

# Table of Contents

Introduction	1.1
Overview of Spark	1.2
Anatomy of Spark Application	1.3
SparkConf - Configuration for Spark Applications	1.3.1
Deploy Mode	1.3.1.1
SparkContext	1.3.2
HeartbeatReceiver RPC Endpoint	1.3.2.1
Inside Creating SparkContext	1.3.2.2
RDD - Resilient Distributed Dataset	1.3.3
Operators	1.3.3.1
Transformations	1.3.3.1.1
Actions	1.3.3.1.2
RDD Lineage — Logical Execution Plan	1.3.3.1.3
Partitions and Partitioning	1.3.3.2
Shuffling	1.3.3.3
Checkpointing	1.3.3.4
Dependencies	1.3.3.5
ParallelCollectionRDD	1.3.3.6
ParallelCollectionRDD	1.3.3.6.1
MapPartitionsRDD	1.3.3.6.2
PairRDDFunctions	1.3.3.6.3
CoGroupedRDD	1.3.3.6.4
HadoopRDD	1.3.3.6.5
ShuffledRDD	1.3.3.6.6
BlockRDD	1.3.3.6.7
Spark Tools	1.4
Spark Shell	1.4.1
WebUI — Spark Application's web UI	1.4.2
Stages Tab	1.4.2.1
Stages for All Jobs	1.4.2.1.1

Stage Details	1.4.2.1.2
Pool Details	1.4.2.1.3
Storage Tab	1.4.2.2
Executors Tab	1.4.2.3
SQL Tab	1.4.2.4
SQLListener	1.4.2.4.1
JobProgressListener	1.4.2.5
spark-submit	1.4.3
spark-class	1.4.4
Spark Architecture	1.5
Driver	1.5.1
Master	1.5.2
Workers	1.5.3
Executors	1.5.4
TaskRunner	1.5.4.1
Spark Services	1.6
MemoryManager — Memory Management	1.6.1
UnifiedMemoryManager	1.6.1.1
SparkEnv — Spark Runtime Environment	1.6.2
DAGScheduler	1.6.3
Jobs	1.6.3.1
Stages	1.6.3.2
ShuffleMapStage — Intermediate Stage in Job	1.6.3.2.1
ResultStage — Final Stage in Job	1.6.3.2.2
Task Scheduler	1.6.4
Tasks	1.6.4.1
TaskSets	1.6.4.2
Schedulable	1.6.4.3
TaskSetManager	1.6.4.3.1
Schedulable Pool	1.6.4.3.2
Schedulable Builders	1.6.4.3.3
FIFOSchedulableBuilder	1.6.4.3.3.1
FairSchedulableBuilder	1.6.4.3.3.2
Scheduling Mode	1.6.4.3.4

TaskSchedulerImpl - Default TaskScheduler	1.6.4.4
TaskContext	1.6.4.5
TaskMemoryManager	1.6.4.6
MemoryConsumer	1.6.4.6.1
TaskMetrics	1.6.4.7
Scheduler Backend	1.6.5
CoarseGrainedSchedulerBackend	1.6.5.1
Executor Backend	1.6.6
CoarseGrainedExecutorBackend	1.6.6.1
BlockManager	1.6.7
MemoryStore	1.6.7.1
DiskStore	1.6.7.2
BlockDataManager	1.6.7.3
ShuffleClient	1.6.7.4
BlockTransferService	1.6.7.5
BlockManagerMaster	1.6.7.6
BlockInfoManager	1.6.7.7
BlockInfo	1.6.7.7.1
Dynamic Allocation (of Executors)	1.6.8
ExecutorAllocationManager	1.6.8.1
ExecutorAllocationClient	1.6.8.2
ExecutorAllocationListener	1.6.8.3
ExecutorAllocationManagerSource	1.6.8.4
Shuffle Manager	1.6.9
ExternalShuffleService	1.6.9.1
ExternalClusterManager	1.6.10
HTTP File Server	1.6.11
Broadcast Manager	1.6.12
Data Locality	1.6.13
Cache Manager	1.6.14
Spark, Akka and Netty	1.6.15
OutputCommitCoordinator	1.6.16
RPC Environment (RpcEnv)	1.6.17

<a href="#">Netty-based RpcEnv</a>	1.6.17.1
<a href="#">ContextCleaner</a>	1.6.18
<a href="#">MapOutputTracker</a>	1.6.19
<a href="#">Deployment Environments — Run Modes</a>	1.7
<a href="#">Spark local (pseudo-cluster)</a>	1.7.1
<a href="#">Spark on cluster</a>	1.7.2
<a href="#">Spark on YARN</a>	1.7.2.1
<a href="#">YarnShuffleService — ExternalShuffleService on YARN</a>	1.7.2.1.1
<a href="#">ExecutorRunnable</a>	1.7.2.1.2
<a href="#">Client</a>	1.7.2.1.3
<a href="#">YarnRMClient</a>	1.7.2.1.4
<a href="#">ApplicationMaster</a>	1.7.2.1.5
<a href="#">AMEndpoint — ApplicationMaster RPC Endpoint</a>	1.7.2.1.5.1
<a href="#">YarnClusterManager — ExternalClusterManager for YARN</a>	1.7.2.1.6
<a href="#">TaskSchedulers for YARN</a>	1.7.2.1.7
<a href="#">YarnScheduler</a>	1.7.2.1.7.1
<a href="#">YarnClusterScheduler</a>	1.7.2.1.7.2
<a href="#">SchedulerBackends for YARN</a>	1.7.2.1.8
<a href="#">YarnSchedulerBackend</a>	1.7.2.1.8.1
<a href="#">YarnClientSchedulerBackend</a>	1.7.2.1.8.2
<a href="#">YarnClusterSchedulerBackend</a>	1.7.2.1.8.3
<a href="#">YarnSchedulerEndpoint RPC Endpoint</a>	1.7.2.1.8.4
<a href="#">YarnAllocator</a>	1.7.2.1.9
<a href="#">Introduction to Hadoop YARN</a>	1.7.2.1.10
<a href="#">Setting up YARN Cluster</a>	1.7.2.1.11
<a href="#">Kerberos</a>	1.7.2.1.12
<a href="#">ConfigurableCredentialManager</a>	1.7.2.1.12.1
<a href="#">ClientDistributedCacheManager</a>	1.7.2.1.13
<a href="#">YarnSparkHadoopUtil</a>	1.7.2.1.14
<a href="#">Settings</a>	1.7.2.1.15
<a href="#">Spark Standalone</a>	1.7.2.2
<a href="#">Standalone Master</a>	1.7.2.2.1
<a href="#">Standalone Worker</a>	1.7.2.2.2
<a href="#">web UI</a>	1.7.2.2.3

Submission Gateways	1.7.2.2.4
Management Scripts for Standalone Master	1.7.2.2.5
Management Scripts for Standalone Workers	1.7.2.2.6
Checking Status	1.7.2.2.7
Example 2-workers-on-1-node Standalone Cluster (one executor per worker)	1.7.2.2.8
StandaloneSchedulerBackend	1.7.2.2.9
Spark on Mesos	1.7.2.3
MesosCoarseGrainedSchedulerBackend	1.7.2.3.1
About Mesos	1.7.2.3.2
Execution Model	1.8
Optimising Spark	1.9
Caching and Persistence	1.9.1
Broadcast variables	1.9.2
Accumulators	1.9.3
Spark Security	1.10
Spark Security	1.10.1
Securing Web UI	1.10.2
Data Sources in Spark	1.11
Using Input and Output (I/O)	1.11.1
Spark and Parquet	1.11.1.1
Serialization	1.11.1.2
Spark and Cassandra	1.11.2
Spark and Kafka	1.11.3
Couchbase Spark Connector	1.11.4
Spark Application Frameworks	1.12
Spark SQL	1.12.1
Logical Query Plan Analyzer	1.12.1.1
SparkSession — The Entry Point	1.12.1.2
Builder — Building SparkSession with Fluent API	1.12.1.2.1
CacheManager	1.12.1.3
Caching	1.12.1.4
Debugging Query Execution	1.12.1.5
Dataset	1.12.1.6

LogicalPlan	1.12.1.6.1
Schema — Shape of Records	1.12.1.6.2
Data Types	1.12.1.6.3
Encoders — Internal Row Converters	1.12.1.6.4
InternalRow — Internal Binary Row Format	1.12.1.6.4.1
Columns	1.12.1.6.5
DataFrame (Dataset[Row])	1.12.1.6.6
Row	1.12.1.6.7
DataSource API — Loading and Saving Datasets	1.12.1.7
DataFrameReader	1.12.1.7.1
DataFrameWriter	1.12.1.7.2
DataSource	1.12.1.7.3
DataSourceRegister	1.12.1.7.4
Custom Formats	1.12.1.7.5
Standard Functions (functions object)	1.12.1.8
Standard Functions (functions object)	1.12.1.8.1
Aggregation (GroupedData)	1.12.1.8.2
User-Defined Functions (UDFs)	1.12.1.8.3
Window Aggregates (Windows)	1.12.1.8.4
Structured Streaming	1.12.1.9
DataStreamReader	1.12.1.9.1
DataStreamWriter	1.12.1.9.2
Streaming Sources	1.12.1.9.3
FileStreamSource	1.12.1.9.3.1
Streaming Sinks	1.12.1.9.4
ConsoleSink	1.12.1.9.4.1
ForeachSink	1.12.1.9.4.2
StreamSourceProvider	1.12.1.9.5
StreamSinkProvider	1.12.1.9.6
StreamingQueryManager	1.12.1.9.7
StreamingQuery	1.12.1.9.8
Trigger	1.12.1.9.9
StreamExecution	1.12.1.9.10

<a href="#">StreamingRelation</a>	1.12.1.9.11
<a href="#">StreamingQueryListenerBus</a>	1.12.1.9.12
<a href="#">Joins</a>	1.12.1.10
<a href="#">SQLConf</a>	1.12.1.11
<a href="#">Catalog</a>	1.12.1.12
<a href="#">Datasets vs RDDs</a>	1.12.1.13
<a href="#">SessionState</a>	1.12.1.14
<a href="#">SQL Parser Framework</a>	1.12.1.15
<a href="#">SQLExecution Helper Object</a>	1.12.1.16
<a href="#">SQLContext</a>	1.12.1.17
<a href="#">Catalyst Query Optimizer</a>	1.12.1.18
<a href="#">Catalyst Query Optimizer</a>	1.12.1.18.1
<a href="#">Predicate Pushdown Optimizer</a>	1.12.1.18.1.1
<a href="#">Constant Folding Optimizer</a>	1.12.1.18.1.2
<a href="#">Nullability (NULL Value) Propagation Optimizer</a>	1.12.1.18.1.3
<a href="#">Vectorized Parquet Decoder</a>	1.12.1.18.1.4
<a href="#">QueryPlan</a>	1.12.1.18.1.5
<a href="#">SparkPlan — Spark Query Planner</a>	1.12.1.18.1.6
<a href="#">QueryPlanner</a>	1.12.1.18.1.7
<a href="#">QueryExecution</a>	1.12.1.18.1.8
<a href="#">Whole-Stage Code Generation (CodeGen)</a>	1.12.1.18.1.9
<a href="#">Project Tungsten</a>	1.12.1.18.2
<a href="#">Hive Integration</a>	1.12.1.19
<a href="#">Spark SQL CLI - spark-sql</a>	1.12.1.19.1
<a href="#">Settings</a>	1.12.1.20
<a href="#">Spark Streaming</a>	1.12.2
<a href="#">StreamingContext</a>	1.12.2.1
<a href="#">Stream Operators</a>	1.12.2.2
<a href="#">Windowed Operators</a>	1.12.2.2.1
<a href="#">SaveAs Operators</a>	1.12.2.2.2
<a href="#">Stateful Operators</a>	1.12.2.2.3
<a href="#">web UI and Streaming Statistics Page</a>	1.12.2.3
<a href="#">Streaming Listeners</a>	1.12.2.4
<a href="#">Checkpointing</a>	1.12.2.5

JobScheduler	1.12.2.6
JobGenerator	1.12.2.7
DStreamGraph	1.12.2.8
Discretized Streams (DStreams)	1.12.2.9
Input DStreams	1.12.2.9.1
ReceiverInputDStreams	1.12.2.9.2
ConstantInputDStreams	1.12.2.9.3
ForEachDStreams	1.12.2.9.4
WindowedDStreams	1.12.2.9.5
MapWithStateDStreams	1.12.2.9.6
StateDStreams	1.12.2.9.7
TransformedDStream	1.12.2.9.8
Receivers	1.12.2.10
ReceiverTracker	1.12.2.10.1
ReceiverSupervisors	1.12.2.10.2
ReceivedBlockHandlers	1.12.2.10.3
Ingesting Data from Kafka	1.12.2.11
KafkaRDD	1.12.2.11.1
RecurringTimer	1.12.2.12
Backpressure	1.12.2.13
Dynamic Allocation (Elastic Scaling)	1.12.2.14
ExecutorAllocationManager	1.12.2.14.1
Settings	1.12.2.15
Spark MLlib - Machine Learning in Spark	1.12.3
ML Pipelines (spark.ml)	1.12.3.1
Transformers	1.12.3.1.1
Estimators	1.12.3.1.2
Models	1.12.3.1.3
Evaluators	1.12.3.1.4
CrossValidator	1.12.3.1.5
Persistence (MLWriter and MLReader)	1.12.3.1.6
Example — Text Classification	1.12.3.1.7
Example — Linear Regression	1.12.3.1.8

<a href="#">Latent Dirichlet Allocation (LDA)</a>	1.12.3.2
<a href="#">Vector</a>	1.12.3.3
<a href="#">LabeledPoint</a>	1.12.3.4
<a href="#">Streaming MLlib</a>	1.12.3.5
<a href="#">Spark GraphX - Distributed Graph Computations</a>	1.12.4
<a href="#">Graph Algorithms</a>	1.12.4.1
<a href="#">Monitoring, Tuning and Debugging</a>	1.13
<a href="#">Unified Memory Management</a>	1.13.1
<a href="#">HistoryServer</a>	1.13.2
<a href="#">SQLHistoryListener</a>	1.13.2.1
<a href="#">FsHistoryProvider</a>	1.13.2.2
<a href="#">Logging</a>	1.13.3
<a href="#">Performance Tuning</a>	1.13.4
<a href="#">Spark Metrics System</a>	1.13.5
<a href="#">Spark Listeners</a>	1.13.6
<a href="#">LiveListenerBus</a>	1.13.6.1
<a href="#">ReplayListenerBus</a>	1.13.6.2
<a href="#">EventLoggingListener — Event Logging</a>	1.13.6.3
<a href="#">StatsReportListener — Logging Summary Statistics</a>	1.13.6.4
<a href="#">Debugging Spark using sbt</a>	1.13.7
<a href="#">Building Spark</a>	1.14
<a href="#">Building Spark</a>	1.14.1
<a href="#">Spark and Hadoop</a>	1.14.2
<a href="#">Spark and software in-memory file systems</a>	1.14.3
<a href="#">Spark and The Others</a>	1.14.4
<a href="#">Distributed Deep Learning on Spark</a>	1.14.5
<a href="#">Spark Packages</a>	1.14.6
<a href="#">TransportConf — Transport Configuration</a>	1.14.7
<a href="#">Interactive Notebooks</a>	1.15
<a href="#">Apache Zeppelin</a>	1.15.1
<a href="#">Spark Notebook</a>	1.15.2
<a href="#">Spark Tips and Tricks</a>	1.16
<a href="#">Access private members in Scala in Spark shell</a>	1.16.1
<a href="#">SparkException: Task not serializable</a>	1.16.2

Running Spark on Windows	1.16.3
Exercises	1.17
One-liners using PairRDDFunctions	1.17.1
Learning Jobs and Partitions Using take Action	1.17.2
Spark Standalone - Using ZooKeeper for High-Availability of Master	1.17.3
Spark's Hello World using Spark shell and Scala	1.17.4
WordCount using Spark shell	1.17.5
Your first complete Spark application (using Scala and sbt)	1.17.6
Spark (notable) use cases	1.17.7
Using Spark SQL to update data in Hive using ORC files	1.17.8
Developing Custom SparkListener to monitor DAGScheduler in Scala	1.17.9
Developing RPC Environment	1.17.10
Developing Custom RDD	1.17.11
Creating DataFrames from Tables using JDBC and PostgreSQL	1.17.12
Causing Stage to Fail	1.17.13
Courses	1.18
Courses	1.18.1
Books	1.18.2
DataStax Enterprise	1.19
DataStax Enterprise	1.19.1
MapR Sandbox for Hadoop (Spark 1.5.2 only)	1.19.2
Commercial Products using Apache Spark	1.20
IBM Analytics for Apache Spark	1.20.1
Google Cloud Dataproc	1.20.2
Spark Advanced Workshop	1.21
Requirements	1.21.1
Day 1	1.21.2
Day 2	1.21.3
Spark Talks Ideas (STI)	1.22
10 Lesser-Known Tidbits about Spark Standalone	1.22.1
Learning Spark internals using groupBy (to cause shuffle)	1.22.2



# Mastering Apache Spark

Welcome to Mastering Apache Spark (aka #SparkNotes)!

