrame API](#), e.g. `rank().over(...)`.

Window Specification

Any window function needs a **Window specification**. A window specification defines which rows are included in the frame associated with a given input row.

A window specification includes three parts:

1. **Partitioning Specification** controls which rows will be in the same partition with the given row. Also, the user might want to make sure all rows having the same value for the category column are collected to the same machine before ordering and calculating

the frame. If no partitioning specification is given, then all data must be collected to a single machine.

2. **Ordering Specification** controls the way that rows in a partition are ordered, determining the position of the given row in its partition.
3. **Frame Specification** (unsupported in Hive; see [Why do Window functions fail with "Window function X does not take a frame specification"?](#)) states which rows are included in the frame for the current input row, based on their relative position to the current row. For example, “*the three rows preceding the current row to the current row*” describes a frame including the current input row and three rows appearing before the current row.

Frames

At its core, a window function calculates a return value for every input row of a table based on a group of rows, called the **frame**. Every input row can have a unique frame associated with it.

When you define a frame you have to specify three components of a frame specification - the **start and end boundaries**, and the **type**.

Types of boundaries (two positions and three offsets):

- UNBOUNDED PRECEDING - the first row of the partition
- UNBOUNDED FOLLOWING - the last row of the partition
- CURRENT ROW
- <value> PRECEDING
- <value> FOLLOWING

Offsets specify the offset from the current input row.

Types of frames:

- ROW - based on *physical offsets* from the position of the current input row
- RANGE - based on *logical offsets* from the position of the current input row

Examples

Two samples from [org.apache.spark.sql.expressions.Window scaladoc](#):

```
// PARTITION BY country ORDER BY date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
Window.partitionBy("country").orderBy("date").rowsBetween(Long.MinValue, 0)
```

```
// PARTITION BY country ORDER BY date ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING
Window.partitionBy("country").orderBy("date").rowsBetween(-3, 3)
```

Important	Present input data as a colorful table (headers, rows) and ask the question to be answered in an example.
-----------	---

Computing the first and second best-sellers in category

Note	This example is borrowed from an excellent article Introducing Window Functions in Spark SQL .
------	--

Table 2. Table PRODUCT_REVENUE

product	category	revenue
Thin	cell phone	6000
Normal	tablet	1500
Mini	tablet	5500
Ultra thin	cell phone	5000
Very thin	cell phone	6000
Big	tablet	2500
Bendable	cell phone	3000
Foldable	cell phone	3000
Pro	tablet	4500
Pro2	tablet	6500

Question: What are the best-selling and the second best-selling products in every category?

```

scala> val schema = Seq("product", "category", "revenue")
schema: Seq[String] = List(product, category, revenue)

scala> val data = Seq(
|   ("Thin",      "cell phone", 6000),
|   ("Normal",    "tablet",     1500),
|   ("Mini",      "tablet",     5500),
|   ("Ultra thin", "cell phone", 5000),
|   ("Very thin",  "cell phone", 6000),
|   ("Big",        "tablet",     2500),
|   ("Bendable",   "cell phone", 3000),
|   ("Foldable",   "cell phone", 3000),
|   ("Pro",        "tablet",     4500),
|   ("Pro2",       "tablet",     6500)
| )
data: Seq[(String, String, Int)] = List((Thin,cell phone,6000), (Normal,tablet,1500), (Mi

scala> val df = sc.parallelize(data).toDF(schema: _*)
df: org.apache.spark.sql.DataFrame = [product: string, category: string, revenue: int]

scala> df.select('*).show
+-----+-----+-----+
| product| category|revenue|
+-----+-----+-----+
|     Thin|cell phone| 6000|
|   Normal|    tablet| 1500|
|     Mini|    tablet| 5500|
|Ultra thin|cell phone| 5000|
| Very thin|cell phone| 6000|
|      Big|    tablet| 2500|
| Bendable|cell phone| 3000|
| Foldable|cell phone| 3000|
|       Pro|    tablet| 4500|
|      Pro2|    tablet| 6500|
+-----+-----+-----+

scala> df.where(df("category") === "tablet").show
+-----+-----+-----+
|product|category|revenue|
+-----+-----+-----+
| Normal|    tablet| 1500|
|   Mini|    tablet| 5500|
|     Big|    tablet| 2500|
|     Pro|    tablet| 4500|
|    Pro2|    tablet| 6500|
+-----+-----+-----+

```

The question boils down to ranking products in a category based on their revenue, and to pick the best selling and the second best-selling products based the ranking.

```

scala> import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.expressions.Window

scala> val overCategory = Window.partitionBy("category").orderBy(desc("revenue"))
overCategory: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressi

// or alternatively using $
scala> val overCategory = Window.partitionBy("category").orderBy($"revenue".desc)
overCategory: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressi

```

ImportantExplain the difference between `desc` and `$...desc`.

```

scala> val rank = denseRank.over(overCategory)
rank: org.apache.spark.sql.Column = 'dense_rank() WindowSpecDefinition UnspecifiedFrame

scala> val ranked = df.withColumn("rank", rank)

scala> ranked.show
+-----+-----+-----+----+
| product| category|revenue|rank|
+-----+-----+-----+----+
|    Pro2|   tablet|  6500|  1|
|     Mini|   tablet|  5500|  2|
|      Pro|   tablet|  4500|  3|
|      Big|   tablet|  2500|  4|
|  Normal|   tablet|  1500|  5|
|     Thin|cell phone|  6000|  1|
| Very thin|cell phone|  6000|  1|
|Ultra thin|cell phone|  5000|  2|
| Bendable|cell phone|  3000|  3|
| Foldable|cell phone|  3000|  3|
+-----+-----+-----+----+

scala> ranked.where(ranked("rank") <= 2).show
+-----+-----+-----+----+
| product| category|revenue|rank|
+-----+-----+-----+----+
|    Pro2|   tablet|  6500|  1|
|     Mini|   tablet|  5500|  2|
|     Thin|cell phone|  6000|  1|
| Very thin|cell phone|  6000|  1|
|Ultra thin|cell phone|  5000|  2|
+-----+-----+-----+----+

```

Computing the first and second best-sellers in category

Note

This example is the 2nd example from an excellent article [Introducing Window Functions in Spark SQL](#).

```
scala> import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.expressions.Window

scala> val overCategory = Window.partitionBy("category").orderBy($"revenue".desc)
overCategory: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.Window@43333333

scala> val reveDiff = max(df("revenue")).over(overCategory) - df("revenue")
reveDiff: org.apache.spark.sql.Column = ('max(revenue) WindowSpecDefinition UnspecifiedFrame)

scala> df.select('*', reveDiff as "revenue_difference").show
+-----+-----+-----+
| product| category|revenue|revenue_difference|
+-----+-----+-----+
|     Pro2|    tablet|   6500|          0|
|      Mini|    tablet|   5500|        1000|
|       Pro|    tablet|   4500|        2000|
|       Big|    tablet|   2500|        4000|
|  Normal|    tablet|   1500|        5000|
|     Thin|cell phone|  6000|          0|
| Very thin|cell phone|  6000|          0|
|Ultra thin|cell phone|  5000|        1000|
| Bendable|cell phone|  3000|        3000|
| Foldable|cell phone|  3000|        3000|
+-----+-----+-----+
```

Compute difference on column

Compute a difference between values in rows in a column.

```
scala> val pairs = (1 to 10).zip(10 to 100 by 10).flatMap(x => Seq(x, x))
pairs: scala.collection.immutable.IndexedSeq[(Int, Int)] = Vector((1,10), (1,10), (2,20), (2,20), (3,30), (3,30), (4,40), (4,40), (5,50), (5,50), (6,60), (6,60), (7,70), (7,70), (8,80), (8,80), (9,90), (9,90), (10,100), (10,100))

scala> val df = sc.parallelize(pairs).toDF("ns", "tens")
df: org.apache.spark.sql.DataFrame = [ns: int, tens: int]

scala> df.show
+---+---+
| ns|tens|
+---+---+
|  1|  10|
|  1|  10|
|  2|  20|
|  2|  20|
|  3|  30|
|  3|  30|
```

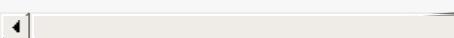
```
| 4| 40|
| 4| 40|
| 5| 50|
| 5| 50|
| 6| 60|
| 6| 60|
| 7| 70|
| 7| 70|
| 8| 80|
| 8| 80|
| 9| 90|
| 9| 90|
| 10| 100|
| 10| 100|
+---+---+
```

```
scala> import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.expressions.Window

scala> val overNs = Window.partitionBy("ns").orderBy("tens")
overNs: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.Wi

scala> val diff = lead(df("tens"), 1).over(overNs)
diff: org.apache.spark.sql.Column = 'lead(tens,0,null) WindowSpecDefinition ROWS BETWEEN

scala> df.withColumn("diff", diff - df("tens")).show
+---+---+---+
| ns|tens|diff|
+---+---+---+
| 1| 10| 0|
| 1| 10|null|
| 2| 20| 0|
| 2| 20|null|
| 3| 30| 0|
| 3| 30|null|
| 4| 40| 0|
| 4| 40|null|
| 5| 50| 0|
| 5| 50|null|
| 6| 60| 0|
| 6| 60|null|
| 7| 70| 0|
| 7| 70|null|
| 8| 80| 0|
| 8| 80|null|
| 9| 90| 0|
| 9| 90|null|
| 10| 100| 0|
| 10| 100|null|
+---+---+---+
```



Please note that [Why do Window functions fail with "Window function X does not take a frame specification"?](#)

The key here is to remember that DataFrames are RDDs under the covers and hence aggregation like grouping by a key in DataFrames is RDD's `groupByKey` (or worse, `reduceByKey` or `aggregateByKey` transformations).

Accessing values of earlier rows

[FIXME](#) What's the value of rows before current one?

Calculate rank of row

Calculate moving average

Calculate cumulative sum

Interval data type for Date and Timestamp types

See [\[SPARK-8943\] CalendarIntervalType for time intervals.](#)

With the Interval data type, you could use intervals as values specified in `<value> PRECEDING` and `<value> FOLLOWING` for `RANGE` frame. It is specifically suited for time-series analysis with window functions.

User-defined aggregate functions

See [\[SPARK-3947\] Support Scala/Java UDAF.](#)

With the window function support, you could use user-defined aggregate functions as window functions.

Catalyst optimizer

Review [sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/Optimizer.scala](https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/Optimizer.scala).

Into the depths

- code generation
- memory management

Datasets vs RDDs

Many may have been asking yourself why they should be using Datasets rather than the foundation of all Spark - RDDs using case classes.

This document collects advantages of `Dataset` vs `RDD[CaseClass]` to answer the question [Dan has asked on twitter](#):

"In #Spark, what is the advantage of a DataSet over an RDD[CaseClass]?"

Saving to or Writing from Data Sources

In Datasets, reading or writing boils down to using `SQLContext.read` or `SQLContext.write` methods, appropriately.

Accessing Fields / Columns

You `select` columns in a datasets without worrying about the positions of the columns.

In RDD, you have to do an additional hop over a case class and access fields by name.

Settings

The following list are the settings used to configure Spark SQL applications.

- `spark.sql.allowMultipleContexts` (default: `true`) controls whether creating multiple SQLContexts/HiveContexts is allowed.

Spark MLlib

Caution	I'm new to Machine Learning as a discipline and Spark MLlib in particular so mistakes in this document are considered a norm (not an exception).
---------	--

Spark MLlib is a module or a library of Apache Spark to provide distributed machine learning algorithms on top of Spark. Its goal is to simplify the development and usage of large scale machine learning.

You can find the following types of machine learning algorithms in MLlib:

- Classification
- Regression
- Frequent itemsets (via [FP-growth Algorithm](#))
- Recommendation
- Feature extraction and selection
- Clustering
- Statistics
- Linear Algebra

You can also do the following using MLlib:

- Model import and export
- [Pipelines](#)

Machine Learning uses large datasets to identify patterns and make decisions (aka *predictions*). Automated decision making is what makes Machine Learning so appealing.

The amount of data (measured in TB or PB) is what makes Spark MLlib especially important since a human could not possibly extract much value from the dataset in a short time.

Spark handles data distribution and makes the huge data available by means of [RDDs](#), [DataFrames](#), and recently [Datasets](#).

Use cases for Machine Learning (and hence Spark MLlib that comes with appropriate algorithms):

- Marketing and Advertising Optimization
- Security Monitoring and Fraud Detection

- Operational Optimizations

FP-growth Algorithm

Spark 1.5 have significantly improved on frequent pattern mining capabilities with new algorithms for association rule generation and sequential pattern mining.