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

Contact me at [jacek@japila.pl](mailto:jacek@japila.pl) or [@jaceklaskowski](https://twitter.com/jaceklaskowski) to discuss Spark opportunities, e.g. courses, workshops, or other mentoring or development services.

If you like the Apache Spark notes you should seriously consider participating in my own, very hands-on [Spark Workshops for Developers, Administrators and Operators](#).

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](https://twitter.com/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.

# Apache Spark

Apache Spark is an **open-source distributed general-purpose cluster computing framework** with **in-memory data processing engine** that can do ETL, analytics, machine learning and graph processing on large volumes of data at rest (batch processing) or in motion (streaming processing) with **rich concise high-level APIs** for the programming languages: Scala, Python, Java, R, and SQL.

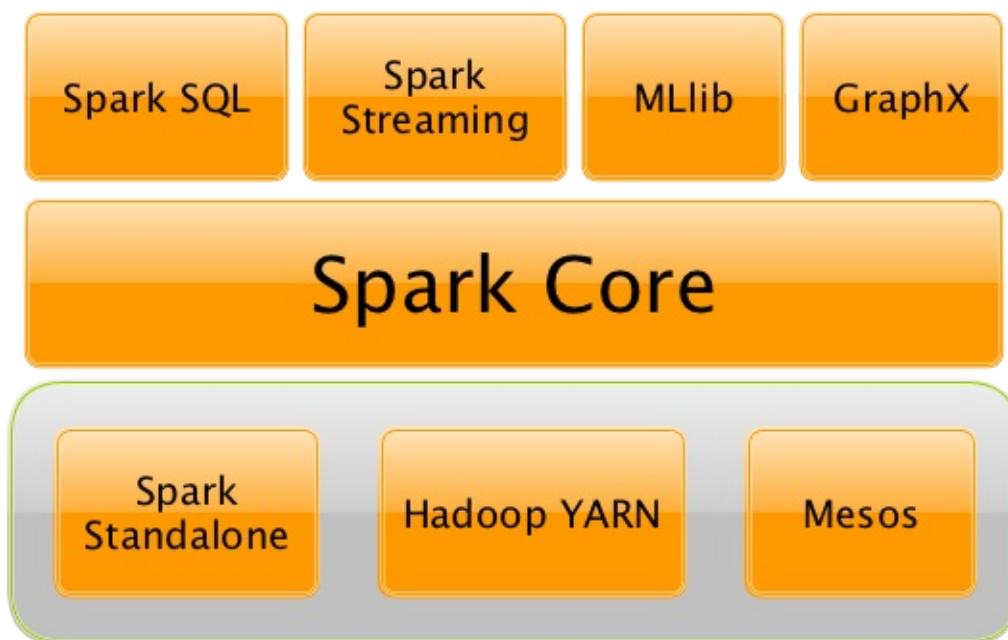


Figure 1. The Spark Platform

You could also describe Spark as 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.

Note	When you hear "Apache Spark" it can be two things — the Spark engine aka <b>Spark Core</b> or the Apache Spark open source project which is an "umbrella" term for Spark Core and the accompanying <a href="#">Spark Application Frameworks</a> , i.e. <a href="#">Spark SQL</a> , <a href="#">Spark Streaming</a> , <a href="#">Spark MLlib</a> and <a href="#">Spark GraphX</a> that sit on top of Spark Core and the main data abstraction in Spark called <a href="#">RDD - Resilient Distributed Dataset</a> .
------	---

## 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.

### Easy to Get Started

Spark offers [spark-shell](#) that makes for a very easy head start to writing and running Spark applications on the command line on your laptop.

You could then use [Spark Standalone](#) built-in cluster manager to deploy your Spark applications to a production-grade cluster to run on a full dataset.

### Unified Engine for 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 combines batch, interactive, and streaming workloads under one rich concise API.

Spark 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 / Exploratory Analytics

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, R or SQL, 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 (Datasets), Spark MLlib (ML Pipelines), Spark GraphX (Graphs) 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.

## Data Integration Toolkit with Rich Set of Supported Data Sources

Spark can read from many types of data sources — relational, NoSQL, file systems, etc. — using many types of data formats - Parquet, Avro, CSV, JSON.

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 warehouses the 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 Concise High-Level API

Spark offers a **unified, concise, high-level APIs** for batch analytics (RDD API), SQL queries (Dataset API), real-time analysis (DStream API), machine learning (ML Pipeline API) and graph processing (Graph API).

Developers no longer have to learn many different processing engines and platforms, and let the time be spent on mastering framework APIs per use case (atop a single computation engine Spark).

## 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.

## Further reading or watching

- (video) [Keynote: Spark 2.0 - Matei Zaharia, Apache Spark Creator and CTO of Databricks](#)

# 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](#).

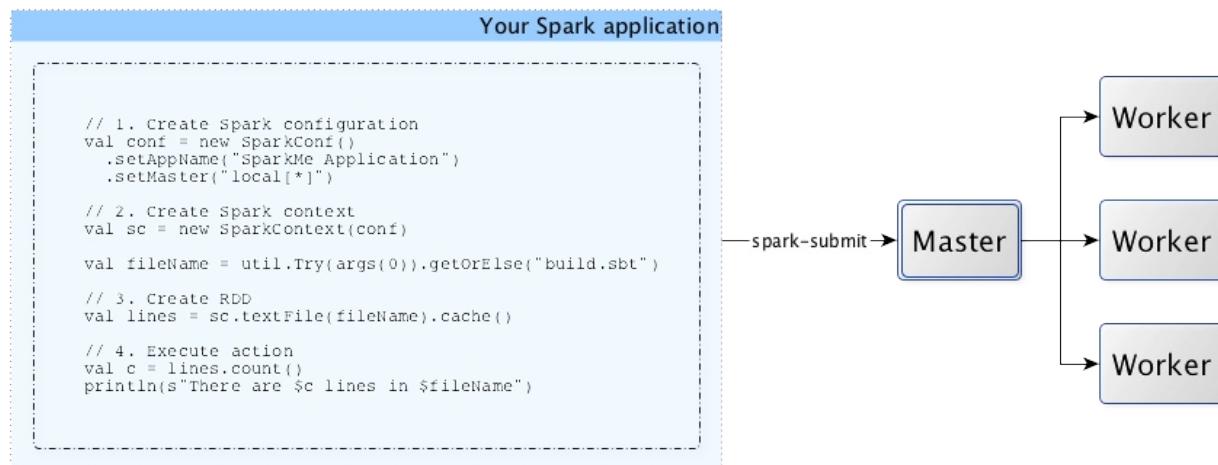


Figure 1. Submitting Spark application to master using master URL

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 <a href="#">Deployment Modes</a> .
------	--

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

There are three ways to configure Spark and user programs:

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

## Mandatory Settings - `spark.master` and `spark.app.name`

There are two mandatory settings of any Spark application that have to be defined before this Spark application could be run — `spark.master` and `spark.app.name`.

### `spark.master` - Master URL

Caution	<a href="#">FIXME</a>
---------	-----------------------

### `spark.app.name` - Application Name

## 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. The instance of `SparkConf` can be used to create [SparkContext](#).

Start [Spark shell](#) with `--conf spark.logConf=true` to log the effective Spark configuration as INFO when `SparkContext` is started.

```
$ ./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
...
```

Tip

Use `sc.getConf.toDebugString` to have a richer output once `SparkContext` has finished initializing.

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` or `-c` - 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 simply loads `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,""), (spark.master,local[*]), (spark.submit.deployMode,client))

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
```

# Deploy Mode

**Deploy mode** specifies the location of where `driver` executes in the [deployment environment](#).

Deploy mode can be one of the following options:

- `client` (default) - the driver runs on the machine that the Spark application was launched.
- `cluster` - the driver runs on a random node in a cluster.

Note	<code>cluster</code> deploy mode is only available for <a href="#">non-local cluster deployments</a> .
------	--

You can control deploy mode using `spark-submit`'s `--deploy-mode` or `--conf` command-line options with `spark.submit.deployMode` setting.

Note	<code>spark.submit.deployMode</code> setting can be <code>client</code> or <code>cluster</code> .
------	---

## Client Mode

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Cluster Mode

Caution	<a href="#">FIXME</a>
---------	-----------------------

## `spark.submit.deployMode`

`spark.submit.deployMode` (default: `client`) can be `client` or `cluster`.

# SparkContext - Entry Point to Spark

`SparkContext` (aka **Spark context**) is the entry point to Spark for a Spark application.

Note	You could also assume that a <code>SparkContext</code> instance <i>is</i> a Spark application.
------	--

It [sets up internal services](#) and establishes a connection to a [Spark execution environment](#) ([deployment mode](#)).

Once a [SparkContext instance is created](#) you can use it 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).

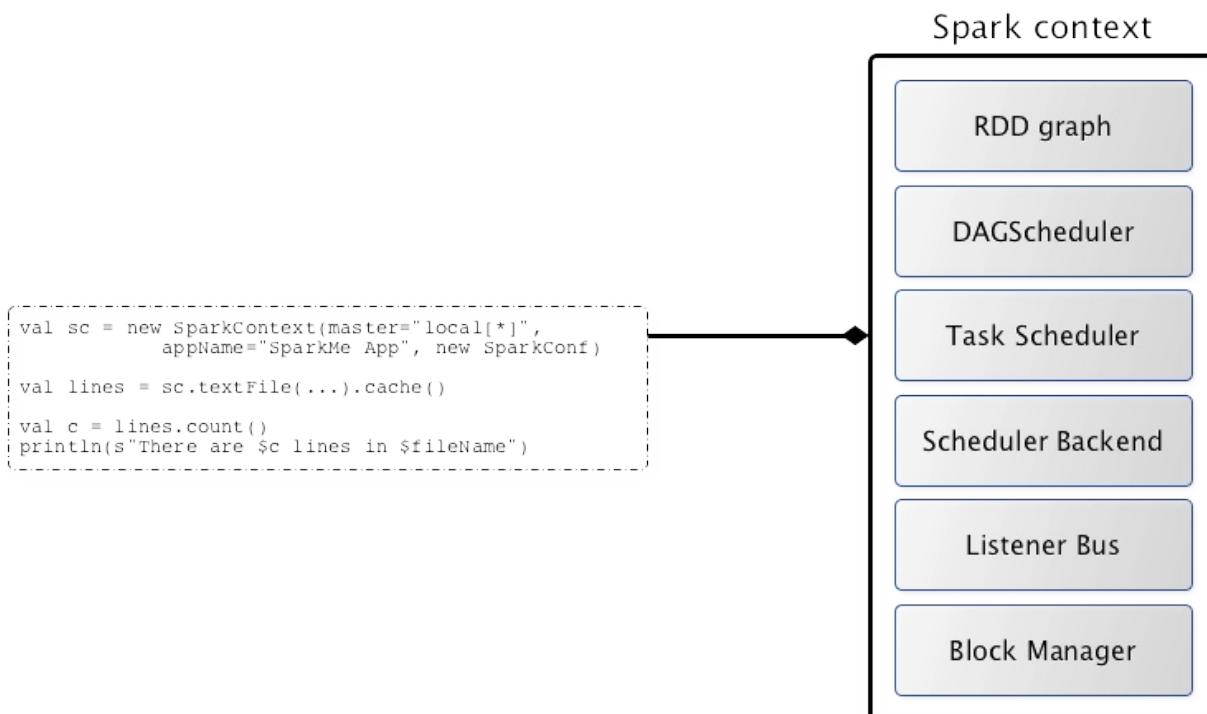


Figure 1. Spark context acts as the master of your Spark application

`SparkContext` offers the following functions:

- Getting current configuration
  - [SparkConf](#)
  - [deployment environment \(as master URL\)](#)
  - [application name](#)

- deploy mode
  - default level of parallelism
  - Spark user
  - the time (in milliseconds) when `SparkContext` was created
  - Spark version
- Setting configuration
    - mandatory master URL
    - local properties
    - default log level
  - Creating objects
    - RDDs
    - accumulators
    - broadcast variables
  - Accessing services, e.g. TaskScheduler, LiveListenerBus, BlockManager, SchedulerBackends, ShuffleManager.
  - Running jobs
  - Setting up custom Scheduler Backend, TaskScheduler and DAGScheduler
  - Closure Cleaning
  - Submitting Jobs Asynchronously
  - Unpersisting RDDs, i.e. marking RDDs as non-persistent
  - Registering SparkListener
  - Programmable Dynamic Allocation

Tip	Read the scaladoc of <a href="#">org.apache.spark.SparkContext</a> .
-----	--

## Persisted RDDs

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

### **persistRDD**

```
persistRDD(rdd: RDD[_])
```

`persistRDD` is a `private[spark]` method to register `rdd` in `persistentRdds` registry.

## Programmable Dynamic Allocation

`SparkContext` offers a developer API for [dynamic allocation of executors](#):

- [requestExecutors](#)
- [killExecutors](#)
- (private!) [requestTotalExecutors](#)
- (private!) [getExecutorIds](#)

## Getting Executor Ids (`getExecutorIds` method)

`getExecutorIds` is a `private[spark]` method that is a part of [ExecutorAllocationClient contract](#). It simply [passes the call on to the current coarse-grained scheduler backend](#), i.e. calls `getExecutorIds`.

Note

It works for [coarse-grained scheduler backends](#) only.

When called for other scheduler backends you should see the following WARN message in the logs:

```
WARN Requesting executors is only supported in coarse-grained mode
```

Caution

[FIXME](#) Why does `SparkContext` implement the method for coarse-grained scheduler backends? Why doesn't `SparkContext` throw an exception when the method is called? Nobody seems to be using it (!)

## requestExecutors method

```
requestExecutors(numAdditionalExecutors: Int): Boolean
```

`requestExecutors` requests `numAdditionalExecutors` executors from [CoarseGrainedSchedulerBackend](#).

## Requesting to Kill Executors (`killExecutors` method)

```
killExecutors(executorIds: Seq[String]): Boolean
```

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

## requestTotalExecutors method

```
requestTotalExecutors(  
    numExecutors: Int,  
    localityAwareTasks: Int,  
    hostToLocalTaskCount: Map[String, Int]): Boolean
```

`requestTotalExecutors` is a `private[spark]` method that [requests the exact number of executors from a coarse-grained scheduler backend](#).

Note	It works for <a href="#">coarse-grained scheduler backends only</a> .
------	---

When called for other scheduler backends you should see the following WARN message in the logs:

```
WARN Requesting executors is only supported in coarse-grained mode
```

## Creating SparkContext

You can create a `SparkContext` instance with or without creating a `SparkConf` object first.

## Getting Existing or Creating New SparkContext (getOrCreate methods)

```
getOrCreate(): SparkContext  
getOrCreate(conf: SparkConf): SparkContext
```

`SparkContext.getOrCreate` methods allow you to get the existing `SparkContext` or create a new one.

```

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

// Using an explicit SparkConf object
import org.apache.spark.SparkConf
val conf = new SparkConf()
.setMaster("local[*]")
.setAppName("SparkMe App")
val sc = SparkContext.getOrCreate(conf)

```

The no-param `getOrCreate` method requires that the two mandatory Spark settings - `master` and `application name` - are specified using `spark-submit`.

## Constructors

```

SparkContext()
SparkContext(conf: SparkConf)
SparkContext(master: String, appName: String, conf: SparkConf)
SparkContext(
  master: String,
  appName: String,
  sparkHome: String = null,
  jars: Seq[String] = Nil,
  environment: Map[String, String] = Map())

```

You can create a `SparkContext` instance using the four constructors.

```

import org.apache.spark.SparkConf
val conf = new SparkConf()
.setMaster("local[*]")
.setAppName("SparkMe App")

import org.apache.spark.SparkContext
val sc = new SparkContext(conf)

```

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

```
INFO SparkContext: Running Spark version 2.0.0-SNAPSHOT
```

Note

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).

## Getting Current SparkConf (getConf method)

```
getConf: SparkConf
```

`getConf` returns the current [SparkConf](#).

Note	Changing the <code>SparkConf</code> object does not change the current configuration (as the method returns a copy).
------	--

## Getting Deployment Environment (master method)

```
master: String
```

`master` method returns the current value of [spark.master](#) which is the deployment environment in use.

## Getting Application Name (appName method)

```
appName: String
```

`appName` returns the value of the mandatory [spark.app.name](#) setting.

Note	It is used in <a href="#">SparkDeploySchedulerBackend</a> (to create a <a href="#">ApplicationDescription</a> when it starts), for <a href="#">SparkUI.createLiveUI</a> (when <a href="#">spark.ui.enabled</a> is enabled), when <code>postApplicationStart</code> is executed, and for Mesos and checkpointing in Spark Streaming.
------	---

## Getting Deploy Mode (deployMode method)

```
deployMode: String
```

`deployMode` returns the current value of [spark.submit.deployMode](#) setting or `client` if not set.

## Getting Scheduling Mode (getSchedulingMode method)

```
getSchedulingMode: SchedulingMode.SchedulingMode
```

`getSchedulingMode` returns the current [Scheduling Mode](#).

## Getting Schedulable (Pool) by Name (getPoolForName method)

```
getPoolForName(pool: String): Option[Schedulable]
```

`getPoolForName` returns a [Schedulable](#) by the `pool` name, if one exists.

Note	<code>getPoolForName</code> is part of the Developer's API and may change in the future.
------	--

Internally, it requests the [TaskScheduler](#) for the root pool and looks up the [Schedulable](#) by the `pool` name.

It is exclusively used to [show pool details in web UI \(for a stage\)](#).

## Getting All Pools (getAllPools method)

```
getAllPools: Seq[Schedulable]
```

`getAllPools` collects the [Pools](#) in [TaskScheduler.rootPool](#).

Note	<code>TaskScheduler.rootPool</code> is part of the <a href="#">TaskScheduler Contract</a> .
------	---

Note	<code>getAllPools</code> is part of the Developer's API.
------	--

Caution	<a href="#">FIXME</a> Where is the method used?
---------	---

Note	<code>getAllPools</code> is used to calculate pool names for <a href="#">Stages tab in web UI</a> with FAIR scheduling mode used.
------	---

## Computing Default Level of Parallelism

**Default level of parallelism** is the number of [partitions](#) in RDDs when created without specifying them 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](#).

Internally, `defaultParallelism` relays requests for the default level of parallelism to [TaskScheduler](#) (it is a part of its contract).

## Getting Spark Version

```
version: String
```

`version` returns the Spark version this `SparkContext` uses.

## Setting Local Properties

```
setLocalProperty(key: String, value: String): Unit
```

`setLocalProperty` sets a local thread-scoped `key` property to `value`.

```
sc.setLocalProperty("spark.scheduler.pool", "myPool")
```

The goal of the local property concept is to differentiate between or group jobs submitted from different threads by local properties.

Note

It is used to [group jobs into pools in FAIR job scheduler by spark.scheduler.pool per-thread property](#) and in [SQLExecution.withNewExecutionId Helper Methods](#)

If `value` is `null` the `key` property is removed the `key` from the local properties

```
sc.setLocalProperty("spark.scheduler.pool", null)
```

A common use case for the local property concept is to set a local property in a thread, say `spark.scheduler.pool`, after which all jobs submitted within the thread will be grouped, say into a pool by FAIR job scheduler.

```
val rdd = sc.parallelize(0 to 9)

sc.setLocalProperty("spark.scheduler.pool", "myPool")

// these two jobs (one per action) will run in the myPool pool
rdd.count
rdd.collect

sc.setLocalProperty("spark.scheduler.pool", null)

// this job will run in the default pool
rdd.count
```

## 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
---------	-------

## 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.

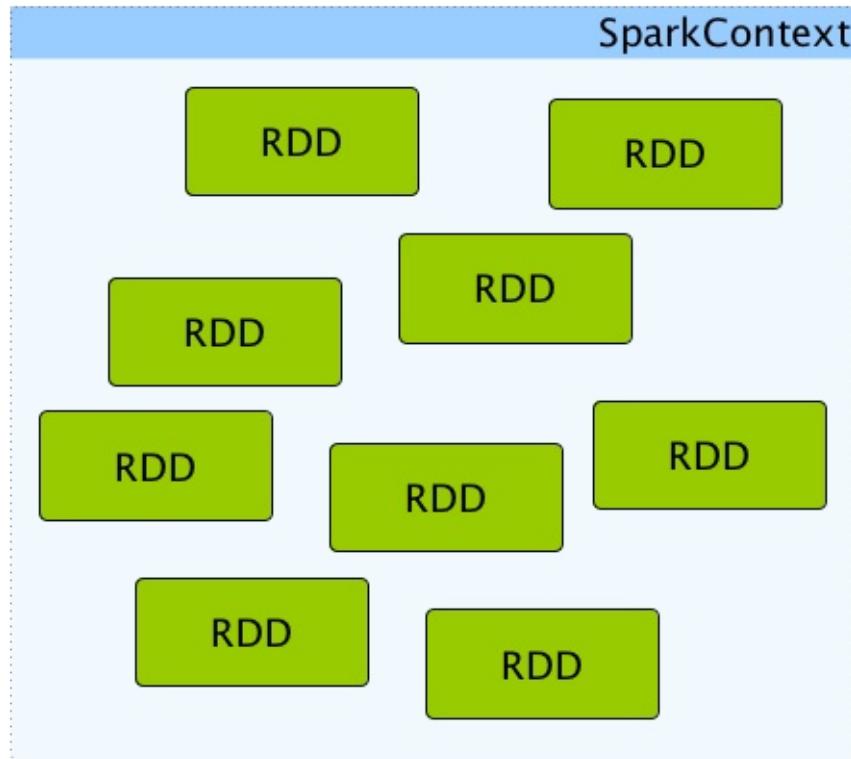


Figure 2. A Spark context creates a living space for RDDs.

## 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 `SparkListenerUnpersistRDD` message to `listenerBus`.

## Setting Checkpoint Directory (`setCheckpointDir` method)

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

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

Caution

[FIXME](#)

## Registering Custom Accumulators (register methods)

```
register(acc: AccumulatorV2[_, _]): Unit  
register(acc: AccumulatorV2[_, _], name: String): Unit
```

`register` registers the `acc` [accumulator](#). You can optionally give an accumulator a `name`.

Tip

You can create built-in accumulators for longs, doubles, and collection types using [specialized methods](#).

Internally, `register` registers the `SparkContext` to the accumulator.

## Creating Built-In Accumulators

```
longAccumulator: LongAccumulator  
longAccumulator(name: String): LongAccumulator  
doubleAccumulator: DoubleAccumulator  
doubleAccumulator(name: String): DoubleAccumulator  
collectionAccumulator[T]: CollectionAccumulator[T]  
collectionAccumulator[T](name: String): CollectionAccumulator[T]
```

You can use `longAccumulator`, `doubleAccumulator` or `collectionAccumulator` to create and register [accumulators](#) for simple and collection values.

`longAccumulator` returns [LongAccumulator](#) with the zero value `0`.

`doubleAccumulator` returns [DoubleAccumulator](#) with the zero value `0.0`.

`collectionAccumulator` returns [CollectionAccumulator](#) with the zero value `java.util.List[T]`.

```

scala> val acc = sc.longAccumulator
acc: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: None, value: 0)

scala> val counter = sc.longAccumulator("counter")
counter: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 1, name: Some(counter), value: 0)

scala> counter.value
res0: Long = 0

scala> sc.parallelize(0 to 9).foreach(n => counter.add(n))

scala> counter.value
res3: Long = 45

```

The `name` input parameter allows you to give a name to an accumulator and have it displayed in [Spark UI](#) (under Stages tab for a given stage).

Accumulators										
Tasks										
Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Figure 3. Accumulators in the Spark UI

Tip	You can register custom accumulators using <a href="#">register</a> methods.
-----	--

## Creating Broadcast Variables

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

`broadcast` method creates a [broadcast variable](#) that is a shared memory with `value` on all Spark executors.

```
scala> val hello = sc.broadcast("hello")
hello: 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.

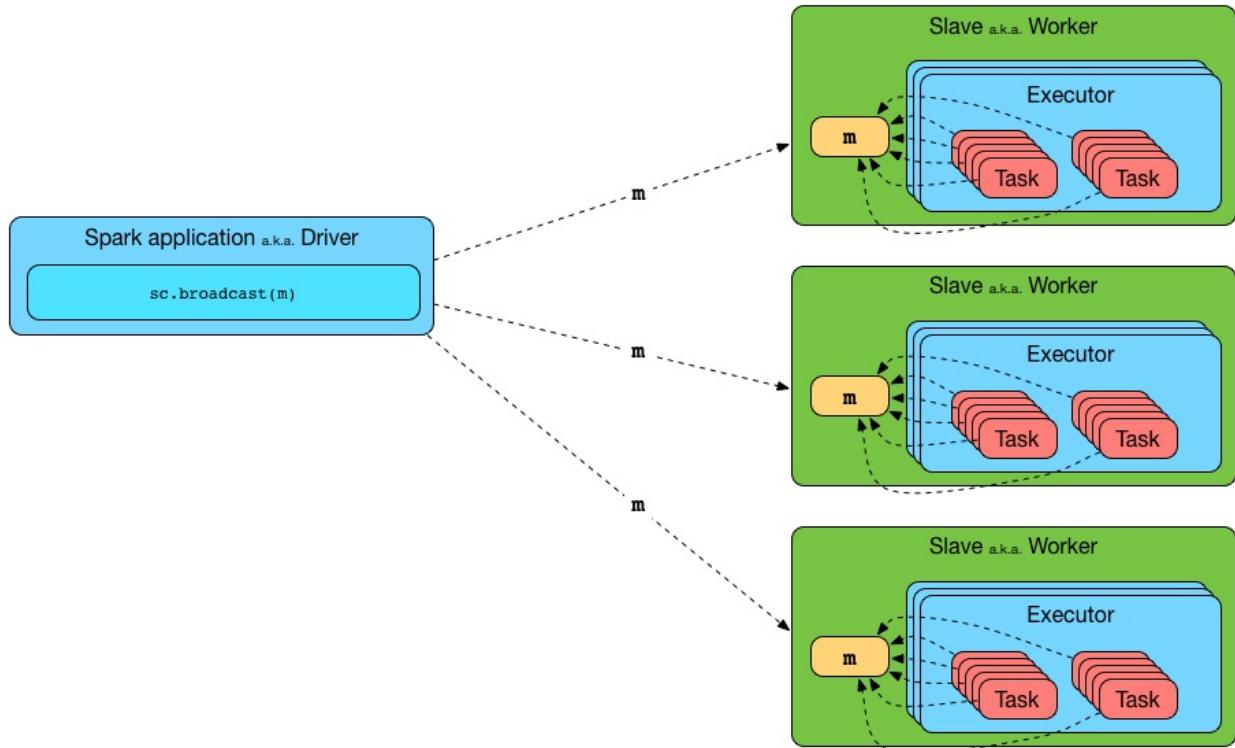


Figure 4. Broadcasting a value to executors

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

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

	Spark does not support broadcasting RDDs.
Note	<pre>scala&gt; sc.broadcast(sc.range(0, 10)) java.lang.IllegalArgumentException: requirement failed: Can not directly broadcast RDDs         at scala.Predef\$.require(Predef.scala:224)         at org.apache.spark.SparkContext.broadcast(SparkContext.scala:1370) ... 48 elided</pre>

## 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/build.sbt with timestamp 1447275294457
```

Caution	<a href="#">FIXME</a> 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 (runJob methods)

```
runJob[T, U](
  rdd: RDD[T],
  func: (TaskContext, Iterator[T]) => U,
  partitions: Seq[Int],
  resultHandler: (Int, U) => Unit)
runJob[T, U](
  rdd: RDD[T],
  func: (TaskContext, Iterator[T]) => U,
  partitions: Seq[Int]): Array[U]
runJob[T, U](
  rdd: RDD[T],
  func: Iterator[T] => U,
  partitions: Seq[Int]): Array[U]
runJob[T, U](rdd: RDD[T], func: (TaskContext, Iterator[T]) => U): Array[U]
runJob[T, U](rdd: RDD[T], func: Iterator[T] => U): Array[U]
runJob[T, U](
  rdd: RDD[T],
  processPartition: (TaskContext, Iterator[T]) => U,
  resultHandler: (Int, U) => Unit)
runJob[T, U: ClassTag](
  rdd: RDD[T],
  processPartition: Iterator[T] => U,
  resultHandler: (Int, U) => Unit)
```

[RDD actions](#) in Spark run [jobs](#) using one of `runJob` methods. It executes a function on one or many partitions of a RDD to produce a collection of values per partition.

**Tip**

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

```
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?

**Tip**

Read about `TaskContext` in [TaskContext](#).

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).

When executed, `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](#).

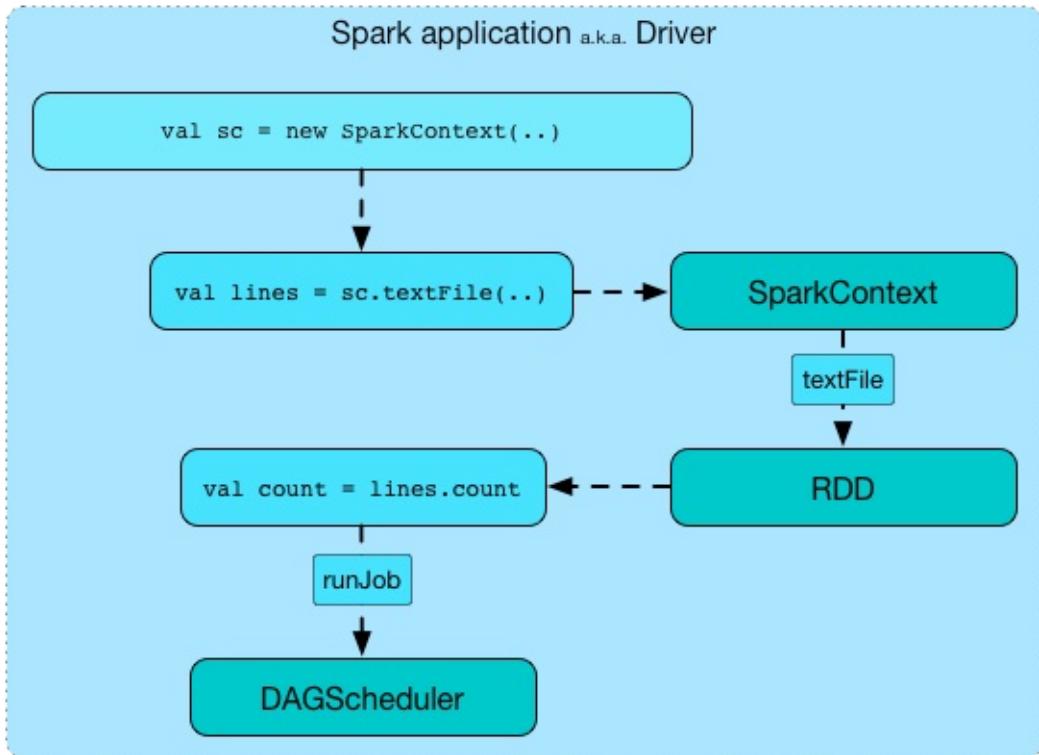


Figure 5. Executing action

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` (`stop` method)

```
stop(): Unit
```

You can stop a `SparkContext` using `stop` method. Stopping a Spark context stops the [Spark Runtime Environment](#) and 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 [LiveListenerBus](#)
- Stops [web UI](#)
- Requests [MetricSystem](#) to report metrics from all registered sinks (using `MetricsSystem.report()`)
- `metadataCleaner.cancel()`
- Stops [ContextCleaner](#)
- Stops [ExecutorAllocationManager](#)
- Stops [DAGScheduler](#)
- Stops [LiveListenerBus](#)
- Stops [EventLoggingListener](#)
- Stops [HeartbeatReceiver](#)
- Stops optional [ConsoleProgressBar](#)
- It clears the reference to TaskScheduler (i.e. `_taskScheduler is null`)
- Stops [SparkEnv](#) and calls `sparkEnv.set(null)`

Caution	<a href="#">FIXME</a> <code>sparkEnv.set(null)</code> what is this doing?
---------	---

- It clears [SPARK\\_YARN\\_MODE flag](#).
- It calls `SparkContext.clearActiveContext()`.

Caution	<a href="#">FIXME</a> What is <code>SparkContext.clearActiveContext()</code> doing?
---------	---

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

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

## Registering SparkListener (addSparkListener method)

```
addSparkListener(listener: SparkListenerInterface): Unit
```

You can register a custom [SparkListenerInterface](#) using `addSparkListener` method

Note

You can also register custom listeners using [spark.extraListeners](#) setting.

## 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.

## Events

When a Spark context starts, it triggers [SparkListenerEnvironmentUpdate](#) and [SparkListenerApplicationStart](#) messages.