- **Frequent Itemset Mining** using the **Parallel FP-growth** algorithm (since Spark 1.3)
 - [Frequent Pattern Mining in MLlib User Guide](#)
 - **frequent pattern mining**
 - reveals the most frequently visited site in a particular period
 - finds popular routing paths that generate most traffic in a particular region
 - models its input as a set of **transactions**, e.g. a path of nodes.
 - A transaction is a set of **items**, e.g. network nodes.
 - the algorithm looks for common **subsets of items** that appear across transactions, e.g. sub-paths of the network that are frequently traversed.
 - A naive solution: generate all possible itemsets and count their occurrence
 - A subset is considered **a pattern** when it appears in some minimum proportion of all transactions - **the support**.
 - the items in a transaction are unordered
 - analyzing traffic patterns from network logs
 - the algorithm finds all frequent itemsets without generating and testing all candidates
- suffix trees (FP-trees) constructed and grown from filtered transactions
- Also available in Mahout, but slower.
- Distributed generation of [association rules](#) (since Spark 1.5).
 - in a retailer's transaction database, a rule `{toothbrush, floss} → {toothpaste}` with a confidence value `0.8` would indicate that `80%` of customers who buy a toothbrush and floss also purchase a toothpaste in the same transaction. The retailer could then use this information, put both toothbrush and floss on sale, but raise the price of toothpaste to increase overall profit.
 - [FPGrowth](#) model

- **parallel sequential pattern mining** (since Spark 1.5)
 - **PrefixSpan** algorithm with modifications to parallelize the algorithm for Spark.
 - extract frequent sequential patterns like routing updates, activation failures, and broadcasting timeouts that could potentially lead to customer complaints and proactively reach out to customers when it happens.

Power Iteration Clustering

- since Spark 1.3
- unsupervised learning including clustering
- identifying similar behaviors among users or network clusters
- **Power Iteration Clustering (PIC)** in MLlib, a simple and scalable graph clustering method
 - [PIC in MLlib User Guide](#)
 - `org.apache.spark.mllib.clustering.PowerIterationClustering`
 - a graph algorithm
 - Among the first MLlib algorithms built upon [GraphX](#).
 - takes an undirected graph with similarities defined on edges and outputs clustering assignment on nodes
 - uses truncated [power iteration](#) to find a very low-dimensional embedding of the nodes, and this embedding leads to effective graph clustering.
 - stores the normalized similarity matrix as a graph with normalized similarities defined as edge properties
 - The edge properties are cached and remain static during the power iterations.
 - The embedding of nodes is defined as node properties on the same graph topology.
 - update the embedding through power iterations, where `aggregateMessages` is used to compute matrix-vector multiplications, the essential operation in a power iteration method
 - k-means is used to cluster nodes using the embedding.
 - able to distinguish clearly the degree of similarity – as represented by the Euclidean distance among the points – even though their relationship is non-linear

LabeledPoint

Caution **FIXME**

`LabeledPoint` is a convenient class for declaring a schema for DataFrames that are used as input data for [Linear Regression](#) algorithm in Spark MLlib.

LinearRegression

`LinearRegression` class represents the linear regression algorithm in machine learning.

Note The class belongs to org.apache.spark.ml.regression package.

Example

```
$ bin/spark-shell
...
SQL context available as sqlContext.
Welcome to

   _/ \
  /  \_ _ \_ _ _ / / \
 _\ \ \_ \ \_ ` \_ / \_ / \
/_ / . / \_, / / / \_ \ version 2.0.0-SNAPSHOT
 /_ /


Using Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_74)
Type in expressions to have them evaluated.
Type :help for more information.

scala> import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.Vectors

scala> import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LabeledPoint

scala> val data = (0 to 9).map(_.toDouble).map(n => (n, n)).map { case (label, feature) =
  data: org.apache.spark.sql.DataFrame = [label: double, features: vector]

scala> data.show
+-----+
|label|features|
+-----+
|  0.0|  [0.0]|
|  1.0|  [1.0]|
|  2.0|  [2.0]|
|  3.0|  [3.0]|
|  4.0|  [4.0]|
|  5.0|  [5.0]|
```

```

| 6.0| [6.0]|
| 7.0| [7.0]|
| 8.0| [8.0]|
| 9.0| [9.0]|
+---+-----+

```

```

scala> import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.regression.LinearRegression

scala> val lr = new LinearRegression
lr: org.apache.spark.ml.regression.LinearRegression = linReg_c227301cf2c3

scala> val model = lr.fit(data)
16/03/04 10:07:45 WARN WeightedLeastSquares: regParam is zero, which might cause numerical
16/03/04 10:07:45 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib
16/03/04 10:07:45 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib
16/03/04 10:07:45 WARN LAPACK: Failed to load implementation from: com.github.fommil.netl
16/03/04 10:07:45 WARN LAPACK: Failed to load implementation from: com.github.fommil.netl
model: org.apache.spark.ml.regression.LinearRegressionModel = linReg_c227301cf2c3

scala> model.intercept
res1: Double = 0.0

scala> model.coefficients
res2: org.apache.spark.mllib.linalg.Vector = [1.0]

// make predictions
scala> val predictions = model.transform(data)
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 more

scala> predictions.show
+---+-----+-----+
|label|features|prediction|
+---+-----+-----+
| 0.0| [0.0]| 0.0|
| 1.0| [1.0]| 1.0|
| 2.0| [2.0]| 2.0|
| 3.0| [3.0]| 3.0|
| 4.0| [4.0]| 4.0|
| 5.0| [5.0]| 5.0|
| 6.0| [6.0]| 6.0|
| 7.0| [7.0]| 7.0|
| 8.0| [8.0]| 8.0|
| 9.0| [9.0]| 9.0|
+---+-----+-----+

scala> import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.evaluation.RegressionEvaluator

// rmse is the default metric
// We're explicit here for learning purposes
scala> val evaluator = new RegressionEvaluator().setMetricName("rmse")

```

```

evaluator: org.apache.spark.ml.evaluation.RegressionEvaluator = regEval_bed1f4fc840d

scala> val rmse = evaluator.evaluate(predictions)
rmse: Double = 0.0

scala> println(s"Root Mean Squared Error: $rmse")
Root Mean Squared Error: 0.0

scala> import org.apache.spark.mllib.linalg.DenseVector
import org.apache.spark.mllib.linalg.DenseVector

scala> predictions.rdd.map(r => (r(0).asInstanceOf[Double], r(1).asInstanceOf[DenseVector]
+----+-----+-----+
|label|feature0|prediction|
+----+-----+-----+
| 0.0|    0.0|      0.0|
| 1.0|    1.0|      1.0|
| 2.0|    2.0|      2.0|
| 3.0|    3.0|      3.0|
| 4.0|    4.0|      4.0|
| 5.0|    5.0|      5.0|
| 6.0|    6.0|      6.0|
| 7.0|    7.0|      7.0|
| 8.0|    8.0|      8.0|
| 9.0|    9.0|      9.0|
+----+-----+-----+

// Let's make it nicer to the eyes using a Scala case class
scala> :pa
// Entering paste mode (ctrl-D to finish)

import org.apache.spark.sql.Row
import org.apache.spark.mllib.linalg.DenseVector
case class Prediction(label: Double, feature0: Double, prediction: Double)
object Prediction {
  def apply(r: Row) = new Prediction(
    label = r(0).asInstanceOf[Double],
    feature0 = r(1).asInstanceOf[DenseVector](0).toDouble,
    prediction = r(2).asInstanceOf[Double])
}

// Exiting paste mode, now interpreting.

import org.apache.spark.sql.Row
import org.apache.spark.mllib.linalg.DenseVector
defined class Prediction
defined object Prediction

scala> predictions.rdd.map(Prediction.apply).toDF.show
+----+-----+-----+
|label|feature0|prediction|
+----+-----+-----+
| 0.0|    0.0|      0.0|

```

```
| 1.0| 1.0| 1.0|
| 2.0| 2.0| 2.0|
| 3.0| 3.0| 3.0|
| 4.0| 4.0| 4.0|
| 5.0| 5.0| 5.0|
| 6.0| 6.0| 6.0|
| 7.0| 7.0| 7.0|
| 8.0| 8.0| 8.0|
| 9.0| 9.0| 9.0|
+---+---+---+
```



Further reading

- [Improved Frequent Pattern Mining in Spark 1.5: Association Rules and Sequential Patterns](#)
- [New MLlib Algorithms in Spark 1.3: FP-Growth and Power Iteration Clustering](#)

ML Pipelines - High-Level API for MLlib

Note

Both [scikit-learn](#) and [GraphLab](#) have the concept of **pipelines** built into their system.

Use of a machine learning algorithm is only one component of a **predictive analytic workflow**. There can also be **pre-processing steps** for the machine learning algorithm to work.

A typical standard machine learning workflow is to:

1. Load the data
2. Extract features (aka *feature extraction*)
3. Train model
4. Evaluate (or *predictionize*)

The goal of the **Pipeline API** (aka `spark.ml`) is to let users quickly and easily assemble and configure practical machine learning pipelines (aka workflows).

Note

The Pipeline API lives under [org.apache.spark.ml](#) package.

Concepts:

- [Pipelines](#) and [PipelineStages](#)
- [Transformers](#)
 - [UnaryTransformers](#)
- [Estimators](#)
- [Evaluators](#)
- [Models](#)

The beauty of using Spark MLlib is that the ML dataset is simply a [DataFrame](#).

Example: In text classification, preprocessing steps like n-gram extraction, and TF-IDF feature weighting are often necessary before training of a classification model like an SVM.

Upon deploying a model, your system must not only know the SVM weights to apply to input features, but also transform raw data into the format the model is trained on.

- Pipeline for text categorization

- Pipeline for image classification

Pipelines are like a query plan in a database system.

Components of ML Pipeline:

- **Pipeline Construction Framework** – A DSL for the construction of pipelines that includes concepts of **Nodes** and **Pipelines**.
 - Nodes are data transformation steps (**transformers**)
 - Pipelines are a DAG of Nodes.

Pipelines become objects that can be saved out and applied in real-time to new data.

It can help creating domain-specific feature transformers, general purpose transformers, statistical utilities and nodes.

You could eventually `save` or `load` machine learning components as described in [Persisting Machine Learning Components](#).

Note	A machine learning component is any object that belongs to Pipeline API, e.g. Pipeline , LinearRegressionModel , etc.
------	--

Features of Pipeline API

The features of the Pipeline API in Spark MLlib:

- [DataFrame](#) as a dataset format
- ML Pipelines API is similar to [scikit-learn](#)
- Easy debugging (via inspecting columns added during execution)
- Parameter tuning
- Compositions (to build more complex pipelines out of existing ones)

Pipelines

A **ML pipeline** (or a **ML workflow**) is a sequence of [Transformers](#) and [Estimators](#) to build a model out of input dataset.

A pipeline is represented by [Pipeline class](#).

```
import org.apache.spark.ml.Pipeline
```

`Pipeline` is an [Estimator](#) (so it is acceptable to set up a `Pipeline` with other `Pipeline` instances).

The `Pipeline` object can `read` or `load` pipelines (refer to [Persisting Machine Learning Components](#) page).

```
read: MLReader[Pipeline]
load(path: String): Pipeline
```

You can create a `Pipeline` with an optional `uid` identifier. It is of the format `pipeline_[randomUid]` when unspecified.

```
scala> val pipeline = new Pipeline()
pipeline: org.apache.spark.ml.Pipeline = pipeline_54bf3f5002e2

scala> pipeline.uid
res1: String = pipeline_54bf3f5002e2

scala> val pipeline = new Pipeline("my_pipeline")
pipeline: org.apache.spark.ml.Pipeline = my_pipeline

scala> pipeline.uid
res2: String = my_pipeline
```

The identifier `uid` is used to create an instance of [PipelineModel](#) to return from `fit(dataset: DataFrame): PipelineModel` method.

```
scala> val pipeline = new Pipeline("my_pipeline")
pipeline: org.apache.spark.ml.Pipeline = my_pipeline

scala> val df = sc.parallelize(0 to 9).toDF("num")
df: org.apache.spark.sql.DataFrame = [num: int]

scala> val model = pipeline.setStages(Array()).fit(df)
model: org.apache.spark.ml.PipelineModel = my_pipeline
```

The `stages` mandatory parameter can be set using `setStages(value: Array[PipelineStage]): this.type` method.

Pipeline Fitting (fit method)

```
fit(dataset: DataFrame): PipelineModel
```

The `fit` method returns a [PipelineModel](#) that holds a collection of `Transformer` objects that are results of `Estimator.fit` method for every `Estimator` in the Pipeline (with possibly-modified `dataset`) or simply input `Transformer` objects. The input `dataset` [DataFrame](#) is passed to `transform` for every `Transformer` instance in the Pipeline.

It first transforms the schema of the input `dataset` [DataFrame](#).

It then searches for the index of the last `Estimator` to calculate `Transformers` for `Estimator` and simply return `Transformers` back up to the index in the pipeline. For each `Estimator` the `fit` method is called with the input `dataset`. The result [DataFrame](#) is passed to the next `Transformer` in the chain.

Note

An `IllegalArgumentException` exception is thrown when a stage is neither `Estimator` or `Transformer`.

`transform` method is called for every `Transformer` calculated but the last one (that is the result of executing `fit` on the last `Estimator`).

The calculated `Transformers` are collected.

After the last `Estimator` there can only be `Transformer` stages.

The method returns a `PipelineModel` with `uid` and `transformers`. The parent `Estimator` is the `Pipeline` itself.

PipelineStage

The [PipelineStage](#) abstract class represents a single stage in a [Pipeline](#).