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

## Setting Default Log Level (setLogLevel method)

```
setLogLevel(logLevel: String)
```

`setLogLevel` allows you to set the root logging level in a Spark application, e.g. [Spark shell](#).

Internally, `setLogLevel` calls `org.apache.log4j.Level.toLevel(logLevel)` and `org.apache.log4j.Logger.getRootLogger().setLevel(1)`.

## 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.

```
[Stage 0:====>          (316 + 4) / 1000][Stage 1:>          (0 + 0) / 1000][Sta
ge 2:>          (0 + 0) / 1000]]]
```

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
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 <code>DEBUG</code> logging level for <code>org.apache.spark.util.ClosureCleaner</code> 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 [closure Serializer](#)). Refer to [Serialization](#).

Caution	<a href="#">FIXME</a> an example, please.
---------	---

## Hadoop Configuration

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

Note	<a href="#">SparkHadoopUtil.get.newConfiguration</a> 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`.

## listenerBus

`listenerBus` is a [LiveListenerBus](#) object that acts as a mechanism to announce events to other services on the [driver](#).

Note	It is created while <a href="#">SparkContext is being created</a> and, since it is a single-JVM event bus, it is exclusively used on the driver.
------	--

Note	<code>listenerBus</code> is a <code>private[spark]</code> value in <code>SparkContext</code> .
------	--

## Time when SparkContext was Created (`startTime` value)

<code>startTime: Long</code>
------------------------------

`startTime` is the time in milliseconds when [SparkContext was created](#).

```
scala> sc.startTime
res0: Long = 1464425605653
```

## Spark User (`sparkUser` value)

```
sparkUser: String
```

`sparkUser` is the user who started the `SparkContext` instance.

Note	It is computed when <a href="#">SparkContext is created</a> using <code>Utils.getCurrentUserName</code> .
------	---

## Settings

### `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.

You can however control the behaviour using `spark.driver.allowMultipleContexts` flag.

It is disabled, i.e. `false`, by default.

If enabled (i.e. `true`), Spark prints the following WARN message to the logs:

```
WARN Multiple running SparkContexts detected in the same JVM!
```

If disabled (default), it will throw an `sparkException` exception:

```
Only one SparkContext may be running in this JVM (see SPARK-2243). To ignore this error, set spark.driver.allowMultipleContexts = true. The currently running SparkContext was created at:
[ctx.creationSite.longForm]
```

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.
---------	--

## Environment Variables

### **SPARK\_EXECUTOR\_MEMORY**

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

### **SPARK\_USER**

`SPARK_USER` is the user who is running `SparkContext`. It is available later as `sparkUser`.

# HeartbeatReceiver RPC Endpoint

`HeartbeatReceiver` RPC endpoint is a [ThreadSafeRpcEndpoint](#) and a [SparkListener](#).

It keeps track of executors (through [messages](#)) and informs [TaskScheduler](#) and [SparkContext](#) about lost executors.

When created, it requires a [SparkContext](#) and a [clock](#). Later, it uses the [SparkContext](#) to register itself as a [sparkListener](#) and [TaskScheduler](#) (as `scheduler`).

## Note

`HeartbeatReceiver` RPC endpoint is registered while [SparkContext](#) is being created.

## Tip

Enable `DEBUG` or `TRACE` logging levels for `org.apache.spark.HeartbeatReceiver` to see what happens inside.

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

```
log4j.logger.org.apache.spark.HeartbeatReceiver=TRACE
```

Refer to [Logging](#).

## Starting (onStart method)

## Note

`onStart` is part of the [RpcEndpoint Contract](#)

When called, `HeartbeatReceiver` sends a blocking [ExpireDeadHosts](#) every `spark.network.timeoutInterval` on `eventLoopThread` - Heartbeat Receiver Event Loop Thread.

## Stopping (onStop method)

## Note

`onStop` is part of the [RpcEndpoint Contract](#)

When called, `HeartbeatReceiver` cancels the checking task (that sends a blocking [ExpireDeadHosts](#) every `spark.network.timeoutInterval` on `eventLoopThread` - Heartbeat Receiver Event Loop Thread - see [Starting \(onStart method\)](#)) and shuts down `eventLoopThread` and `killExecutorThread` executors.

## killExecutorThread - Kill Executor Thread

`killExecutorThread` is a daemon [ScheduledThreadPoolExecutor](#) with a single thread.

The name of the thread pool is **kill-executor-thread**.

Note	It is used to request SparkContext to kill the executor.
------	--

## eventLoopThread - Heartbeat Receiver Event Loop Thread

`eventLoopThread` is a daemon [ScheduledThreadPoolExecutor](#) with a single thread.

The name of the thread pool is **heartbeat-receiver-event-loop-thread**.

## Messages

### ExecutorRegistered

```
ExecutorRegistered(executorId: String)
```

When `ExecutorRegistered` arrives, `executorId` is simply added to [executorLastSeen](#) internal registry.

Note	HeartbeatReceiver sends a <code>ExecutorRegistered</code> message to itself (from <code>addExecutor</code> internal method). It is as a follow-up to <code>SparkListener.onExecutorAdded</code> when a driver announces a new executor registration.
------	--

Note	It is an internal message.
------	----------------------------

### ExecutorRemoved

```
ExecutorRemoved(executorId: String)
```

When `ExecutorRemoved` arrives, `executorId` is simply removed from [executorLastSeen](#) internal registry.

Note	HeartbeatReceiver itself sends a <code>ExecutorRegistered</code> message (from <code>removeExecutor</code> internal method). It is as a follow-up to <code>SparkListener.onExecutorRemoved</code> when a driver removes an executor.
------	--

Note	It is an internal message.
------	----------------------------

### ExpireDeadHosts

```
ExpireDeadHosts
```

When `ExpireDeadHosts` arrives the following TRACE is printed out to the logs:

```
TRACE HeartbeatReceiver: Checking for hosts with no recent heartbeats in HeartbeatReceiver.
```

Each executor (in `executorLastSeen` registry) is checked whether the time it was last seen is not longer than `spark.network.timeout`.

For any such executor, the following WARN message is printed out to the logs:

```
WARN HeartbeatReceiver: Removing executor [executorId] with no recent heartbeats: [time] ms exceeds timeout [timeout] ms
```

`TaskScheduler.executorLost` is called (with `slaveLost("Executor heartbeat timed out after [timeout] ms" )`).

`SparkContext.killAndReplaceExecutor` is asynchronously called for the executor (i.e. on `killExecutorThread`).

The executor is removed from `executorLastSeen`.

Note	It is an internal message.
------	----------------------------

## Heartbeat

```
Heartbeat(executorId: String,
  accumUpdates: Array[(Long, Seq[AccumulatorV2[_, _]])],
  blockManagerId: BlockManagerId)
```

When `Heartbeat` arrives and the internal `scheduler` is not set yet (no `TaskSchedulerIsSet` earlier), the following WARN is printed out to the logs:

```
WARN HeartbeatReceiver: Dropping [heartbeat] because TaskScheduler is not ready yet
```

And the response is `HeartbeatResponse(reregisterBlockManager = true)`.

Note	Heartbeats messages are the mechanism of executors to inform that they are alive and update about the state of active tasks.
------	--

If however the internal `scheduler` was set already, `HeartbeatReceiver` checks whether the executor `executorId` is known (in `executorLastSeen`).

If the executor is not recognized, the following DEBUG message is printed out to the logs:

```
DEBUG HeartbeatReceiver: Received heartbeat from unknown executor [executorId]
```

And the response is `HeartbeatResponse(reregisterBlockManager = true)`.

If however the internal `scheduler` is set and the executor is recognized (in `executorLastSeen`), the current time is recorded in `executorLastSeen` and `TaskScheduler.executorHeartbeatReceived` is called asynchronously (i.e. on a separate thread) on `eventLoopThread`.

The response is `HeartbeatResponse(reregisterBlockManager = unknownExecutor)` where `unknownExecutor` corresponds to the result of calling `TaskScheduler.executorHeartbeatReceived`.

Caution	<a href="#">FIXME Figure</a>
---------	------------------------------

## TaskSchedulerIsSet

When `TaskSchedulerIsSet` arrives, `HeartbeatReceiver` sets `scheduler` internal attribute (using `SparkContext.taskScheduler`).

Note	TaskSchedulerIsSet is sent by <code>SparkContext</code> (while it is being created) to inform that the <code>TaskScheduler</code> is now available.
------	---

Note	It is an internal message.
------	----------------------------

## Internal Registries

- `executorLastSeen` - a registry of executor ids and the timestamps of when the last heartbeat was received.

## Settings

### spark.storage.blockManagerSlaveTimeoutMs

`spark.storage.blockManagerSlaveTimeoutMs` (default: `120s`)

### spark.network.timeout

`spark.network.timeout` (**default:** `spark.storage.blockManagerSlaveTimeoutMs` )

See [spark.network.timeout](#) in [RPC Environment \(RpcEnv\)](#).

## Other

- `spark.storage.blockManagerTimeoutIntervalMs` (**default:** `60s` )
- `spark.network.timeoutInterval` (**default:** `spark.storage.blockManagerTimeoutIntervalMs` )

# Inside Creating SparkContext

This document describes what happens when you [create a new SparkContext](#).

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

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

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

**Note**

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

Creating `SparkContext` instance starts by setting the internal `allowMultipleContexts` field with the value of `spark.driver.allowMultipleContexts` and marking this `SparkContext` instance as partially constructed. It makes sure that no other thread is creating a `SparkContext` instance in this JVM. It does so by synchronizing on `SPARK_CONTEXT_CONSTRUCTOR_LOCK` and using the internal atomic reference `activeContext` (that eventually has a fully-created `SparkContext` instance).

**Note**

The entire code of `SparkContext` that creates a fully-working `SparkContext` instance is between two statements:

```
SparkContext.markPartiallyConstructed(this, allowMultipleContexts)
// the SparkContext code goes here
SparkContext setActiveContext(this, allowMultipleContexts)
```

`startTime` is set to the current time in milliseconds.

`stopped` internal flag is set to `false`.

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

```
INFO SparkContext: Running Spark version 2.0.0-SNAPSHOT
```

**Tip**

You can use `version` method to learn about the current Spark version or `org.apache.spark.SPARK_VERSION` value.

A [LiveListenerBus](#) instance is created (as `listenerBus` ).

The [current user name](#) is computed.

Caution

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

It saves the input `SparkConf` (as `_conf` ).

Caution

[FIXME](#) Review `_conf.validateSettings()`

It ensures that the first mandatory setting - `spark.master` is defined. `SparkException` is thrown if not.

A master URL must be set in your configuration

It ensures that the other mandatory setting - `spark.app.name` is defined. `SparkException` is thrown if not.

An application name must be set in your configuration

For [Spark on YARN in cluster deploy mode](#), it checks existence of `spark.yarn.app.id`.

`SparkException` is thrown if it does not exist.

Detected yarn cluster mode, but isn't running on a cluster. Deployment to YARN is not supported directly by `SparkContext`. Please use `spark-submit`.

Caution

[FIXME](#) How to "trigger" the exception? What are the steps?

When `spark.logConf` is enabled [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 driver's host and port are set if missing. `spark.driver.host` becomes the value of [Utils.localHostName](#) (or an exception is thrown) while `spark.driver.port` is set to `0`.

Note

`spark.driver.host` and `spark.driver.port` are expected to be set on the driver. It is later asserted by [SparkEnv.createDriverEnv](#).

`spark.executor.id` setting is set to `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 event logging is enabled, i.e. `spark.eventLog.enabled` is `true`, the internal field `_eventLogDir` is set to the value of `spark.eventLog.dir` setting or the default value `/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 Spark on YARN in client deploy mode, `SPARK_YARN_MODE` flag is enabled.

A `JobProgressListener` is created and registered to `LiveListenerBus`.

A `SparkEnv` is created.

`MetadataCleaner` is created.

**Caution**

[FIXME](#) What's `MetadataCleaner`?

Optional `ConsoleProgressBar` with `SparkStatusTracker` are created.

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

**Caution**

[FIXME](#) Where's `_ui` 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`.

**FIXME****Caution**

- What's `_executorMemory` ?
- What's the unit of the value of `_executorMemory` exactly?
- What are "SPARK\_TESTING", "spark.testing"? How do they contribute to `executorEnvs` ?
- What's `executorEnvs` ?

The Mesos scheduler backend's configuration is included in `executorEnvs`, i.e.

[SPARK\\_EXECUTOR\\_MEMORY](#), `_conf.getExecutorEnv`, and `SPARK_USER`.

[HeartbeatReceiver RPC endpoint](#) is registered (as `_heartbeatReceiver`).

[SparkContext.createTaskScheduler](#) is executed (using the master URL) and the result becomes the internal `_schedulerBackend` and `_taskScheduler`.

**Note**

The internal `_schedulerBackend` and `_taskScheduler` are used by `schedulerBackend` and `taskScheduler` methods, respectively.

[DAGScheduler is created](#) (as `_dagScheduler`).

`SparkContext` sends a blocking [TaskSchedulerIsSet](#) message to [HeartbeatReceiver RPC endpoint](#) (to inform that the `TaskScheduler` is now available).

[TaskScheduler is started](#).

The internal fields, `_applicationId` and `_applicationAttemptId`, are set (using `applicationId` and `applicationAttemptId` from the [TaskScheduler Contract](#)).

The setting `spark.app.id` is set to the current application id and Web UI gets notified about it if used (using `setappId(_applicationId)` ).

The [BlockManager \(for the driver\)](#) is initialized (with `_applicationId` ).

**Caution**

**FIXME** Why should UI 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 [LiveListenerBus](#).

**Caution**

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

If `dynamic allocation is enabled`, `ExecutorAllocationManager` is created (as `_executorAllocationManager`) and immediately started.

**Note**

`_executorAllocationManager` is exposed (as a method) to `YARN scheduler backends to reset their state to the initial state`.

`_cleaner` is set to `ContextCleaner` if `spark.cleaner.referenceTracking` is enabled (i.e. `true`). By default it is enabled.

**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`.

It registers user-defined listeners and starts `SparkListenerEvent` event delivery to the listeners.

`postEnvironmentUpdate` is called that posts `SparkListenerEnvironmentUpdate` message on `LiveListenerBus` 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`.

`SparkListenerApplicationStart` message is posted to `LiveListenerBus` (using the internal `postApplicationStart` method).

`TaskScheduler.postStartHook` is called.

**Note**

`TaskScheduler.postStartHook` does nothing by default, but the **only implementation** `TaskSchedulerImpl` comes with its own `postStartHook` and blocks the current thread until a `SchedulerBackend` is ready.

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

1. `DAGSchedulerSource`
2. `BlockManagerSource`
3. `ExecutorAllocationManagerSource` (only if `dynamic allocation is enabled`).

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

**Caution**

**FIXME** What exactly does `ShutdownHookManager.addShutdownHook()` do?

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

**Caution**

**FIXME** What does `NonFatal` represent in Scala?

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

Caution	<a href="#">FIXME</a> Where are <code>nextShuffleId</code> and <code>nextRddId</code> used?
---------	---

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

## Creating SchedulerBackend and TaskScheduler (createTaskScheduler method)

```
createTaskScheduler(
    sc: SparkContext,
    master: String,
    deployMode: String): (SchedulerBackend, TaskScheduler)
```

The private `createTaskScheduler` is executed as part of [creating an instance of SparkContext](#) to create `TaskScheduler` and `SchedulerBackend` objects.

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

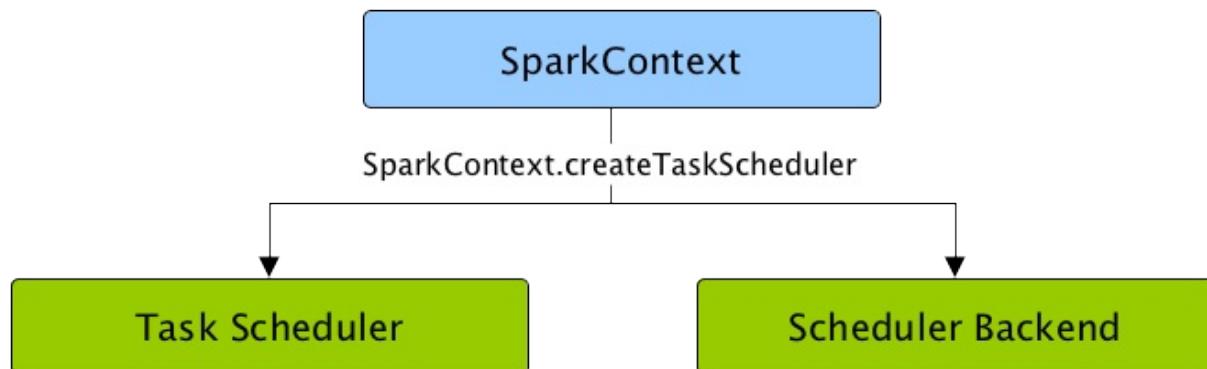


Figure 1. `SparkContext` creates Task Scheduler and Scheduler Backend  
`createTaskScheduler` understands the following master URLs:

- `local` - local mode with 1 thread only
- `local[n]` or `local[*]` - local mode with `n` threads.
- `local[n, m]` or `local[*, m]` — local mode with `n` threads and `m` number of failures.
- `spark://hostname:port` for Spark Standalone.
- `local-cluster[n, m, z]` — local cluster with `n` workers, `m` cores per worker, and `z` memory per worker.
- `mesos://hostname:port` for Spark on Apache Mesos.
- any other URL is passed to [getClusterManager](#) to load an external cluster manager.

Caution

FIXME

## Loading External Cluster Manager for URL (`getClusterManager` method)

```
getClusterManager(url: String): Option[ExternalClusterManager]
```

`getClusterManager` loads [ExternalClusterManager](#) that can handle the input `url`.

If there are two or more external cluster managers that could handle `url`, a `SparkException` is thrown:

```
Multiple Cluster Managers ([serviceLoaders]) registered for the url [url].
```

Note

`getClusterManager` uses Java's [ServiceLoader.load](#) method.

Note

`getClusterManager` is used to find a cluster manager for a master URL when creating a [SchedulerBackend](#) and a [TaskScheduler](#) for the driver.

## setupAndStartListenerBus

```
setupAndStartListenerBus(): Unit
```

`setupAndStartListenerBus` is an internal method that reads `spark.extraListeners` setting from the current [SparkConf](#) to create and register [SparkListenerInterface](#) listeners.

It expects that the class name represents a `SparkListenerInterface` listener with one of the following constructors (in this order):

- a single-argument constructor that accepts [SparkConf](#)
- a zero-argument constructor

`setupAndStartListenerBus` registers every listener class.

You should see the following INFO message in the logs:

```
INFO Registered listener [className]
```

It starts [LiveListenerBus](#) and records it in the internal `_listenerBusStarted`.

When no single- `SparkConf` or zero-argument constructor could be found for a class name in `spark.extraListeners`, a `SparkException` is thrown with the message:

```
[className] did not have a zero-argument constructor or a single-argument constructor that accepts SparkConf. Note: if the class is defined inside of another Scala class, then its constructors may accept an implicit parameter that references the enclosing class; in this case, you must define the listener as a top-level class in order to prevent this extra parameter from breaking Spark's ability to find a valid constructor.
```

Any exception while registering a `SparkListenerInterface` listener stops the `SparkContext` and a `SparkException` is thrown and the source exception's message.

Exception when registering SparkListener

**Tip** Set `INFO` on `org.apache.spark.SparkContext` logger to see the extra listeners being registered.

```
INFO SparkContext: Registered listener pl.japila.spark.CustomSparkListener
```

## Creating SparkEnv for Driver (createSparkEnv method)

```
createSparkEnv(  
    conf: SparkConf,  
    isLocal: Boolean,  
    listenerBus: LiveListenerBus): SparkEnv
```

`createSparkEnv` simply delegates the call to `SparkEnv` to create a `SparkEnv` for the driver.

It calculates the number of cores to `1` for `local` master URL, the number of processors available for JVM for `*` or the exact number in the master URL, or `0` for the cluster master URLs.

## Utils.getCurrentUserNames

```
getCurrentUserName(): String
```

`getCurrentUserName` computes the user name who has started the `SparkContext` instance.

<b>Note</b>	It is later available as <code>SparkContext.sparkUser</code> .
-------------	--

Internally, it reads `SPARK_USER` environment variable and, if not set, reverts to Hadoop Security API's `UserGroupInformation.getCurrentUser().getShortUserName()`.

Note	It is another place where Spark relies on Hadoop API for its operation.
------	---

## Utils.localHostName

`localHostName` computes the local host name.

It starts by checking `SPARK_LOCAL_HOSTNAME` environment variable for the value. If it is not defined, it uses `SPARK_LOCAL_IP` to find the name (using `InetAddress.getByName`). If it is not defined either, it calls `InetAddress.getLocalHost` for the name.

Note	<code>Utils.localHostName</code> is executed while <a href="#">SparkContext is being created</a> .
------	--

Caution	<a href="#">FIXME</a> Review the rest.
---------	--

## stopped flag

Caution	<a href="#">FIXME</a> Where is this used?
---------	---

# 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 Apache Spark and the core of Spark (that many often refer to as **Spark Core**).

A RDD is a resilient and distributed collection of records. One could compare RDD to a Scala collection (that sits on a single JVM) to its distributed variant (that sits on many JVMs, possibly on separate nodes in a cluster).

Tip

RDD 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 with the help of [RDD lineage graph](#) and so able to recompute missing or damaged partitions due to node failures.
- **Distributed** with data residing on multiple nodes in a [cluster](#).
- **Dataset** is a collection of [partitioned data](#) with primitive values or values of values, e.g. tuples or other objects (that represent records of the data you work with).

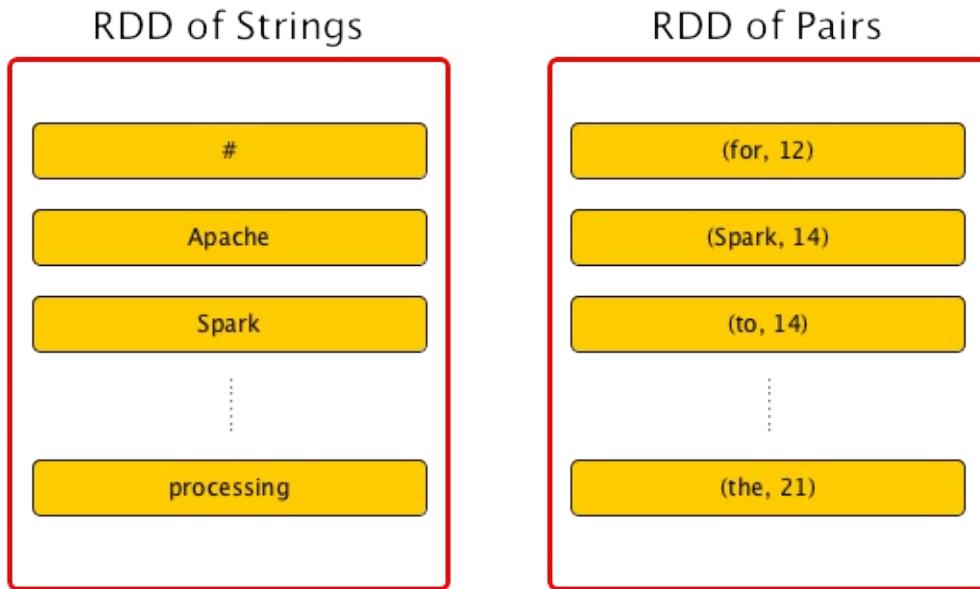


Figure 1. RDDs

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:

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable or Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **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.
- **Typed**, i.e. values in a RDD have types, e.g. `RDD[Long]` Or `RDD[(Int, String)]` .

- **Partitioned**, i.e. the data inside a RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node).

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**.

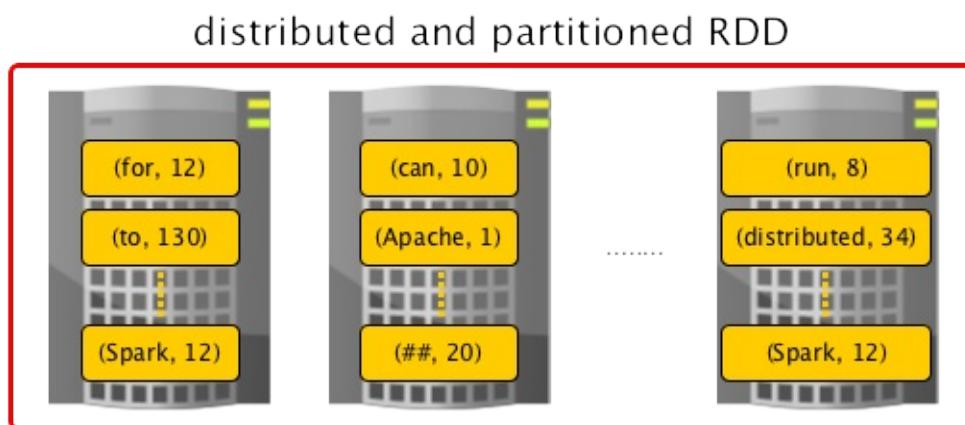


Figure 2. RDDs

**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.

An RDD is defined by five main intrinsic properties:

- List of [parent RDDs](#) that is the list of the dependencies an RDD depends on for records.
- An array of [partitions](#) that a dataset is divided to.
- A [compute function](#) to do a computation on partitions.
- An optional [partitioner](#) that defines how keys are hashed, and the pairs partitioned (for key-value RDDs)
- Optional [preferred locations](#) (aka **locality info**), 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>:24

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 by [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.

Tip	Go in-depth in the section <a href="#">Transformations</a> .
-----	--

## Actions

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

Tip	Go in-depth in the section <a href="#">Actions</a> .
-----	--

## 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>:25
```

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, 310967
683, 1114081267, 1244509086, 1797452433, 124035586)

scala> val rdd = sc.parallelize(data)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:29
```

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	<a href="#">FIXME</a> What's the use case for <code>makeRDD</code> ?
---------	--

```
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 `flatMap` over the lines inside to have an RDD of words):

```
scala> val words = sc.textFile("README.md").flatMap(_.split("\\\\W+")).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 for performance reasons that also makes it visible in Storage tab in the web UI
4. Executes action (and materializes the RDD)

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

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

## Storage

### RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore	Size on Disk
Hundred 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
```

The screenshot shows the Spark shell application UI at localhost:4040/stages/. The 'Stages' tab is selected. It displays two completed stages:

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	count at <console>:27	2015/09/23 13:29:42	34 ms	2/2			2.4 KB	
3	repartition at <console>:27	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 (compute method)

```
compute(split: Partition, context: TaskContext): Iterator[T]
```

The abstract `compute` method computes the input `split` `partition` in the `TaskContext` to produce a collection of values (of type `T`).

It is implemented by any type of RDD in Spark and is called every time the records are requested unless RDD is `cached` or `checkpointed` (and the records can be read from an external storage, but this time closer to the compute node).

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 the `driver`.

## Preferred Locations (aka Locality Info)

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

A **preferred location** (aka *locality preferences* or *placement preferences* or *locality info*) is information about the locations of the `split` block for an HDFS file (to place computing the partition on).

`getPreferredLocations` returns the preferred locations for the input `split` partition (of an RDD).

## Getting Partition Count (getNumPartitions method)

```
getNumPartitions: Int
```

`getNumPartitions` calculates the number of partitions of the RDD.

```
scala> sc.textFile("README.md").getNumPartitions
res0: Int = 2

scala> sc.textFile("README.md", 5).getNumPartitions
res1: Int = 5
```

# Operators - Transformations and Actions

RDDs have two types of operations: [transformations](#) and [actions](#).

Note

Operators are also called **operations**.

## 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>

# Transformations

**Transformations** are lazy operations on a RDD that create one or many new RDDs, e.g.

```
map , filter , reduceByKey , join , cogroup , randomSplit .
```

```
transformation: RDD => RDD
transformation: RDD => Seq[RDD]
```

In other words, transformations are *functions* that take a RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since [RDDs are immutable](#) and hence cannot be modified), but always produce one or more new RDDs by applying the computations they represent.

By applying transformations you incrementally build a [RDD lineage](#) with all the parent RDDs of the final RDD(s).

Transformations are lazy, i.e. are not executed immediately. Only after calling an action are transformations executed.

After executing a transformation, the result RDD(s) will always be different from their parents and can be smaller (e.g. `filter` , `count` , `distinct` , `sample` ), bigger (e.g. `flatMap` , `union` , `cartesian` ) or the same size (e.g. `map` ).

Caution	There are transformations that may trigger jobs, e.g. <code>sortBy</code> , <code>zipWithIndex</code> , etc.
---------	--

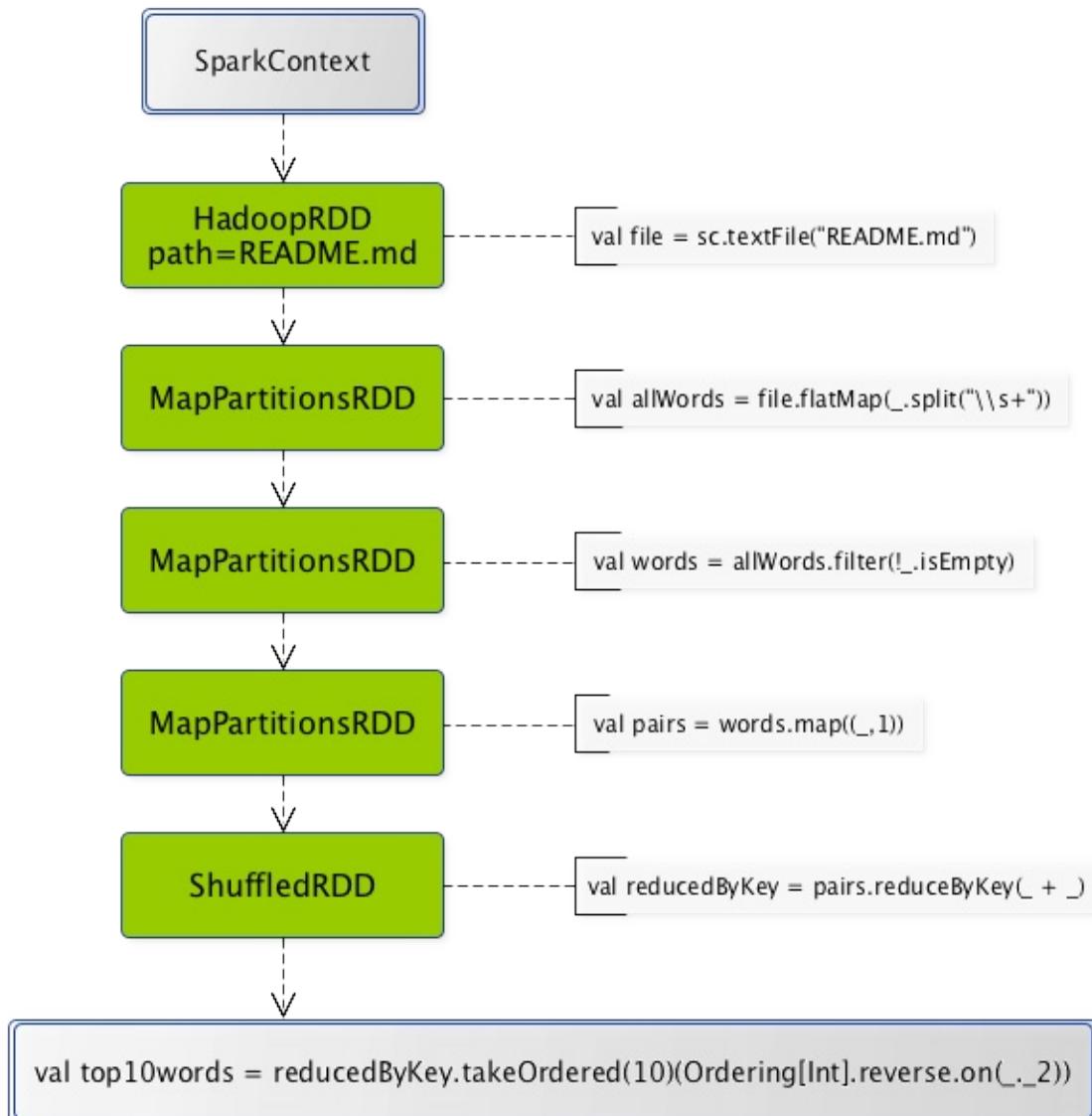


Figure 1. From `SparkContext` by transformations to the result

Certain transformations can be **pipelined** which is an optimization that Spark uses to improve performance of 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("\\\\w+"))
allWords: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[55] at flatMap at <console>:26

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>:30

scala> val reducedByKey = pairs.reduceByKey(_ + _)
reducedByKey: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[59] at reduceByKey at <console>:32

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), (##,8), (a,8), (run,7), (can,6), (is,6))

```

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 which is called **pipelining**.

## 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.

**Note**

Wide transformations are also called shuffle transformations as they may or may not depend on a shuffle.

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.

## mapPartitions

**Caution****FIXME**

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

## zipWithIndex

```
zipWithIndex(): RDD[(T, Long)]
```

`zipWithIndex` zips this `RDD[T]` with its element indices.