`PipelineStage` has the following direct implementations (of which few are abstract classes, too):

- [Estimator](#)
- [Model](#)
- [Pipeline](#)
- [Predictor](#)
- [Transformer](#)

Each `PipelineStage` transforms schema using `transformSchema` family of methods:

```
transformSchema(schema: StructType): StructType
transformSchema(schema: StructType, logging: Boolean): StructType
```

Note	<code>StructType</code> is a Spark SQL type. Read up on it in Traits of DataFrame .
------	---

Tip	Enable <code>DEBUG</code> logging level for the respective <code>PipelineStage</code> implementations to see what happens beneath.
-----	--

Transformers

A **transformer** is a function that maps a `DataFrame` into another `DataFrame`.

Transformers are instances of [org.apache.spark.ml.Transformer](#) abstract class that offers `transform` family of methods:

```
transform(dataset: DataFrame): DataFrame
transform(dataset: DataFrame, paramMap: ParamMap): DataFrame
transform(dataset: DataFrame, firstParamPair: ParamPair[_], otherParamPairs: ParamPair[_])
```

A `Transformer` is a [PipelineStage](#) (so it can be a part of a [Pipeline](#)).

The direct descendants of the `Transformer` abstract class are:

- [Model](#)
- [UnaryTransformers](#)

UnaryTransformers

The [UnaryTransformer](#) abstract class is a specialized `Transformer` that applies transformation to one input column and writes results to another (by appending a new column).

Each `UnaryTransformer` defines the input and output columns using the following "chain" methods (they return the transformer on which they were executed):

- `setInputCol(value: String)`
- `setOutputCol(value: String)`

Each `UnaryTransformer` calls `validateInputType` while executing `transformSchema(schema: StructType)` (that is part of [PipelineStage](#) contract).

Note	A <code>UnaryTransformer</code> is a <code>PipelineStage</code> .
------	---

When `transform` is called, it first calls `transformSchema` (with `DEBUG` logging enabled) and then adds the column as a result of calling a protected abstract `createTransformFunc`.

Note

`createTransformFunc` function is abstract and defined by concrete `UnaryTransformer` objects.

Internally, `transform` methods uses Spark SQL's `udf` to define a function (based on `createTransformFunc` function described above) that will create the new output column (with appropriate `outputDataType`). The UDF is later applied to the input column of the input `DataFrame` and the result becomes the output column (using `DataFrame.withColumn` method).

Note

Using `udf` and `withColumn` methods from Spark SQL demonstrates integration between the Spark modules: MLlib and SQL.

Examples of Transformers

One example of a transformer is [org.apache.spark.ml.feature.RegexTokenizer](#).

```
scala> import org.apache.spark.ml.feature.RegexTokenizer
scala> val regexTok = new RegexTokenizer().setInputCol("text").setOutputCol("words").setP
regexTok: org.apache.spark.ml.feature.RegexTokenizer = regexTok_31b044abd10c
```

Another example of a transformer could be [org.apache.spark.ml.feature.HashingTF](#).

```
scala> val hashingTF = new HashingTF().setInputCol("text").setOutputCol("features").setNu
hashingTF: org.apache.spark.ml.feature.HashingTF = hashingTF_16ecd3b7e333
```

In this example you use [org.apache.spark.ml.feature.NGram](#) that converts the input collection of strings into a collection of n-grams (of `n` words).

```
import org.apache.spark.ml.feature.NGram

val bigram = new NGram("bigrams")
val df = Seq((0, Seq("hello", "world"))).toDF("id", "tokens")
bigram.setInputCol("tokens").transform(df).show

+---+-----+-----+
| id|      tokens|bigrams__output|
+---+-----+-----+
|  0|[hello, world]| [hello world]|
+---+-----+-----+
```

Estimators

An **estimator** takes a `DataFrame` and parameters (as `ParamMap`) and fits a model. It is a function that maps a `DataFrame` into a `Model` that takes a `DataFrame`, trains on it and produces a `Model`.

Caution	FIXME What does <i>fitting a model</i> mean?
---------	--

It is a [PipelineStage](#) (and so can be a part of [Pipeline](#)).

Evaluators

A **evaluator** is a function that maps a `DataFrame` into a metric indicating how well the model is.

Models

`Model` abstract class is a [Transformer](#) with the optional [Estimator](#) that has produced it (as a transient `parent` field).

Note	Estimator is optional.
------	--

Caution	FIXME What does it mean when a Estimator is not known? When could an Estimator be missing?
---------	--

Caution	FIXME What does a fitted model mean? What are the other kinds of models?
---------	--

There are two direct implementations of the `Model` class that are not directly related to a ML algorithm:

- [PipelineModel](#)
- [PredictionModel](#)

PipelineModel

Caution	<code>PipelineModel</code> is a <code>private[ml]</code> class so perhaps of less interest to end users like me (as of today).
---------	--

Caution	FIXME
---------	-----------------------

PredictionModel

`PredictionModel` is an abstract model for prediction algorithms like regression and classification (that have their own specialized models).

The direct non-algorithm-specific extensions of `PredictionModel` are:

- `ClassificationModel`
- `RegressionModel`

LinearRegressionModel

Caution

FIXME

Further reading or watching

- [ML Pipelines](#)
- [ML Pipelines: A New High-Level API for MLlib](#)
- (video) [Building, Debugging, and Tuning Spark Machine Learning Pipelines - Joseph Bradley \(Databricks\)](#)

Persisting Machine Learning Components

[MLWriter](#) and [MLReader](#) belong to `org.apache.spark.ml.util` package.

MLWriter

`MLWriter` abstract class comes with `save(path: String)` method to save a machine learning component to a given `path`.

```
save(path: String): Unit
```

It comes with another (chainable) method `overwrite` to overwrite the output path if it already exists.

```
overwrite(): this.type
```

The component is saved into a JSON file (see [MLWriter Example](#) section below).

Tip

Enable `INFO` logging level for the `MLWriter` implementation logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.ml.Pipeline$.PipelineWriter=INFO
```

Refer to [Logging](#).

Caution

FIXME The logging doesn't work and overwriting does not print out INFO message to the logs :(

MLWriter Example

```
import org.apache.spark.ml._  
val pipeline = new Pipeline().setStages(Array())  
pipeline.write.overwrite().save("hello-pipeline")
```

The result of `save` is a JSON file (as shown below).

```
$ cat hello-pipeline/metadata/part-00000 | jq
{
  "class": "org.apache.spark.ml.Pipeline",
  "timestamp": 1457685293319,
  "sparkVersion": "2.0.0-SNAPSHOT",
  "uid": "pipeline_12424a3716b2",
  "paramMap": {
    "stageUids": []
  }
}
```

MLReader

`MLReader` abstract class comes with `load(path: String)` method to `load` a machine learning component from a given `path`.

```
import org.apache.spark.ml._
val pipeline = Pipeline.read.load("hello-pipeline")
```

Example — Text Classification

Note

The example was inspired by the video [Building, Debugging, and Tuning Spark Machine Learning Pipelines - Joseph Bradley \(Databricks\)](#).

Problem: Given a text document, classify it as a scientific or non-scientific one.

When loading the input data it usually becomes a `DataFrame`.

Note

The example uses a case class `LabeledText` to have the schema described nicely.

```
import sqlContext.implicits._

sealed trait Category
case object Scientific extends Category
case object NonScientific extends Category

// FIXME: Define schema for Category

case class LabeledText(id: Long, category: Category, text: String)

val data = Seq(LabeledText(0, Scientific, "hello world"), LabeledText(1, NonScientific, "witaj swiecie"))

scala> data.show
+---+-----+
|label|      text|
+---+-----+
|    0| hello world|
|    1|witaj swiecie|
+---+-----+
```

It is then *tokenized* and transformed into another `DataFrame` with an additional column called `features` that is a `vector` of numerical values.

Note

Paste the code below into Spark Shell using `:paste` mode.

```
import sqlContext.implicits._

case class Article(id: Long, topic: String, text: String)
val articles = Seq(Article(0, "sci.math", "Hello, Math!"),
Article(1, "alt.religion", "Hello, Religion!"),
Article(2, "sci.physics", "Hello, Physics!")).toDF

org.apache.spark.sql.catalyst.encoders.OuterScopes.addOuterScope(this) (1)

val papers = articles.as[Article]
```

1. The line is required due to the way Spark Shell and Datasets interact. See [Dataset](#) for more coverage.

Now, the tokenization part comes that maps the input text of each text document into tokens (a `seq[String]`) and then into a `Vector` of numerical values that can only then be understood by a machine learning algorithm (that operates on `Vector` instances).

```
scala> papers.show
+---+-----+-----+
| id|      topic|      text|
+---+-----+-----+
| 0|sci.math|Hello, Math!|
| 1|alt.religion|Hello, Religion!|
| 2|sci.physics|Hello, Physics!|
+---+-----+-----+

// FIXME Use Dataset API (not DataFrame API)
val labelled = papers.toDF.withColumn("label", $"topic".like("sci%")).cache

val topic2Label: Boolean => Double = isSci => if (isSci) 1 else 0
val toLabel = udf(topic2Label)

val training = papers.toDF.withColumn("label", toLabel($"topic".like("sci%"))).cache

scala> training.show
+---+-----+-----+-----+
| id|      topic|      text|label|
+---+-----+-----+-----+
| 0|sci.math|Hello, Math!| 1.0|
| 1|alt.religion|Hello, Religion!| 0.0|
| 2|sci.physics|Hello, Physics!| 1.0|
+---+-----+-----+-----+

scala> training.groupBy("label").count.show
+-----+-----+
|label|count|
+-----+-----+
| 0.0|    1|
| 1.0|    2|
+-----+-----+
```

The *train a model* phase uses the logistic regression machine learning algorithm to build a model and predict `label` for future input text documents (and hence classify them as scientific or non-scientific).

```
scala> import org.apache.spark.ml.feature.RegexTokenizer

scala> val tokenizer = new RegexTokenizer().setInputCol("text").setOutputCol("words").set
tokenizer: org.apache.spark.ml.feature.RegexTokenizer = regexTok_f5a01fb6646a

scala> import org.apache.spark.ml.feature.HashingTF
import org.apache.spark.ml.feature.HashingTF

scala> val hashingTF = new HashingTF().setInputCol(tokenizer.getOutputCol).setOutputCol("w
hashingTF: org.apache.spark.ml.feature.HashingTF = hashingTF_152427802099

scala> import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.classification.LogisticRegression

scala> val lr = new LogisticRegression().setMaxIter(20).setRegParam(0.01)
lr: org.apache.spark.ml.classification.LogisticRegression = logreg_c346fddce901

scala> import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.Pipeline

scala> val pipeline = new Pipeline().setStages(Array(tokenizer, hashingTF, lr))
pipeline: org.apache.spark.ml.Pipeline = pipeline_c1feff10b9bb
```

It uses two columns, namely `label` and `features` vector to build a logistic regression model to make predictions.

Let's tune the model (using "tools" from `org.apache.spark.ml.tuning` package).

Caution	FIXME Review the available classes in the org.apache.spark.ml.tuning package.
---------	--

```
import org.apache.spark.ml.tuning.ParamGridBuilder
val paramGrid = new ParamGridBuilder()
  .addGrid(hashingTF.numFeatures, Array(1000, 10000))
  .addGrid(lr.regParam, Array(0.05, 0.2))
  .build

import org.apache.spark.ml.tuning.CrossValidator
import org.apache.spark.ml.param._
val cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEvaluator(evaluator)
  .setEstimatorParamMaps(paramGrid)
  .setNumFolds(2)

val cvModel = cv.fit(training)
```

Caution**FIXME Review**

<https://github.com/apache/spark/blob/master/mllib/src/test/scala/org/apache/sp>

You can eventually save the model for later use (using `DataFrame.write`).

```
cvModel.transform(test).select("id", "prediction")
  .write
  .json("/demo/predictions")
```

Example — Linear Regression

The `DataFrame` used for Linear Regression has to have `features` column of `org.apache.spark.mllib.linalg.VectorUDT` type.

Note You can change the name of the column using `featuresCol` parameter.

The list of the parameters of `LinearRegression`:

```
scala> println(lr.explainParams)
elasticNetParam: the ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the pen
featuresCol: features column name (default: features)
fitIntercept: whether to fit an intercept term (default: true)
labelCol: label column name (default: label)
maxIter: maximum number of iterations (>= 0) (default: 100)
predictionCol: prediction column name (default: prediction)
regParam: regularization parameter (>= 0) (default: 0.0)
solver: the solver algorithm for optimization. If this is not set or empty, default value
standardization: whether to standardize the training features before fitting the model (d
tol: the convergence tolerance for iterative algorithms (default: 1.0E-6)
weightCol: weight column name. If this is not set or empty, we treat all instance weights
```

Caution

FIXME The following example is work in progress.

```
import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline("my_pipeline")

import org.apache.spark.ml.regression._
val lr = new LinearRegression

val df = sc.parallelize(0 to 9).toDF("num")
val stages = Array(lr)
val model = pipeline.setStages(stages).fit(df)

// the above lines gives:
java.lang.IllegalArgumentException: requirement failed: Column features must be of type o
at scala.Predef$.require(Predef.scala:219)
at org.apache.spark.ml.util.SchemaUtils$.checkColumnType(SchemaUtils.scala:42)
at org.apache.spark.ml.PredictorParams$class.validateAndTransformSchema(Predictor.scala
at org.apache.spark.ml.Predictor.validateAndTransformSchema(Predictor.scala:72)
at org.apache.spark.ml.Predictor.transformSchema(Predictor.scala:117)
at org.apache.spark.ml.Pipeline$$anonfun$transformSchema$4.apply(Pipeline.scala:182)
at org.apache.spark.ml.Pipeline$$anonfun$transformSchema$4.apply(Pipeline.scala:182)
at scala.collection.IndexedSeqOptimized$class.foldl(IndexedSeqOptimized.scala:57)
at scala.collection.IndexedSeqOptimized$class.foldLeft(IndexedSeqOptimized.scala:66)
at scala.collection.mutable.ArrayOps$ofRef.foldLeft(ArrayOps.scala:186)
at org.apache.spark.ml.Pipeline.transformSchema(Pipeline.scala:182)
at org.apache.spark.ml.PipelineStage.transformSchema(Pipeline.scala:66)
at org.apache.spark.ml.Pipeline.fit(Pipeline.scala:133)
... 51 elided
```



GraphX

Apache Spark comes with a library for executing distributed computation on graph data, [GraphX](#).

- Apache Spark graph analytics
- GraphX is a pure programming API
 - missing a graphical UI to visually explore datasets
 - Could TitanDB be a solution?

From the article [Merging datasets using graph analytics](#):

Such a situation, in which we need to find the best matching in a weighted bipartite graph, poses what is known as the [stable marriage problem](#). It is a classical problem that has a well-known solution, the Gale–Shapley algorithm.

A popular [model of distributed computation on graphs](#) known as Pregel was published by Google researchers in 2010. Pregel is based on passing messages along the graph edges in a series of iterations. Accordingly, it is a good fit for the Gale–Shapley algorithm, which starts with each “gentleman” (a vertex on one side of the bipartite graph) sending a marriage proposal to its most preferred single “lady” (a vertex on the other side of the bipartite graph). The “ladies” then marry their most preferred suitors, after which the process is repeated until there are no more proposals to be made.

The Apache Spark distributed computation engine includes GraphX, a library specifically made for executing distributed computation on graph data. GraphX provides an elegant Pregel interface but also permits more general computation that is not restricted to the message-passing pattern.

Logging

Spark uses log4j for logging.

The valid log levels are "ALL", "DEBUG", "ERROR", "FATAL", "INFO", "OFF", "TRACE", "WARN".

Caution

FIXME importance

conf/log4j.properties

You can set up the default logging for Spark shell in `conf/log4j.properties`. Use `conf/log4j.properties.template` as a starting point.

Setting Default Log Level Programatically

See [Setting Default Log Level Programatically](#) in [SparkContext - the door to Spark](#).

Spark applications

In standalone Spark applications or while in [Spark Shell](#) session, use the following:

```
import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.OFF)
Logger.getLogger("akka").setLevel(Level.OFF)
```

sbt

When running a Spark application from within sbt using `run` task, you can use the following `build.sbt` to configure logging levels:

```
fork in run := true
javaOptions in run ++= Seq(
  "-Dlog4j.debug=true",
  "-Dlog4j.configuration=log4j.properties")
outputStrategy := Some(StdoutOutput)
```

With the above configuration `log4j.properties` file should be on CLASSPATH which can be in `src/main/resources` directory (that is included in CLASSPATH by default).

When `run` starts, you should see the following output in sbt:

```
[spark-activator]> run
[info] Running StreamingApp
log4j: Trying to find [log4j.properties] using context classloader sun.misc.Launcher$AppC
log4j: Using URL [file:/Users/jacek/dev/oss/spark-activator/target/scala-2.11/classes/log
log4j: Reading configuration from URL file:/Users/jacek/dev/oss/spark-activator/target/sc
```

Performance Tuning

Goal: Improve Spark's performance where feasible.

From [Investigating Spark's performance](#):

- measure performance bottlenecks using new metrics, including **block-time analysis**
- a live demo of a new **performance analysis tool**
- CPU — not I/O (network) — is often a critical bottleneck
- *community dogma* = network and disk I/O are major bottlenecks
- a TPC-DS workload, of two sizes: a 20 machine cluster with 850GB of data, and a 60 machine cluster with 2.5TB of data.
 - network is almost irrelevant for performance of these workloads
 - network optimization could only reduce job completion time by, at most, 2%
 - 10Gbps networking hardware is likely not necessary
- serialized compressed data

From [Making Sense of Spark Performance - Kay Ousterhout \(UC Berkeley\)](#) at Spark Summit 2015:

- `reduceByKey` is better
- mind serialization time
 - impacts CPU - time to serialize and network - time to send the data over the wire
- Tungsten - recent initiative from Databricks - aims at reducing CPU time
 - jobs become more bottlenecked by IO

Metrics System

Spark uses [Metrics](#) - a Java library to measure the behavior of the components.

`org.apache.spark.metrics.source.Source` is the top-level class for the metric registries in Spark.

Caution	<p>FIXME Review</p> <ul style="list-style-type: none"> • How to use the metrics to monitor Spark using jconsole? • ApplicationSource • WorkerSource • ExecutorSource • JvmSource • MesosClusterSchedulerSource • StreamingSource
---------	---

- Review `MetricsServlet`
- Review `org.apache.spark.metrics` package, esp. `MetricsSystem` class.
- Default properties
 - `"*.sink.servlet.class"`, `"org.apache.spark.metrics.sink.MetricsServlet"`
 - `"*.sink.servlet.path"`, `"/metrics/json"`
 - `"master.sink.servlet.path"`, `"/metrics/master/json"`
 - `"applications.sink.servlet.path"`, `"/metrics/applications/json"`
- `spark.metrics.conf` (default: `metrics.properties` on `CLASSPATH`)
- `spark.metrics.conf.` prefix in `SparkConf`

Executors

A non-local executor registers executor source.

[FIXME](#) See `Executor` class.

Master

```
$ http http://192.168.1.4:8080/metrics/master/json/path
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Content-Length: 207
Content-Type: text/json;charset=UTF-8
Server: Jetty(8.y.z-SNAPSHOT)
X-Frame-Options: SAMEORIGIN

{
  "counters": {},
  "gauges": {
    "master.aliveWorkers": {
      "value": 0
    },
    "master.apps": {
      "value": 0
    },
    "master.waitingApps": {
      "value": 0
    },
    "master.workers": {
      "value": 0
    }
  },
  "histograms": {},
  "meters": {},
  "timers": {},
  "version": "3.0.0"
}
```

Scheduler Listeners

A Spark **listener** is a class that listens to execution events from [DAGScheduler](#). It extends [org.apache.spark.scheduler.SparkListener](#).

Tip

Developing a custom SparkListener can be an excellent introduction to low-level details of [Spark's Execution Model](#). Check out the exercise [Developing Custom SparkListener to monitor DAGScheduler in Scala](#).

A Spark listener can receive events about:

- when a stage completes successfully or fails
- when a stage is submitted
- when a task starts
- when a task begins remotely fetching its result
- when a task ends
- when a job starts
- when a job ends
- when environment properties have been updated
- when a new block manager has joined
- when an existing block manager has been removed
- when an RDD is manually unpersisted by the application
- when the application starts (as `sparkListenerApplicationStart`)
- when the application ends (as `sparkListenerApplicationEnd`)
- when the driver receives task metrics from an executor in a heartbeat.
- [when the driver registers a new executor \(FIXME or is this to let the driver know about the new executor?\).](#)
- when the driver removes an executor.
- when the driver receives a block update info.

Listener Bus

A **Listener Bus** asynchronously passes [listener events](#) to registered [Spark listeners](#).

An instance of Listener Bus runs on [a driver](#) as an instance of `LiveListenerBus` (as `listenerBus`). It is created and started when [a Spark context starts](#).

A Listener Bus is a daemon thread called **SparkListenerBus** that asynchronously processes events from a `java.util.concurrent.LinkedBlockingQueue` capped at 10000 events.

Listener Events

Caution

[FIXME](#) What are SparkListenerEvents? Where and why are they posted? What do they cause?

- [SparkListenerEnvironmentUpdate](#)

SparkListenerApplicationStart

[FIXME](#)

SparkListenerExecutorAdded

`SparkListenerExecutorAdded` is posted as a result of:

- A `RegisterExecutor` event having been received by [CoarseGrainedSchedulerBackend](#)
- Calling [MesosSchedulerBackend.resourceOffers](#).
- [LocalBackend](#) being started.

`SparkListenerExecutorAdded` is passed along to [Spark Listeners](#) using

`SparkListener.onExecutorAdded(executorAdded)` method.

Spark Listeners

Caution

[FIXME](#) What do the listeners do? Move them to appropriate sections.

- [EventLoggingListener](#)
- [ExecutorsListener](#) that prepares information to be displayed on the **Executors** tab in [web UI](#).
- [SparkFirehoseListener](#) that allows users to receive all SparkListener events by overriding the `onEvent` method only.
- [ExecutorAllocationListener](#)

- [HeartbeatReceiver](#)

Registering Spark Listener

You can register a Spark listener in a Spark application using

`SparkContext.addSparkListener(listener: SparkListener)` method or [spark.extraListeners](#) setting.

It is assumed that the listener comes with one of the following (in this order):

- a single-argument constructor that accepts `SparkConf`
- a zero-argument constructor

Tip Set `INFO` on `org.apache.spark.SparkContext` logger to see the extra listeners being registered.

```
INFO SparkContext: Registered listener pl.japila.spark.CustomSparkListener
```

Internal Listeners

- web UI and [event logging](#) listeners

Event Logging

Spark comes with its own [EventLoggingListener](#) Spark listener that logs events to persistent storage.

When the listener starts, it prints out the INFO message `Logging events to [logPath]` and logs JSON-encoded events using `JsonProtocol` class.

`EventLoggingListener` uses the following properties:

- `spark.eventLog.enabled` (default: `false`) - whether to log Spark events that encode the information displayed in the UI to persisted storage. It is useful for reconstructing the Web UI after a Spark application has finished.
- `spark.eventLog.compress` (default: `false`) - whether to compress logged events.
- `spark.eventLog.overwrite` (default: `false`) - whether to overwrite any existing files.
- `spark.eventLog.dir` (default: `/tmp/spark-events`) - path to the directory in which events are logged, e.g. `hdfs://namenode:8021/directory`. The directory must exist before Spark starts up.

- `spark.eventLog.buffer.kb` (default: `100`) - buffer size to use when writing to output streams.
- `spark.eventLog.testing` (default: `false`)

StatsReportListener - Logging summary statistics

`org.apache.spark.scheduler.StatsReportListener` (see [the class' scaladoc](#)) is a `SparkListener` that logs a few summary statistics when each stage completes.

It listens to `SparkListenerTaskEnd`, `SparkListenerStageCompleted` events.

```
$ ./bin/spark-shell --conf \
    spark.extraListeners=org.apache.spark.scheduler.StatsReportListener
...
INFO SparkContext: Registered listener org.apache.spark.scheduler.StatsReportListener
...
scala> sc.parallelize(0 to 10).count
...
15/11/04 15:39:45 INFO StatsReportListener: Finished stage: org.apache.spark.scheduler.St
15/11/04 15:39:45 INFO StatsReportListener: task runtime:(count: 8, mean: 36.625000, std
15/11/04 15:39:45 INFO StatsReportListener:      0%      5%      10%      25%      50%    7
15/11/04 15:39:45 INFO StatsReportListener:      33.0 ms 33.0 ms 33.0 ms 34.0 ms 35.0 ms 3
15/11/04 15:39:45 INFO StatsReportListener: task result size:(count: 8, mean: 953.000000,
15/11/04 15:39:45 INFO StatsReportListener:      0%      5%      10%      25%      50%    7
15/11/04 15:39:45 INFO StatsReportListener:      953.0 B 953.0 B 953.0 B 953.0 B 953.0 B 9
15/11/04 15:39:45 INFO StatsReportListener: executor (non-fetch) time pct: (count: 8, mea
15/11/04 15:39:45 INFO StatsReportListener:      0%      5%      10%      25%      50%    7
15/11/04 15:39:45 INFO StatsReportListener:      13 %     13 %     13 %     17 %     18 %    2
15/11/04 15:39:45 INFO StatsReportListener: other time pct: (count: 8, mean: 82.339780, s
15/11/04 15:39:45 INFO StatsReportListener:      0%      5%      10%      25%      50%    7
15/11/04 15:39:45 INFO StatsReportListener:      80 %     80 %     80 %     82 %     82 %    8
```

Settings

- `spark.extraListeners` (default: empty) is a comma-separated list of listener class names that should be registered with Spark's listener bus when [SparkContext is initialized](#).

```
$ ./bin/spark-shell --conf spark.extraListeners=pl.japila.spark.CustomSparkListener
```

Exercise

In [Developing Custom SparkListener to monitor DAGScheduler in Scala](#) you can find a complete custom Scheduler Listener using Scala and sbt.

Debugging Spark using sbt

Use `sbt -jvm-debug 5005`, connect to the remote JVM at the port `5005` using IntelliJ IDEA, place breakpoints on the desired lines of the source code of Spark.