If the number of partitions of the source RDD is greater than 1, it will submit an additional job to calculate start indices.

**Caution**

```

val onePartition = sc.parallelize(0 to 9, 1)
scala> onePartition.partitions.length
res0: Int = 1

// no job submitted
onePartition.zipWithIndex

val eightPartitions = sc.parallelize(0 to 9, 8)
scala> eightPartitions.partitions.length
res1: Int = 8

// submits a job
eightPartitions.zipWithIndex

```

The screenshot shows the Spark application UI interface. At the top, there's a navigation bar with the Spark logo (2.0.0-SNAPSHOT), followed by tabs for Jobs, Stages, Storage, Environment, Executors, and a link to the Spark shell application UI. Below the navigation bar, the title "Spark Jobs" is displayed with a question mark icon. Underneath, it shows user information: User: jacek, Total Uptime: 23 s, Scheduling Mode: FIFO, and Completed Jobs: 1. There is a link to Event Timeline. A table titled "Completed Jobs (1)" lists the single completed job. The table has columns: Job Id, Description, Submitted, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total. The data row shows: Job Id 0, Description zipWithIndex at <console>:28, Submitted 2016/04/25 11:02:58, Duration 0.3 s, Stages: Succeeded/Total 1/1, and Tasks (for all stages): Succeeded/Total 7/7.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	zipWithIndex at <console>:28	2016/04/25 11:02:58	0.3 s	1/1	7/7

Figure 2. Spark job submitted by zipWithIndex transformation

# Actions

**Actions** are [RDD operations](#) that produce non-RDD values. They materialize a value in a Spark program. In other words, a RDD operation that returns a value of any type but `RDD[T]` is an action.

```
action: RDD => a value
```

**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 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 are one of two ways to send data from [executors](#) to the [driver](#) (the other being [accumulators](#)).

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 RDDs you work with when you want to execute two or more actions on it for a better performance. Refer to [RDD Caching and 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	<a href="#">FIXME</a>
---------	-----------------------

# RDD Lineage — Logical Execution Plan

A **RDD Lineage Graph** (aka *RDD operator graph*) is a graph of all the parent RDDs of a RDD. It is built as a result of applying transformations to the RDD and creates a [logical execution plan](#).

## Note

The following diagram uses `cartesian` or `zip` for learning purposes only. You may use other operators to build a RDD graph.

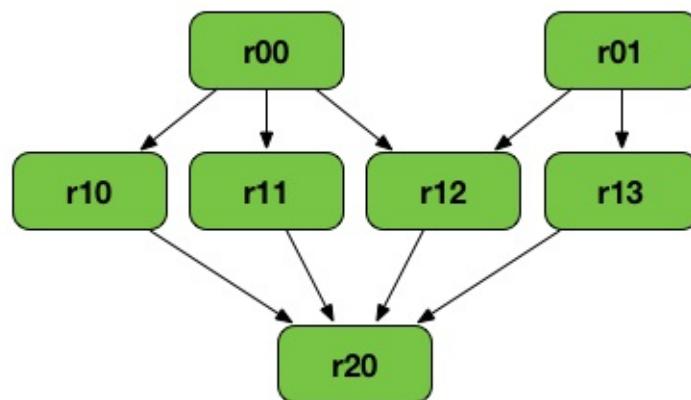


Figure 1. RDD lineage

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

```
toDebugString: String
```

You can learn about a [RDD lineage graph](#) using `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 a
t <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 []

```

`toDebugString` uses indentations to indicate a shuffle boundary.

The numbers in round brackets show the level of parallelism at each stage.

## 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 []

```

## Logical Execution Plan

**Logical 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.

Note	A logical plan — a DAG — is materialized using <a href="#">SparkContext.runJob</a> .
------	--

# Partitions and Partitioning

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

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"><li>• <a href="#">How Many Partitions Does An RDD Have?</a></li><li>• <a href="#">Tuning Spark</a> (the official documentation of Spark)</li></ul>
-----	---

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 Spark shell application UI with the title "Stages for All Jobs". 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
```

The screenshot shows the Spark UI interface with the title "Spark shell - Stages for All Jobs". The "Stages" tab is selected. Below it, a table lists completed stages. Stage 1 has a duration of 6 ms and 2/2 tasks succeeded. Stage 0 has a duration of 0.1 s and 8/8 tasks succeeded. The "Tasks: Succeeded/Total" section for Stage 1 is highlighted with a red box.

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>:24

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 (Ma
pPartitionsRDD[19] at repartition at <console>:27)
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, localho
st, partition 0, NODE_LOCAL, 2089 bytes)
15/10/07 08:10:00 INFO TaskSetManager: Starting task 1.0 in stage 7.0 (TID 18, localho
st, partition 1, NODE_LOCAL, 2089 bytes)
15/10/07 08:10:00 INFO TaskSetManager: Starting task 2.0 in stage 7.0 (TID 19, localho
st, partition 2, NODE_LOCAL, 2089 bytes)
15/10/07 08:10:00 INFO TaskSetManager: Starting task 3.0 in stage 7.0 (TID 20, localho
st, partition 3, NODE_LOCAL, 2089 bytes)
15/10/07 08:10:00 INFO TaskSetManager: Starting task 4.0 in stage 7.0 (TID 21, localho
st, partition 4, NODE_LOCAL, 2089 bytes)
...
```

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
(MapPartitionsRDD[20] at repartition at <console>:27)
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, localho
st, partition 0, PROCESS_LOCAL, 2058 bytes)
15/10/07 08:14:09 INFO TaskSetManager: Starting task 1.0 in stage 8.0 (TID 23, localho
st, partition 1, PROCESS_LOCAL, 2058 bytes)
...
```

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): RDD[T]
```

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 lineage graph](#).

```
scala> val rdd = sc.parallelize(0 to 10, 8)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

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`.

## 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`.

## Settings

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

# RDD shuffling

Tip

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

**Shuffling** is a process of [redistributing data across partitions](#) (aka *repartitioning*) that may or may not cause moving data across JVM processes or even over the wire (between executors on separate machines).

Shuffling is the process of data transfer between stages.

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), (4,5), (5,5))

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), (4,10), (5,10))

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>:26

scala> val joined = kvR join kwR
joined: org.apache.spark.rdd.RDD[(Int, (Int, Int))] = MapPartitionsRDD[10] at join at <console>:32

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.

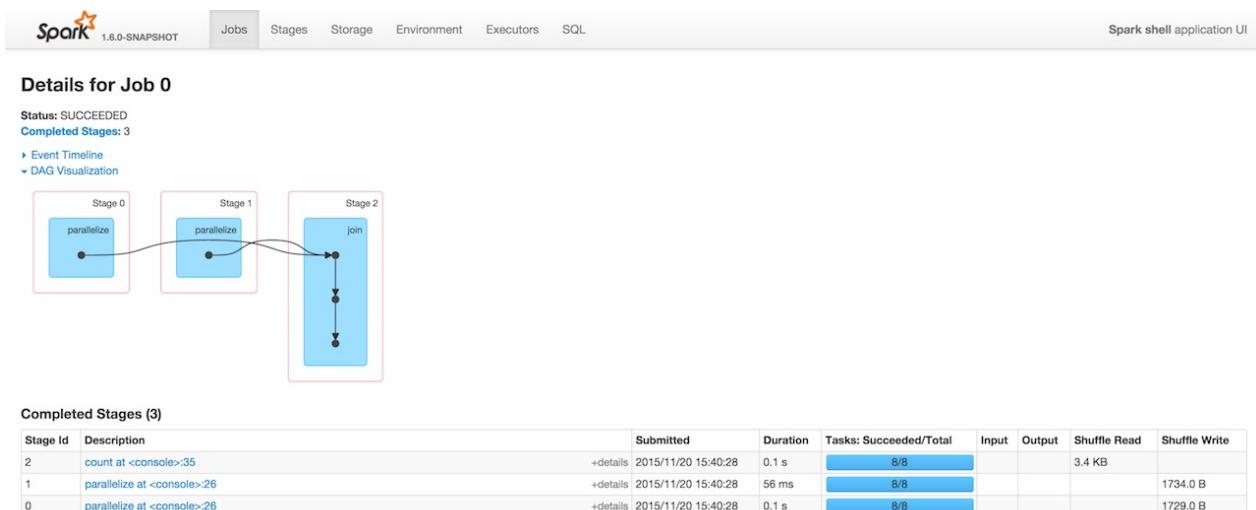


Figure 1. Executing `joined.count`

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 [SchedulerBackend](#).

`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/jacek/dev/oss/spark/checkpoints/91514c29-d44b-4d95-ba02-480027b7c174/rdd-5, new parent is RDD 6
```

## 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>:18

scala> val r2 = sc.parallelize(0 to 9)
r2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[21] at parallelize at <console>:18

scala> val r3 = sc.parallelize(0 to 9)
r3: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[22] at parallelize at <console>:18

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@6f2ab3f6, org.apache.spark.RangeDependency@7aa0e351, org.apache.spark.RangeDependency@26468)

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, 2, 3, 4, 5, 6, 7, 8, 9)
```

## 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).groupBy(identity)
r: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[12] at groupBy at <console>:18

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 <console>:18

scala> r.dependencies
res0: Seq[org.apache.spark.Dependency[_]] = List(org.apache.spark.ShuffleDependency@493b0b09)
```

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>:18

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@7353a0fb)

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>:18

scala> val r2 = sc.parallelize(10 to 19)
r2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[14] at parallelize at <console>:18

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@28408ad7, org.apache.spark.RangeDependency@6e1d2e9f)

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:

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

# PairRDDFunctions

Tip

Read up the scaladoc of [PairRDDFunctions](#).

`PairRDDFunctions` are available in RDDs of key-value pairs via Scala's implicit conversion.

Tip

**Partitioning** is an advanced feature that is directly linked to (or inferred by) use of `PairRDDFunctions`. Read up about it in [Partitions and Partitioning](#).

## 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*.

## mapValues, flatMapValues

Caution

[FIXME](#)

## 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.

## 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:0+1784
15/10/10 18:03:21 INFO HadoopRDD: Input split: file:/Users/jacek/dev/oss/spark/README.md:1784+1784
...
```

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

## 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 <a href="#">the sources of org.apache.hadoop.mapred.FileInputFormat.getSplits</a> 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>:18

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`.

# BlockRDD

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

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

# Spark Tools

# 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 `spark`.

Note	<p>When you execute <code>spark-shell</code> it executes <a href="#">Spark submit</a> 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 <a href="#">Command of Spark Scripts</a>.</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).

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

```
scala> spark
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
```

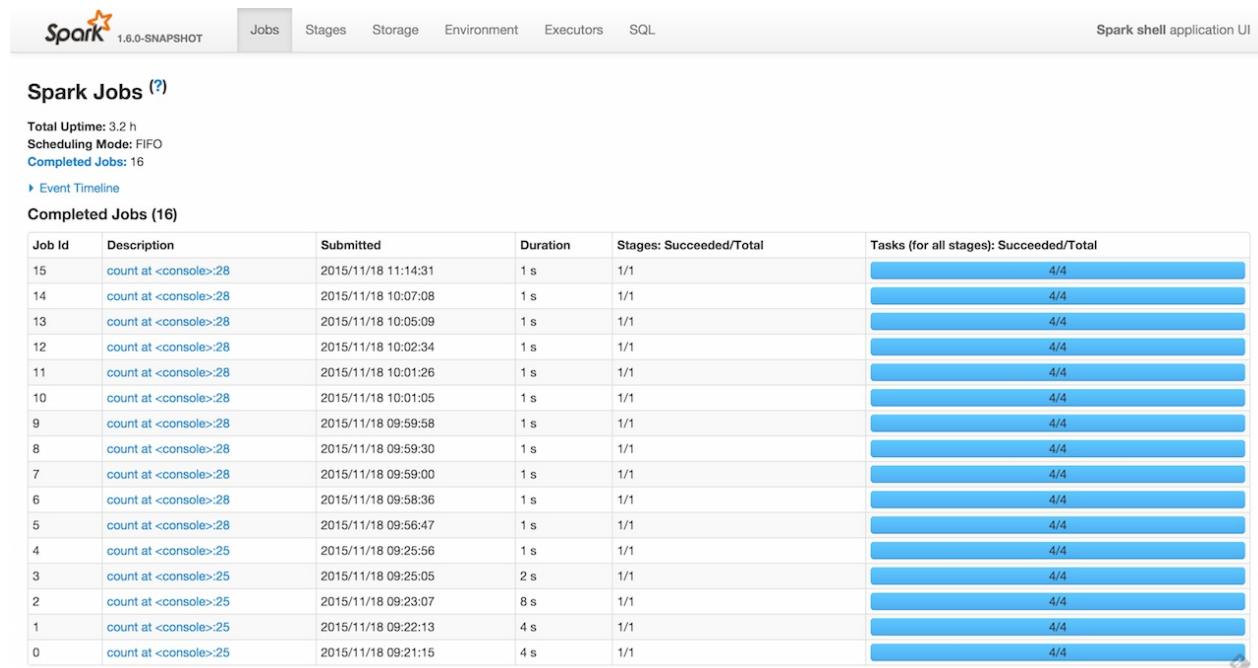
Spark properties used, including those specified through  
`--conf` and those from the properties file `null`:

```
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 — Spark Application's web UI

**Web UI** (aka **webUI** or **Spark UI** after **SparkUI**) is the web interface of a Spark application to inspect job executions in the `SparkContext` using a browser.



The screenshot shows the 'Jobs' tab of the Spark Web UI. At the top, it displays system statistics: Total Uptime: 3.2 h, Scheduling Mode: FIFO, and Completed Jobs: 16. Below this is a link to the Event Timeline. The main section is titled 'Completed Jobs (16)' and contains a table with 16 rows, each representing a completed job. The columns in the table are Job Id, Description, Submitted, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total. All 16 jobs have a duration of 1 second and 4/4 tasks succeeded.

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` setting).

web UI comes with the following tabs:

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

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

Tip

You can use the web UI after the application is finished by persisting events using `EventLoggingListener`.

Note	All the information that are displayed in web UI is available thanks to <a href="#">JobProgressListener</a> . One could say that web UI is a web layer to <a href="#">JobProgressListener</a> .
------	---

## Environment Tab

The screenshot shows the 'Environment' tab of the Spark Web UI. At the top, there are tabs for Jobs, Stages, Storage, Environment (which is selected), Executors, and SQL. On the right, it says 'Spark shell application UI'. Below the tabs, there are two sections: 'Runtime Information' and 'Spark Properties'. The 'Runtime Information' section contains a table with three rows: Java Home, Java Version, and Scala Version. The 'Spark Properties' section contains a table with many rows, each with a 'Name' and 'Value' column.

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.uri	http://192.168.1.4:62705
spark.home	/Users/jacek/dev/oss/spark
spark.jars	
spark.master	local[*]
spark.repl.class.url	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

## SparkUI

SparkUI is...[FIXME](#)

### createLiveUI

Caution	<a href="#">FIXME</a>
---------	-----------------------

### appUIAddress

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Settings

### spark.ui.enabled

`spark.ui.enabled` (default: `true`) setting controls whether the web UI is started at all.

## **spark.ui.port**

`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**

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

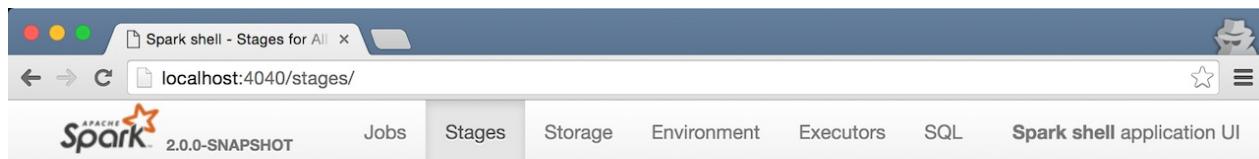
# Stages Tab — Stages for All Jobs

**Stages** tab in web UI shows the current state of all stages of all jobs in a Spark application (i.e. a [SparkContext](#)) with two optional pages for [the tasks and statistics for a stage](#) (when a stage is selected) and [pool details](#) (when the application works in FAIR scheduling mode).

The title of the tab is **Stages for All Jobs**.

You can access the Stages tab under `/stages` URL, i.e. <http://localhost:4040/stages>.

With no jobs submitted yet (and hence no stages to display), the page shows nothing but the title.



## Stages for All Jobs

Figure 1. Stages Page Empty

The Stages page shows the stages in a Spark application per state in their respective sections — **Active Stages**, **Pending Stages**, **Completed Stages**, and **Failed Stages**.

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
0	count at <console>:25 +details	2016/06/29 07:29:35	94 ms	3/3				

Figure 2. Stages Page With One Stage Completed

Note	The state sections are only displayed when there are stages in a given state. Refer to <a href="#">Stages for All Jobs</a> .
------	--

In [FAIR scheduling mode](#) you have access to the table showing the scheduler pools.