```
→ sparkme-app sbt -jvm-debug 5005
Listening for transport dt_socket at address: 5005
...
```

Run Spark context and the breakpoints get triggered.

```
scala> val sc = new SparkContext(conf)
15/11/14 22:58:46 INFO SparkContext: Running Spark version 1.6.0-SNAPSHOT
```

Tip

Read [Debugging chapter in IntelliJ IDEA 15.0 Help](#).

Building Spark

You can download pre-packaged versions of Apache Spark from [the project's web site](#). The packages are built for a different Hadoop versions, but only for Scala 2.10.

Note

Since [\[SPARK-6363\]\[BUILD\]](#) Make Scala 2.11 the default Scala version the default version of Scala is **2.11**.

If you want a **Scala 2.11** version of Apache Spark "*users should download the Spark source package and build with Scala 2.11 support*" (quoted from the Note at [Download Spark](#)).

The build process for Scala 2.11 takes around 15 mins (on a decent machine) and is so simple that it's unlikely to refuse the urge to do it yourself.

You can use [sbt](#) or [Maven](#) as the build command.

Using sbt as the build tool

The build command with sbt as the build tool is as follows:

```
./build/sbt -Pyarn -Phadoop-2.6 -Dhadoop.version=2.7.1 -Phive -Phive-thriftserver -DskipT
```

Using Java 8 to build Spark using sbt takes ca 10 minutes.

```
→ spark git:(master) ✘ ./build/sbt -Pyarn -Phadoop-2.6 -Dhadoop.version=2.7.1 -Phive -Ph...
...
[success] Total time: 496 s, completed Dec 7, 2015 8:24:41 PM
```

Using Apache Maven as the build tool

The build command with Apache Maven is as follows:

```
$ ./build/mvn -Pyarn -Phadoop-2.6 -Dhadoop.version=2.7.1 -Phive -Phive-thriftserver -Dski
```

After a couple of minutes your freshly baked distro is ready to fly!

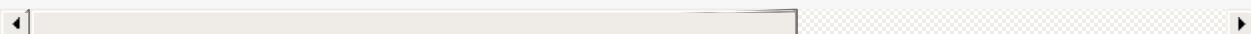
I'm using Oracle Java 8 to build Spark.

```
→ spark git:(master) ✘ java -version
```

```
java version "1.8.0_72"
Java(TM) SE Runtime Environment (build 1.8.0_72-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.72-b15, mixed mode)

→ spark git:(master) ✘ ./build/mvn -Pyarn -Phadoop-2.6 -Dhadoop.version=2.7.1 -Phive -Ph
Using `mvn` from path: /usr/local/bin/mvn
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=512M; support was
[INFO] Scanning for projects...
[INFO] -----
[INFO] Reactor Build Order:
[INFO]
[INFO] Spark Project Parent POM
[INFO] Spark Project Launcher
[INFO] Spark Project Networking
[INFO] Spark Project Shuffle Streaming Service
[INFO] Spark Project Unsafe
[INFO] Spark Project Core
[INFO] Spark Project Bagel
[INFO] Spark Project GraphX
[INFO] Spark Project Streaming
[INFO] Spark Project Catalyst
[INFO] Spark Project SQL
[INFO] Spark Project ML Library
[INFO] Spark Project Tools
[INFO] Spark Project Hive
[INFO] Spark Project REPL
[INFO] Spark Project YARN Shuffle Service
[INFO] Spark Project YARN
[INFO] Spark Project Hive Thrift Server
[INFO] Spark Project Assembly
[INFO] Spark Project External Twitter
[INFO] Spark Project External Flume Sink
[INFO] Spark Project External Flume
[INFO] Spark Project External Flume Assembly
[INFO] Spark Project External MQTT
[INFO] Spark Project External MQTT Assembly
[INFO] Spark Project External ZeroMQ
[INFO] Spark Project External Kafka
[INFO] Spark Project Examples
[INFO] Spark Project External Kafka Assembly
[INFO]
[INFO] -----
[INFO] Building Spark Project Parent POM 2.0.0-SNAPSHOT
[INFO] -----
[INFO] --- maven-clean-plugin:2.6.1:clean (default-clean) @ spark-parent_2.11 ---
...
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] Spark Project Parent POM ..... SUCCESS [ 3.731 s]
[INFO] Spark Project Test Tags ..... SUCCESS [ 3.611 s]
[INFO] Spark Project Sketch ..... SUCCESS [ 5.722 s]
[INFO] Spark Project Launcher ..... SUCCESS [ 9.695 s]
```

```
[INFO] Spark Project Networking ..... SUCCESS [ 11.342 s]
[INFO] Spark Project Shuffle Streaming Service ..... SUCCESS [ 7.307 s]
[INFO] Spark Project Unsafe ..... SUCCESS [ 7.882 s]
[INFO] Spark Project Core ..... SUCCESS [01:58 min]
[INFO] Spark Project GraphX ..... SUCCESS [ 16.775 s]
[INFO] Spark Project Streaming ..... SUCCESS [ 38.474 s]
[INFO] Spark Project Catalyst ..... SUCCESS [01:33 min]
[INFO] Spark Project SQL ..... SUCCESS [01:17 min]
[INFO] Spark Project ML Library ..... SUCCESS [01:20 min]
[INFO] Spark Project Tools ..... SUCCESS [ 5.239 s]
[INFO] Spark Project Hive ..... SUCCESS [ 44.372 s]
[INFO] Spark Project Docker Integration Tests ..... SUCCESS [ 2.009 s]
[INFO] Spark Project REPL ..... SUCCESS [ 6.495 s]
[INFO] Spark Project YARN Shuffle Service ..... SUCCESS [ 6.818 s]
[INFO] Spark Project YARN ..... SUCCESS [ 13.716 s]
[INFO] Spark Project Hive Thrift Server ..... SUCCESS [ 9.813 s]
[INFO] Spark Project Assembly ..... SUCCESS [ 40.605 s]
[INFO] Spark Project External Twitter ..... SUCCESS [ 7.439 s]
[INFO] Spark Project External Flume Sink ..... SUCCESS [ 6.325 s]
[INFO] Spark Project External Flume ..... SUCCESS [ 9.567 s]
[INFO] Spark Project External Flume Assembly ..... SUCCESS [ 1.655 s]
[INFO] Spark Project External Akka ..... SUCCESS [ 6.172 s]
[INFO] Spark Project External MQTT ..... SUCCESS [ 14.146 s]
[INFO] Spark Project External MQTT Assembly ..... SUCCESS [ 2.126 s]
[INFO] Spark Project External ZeroMQ ..... SUCCESS [ 7.035 s]
[INFO] Spark Project External Kafka ..... SUCCESS [ 12.157 s]
[INFO] Spark Project Examples ..... SUCCESS [01:00 min]
[INFO] Spark Project External Kafka Assembly ..... SUCCESS [ 3.033 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 12:15 min
[INFO] Finished at: 2016-01-30T22:29:53+01:00
[INFO] Final Memory: 119M/1526M
[INFO] -----
```



Please note the messages that say the version of Spark (*Building Spark Project Parent POM 2.0.0-SNAPSHOT*), Scala version (*maven-clean-plugin:2.6.1:clean (default-clean) @ spark-parent_2.11*) and the Spark modules built.

The above command gives you the latest version of **Apache Spark 2.0.0-SNAPSHOT** built for **Scala 2.11.7** (see [the configuration of scala-2.11 profile](#)).

Tip	You can also know the version of Spark using <code>./bin/spark-shell --version</code> .
-----	---

Making Distribution

`./make-distribution.sh` is the shell script to make a distribution. It uses the same profiles as for sbt and Maven.

Use `--tgz` option to have a tar gz version of the Spark distribution.

```
→ spark git:(master) ✘ ./make-distribution.sh --tgz -Pyarn -Phadoop-2.6 -Dhadoop.version
```

Once finished, you will have the distribution in the current directory, i.e. `spark-2.0.0-SNAPSHOT-bin-2.7.1.tgz`.

Spark and Hadoop

SparkHadoopUtil

Caution

[FIXME](#)

Introduction to Hadoop

Note

This page is the place to keep information more general about Hadoop and not related to [Spark on YARN](#) or files [Using Input and Output \(I/O\)](#) (HDFS). I don't really know what it could be, though. Perhaps nothing at all. Just saying.

From [Apache Hadoop's web site](#):

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

- Originally, **Hadoop** is an umbrella term for the following (core) **modules**:
 - [HDFS \(Hadoop Distributed File System\)](#) is a distributed file system designed to run on commodity hardware. It is a data storage with files split across a cluster.
 - **MapReduce** - the compute engine for batch processing
 - **YARN** (Yet Another Resource Negotiator) - the resource manager
- Currently, it's more about the ecosystem of solutions that all use Hadoop infrastructure for their work.

People reported to do wonders with the software with [Yahoo! saying](#):

Yahoo has progressively invested in building and scaling Apache Hadoop clusters with a current footprint of more than 40,000 servers and 600 petabytes of storage spread across 19 clusters.

Beside numbers [Yahoo! reported](#) that:

Deep learning can be defined as first-class steps in [Apache Oozie](#) workflows with Hadoop for data processing and Spark pipelines for machine learning.

You can find some *preliminary* information about **Spark pipelines for machine learning** in the chapter [ML Pipelines](#).

HDFS provides fast analytics – scanning over large amounts of data very quickly, but it was not built to handle updates. If data changed, it would need to be appended in bulk after a certain volume or time interval, preventing real-time visibility into this data.

- HBase complements HDFS' capabilities by providing fast and random reads and writes and supporting updating data, i.e. serving small queries extremely quickly, and allowing data to be updated in place.

From [How does partitioning work for data from files on HDFS?](#):

When Spark reads a file from HDFS, it creates a single partition for a single input split. Input split is set by the Hadoop `InputFormat` used to read this file. For instance, if you use `textFile()` it would be `TextInputFormat` in Hadoop, which would return you a single partition for a single block of HDFS (but the split between partitions would be done on line split, not the exact block split), unless you have a compressed text file. In case of compressed file you would get a single partition for a single file (as compressed text files are not splittable).

If you have a 30GB uncompressed text file stored on HDFS, then with the default HDFS block size setting (128MB) it would be stored in 235 blocks, which means that the RDD you read from this file would have 235 partitions. When you call `repartition(1000)` your RDD would be marked as to be repartitioned, but in fact it would be shuffled to 1000 partitions only when you will execute an action on top of this RDD (lazy execution concept)

With HDFS you can store any data (regardless of format and size). It can easily handle **unstructured data** like video or other binary files as well as semi- or fully-structured data like CSV files or databases.

There is the concept of **data lake** that is a huge data repository to support analytics.

HDFS partition files into so called **splits** and distributes them across multiple nodes in a cluster to achieve fail-over and resiliency.

MapReduce happens in three phases: **Map**, **Shuffle**, and **Reduce**.

Further reading

- [Introducing Kudu: The New Hadoop Storage Engine for Fast Analytics on Fast Data](#)

Spark and software in-memory file systems

It appears that there are a few open source projects that can boost performance of any in-memory shared state, akin to file system, including RDDs - [Tachyon](#), [Apache Ignite](#), and [Apache Geode](#).

From [tachyon project's website](#):

Tachyon is designed to function as a software file system that is compatible with the HDFS interface prevalent in the big data analytics space. The point of doing this is that it can be used as a drop in accelerator rather than having to adapt each framework to use a distributed caching layer explicitly.

From [Spark Shared RDDs](#):

Apache Ignite provides an implementation of Spark RDD abstraction which allows to easily share state in memory across multiple Spark jobs, either within the same application or between different Spark applications.

There's another similar open source project [Apache Geode](#).

Spark and The Others

The **others** are the ones that are similar to Spark, but as I haven't yet exactly figured out where and how, they are here.

Note	I'm going to keep the noise (<i>enterprisey adornments</i>) to the very minimum.
------	--

- [Ceph](#) is a unified, distributed storage system.
- [Apache Twill](#) is an abstraction over Apache Hadoop YARN that allows you to use YARN's distributed capabilities with a programming model that is similar to running threads.

Distributed Deep Learning on Spark (using Yahoo's Caffe-on-Spark)

Read the article [Large Scale Distributed Deep Learning on Hadoop Clusters](#) to learn about **Distributed Deep Learning using Caffe-on-Spark**:

To enable deep learning on these enhanced Hadoop clusters, we developed a comprehensive distributed solution based upon open source software libraries, [Apache Spark](#) and [Caffe](#). One can now submit deep learning jobs onto a (Hadoop YARN) cluster of GPU nodes (using `spark-submit`).

Caffe-on-Spark is a result of Yahoo's early steps in bringing Apache Hadoop ecosystem and deep learning together on the same heterogeneous (GPU+CPU) cluster that may be open sourced depending on interest from the community.

In the comments to the article, some people announced their plans of using it with [AWS GPU cluster](#).

Spark Packages

[Spark Packages](#) is a community index of packages for Apache Spark.

Spark Packages is a community site hosting modules that are not part of Apache Spark. It offers packages for reading different files formats (than those natively supported by Spark) or from NoSQL databases like [Cassandra](#), code testing, etc.

When you want to include a Spark package in your application, you should be using `--packages` command line option.

Spark Tips and Tricks

Print Launch Command of Spark Scripts

`SPARK_PRINT_LAUNCH_COMMAND` environment variable controls whether the Spark launch command is printed out to the standard error output, i.e. `System.err`, or not.

```
Spark Command: [here comes the command]
=====

```

All the Spark shell scripts use `org.apache.spark.launcher.Main` class internally that checks `SPARK_PRINT_LAUNCH_COMMAND` and when set (to any value) will print out the entire command line to launch it.

```
$ SPARK_PRINT_LAUNCH_COMMAND=1 ./bin/spark-shell
Spark Command: /Library/Java/JavaVirtualMachines/Current/Contents/Home/bin/java -cp /User
=====

```

Show Spark version in Spark shell

In spark-shell, use `sc.version` or `org.apache.spark.SPARK_VERSION` to know the Spark version:

```
scala> sc.version
res0: String = 1.6.0-SNAPSHOT

scala> org.apache.spark.SPARK_VERSION
res1: String = 1.6.0-SNAPSHOT
```

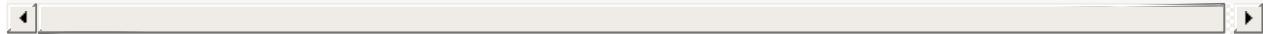
Resolving local host name

When you face networking issues when Spark can't resolve your local hostname or IP address, use the preferred `SPARK_LOCAL_HOSTNAME` environment variable as the custom host name or `SPARK_LOCAL_IP` as the custom IP that is going to be later resolved to a hostname.

Spark checks them out before using [java.net.InetAddress.getLocalHost\(\)](#) (consult [org.apache.spark.util.Utils.findLocalInetAddress\(\)](#) method).

You may see the following WARN messages in the logs when Spark finished the resolving process:

```
WARN Your hostname, [hostname] resolves to a loopback address: [host-address]; using...
WARN Set SPARK_LOCAL_IP if you need to bind to another address
```



Starting standalone Master and workers on Windows 7

Windows 7 users can use [spark-class](#) to start Spark Standalone as there are no launch scripts for the Windows platform.

```
$ ./bin/spark-class org.apache.spark.deploy.master.Master -h localhost
```

```
$ ./bin/spark-class org.apache.spark.deploy.worker.Worker spark://localhost:7077
```

Access private members in Scala in Spark shell

If you ever wanted to use `private[spark]` members in Spark using the Scala programming language, e.g. toy with `org.apache.spark.scheduler.DAGScheduler` or similar, you will have to use the following trick in Spark shell - use `:paste -raw` as described in [REPL: support for package definition](#).

Open `spark-shell` and execute `:paste -raw` that allows you to enter any valid Scala code, including `package`.

The following snippet shows how to access `private[spark]` member

`DAGScheduler.RESUBMIT_TIMEOUT` :

```
scala> :paste -raw
// Entering paste mode (ctrl-D to finish)

package org.apache.spark

object spark {
  def test = {
    import org.apache.spark.scheduler._
    println(DAGScheduler.RESUBMIT_TIMEOUT == 200)
  }
}

scala> spark.test
true

scala> sc.version
res0: String = 1.6.0-SNAPSHOT
```

org.apache.spark.SparkException: Task not serializable

When you run into `org.apache.spark.SparkException: Task not serializable` exception, it means that you use a reference to an instance of a non-serializable class inside a transformation. See the following example:

```
→ spark git:(master) ✘ ./bin/spark-shell
Welcome to

    __
   / \
  _\ \_ \_ \_ \_ \_ \
 /__/ . __/\_,/_/ /__\ \
                           version 1.6.0-SNAPSHOT
  /_/

Using Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_66)
Type in expressions to have them evaluated.
Type :help for more information.

scala> class NotSerializable(val num: Int)
defined class NotSerializable

scala> val notSerializable = new NotSerializable(10)
notSerializable: NotSerializable = NotSerializable@2700f556

scala> sc.parallelize(0 to 10).map(_ => notSerializable.num).count
org.apache.spark.SparkException: Task not serializable
  at org.apache.spark.util.ClosureCleaner$.ensureSerializable(ClosureCleaner.scala:304)
  at org.apache.spark.util.ClosureCleaner$.org$apache$spark$util$ClosureCleaner$$clean(ClosureCleaner.scala:297)
  at org.apache.spark.util.ClosureCleaner$.clean(ClosureCleaner.scala:122)
  at org.apache.spark.SparkContext.clean(SparkContext.scala:2055)
  at org.apache.spark.rdd.RDD$$anonfun$map$1.apply(RDD.scala:318)
  at org.apache.spark.rdd.RDD$$anonfun$map$1.apply(RDD.scala:317)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:150)
  at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:111)
  at org.apache.spark.rdd.RDD.withScope(RDD.scala:310)
  at org.apache.spark.rdd.RDD.map(RDD.scala:317)
  ... 48 elided
Caused by: java.io.NotSerializableException: NotSerializable
Serialization stack:
 - object not serializable (class: NotSerializable, value: NotSerializable@2700f556)
 - field (class: $iw, name: notSerializable, type: class NotSerializable)
 - object (class $iw, $iw@10e542f3)
 - field (class: $iw, name: $iw, type: class $iw)
 - object (class $iw, $iw@729feae8)
 - field (class: $iw, name: $iw, type: class $iw)
 - object (class $iw, $iw@5fc3b20b)
 - field (class: $iw, name: $iw, type: class $iw)
```

```
- object (class $iw, $iw@36dab184)
- field (class: $iw, name: $iw, type: class $iw)
- object (class $iw, $iw@5eb974)
- field (class: $iw, name: $iw, type: class $iw)
- object (class $iw, $iw@79c514e4)
- field (class: $iw, name: $iw, type: class $iw)
- object (class $iw, $iw@5aeaee3)
- field (class: $iw, name: $iw, type: class $iw)
- object (class $iw, $iw@2be9425f)
- field (class: $line18.$read, name: $iw, type: class $iw)
- object (class $line18.$read, $line18.$read@6311640d)
- field (class: $iw, name: $line18$read, type: class $line18$read)
- object (class $iw, $iw@c9cd06e)
- field (class: $iw, name: $outer, type: class $iw)
- object (class $iw, $iw@6565691a)
- field (class: $anonfun$1, name: $outer, type: class $iw)
- object (class $anonfun$1, <function1>
at org.apache.spark.serializer.SerializationDebugger$.improveException(SerializationDeb
at org.apache.spark.serializer.JavaSerializationStream.writeObject(JavaSerializer.scala
at org.apache.spark.serializer.JavaSerializerInstance.serialize(JavaSerializer.scala:10
at org.apache.spark.util.ClosureCleaner$.ensureSerializable(ClosureCleaner.scala:301)
... 57 more
```

Further reading

- Job aborted due to stage failure: Task not serializable
- Add utility to help with NotSerializableException debugging
- Task not serializable: java.io.NotSerializableException when calling function outside closure only on classes not objects

Running Spark on Windows

Running Spark on Windows is not much different from other operating systems like Linux or Mac OS X, but there is one issue that people report often - a permission error when running Spark Shell.

```
15/01/29 17:21:27 ERROR Shell: Failed to locate the winutils binary in the hadoop binary  
java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop bina  
at org.apache.hadoop.util.Shell.getQualifiedBinPath(Shell.java:318)  
at org.apache.hadoop.util.Shell.getWinUtilsPath(Shell.java:333)  
at org.apache.hadoop.util.Shell.<clinit>(Shell.java:326)  
at org.apache.hadoop.util.StringUtils.<clinit>(StringUtils.java:76)
```

Download [winutils.exe](#) and save it to a directory, say `c:\hadoop\bin`.

Set `HADOOP_HOME`.

```
set HADOOP_HOME=c:\hadoop
```

Set `PATH` to include `%HADOOP_HOME%\bin` as follows:

```
set PATH=%HADOOP_HOME%\bin;%PATH%
```

Create `c:\tmp\hive` folder and execute the following command as Administrator (using "Run As Administrator" option while executing `cmd`):

```
winutils.exe chmod -R 777 \tmp\hive
```

You may want to check the permissions as follows:

```
winutils.exe ls \tmp\hive
```

Exercises

Here I'm collecting exercises that aim at strengthening your understanding of Apache Spark.

Exercise: One-liners using PairRDDFunctions

This is a set of one-liners to give you a entry point into using [PairRDDFunctions](#) that are available in RDDs of pairs via Scala's implicit conversion.

Exercise

How would you go about solving a requirement to pair elements of the same key and creating a new RDD out of the matched values?

```
val users = Seq((1, "user1"), (1, "user2"), (2, "user1"), (2, "user3"), (3, "user2"), (3, "  
// Input RDD  
val us = sc.parallelize(users)  
  
// ...your code here  
  
// Desired output  
Seq("user1", "user2"), ("user1", "user3"), ("user1", "user4"), ("user2", "user4"))
```

Exercise: Learning Jobs and Partitions Using take Action

The exercise aims for introducing `take` action and using `spark-shell` and web UI. It should introduce you to the concepts of partitions and jobs.

The following snippet creates an RDD of 16 elements with 16 partitions.

```
scala> val r1 = sc.parallelize(0 to 15, 16)
r1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[26] at parallelize at <console>

scala> r1.partitions.size
res63: Int = 16

scala> r1.foreachPartition(it => println(">>> partition size: " + it.size))
...
>>> partition size: 1
...
// the machine has 8 cores
...
// so first 8 tasks get executed immediately
...
// with the others after a core is free to take on new tasks.
>>> partition size: 1
...
>>> partition size: 1
...
>>> partition size: 1
...
>>> partition size: 1
>>> partition size: 1
...
>>> partition size: 1
>>> partition size: 1
>>> partition size: 1
```

All 16 partitions have one element.

When you execute `r1.take(1)` only one job gets run since it is enough to compute one task on one partition.

Caution

[FIXME](#) Snapshot from web UI - note the number of tasks

However, when you execute `r1.take(2)` two jobs get run as the implementation assumes one job with one partition, and if the elements didn't total to the number of elements requested in `take`, quadruple the partitions to work on in the following jobs.

Caution	FIXME Snapshot from web UI - note the number of tasks
---------	---

Can you guess how many jobs are run for `r1.take(15)`? How many tasks per job?

Caution	FIXME Snapshot from web UI - note the number of tasks
---------	---

Answer: 3.

Spark Standalone - Using ZooKeeper for High-Availability of Master

Tip

Read [Recovery Mode](#) to know the theory.

You're going to start two standalone Masters.

You'll need 4 terminals (adjust addresses as needed):

Start ZooKeeper.

Create a configuration file `ha.conf` with the content as follows:

```
spark.deploy.recoveryMode=ZOOKEEPER
spark.deploy.zookeeper.url=<zookeeper_host>:2181
spark.deploy.zookeeper.dir=/spark
```

Start the first standalone Master.

```
./sbin/start-master.sh -h localhost -p 7077 --webui-port 8080 --properties-file ha.conf
```

Note

It is not possible to start another instance of standalone Master on the same machine using `./sbin/start-master.sh`. The reason is that the script assumes one instance per machine only. We're going to change the script to make it possible.

```
$ cp ./sbin/start-master{,-2}.sh

$ grep "CLASS 1" ./sbin/start-master-2.sh
"${SPARK_HOME}/sbin"/spark-daemon.sh start $CLASS 1 \

$ sed -i -e 's/CLASS 1/CLASS 2/' sbin/start-master-2.sh

$ grep "CLASS 1" ./sbin/start-master-2.sh

$ grep "CLASS 2" ./sbin/start-master-2.sh
"${SPARK_HOME}/sbin"/spark-daemon.sh start $CLASS 2 \

$ ./sbin/start-master-2.sh -h localhost -p 17077 --webui-port 18080 --properties-file ha.
```

You can check how many instances you're currently running using `jps` command as follows:

```
$ jps -lm
5024 sun.tools.jps.Jps -lm
4994 org.apache.spark.deploy.master.Master --ip japila.local --port 7077 --webui-port 808
4808 org.apache.spark.deploy.master.Master --ip japila.local --port 7077 --webui-port 808
4778 org.apache.zookeeper.server.quorum.QuorumPeerMain config/zookeeper.properties
```

Start a standalone Worker.

```
./sbin/start-slave.sh spark://localhost:7077,localhost:17077
```

Start Spark shell.

```
./bin/spark-shell --master spark://localhost:7077,localhost:17077
```

Wait till the Spark shell connects to an active standalone Master.

Find out which standalone Master is active (there can only be one). Kill it. Observe how the other standalone Master takes over and lets the Spark shell register with itself. Check out the master's UI.

Optionally, kill the worker, make sure it goes away instantly in the active master's logs.

Exercise: Spark's Hello World using Spark shell and Scala

Run Spark shell and count the number of words in a file using MapReduce pattern.

- Use `sc.textFile` to read the file into memory
- Use `RDD.flatMap` for a mapper step
- Use `reduceByKey` for a reducer step

WordCount using Spark shell

It is like any introductory big data example should somehow demonstrate how to count words in distributed fashion.

In the following example you're going to count the words in `README.md` file that sits in your Spark distribution and save the result under `README.count` directory.

You're going to use [the Spark shell](#) for the example. Execute `spark-shell`.