Figure 3. Fair Scheduler Pools Table

Internally, the page is represented by [org.apache.spark.ui.jobs.StagesTab](#) class.

The page uses the parent's [SparkUI](#) to access required services, i.e. [SparkContext](#), [SparkConf](#), [JobProgressListener](#), [RDDOperationGraphListener](#), and [whether kill is enabled or not](#).

**Caution**

[FIXME](#) What is [RDDOperationGraphListener](#) ?

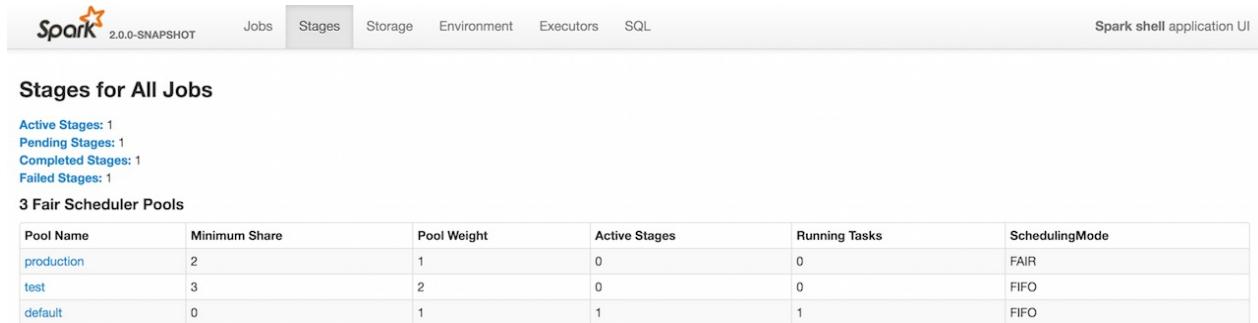
## killEnabled flag

**Caution**

[FIXME](#)

# Stages for All Jobs Page

`AllStagesPage` is a web page (section) that is registered with the [Stages tab](#) that [displays all stages in a Spark application](#) - active, pending, completed, and failed stages with their count.



The screenshot shows the Spark Web UI with the 'Stages' tab selected. At the top, it displays summary statistics: Active Stages: 1, Pending Stages: 1, Completed Stages: 1, and Failed Stages: 1. Below this, a table titled '3 Fair Scheduler Pools' lists the configuration for each pool:

Pool Name	Minimum Share	Pool Weight	Active Stages	Running Tasks	SchedulingMode
production	2	1	0	0	FAIR
test	3	2	0	0	FIFO
default	0	1	1	1	FIFO

Figure 1. Stages Tab in web UI for FAIR scheduling mode (with pools only)

In [FAIR scheduling mode](#) you have access to the table showing the scheduler pools as well as the pool names per stage.

Note	Pool names are calculated using <a href="#">SparkContext.getAllPools</a> .
------	--

Internally, `AllStagesPage` is a `WebUIPage` with access to the parent [Stages tab](#) and more importantly the [JobProgressListener](#) to have access to current state of the entire Spark application.

## Rendering AllStagesPage (render method)

```
render(request: HttpServletRequest): Seq[Node]
```

`render` generates a HTML page to display in a web browser.

It uses the parent's [JobProgressListener](#) to know about:

- active stages (as `activeStages`)
- pending stages (as `pendingStages`)
- completed stages (as `completedStages`)
- failed stages (as `failedStages`)
- the number of completed stages (as `numCompletedStages`)
- the number of failed stages (as `numFailedStages`)

Note	Stage information is available as <code>StageInfo</code> object.
------	--

## Stages Tab

Caution		FIXME StageInfo ???																																	
There are 4 different tables for the different states of stages - active, pending, completed, and failed. They are displayed only when there are stages in a given state.																																			
<b>3 Fair Scheduler Pools</b>																																			
<table border="1"><thead><tr><th>Pool Name</th><th>Minimum Share</th><th>Pool Weight</th><th>Active Stages</th><th>Running Tasks</th><th>SchedulingMode</th></tr></thead><tbody><tr><td>production</td><td>2</td><td>1</td><td>0</td><td>0</td><td>FAIR</td></tr><tr><td>test</td><td>3</td><td>2</td><td>0</td><td>0</td><td>FIFO</td></tr><tr><td>default</td><td>0</td><td>1</td><td>1</td><td>1</td><td>FIFO</td></tr></tbody></table>												Pool Name	Minimum Share	Pool Weight	Active Stages	Running Tasks	SchedulingMode	production	2	1	0	0	FAIR	test	3	2	0	0	FIFO	default	0	1	1	1	FIFO
Pool Name	Minimum Share	Pool Weight	Active Stages	Running Tasks	SchedulingMode																														
production	2	1	0	0	FAIR																														
test	3	2	0	0	FIFO																														
default	0	1	1	1	FIFO																														
<b>Active Stages (1)</b>																																			
<table border="1"><thead><tr><th>Stage Id</th><th>Pool Name</th><th>Description</th><th>Submitted</th><th>Duration</th><th>Tasks: Succeeded/Total</th><th>Input</th><th>Output</th><th>Shuffle Read</th><th>Shuffle Write</th></tr></thead><tbody><tr><td>2</td><td>default</td><td>map at &lt;console&gt;:29 +details (kill)</td><td>2016/06/02 20:56:36</td><td>2 s</td><td>7/8</td><td>168.0 B</td><td></td><td></td><td>414.0 B</td></tr></tbody></table>												Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	2	default	map at <console>:29 +details (kill)	2016/06/02 20:56:36	2 s	7/8	168.0 B			414.0 B				
Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write																										
2	default	map at <console>:29 +details (kill)	2016/06/02 20:56:36	2 s	7/8	168.0 B			414.0 B																										
<b>Pending Stages (1)</b>																																			
<table border="1"><thead><tr><th>Stage Id</th><th>Pool Name</th><th>Description</th><th>Submitted</th><th>Duration</th><th>Tasks: Succeeded/Total</th><th>Input</th><th>Output</th><th>Shuffle Read</th><th>Shuffle Write</th></tr></thead><tbody><tr><td>3</td><td></td><td>count at &lt;console&gt;:29 +details Unknown</td><td></td><td>Unknown</td><td>0/8</td><td></td><td></td><td></td><td></td></tr></tbody></table>												Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	3		count at <console>:29 +details Unknown		Unknown	0/8								
Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write																										
3		count at <console>:29 +details Unknown		Unknown	0/8																														
<b>Completed Stages (1)</b>																																			
<table border="1"><thead><tr><th>Stage Id</th><th>Pool Name</th><th>Description</th><th>Submitted</th><th>Duration</th><th>Tasks: Succeeded/Total</th><th>Input</th><th>Output</th><th>Shuffle Read</th><th>Shuffle Write</th></tr></thead><tbody><tr><td>1</td><td>default</td><td>count at &lt;console&gt;:29 +details</td><td>2016/06/02 20:56:05</td><td>0.1 s</td><td>8/8</td><td>192.0 B</td><td></td><td></td><td></td></tr></tbody></table>												Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	1	default	count at <console>:29 +details	2016/06/02 20:56:05	0.1 s	8/8	192.0 B							
Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write																										
1	default	count at <console>:29 +details	2016/06/02 20:56:05	0.1 s	8/8	192.0 B																													
<b>Failed Stages (1)</b>																																			
<table border="1"><thead><tr><th>Stage Id</th><th>Pool Name</th><th>Description</th><th>Submitted</th><th>Duration</th><th>Tasks: Succeeded/Total</th><th>Input</th><th>Output</th><th>Shuffle Read</th><th>Shuffle Write</th><th>Failure Reason</th></tr></thead><tbody><tr><td>0</td><td>default</td><td>count at &lt;console&gt;:29 +details 20:55:45</td><td>2016/06/02 20:55:45</td><td>0.2 s</td><td>7/8 (1 failed)</td><td></td><td></td><td></td><td></td><td>Job aborted due to stage failure: Task 1 in stage 0.0 failed 1 times, most recent failure: Lost task 1.0 in stage 0.0 (TID 1, localhost): java.lang.Exception: failed +details</td></tr></tbody></table>												Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	Failure Reason	0	default	count at <console>:29 +details 20:55:45	2016/06/02 20:55:45	0.2 s	7/8 (1 failed)					Job aborted due to stage failure: Task 1 in stage 0.0 failed 1 times, most recent failure: Lost task 1.0 in stage 0.0 (TID 1, localhost): java.lang.Exception: failed +details		
Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	Failure Reason																									
0	default	count at <console>:29 +details 20:55:45	2016/06/02 20:55:45	0.2 s	7/8 (1 failed)					Job aborted due to stage failure: Task 1 in stage 0.0 failed 1 times, most recent failure: Lost task 1.0 in stage 0.0 (TID 1, localhost): java.lang.Exception: failed +details																									

Figure 2. Stages Tab in web UI for FAIR scheduling mode (with pools and stages)

You could also notice "retry" for stage when it was retried.

Caution		FIXME A screenshot									

# Stage Details

StagePage shows the task details for a stage given its id and attempt id.

**Details for Stage 2 (Attempt 0)**

Total Time Across All Tasks: 48 ms  
Locality Level Summary: Process local: 4  
Shuffle Write: 506.0 B / 11

- ▶ DAG Visualization
- ▶ Show Additional Metrics
- ▶ Event Timeline

**Summary Metrics for 4 Completed Tasks**

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	9 ms	13 ms	13 ms	13 ms	13 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Write Size / Records	92.0 B / 2	138.0 B / 3	138.0 B / 3	138.0 B / 3	138.0 B / 3

**Aggregated Metrics by Executor**

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Shuffle Write Size / Records
driver	192.168.1.9:65297	70 ms	4	0	0	4	506.0 B / 11

**Tasks**

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Write Time	Shuffle Write Size / Records	Errors
0	16	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	9 ms		1 ms	92.0 B / 2	
1	17	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	13 ms		1 ms	138.0 B / 3	
2	18	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	13 ms		1 ms	138.0 B / 3	
3	19	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	13 ms		1 ms	138.0 B / 3	

Figure 1. Details for Stage 2 and Attempt 0

StagePage renders a page available under `/stage` URL that requires two request parameters — `id` and `attempt`, e.g. <http://localhost:4040/stages/stage/?id=2&attempt=0>.

It is a part of [StagesTab](#).

It uses the parent's `JobProgressListener` and `RDDOperationGraphListener` to calculate the metrics. More specifically, `StagePage` uses `JobProgressListener.stageIdToData` registry to access the stage for given stage `id` and `attempt`.

## Summary Metrics for Completed Tasks in Stage

The summary metrics table shows the metrics for the tasks in a given stage that have already finished with SUCCESS status and metrics available.

The table consists of the following columns: **Metric**, **Min**, **25th percentile**, **Median**, **75th percentile**, **Max**.

- ▶ DAG Visualization
- ▶ Show Additional Metrics
  - (De)select All
  - Scheduler Delay
  - Task Deserialization Time
  - Result Serialization Time
  - Getting Result Time
  - Peak Execution Memory
- ▶ Event Timeline

#### Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	12 ms	12 ms	14 ms	14 ms	14 ms
Scheduler Delay	64 ms	64 ms	72 ms	72 ms	72 ms
Task Deserialization Time	0.5 s	0.5 s	0.6 s	0.6 s	0.6 s
GC Time	25 ms	25 ms	29 ms	29 ms	29 ms
Result Serialization Time	0 ms	0 ms	1 ms	1 ms	1 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Peak Execution Memory	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

Figure 2. Summary Metrics for Completed Tasks in Stage

Note

All the quantiles are doubles using `TaskUIData.metrics` (sorted in ascending order).

The 1st row is **Duration** which includes the quantiles based on `executorRunTime`.

The 2nd row is the optional **Scheduler Delay** which includes the time to ship the task from the scheduler to executors, and the time to send the task result from the executors to the scheduler. It is not enabled by default and you should select **Scheduler Delay** checkbox under **Show Additional Metrics** to include it in the summary table.

Tip

If Scheduler Delay is large, consider decreasing the size of tasks or decreasing the size of task results.

The 3rd row is the optional **Task Deserialization Time** which includes the quantiles based on `executorDeserializeTime` task metric. It is not enabled by default and you should select **Task Deserialization Time** checkbox under **Show Additional Metrics** to include it in the summary table.

The 4th row is **GC Time** which is the time that an executor spent paused for Java garbage collection while the task was running (using `jvmGCTime` task metric).

The 5th row is the optional **Result Serialization Time** which is the time spent serializing the task result on a executor before sending it back to the driver (using `resultSerializationTime` task metric). It is not enabled by default and you should select **Result Serialization Time** checkbox under **Show Additional Metrics** to include it in the summary table.

The 6th row is the optional **Getting Result Time** which is the time that the driver spends fetching task results from workers. It is not enabled by default and you should select **Getting Result Time** checkbox under **Show Additional Metrics** to include it in the summary table.

Tip

If Getting Result Time is large, consider decreasing the amount of data returned from each task.

If [Tungsten is enabled](#) (it is by default), the 7th row is the optional **Peak Execution Memory** which is the sum of the peak sizes of the internal data structures created during shuffles, aggregations and joins (using `peakExecutionMemory` task metric). For SQL jobs, this only tracks all unsafe operators, broadcast joins, and external sort. It is not enabled by default and you should select **Peak Execution Memory** checkbox under **Show Additional Metrics** to include it in the summary table.

If the stage has an input, the 8th row is **Input Size / Records** which is the bytes and records read from Hadoop or from a Spark storage (using `inputMetrics.bytesRead` and `inputMetrics.recordsRead` task metrics).

If the stage has an output, the 9th row is **Output Size / Records** which is the bytes and records written to Hadoop or to a Spark storage (using `outputMetrics.bytesWritten` and `outputMetrics.recordsWritten` task metrics).

If the stage has shuffle read there will be three more rows in the table. The first row is **Shuffle Read Blocked Time** which is the time that tasks spent blocked waiting for shuffle data to be read from remote machines (using `shuffleReadMetrics.fetchWaitTime` task metric). The other row is **Shuffle Read Size / Records** which is the total shuffle bytes and records read (including both data read locally and data read from remote executors using `shuffleReadMetrics.totalBytesRead` and `shuffleReadMetrics.recordsRead` task metrics). And the last row is **Shuffle Remote Reads** which is the total shuffle bytes read from remote executors (which is a subset of the shuffle read bytes; the remaining shuffle data is read locally). It uses `shuffleReadMetrics.remoteBytesRead` task metric.

If the stage has shuffle write, the following row is **Shuffle Write Size / Records** (using `shuffleWriteMetrics.bytesWritten` and `shuffleWriteMetrics.recordsWritten` task metrics).

If the stage has bytes spilled, the following two rows are **Shuffle spill (memory)** (using `memoryBytesSpilled` task metric) and **Shuffle spill (disk)** (using `diskBytesSpilled` task metric).

## Request Parameters

`id` is...

`attempt` is...

Note

`id` and `attempt` uniquely identify the stage in [JobProgressListener.stageIdToData](#) to retrieve `StageUIData`.

`task.page` (default: `1`) is...