```
val lines = sc.textFile("README.md")          (1)  
  
val words = lines.flatMap(_.split("\\s+"))      (2)  
  
val wc = words.map(w => (w, 1)).reduceByKey(_ + _) (3)  
  
wc.saveAsTextFile("README.count")             (4)
```

1. Read the text file - refer to [Using Input and Output \(I/O\)](#).
2. Split each line into words and flatten the result.
3. Map each word into a pair and count them by word (key).
4. Save the result into text files - one per partition.

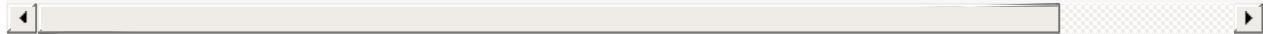
After you have executed the example, see the contents of the `README.count` directory:

```
$ ls -lt README.count  
total 16  
-rw-r--r-- 1 jacek staff 0 9 pa  13:36 _SUCCESS  
-rw-r--r-- 1 jacek staff 1963 9 pa  13:36 part-00000  
-rw-r--r-- 1 jacek staff 1663 9 pa  13:36 part-00001
```

The files `part-0000x` contain the pairs of word and the count.

```
$ cat README.count/part-00000
(package,1)
(this,1)
(Version"])(http://spark.apache.org/docs/latest/building-spark.html#specifying-the-hadoop-
(Because,1)
(Python,2)
(cluster.,1)
(its,1)
([run,1]
...

```



Further (self-)development

Please read the questions and give answers first before looking at the link given.

1. Why are there two files under the directory?
2. How could you have only one?
3. How to `filter` out words by name?
4. How to `count` words?

Please refer to the chapter [Partitions](#) to find some of the answers.

Your first Spark application (using Scala and sbt)

This page gives you the exact steps to develop and run a complete Spark application using [Scala](#) programming language and [sbt](#) as the build tool.

Tip

Refer to Quick Start's [Self-Contained Applications](#) in the official documentation.

The sample application called **SparkMe App** is...[FIXME](#)

Overview

You're going to use [sbt](#) as the project build tool. It uses `build.sbt` for the project's description as well as the dependencies, i.e. the version of Apache Spark and others.

The application's main code is under `src/main/scala` directory, in `SparkMeApp.scala` file.

With the files in a directory, executing `sbt package` results in a package that can be deployed onto a Spark cluster using `spark-submit`.

In this example, you're going to use Spark's [local mode](#).

Project's build - `build.sbt`

Any Scala project managed by sbt uses `build.sbt` as the central place for configuration, including project dependencies denoted as `libraryDependencies`.

`build.sbt`

```
name      := "SparkMe Project"
version   := "1.0"
organization := "pl.japila"

scalaVersion := "2.11.7"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.6.0-SNAPSHOT" (1)
resolvers += Resolver.mavenLocal
```

1. Use the development version of Spark 1.6.0-SNAPSHOT

SparkMe Application

The application uses a single command-line parameter (as `args(0)`) that is the file to process. The file is read and the number of lines printed out.

```
package pl.japila.spark

import org.apache.spark.{SparkContext, SparkConf}

object SparkMeApp {
    def main(args: Array[String]) {
        val conf = new SparkConf().setAppName("SparkMe Application")
        val sc = new SparkContext(conf)

        val fileName = args(0)
        val lines = sc.textFile(fileName).cache()

        val c = lines.count()
        println(s"There are $c lines in $fileName")
    }
}
```

sbt version - project/build.properties

sbt (launcher) uses `project/build.properties` file to set (the real) sbt up

```
sbt.version=0.13.9
```

Tip

With the file the build is more predictable as the version of sbt doesn't depend on the sbt launcher.

Packaging Application

Execute `sbt package` to package the application.

```
→ sparkme-app sbt package
[info] Loading global plugins from /Users/jacek/.sbt/0.13/plugins
[info] Loading project definition from /Users/jacek/dev/sandbox/sparkme-app/project
[info] Set current project to SparkMe Project (in build file:/Users/jacek/dev/sandbox/spa
[info] Compiling 1 Scala source to /Users/jacek/dev/sandbox/sparkme-app/target/scala-2.11
[info] Packaging /Users/jacek/dev/sandbox/sparkme-app/target/scala-2.11/sparkme-project_2
[info] Done packaging.
[success] Total time: 3 s, completed Sep 23, 2015 12:47:52 AM
```

The application uses only classes that comes with Spark so `package` is enough.

In `target/scala-2.11/sparkme-project_2.11-1.0.jar` there is the final application ready for deployment.

Submitting Application to Spark (local)

Note

The application is going to be deployed to `local[*]`. Change it to whatever cluster you have available (refer to [Running Spark in cluster](#)).

`spark-submit` the SparkMe application and specify the file to process (as it is the only and required input parameter to the application), e.g. `build.sbt` of the project.

Note

`build.sbt` is sbt's build definition and is only used as an input file for demonstration purposes. **Any** file is going to work fine.

```
→ sparkme-app ~/dev/oss/spark/bin/spark-submit --master "local[*]" --class pl.japila.sp  
Using Spark's repl log4j profile: org/apache/spark/log4j-defaults-repl.properties  
To adjust logging level use sc.setLogLevel("INFO")  
15/09/23 01:06:02 WARN NativeCodeLoader: Unable to load native-hadoop library for your pl  
15/09/23 01:06:04 WARN MetricsSystem: Using default name DAGScheduler for source because  
There are 8 lines in build.sbt
```

Note

Disregard the two above WARN log messages.

You're done. Sincere congratulations!

Spark (notable) use cases

That's the place where I'm throwing things I'd love exploring further - technology- and business-centric.

Technology "things":

- Spark Streaming on Hadoop YARN cluster processing messages from Apache Kafka using the new direct API.
- Parsing JSONs into Parquet and save it to S3

Business "things":

- **IoT applications** = connected devices and sensors
- **Predictive Analytics** = Manage risk and capture new business opportunities with real-time analytics and probabilistic forecasting of customers, products and partners.
- **Anomaly Detection** = Detect in real-time problems such as financial fraud, structural defects, potential medical conditions, and other anomalies.
- **Personalization** = Deliver a unique experience in real-time that is relevant and engaging based on a deep understanding of the customer and current context.
- data lakes, clickstream analytics, real time analytics, and data warehousing on Hadoop

Using Spark SQL to update data in Hive using ORC files

The example has showed up on Spark's users mailing list.

Caution	<ul style="list-style-type: none"> • FIXME Offer a complete working solution in Scala • FIXME Load ORC files into <code>dataframe</code> <ul style="list-style-type: none"> ◦ <code>val df = hiveContext.read.format("orc").load(to/path)</code>
---------	--

Solution was to use Hive in ORC format with partitions:

- A table in Hive stored as an ORC file (using partitioning)
- Using `SQLContext.sql` to insert data into the table
- Using `SQLContext.sql` to periodically run `ALTER TABLE...CONCATENATE` to merge your many small files into larger files optimized for your HDFS block size
 - Since the `CONCATENATE` command operates on files in place it is transparent to any downstream processing
- Hive solution is just to concatenate the files
 - it does not alter or change records.
 - it's possible to update data in Hive using ORC format
 - With transactional tables in Hive together with insert, update, delete, it does the "concatenate" for you automatically in regularly intervals. Currently this works only with tables in `orc.format` (stored as `orc`)
 - Alternatively, use Hbase with Phoenix as the SQL layer on top
 - Hive was originally not designed for updates, because it was purely warehouse focused, the most recent one can do updates, deletes etc in a transactional way.

Criteria:

- [Spark Streaming](#) jobs are receiving a lot of small events (avg 10kb)
- Events are stored to HDFS, e.g. for Pig jobs
- There are a lot of small files in HDFS (several millions)

Exercise: Developing Custom SparkListener to monitor DAGScheduler in Scala

Introduction

The example shows how to develop a custom DAGScheduler Listener in Spark. You should read [DAGScheduler Listeners](#) first to understand the reasons for the example.

Requirements

1. [Typesafe Activator](#)
2. Access to Internet to download the Spark dependency - `spark-core`

Setting up Scala project using Typesafe Activator

```
$ activator new custom-spark-listener minimal-scala  
$ cd custom-spark-listener
```

Add the following line to `build.sbt` (the main configuration file for the sbt project) that adds the dependency on Spark 1.5.1. Note the double `%` that are to select the proper version of the dependency for Scala 2.11.7.

```
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.5.1"
```

Custom Listener - `pl.japila.spark.CustomSparkListener`

Create the directory of your choice where the custom Spark listener is going to live, i.e.

```
pl/japila/spark .
```

```
$ mkdir -p src/main/scala/pl/japila/spark
```

Create the following file `CustomSparkListener.scala` in `src/main/scala/pl/japila/spark` directory. The aim of the class is to listen to events about jobs being started and tasks completed.

```

package pl.japila.spark

import org.apache.spark.scheduler.{SparkListenerStageCompleted, SparkListener, SparkLister

class CustomSparkListener extends SparkListener {
    override def onJobStart(jobStart: SparkListenerJobStart) {
        println(s"Job started with ${jobStart.stageInfos.size} stages: $jobStart")
    }

    override def onStageCompleted(stageCompleted: SparkListenerStageCompleted): Unit = {
        println(s"Stage ${stageCompleted.stageInfo.stageId} completed with ${stageCompleted.s
    }
}

```

Creating deployable package

package it, i.e. execute the command in activator shell.

```

[custom-spark-listener]> package
[info] Compiling 1 Scala source to /Users/jacek/dev/sandbox/custom-spark-listener/target/
[info] Packaging /Users/jacek/dev/sandbox/custom-spark-listener/target/scala-2.11/custom-
[info] Done packaging.
[success] Total time: 0 s, completed Nov 4, 2015 8:59:30 AM

```

You should have the result jar file with the custom scheduler listener ready (mine is /Users/jacek/dev/sandbox/custom-spark-listener/target/scala-2.11/custom-spark-listener_2.11-1.0.jar)

Activating Custom Listener in Spark shell

Start Spark shell with appropriate configurations for the extra custom listener and the jar that includes the class.

```
$ ./bin/spark-shell --conf spark.logConf=true --conf spark.extraListeners=pl.japila.spark
```

Create an RDD and execute an action to start a job as follows:

```
scala> sc.parallelize(0 to 10).count
15/11/04 12:27:29 INFO SparkContext: Starting job: count at <console>:25
...
Job started with 1 stages: SparkListenerJobStart(0,1446636449441,WrappedArray(org.apache.
...
Stage 0 completed with 8 tasks.
```

The last line that starts with `Job started:` is from the custom Spark listener you've just created. Congratulations! The exercise's over.

BONUS Activating Custom Listener in Spark Application

Tip

Use `sc.addSparkListener(myListener)`

Questions

1. What are the pros and cons of using the command line version vs inside a Spark application?

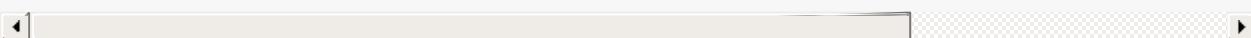
Developing RPC Environment

Caution	<p>FIXME</p> <ul style="list-style-type: none"> • Create the exercise • It could be easier to have an exercise to register a custom RpcEndpoint (it can receive network events known to all endpoints, e.g. RemoteProcessConnected = "a new node connected" or RemoteProcessDisconnected = a node disconnected). That could be the only way to know about the current runtime configuration of RpcEnv. Use <code>SparkEnv.rpcEnv</code> and <code>rpcEnv.setupEndpoint(name, endpointCreator)</code> to register a RPC Endpoint.
---------	---

Start simple using the following command:

```
$ ./bin/spark-shell --conf spark.rpc=doesnotexist
...
15/10/21 12:06:11 INFO SparkContext: Running Spark version 1.6.0-SNAPSHOT
...
15/10/21 12:06:11 ERROR SparkContext: Error initializing SparkContext.
java.lang.ClassNotFoundException: doesnotexist
    at scala.reflect.internal.util.AbstractFileClassLoader.findClass(AbstractFileClas
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:348)
    at org.apache.spark.util.Utils$class.forName(Utils.scala:173)
    at org.apache.spark.rpc.RpcEnv$RpcEnvFactory.getRpcEnvFactory(RpcEnv.scala:38)
    at org.apache.spark.rpc.RpcEnv$RpcEnvFactory.create(RpcEnv.scala:49)
    at org.apache.spark.SparkEnv$.create(SparkEnv.scala:257)
    at org.apache.spark.SparkEnv$.createDriverEnv(SparkEnv.scala:198)
    at org.apache.spark.SparkContext.createSparkEnv(SparkContext.scala:272)
    at org.apache.spark.SparkContext.<init>(SparkContext.scala:441)
    at org.apache.spark.repl.Main$.createSparkContext(Main.scala:79)
    at $line3.$read$$iw$$iw.<init>(<console>:12)
    at $line3.$read$$iw.<init>(<console>:21)
    at $line3.$read.<init>(<console>:23)
    at $line3.$read$.<init>(<console>:27)
    at $line3.$read$.<clinit>(<console>)
    at $line3.$eval$.$print$lzycompute(<console>:7)
    at $line3.$eval$.$print(<console>:6)
    at $line3.$eval$.print(<console>)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.j
    at java.lang.reflect.Method.invoke(Method.java:497)
    at scala.tools.nsc.interpreter.IMain$ReadEvalPrint.call(IMain.scala:784)
    at scala.tools.nsc.interpreter.IMain$Request.loadAndRun(IMain.scala:1039)
```

```
at scala.tools.nsc.interpreter.IMain$WrappedRequest$$anonfun$loadAndRunReq$1.appl
at scala.tools.nsc.interpreter.IMain$WrappedRequest$$anonfun$loadAndRunReq$1.appl
at scala.reflect.internal.util.ScalaClassLoader$class.asContext(ScalaClassLoader.
at scala.reflect.internal.util.AbstractFileClassLoader.asContext(AbstractFileClas
at scala.tools.nsc.interpreter.IMain$WrappedRequest.loadAndRunReq(IMain.scala:635
at scala.tools.nsc.interpreter.IMain.interpret(IMain.scala:567)
at scala.tools.nsc.interpreter.IMain.interpret(IMain.scala:563)
at scala.tools.nsc.interpreter.ILoop.reallyInterpret$1(ILoop.scala:802)
at scala.tools.nsc.interpreter.ILoop.interpretStartingWith(ILoop.scala:836)
at scala.tools.nsc.interpreter.ILoop.command(ILoop.scala:694)
at scala.tools.nsc.interpreter.ILoop.processLine(ILoop.scala:404)
at org.apache.spark.repl.SparkILoop$$anonfun$initializeSpark$1.apply$mcZ$sp(Spark
at org.apache.spark.repl.SparkILoop$$anonfun$initializeSpark$1.apply(SparkILoop.s
at org.apache.spark.repl.SparkILoop$$anonfun$initializeSpark$1.apply(SparkILoop.s
at scala.tools.nsc.interpreter.IMain.beQuietDuring(IMain.scala:213)
at org.apache.spark.repl.SparkILoop.initializeSpark(SparkILoop.scala:38)
at org.apache.spark.repl.SparkILoop.loadFiles(SparkILoop.scala:94)
at scala.tools.nsc.interpreter.ILoop$$anonfun$process$1.apply$mcZ$sp(ILoop.scala:
at scala.tools.nsc.interpreter.ILoop$$anonfun$process$1.apply(ILoop.scala:911)
at scala.tools.nsc.interpreter.ILoop$$anonfun$process$1.apply(ILoop.scala:911)
at scala.reflect.internal.util.ScalaClassLoader$.savingContextLoader(ScalaClassLo
at scala.tools.nsc.interpreter.ILoop.process(ILoop.scala:911)
at org.apache.spark.repl.Main$.main(Main.scala:49)
at org.apache.spark.repl.Main.main(Main.scala)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.j
at java.lang.reflect.Method.invoke(Method.java:497)
at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit$$runM
at org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:180)
at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:205)
at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:120)
at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
```



Developing Custom RDD

Caution	FIXME
---------	-----------------------

Spark courses

- [Spark Fundamentals I](#) from Big Data University.
- [Introduction to Big Data with Apache Spark](#) from Databricks.
- [Scalable Machine Learning](#) from Databricks.

Books

- O'Reilly
 - [Learning Spark \(my review at Amazon.com\)](#)
 - [Advanced Analytics with Spark](#)
 - [Data Algorithms: Recipes for Scaling Up with Hadoop and Spark](#)
 - [Spark Operations: Operationalizing Apache Spark at Scale \(in the works\)](#)
- Manning
 - [Spark in Action \(MEAP\)](#)
 - [Streaming Data \(MEAP\)](#)
 - [Spark GraphX in Action \(MEAP\)](#)
- Packt
 - [Mastering Apache Spark](#)
 - [Spark Cookbook](#)
 - [Learning Real-time Processing with Spark Streaming](#)
 - [Machine Learning with Spark](#)
 - [Fast Data Processing with Spark, 2nd Edition](#)
 - [Fast Data Processing with Spark](#)
 - [Apache Spark Graph Processing](#)
- Apress
 - [Big Data Analytics with Spark](#)
 - [Guide to High Performance Distributed Computing \(Case Studies with Hadoop, Scalding and Spark\)](#)

Commercial Products

Spark has reached the point where companies around the world adopt it to build their own solutions on top of it.

1. [IBM Analytics for Apache Spark](#)
2. [Google Cloud Dataproc](#)

IBM Analytics for Apache Spark

[IBM Analytics for Apache Spark](#) is the Apache Spark-based product from IBM.

In IBM Analytics for Apache Spark you can use IBM SystemML machine learning technology and run it on IBM Bluemix.

IBM Cloud relies on OpenStack Swift as Data Storage for the service.

Google Cloud Dataproc

From [Google Cloud Dataproc's website](#):

Google Cloud Dataproc is a managed Hadoop MapReduce, Spark, Pig, and Hive service designed to easily and cost effectively process big datasets. You can quickly create managed clusters of any size and turn them off when you are finished, so you only pay for what you need. Cloud Dataproc is integrated across several Google Cloud Platform products, so you have access to a simple, powerful, and complete data processing platform.

Spark Advanced Workshop

Taking the notes and leading [Scala/Spark meetups in Warsaw, Poland](#) gave me opportunity to create the initial version of the **Spark Advanced workshop**. It is a highly-interactive in-depth 2-day workshop about Spark with many practical exercises.

Contact me at jacek@japila.pl to discuss having one at your convenient location and/or straight in the office. We could also host the workshop remotely.

It's is a hands-on workshop with lots of exercises and do-fail-fix-rinse-repeat cycles.

1. [Requirements](#)
2. [Day 1](#)
3. [Day 2](#)

Spark Advanced Workshop - Requirements

1. Linux or Mac OS (please no Windows - if you insist, use a virtual machine with Linux using [VirtualBox](#)).
2. The latest release of [Java™ Platform, Standard Edition Development Kit](#).
3. The latest release of Apache Spark **pre-built for Hadoop 2.6 and later** from [Download Spark](#).
4. Basic experience in developing simple applications using [Scala programming language](#) and [sbt](#).

Spark Advanced Workshop - Day 1

Agenda

1. RDD - Resilient Distributed Dataset - 45 mins
2. Setting up Spark Standalone cluster - 45 mins
3. Using Spark shell with Spark Standalone - 45 mins
4. WebUI - UI for Spark Monitoring - 45 mins
5. Developing Spark applications using Scala and sbt and deploying to the Spark Standalone cluster - 2 x 45 mins

Spark Advanced Workshop - Day 2

Agenda

1. [Using Listeners to monitor Spark's Scheduler](#) - 45 mins
2. [TaskScheduler and Speculative execution of tasks](#) - 45 mins
3. [Developing Custom RPC Environment \(RpcEnv\)](#) - 45 mins
4. [Spark Metrics System](#) - 45 mins
5. [Don't fear the logs - Learn Spark by Logs](#) - 45 mins

Spark Talks Ideas (STI)

This is the collection of talks I'm going to present at conferences, meetups, webinars, etc.

- Don't fear the logs - Learn Spark by Logs
- 10 Lesser-Known Tidbits about Spark Standalone
- [Learning Spark internals using groupBy \(to cause shuffle\)](#)
- Fault-tolerant stream processing using Spark Streaming
- Stateful stream processing using Spark Streaming

10 Lesser-Known Tidbits about Spark Standalone

Caution

[FIXME](#) Make sure the title reflects the number of tidbits.

- Duration: ...[FIXME](#)

Multiple Standalone Masters

Multiple standalone Masters in [master URL](#).

REST Server

Read [REST Server](#).

Spark Standalone High-Availability

Read [Recovery Mode](#).

SPARK_PRINT_LAUNCH_COMMAND and debugging

Read [Print Launch Command of Spark Scripts](#).

Note

It's not Standalone mode-specific thing.

spark-shell is spark-submit

Read [Spark shell](#).

Note

It's not Standalone mode-specific thing.

Application Management using spark-submit

Read [Application Management using spark-submit](#).

spark-* scripts and --conf options

You can use `--conf` or `-c`.

Refer to [Command-line Options](#).

Learning Spark internals using groupBy (to cause shuffle)

Execute the following operation and explain transformations, actions, jobs, stages, tasks, partitions using `spark-shell` and web UI.

```
sc.parallelize(0 to 999, 50).zipWithIndex.groupBy(_. _1 / 10).collect
```

You may also make it a little bit heavier with explaining data distribution over cluster and go over the concepts of drivers, masters, workers, and executors.

Glossary

DataFrame

A DataFrame is a programming abstraction for a distributed collection of data organized into named columns. See Spark SQL and DataFrame Guide.

[11.3.1.2. Example — Text Classification](#) [11.3.1.3. Example — Linear Regression](#)
[11.3.1. ML Pipelines](#) [11.2.6. Aggregation \(GroupedData\)](#) [11.2.3. DataFrame](#)
[11.2.3.2. DataFrameWriter](#) [11.2.2. Dataset](#)
[11.2.4. Standard Functions for DataFrames \(functions object\)](#)
[15.8. Using Spark SQL to update data in Hive using ORC files](#) [11.2.1. SQLContext](#)
[11.2.9. Windows in DataFrames](#) [11.2. Spark SQL](#)

FIXME

A place that begs for more relevant content.

[15.11. Developing Custom RDD](#) [15.10. Developing RPC Environment](#)
[15.2. Learning Jobs and Partitions Using take Action](#) [5.6. Block Manager](#)
[8.1. Broadcast variables](#) [5.11. Cache Manager](#) [6.2. Spark on cluster](#) [5.1.1. Jobs](#)
[5.1.2. Stages](#) [5.1. DAGScheduler](#) [4.1. Driver](#) [5.9. Dynamic Allocation](#)
[7. Execution Model](#) [5.4.1. CoarseGrainedExecutorBackend](#) [5.4. Executor Backend](#)
[4.4. Executors](#) [15.6. Your first complete Spark application \(using Scala and sbt\)](#)
[13.2. Spark and Hadoop](#) [5.7. HTTP File Server](#) [10.1. Using Input and Output \(I/O\)](#)
[10.3. Using Apache Kafka](#) [6.1. Spark local](#) [12.1. Logging](#) [4.2. Master](#)
[6.2.2. Spark on Mesos](#) [12.3. Spark Metrics System](#)
[11.3.1.2. Example — Text Classification](#) [11.3.1.3. Example — Linear Regression](#)
[11.3.1.1. Persistance](#) [11.3.1. ML Pipelines](#)
[11.3. Spark MLlib - Machine Learning in Spark](#) [2.3.7.6. BlockRDD](#)
[2.3.5. Checkpointing](#) [2.3.6. Dependencies](#) [2.3.7.4. HadoopRDD](#)
[2.3.1. Operators - Transformations and Actions](#) [2.3.1.1. mapPartitions](#)
[2.3.2. Partitions](#) [2.3.4. Shuffling](#) [2.3. RDD - Resilient Distributed Dataset](#)
[5.14.1. Netty-based RpcEnv](#) [5.14. RPC Environment \(RpcEnv\)](#)
[5. Spark Runtime Environment](#) [5.3.1. CoarseGrainedSchedulerBackend](#)
[5.3. Scheduler Backend](#) [12.4. Scheduler Listeners](#) [5.15. ContextCleaner](#)
[5.17. ExecutorAllocationManager](#) [5.16. MapOutputTracker](#) [5.5. Shuffle Manager](#)

2.2. `SparkContext` - the door to Spark 11.2.5.1. `ContinuousQueryManager`
11.2.3. `DataFrame` 11.2.3.1. `DataFrameReader` 11.2.2. `Dataset`
15.8. Using Spark SQL to update data in Hive using ORC files 11.2.7. `Joins`
11.2.1. `SQLContext` 11.2.8. UDFs — User-Defined Functions
11.2.9. Windows in DataFrames 11.2. Spark SQL
6.2.1.3. Management Scripts for Standalone Master 6.2.1.1. `Master`
6.2.1. Spark Standalone 11.1.14. Backpressure 11.1.5. Checkpointing
11.1.8. `DStreamGraph` 11.1.9. Discretized Streams (DStreams) 11.1.7. `JobGenerator`
11.1.6. `JobScheduler` 11.1.11.1. `KafkaRDD` 11.1.11. Ingesting Data from Kafka
11.1.9.6. `MapWithStateDStreams` 11.1.10.3. `ReceivedBlockHandlers`
11.1.9.2. `ReceiverInputDStreams` 11.1.10.2. `ReceiverSupervisors`
11.1.10.1. `ReceiverTracker` 11.1.16. `Settings` 11.1.9.7. `StateDStreams`
11.1.1. `StreamingContext` 11.1.4. Streaming Listeners 11.1.9.8. `TransformedDStream`
11.1. Spark Streaming 3.3. `spark-submit`
19.1. 10 Lesser-Known Tidbits about Spark Standalone 5.2.1. `Tasks` 5.2.2. `TaskSets`
5.2. Task Scheduler 5.2.4. `TaskSchedulerImpl` - Default TaskScheduler
5.2.3. `TaskSetManager` 3.2.1. Executors Tab 3.2. WebUI - UI for Spark Monitoring
4.3. Workers 6.2.3. Spark on YARN