`task.sort` (default: `Index`)

```
task.desc (default: false )  
task.pageSize (default: 100 )  
task.prevPageSize (default: task.pageSize )
```

## Metrics

Scheduler Delay is...[FIXME](#)

Task Deserialization Time is...[FIXME](#)

Result Serialization Time is...[FIXME](#)

Getting Result Time is...[FIXME](#)

Peak Execution Memory is...[FIXME](#)

Shuffle Read Time is...[FIXME](#)

Executor Computing Time is...[FIXME](#)

Shuffle Write Time is...[FIXME](#)

# Details for Stage 2 (Attempt 0)

**Total Time Across All Tasks:** 48 ms

**Locality Level Summary:** Process local: 4

**Shuffle Write:** 506.0 B / 11

## ▼ DAG Visualization

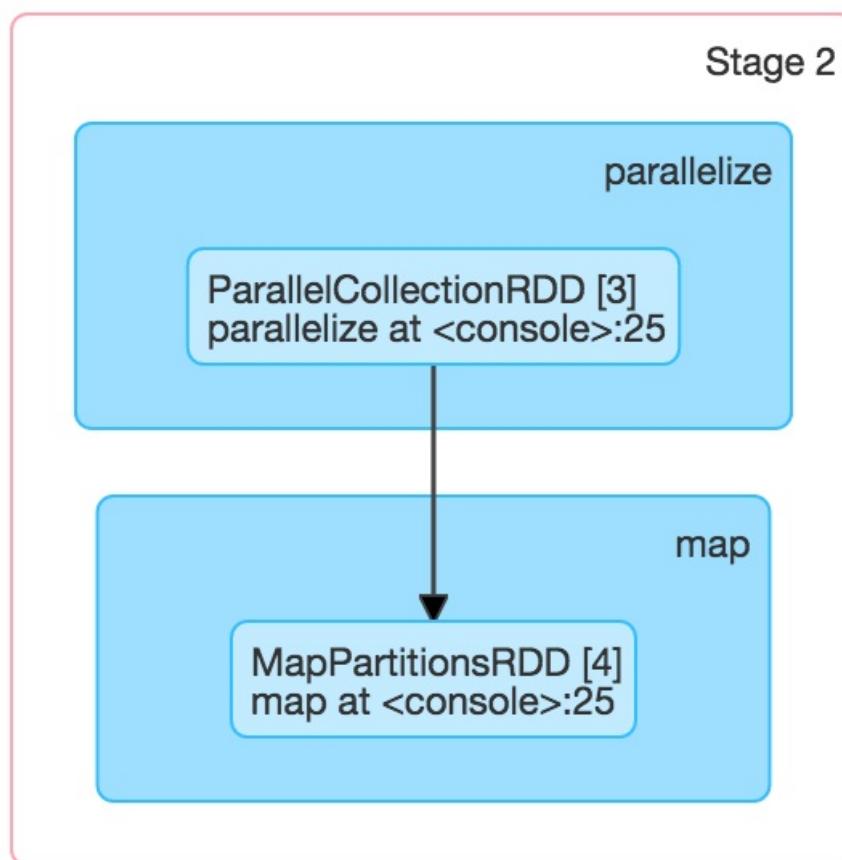


Figure 3. DAG Visualization

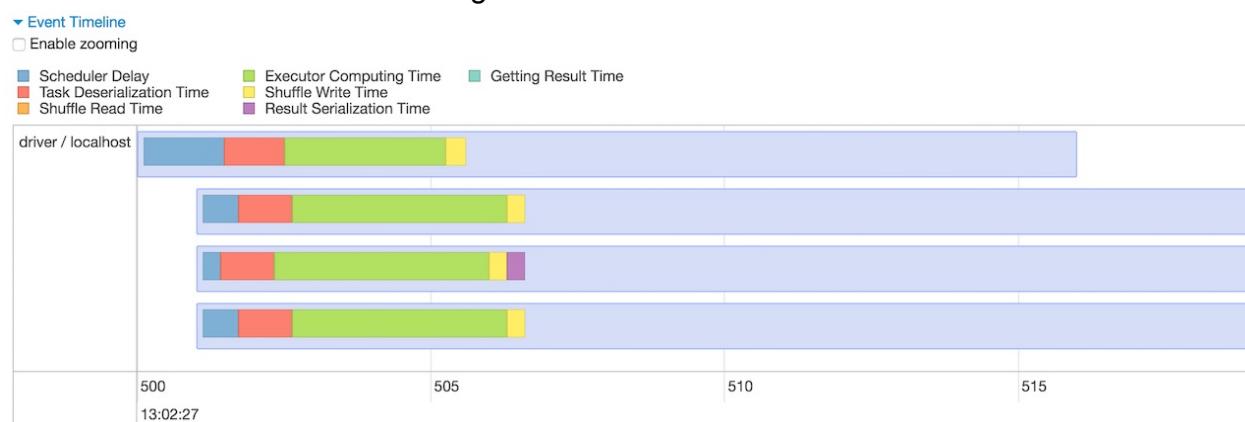


Figure 4. Event Timeline

## Details for Stage 2 (Attempt 0)

**Total Time Across All Tasks:** 48 ms

**Locality Level Summary:** Process local: 4

**Shuffle Write:** 506.0 B / 11

Figure 5. Stage Task and Shuffle Stats

## Aggregated Metrics by Executor

`ExecutorTable` table shows the following columns:

- Executor ID
- Address
- Task Time
- Total Tasks
- Failed Tasks
- Killed Tasks
- Succeeded Tasks
- (optional) Input Size / Records (only when the stage has an input)
- (optional) Output Size / Records (only when the stage has an output)
- (optional) Shuffle Read Size / Records (only when the stage read bytes for a shuffle)
- (optional) Shuffle Write Size / Records (only when the stage wrote bytes for a shuffle)
- (optional) Shuffle Spill (Memory) (only when the stage spilled memory bytes)
- (optional) Shuffle Spill (Disk) (only when the stage spilled bytes to disk)

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Shuffle Write Size / Records
driver	192.168.1.9:65297	70 ms	4	0	0	4	506.0 B / 11

Figure 6. Aggregated Metrics by Executor

It gets `executorSummary` from `StageUIData` (for the stage and stage attempt id) and creates rows per executor.

It also [requests BlockManagers \(from JobProgressListener\)](#) to map executor ids to a pair of host and port to display in Address column.

## Accumulators

Stage page displays the table with [named accumulators](#) (only if they exist). It contains the name and value of the accumulators.

### Accumulators

Accumulable	Value
counter	110

Figure 7. Accumulators Section

#### Note

The information with name and value is stored in [AccumulableInfo](#) (that is available in [StageUIData](#)).

## Tasks

### Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Write Time	Shuffle Write Size / Records	Errors
0	16	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	9 ms		1 ms	92.0 B / 2	
1	17	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	13 ms		1 ms	138.0 B / 3	
2	18	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	13 ms		1 ms	138.0 B / 3	
3	19	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/06/04 13:02:27	13 ms		1 ms	138.0 B / 3	

Figure 8. Tasks Section

## Settings

### spark.ui.timeline.tasks.maximum

`spark.ui.timeline.tasks.maximum` (default: 1000) ...[FIXME](#)

### spark.sql.unsafe.enabled

`spark.sql.unsafe.enabled` (default: true) ...[FIXME](#)

# Fair Scheduler Pool Details Page

The Fair Scheduler Pool Details page shows information about a `Schedulable pool` and is only available when a Spark application uses the `FAIR scheduling mode` (which is controlled by `spark.scheduler.mode` setting).

The screenshot shows the Apache Spark 2.0.0-SNAPSHOT interface with the Stages tab selected. The title bar says "Fair Scheduler Pool: production". Below it is a "Summary" table:

Pool Name	Minimum Share	Pool Weight	Active Stages	Running Tasks	SchedulingMode
production	2	1	1	2	FAIR

Below the summary is a section titled "1 Active Stages" with a table:

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	production	count at <console>:26 +details (kill)	2016/06/17 13:07:04	10 s	0/2 (2 failed)				

Figure 1. Details Page for production Pool

`PoolPage` renders a page under `/pool` URL and requires one request parameter `poolname` that is the name of the pool to display, e.g. <http://localhost:4040/stages/pool/?poolname=production>. It is made up of two tables: `Summary` (with the details of the pool) and `Active Stages` (with the active stages in the pool).

It is a part of [StagesTab](#).

`PoolPage` uses the parent's `SparkContext` to access information about the pool and `JobProgressListener` for active stages in the pool (sorted by `submissionTime` in descending order by default).

## Summary Table

The **Summary** table shows the details of a `schedulable pool`.

Summary					
Pool Name	Minimum Share	Pool Weight	Active Stages	Running Tasks	SchedulingMode
production	2	1	1	2	FAIR

Figure 2. Summary for production Pool

It uses the following columns:

- **Pool Name**
- **Minimum Share**
- **Pool Weight**

- **Active Stages** - the number of the active stages in a `Schedulable` pool.
- **Running Tasks**
- **SchedulingMode**

All the columns are the attributes of a `Schedulable` but the number of active stages which is calculated using the [list of active stages of a pool](#) (from the parent's `JobProgressListener`).

## Active Stages Table

The **Active Stages** table shows the active stages in a pool.

1 Active Stages

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	production	count at <console>:26 +details (kill)	2016/06/17 13:07:04	10 s	0/2 (2 failed)				

Figure 3. Active Stages for production Pool

It uses the following columns:

- **Stage Id**
- (optional) **Pool Name** - only available when in FAIR scheduling mode.
- **Description**
- **Submitted**
- **Duration**
- **Tasks: Succeeded/Total**
- **Input** — Bytes and records read from Hadoop or from Spark storage.
- **Output** — Bytes and records written to Hadoop.
- **Shuffle Read** — Total shuffle bytes and records read (includes both data read locally and data read from remote executors).
- **Shuffle Write** — Bytes and records written to disk in order to be read by a shuffle in a future stage.

The table uses `JobProgressListener` for information per stage in the pool.

## Request Parameters

### poolname

`poolname` is the name of the scheduler pool to display on the page. It is a mandatory request parameter.

## Storage Tab

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

## Executors Tab

Caution	<a href="#">FIXME</a>
---------	-----------------------

# SQL Tab

**SQL tab** in [web UI](#) displays accumulator values per operator.

Caution	<a href="#">FIXME</a> Intro
---------	-----------------------------

You can access the SQL tab under `/SQL` URL, e.g. <http://localhost:4040/SQL/>.

By default, it displays [all SQL query executions](#). However, after a query has been selected, the SQL tab [displays the details of the SQL query execution](#).

## AllExecutionsPage

`AllExecutionsPage` displays all SQL query executions in a Spark application per state sorted by their submission time reversed.

**Running Queries**

ID	Description	Submitted	Duration	Running Jobs	Succeeded Jobs	Failed Jobs
2	<a href="#">foreach at &lt;console&gt;:24</a> +details	2016/06/29 22:30:45	2 s	1		

**Completed Queries**

ID	Description	Submitted	Duration	Jobs
0	<a href="#">show at &lt;console&gt;:24</a> +details	2016/06/29 22:29:46	19 ms	

**Failed Queries**

ID	Description	Submitted	Duration	Succeeded Jobs	Failed Jobs
1	<a href="#">foreach at &lt;console&gt;:24</a> +details	2016/06/29 22:30:02	0.9 s		0

Figure 1. SQL Tab in web UI (AllExecutionsPage)

Internally, the page requests [SQLListener](#) for query executions in running, completed, and failed states (the states correspond to the respective tables on the page).

## ExecutionPage

`ExecutionPage` displays SQL query execution details for a given query execution `id`.

Note	The <code>id</code> request parameter is mandatory.
------	---

`ExecutionPage` displays a summary with **Submitted Time**, **Duration**, the clickable identifiers of the **Running Jobs**, **Succeeded Jobs**, and **Failed Jobs**.

It also displays a visualization (using `accumulator updates` and the `SparkPlanGraph` for the query) with the expandable **Details** section (that corresponds to `SQLExecutionUIData.physicalPlanDescription` ).

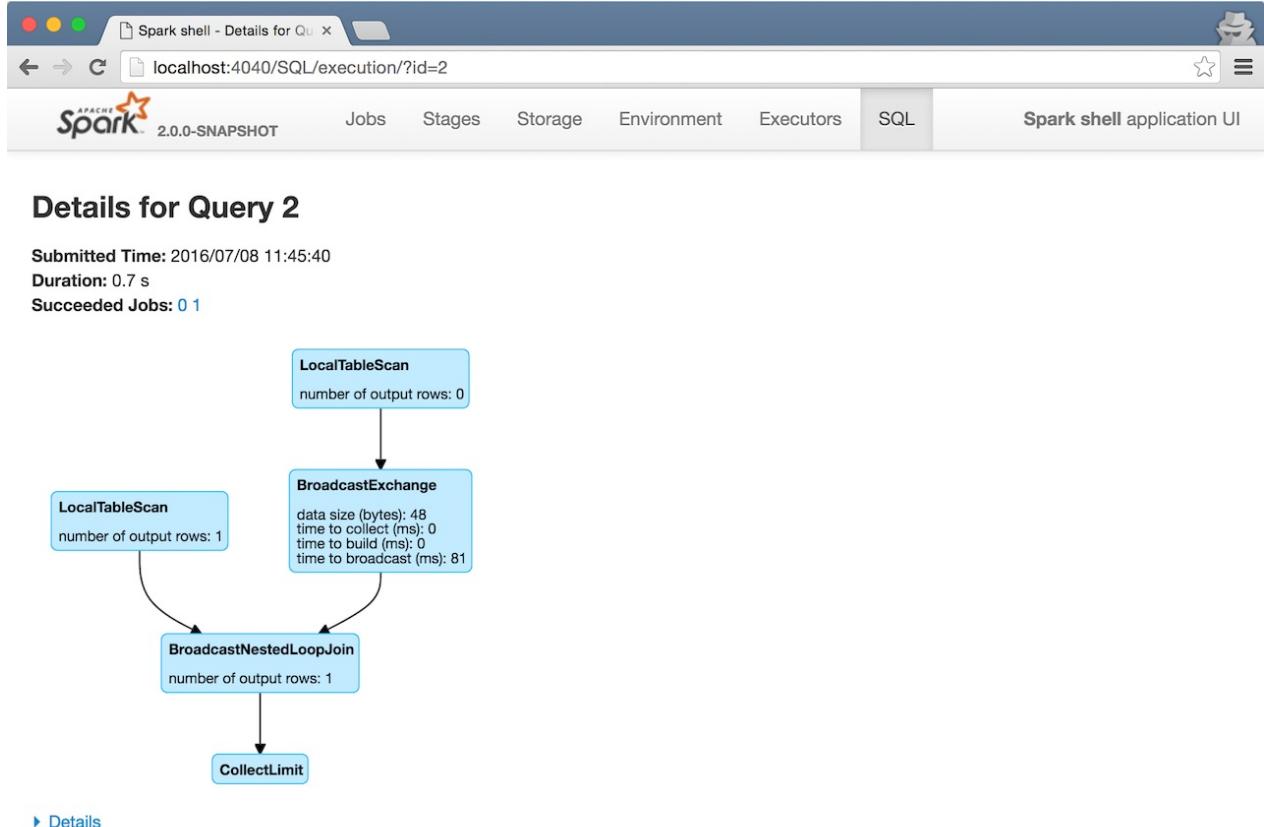


Figure 2. Details for SQL Query in web UI

If there is no information to display for a given query `id`, you should see the following page.



Figure 3. No Details for SQL Query

Internally, it uses `SQLListener` exclusively to get the SQL query execution metrics. It requests `SQLListener` for `SQL execution data` to display for the `id` request parameter.

## Creating SQLTab Instance

`SQLTab` is created when `SharedState` is or at the first `SparkListenerSQLExecutionStart` event when `Spark History Server` is used.

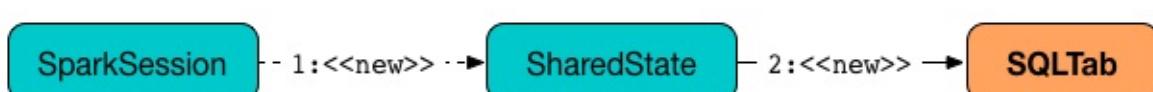


Figure 4. Creating SQLTab Instance

Note	SharedState represents the shared state across all active SQL sessions.
------	---

# SQLListener

`SQLListener` is a custom [SparkListener](#) that collects information about SQL query executions for web UI (to display in [SQL tab](#)). It relies on `spark.sql.execution.id` key to distinguish between queries.

Internally, it uses [SQLExecutionUIData](#) data structure exclusively to record all the necessary data for a single SQL query execution. `SQLExecutionUIData` is tracked in the internal registries, i.e. `activeExecutions`, `failedExecutions`, and `completedExecutions` as well as lookup tables, i.e. `_executionIdToData`, `_jobIdToExecutionId`, and `_stageIdToStageMetrics`.

`SQLListener` starts recording a query execution by intercepting a [SparkListenerSQLExecutionStart](#) event (using `onOtherEvent` callback).

`SQLListener` stops recording information about a SQL query execution when [SparkListenerSQLExecutionEnd](#) event arrives.

It defines the other callbacks (from [SparkListener](#) interface):

- [onJobStart](#)
- [onJobEnd](#)
- [onExecutorMetricsUpdate](#)
- [onStageSubmitted](#)
- [onTaskEnd](#)

## Registering Job and Stages under Active Execution ([onJobStart](#) callback)

```
onJobStart(jobStart: SparkListenerJobStart): Unit
```

`onJobStart` reads the `spark.sql.execution.id` key, the identifiers of the job and the stages and then updates the [SQLExecutionUIData](#) for the execution id in `activeExecutions` internal registry.

Note	When <code>onJobStart</code> is executed, it is assumed that <a href="#">SQLExecutionUIData</a> has already been created and available in the internal <code>activeExecutions</code> registry.
------	--

The job in `SQLExecutionUIData` is marked as running with the stages added (to `stages`). For each stage, a `SQLStageMetrics` is created in the internal `_stageIdToStageMetrics` registry. At the end, the execution id is recorded for the job id in the internal `_jobIdToExecutionId`.

## onOtherEvent

In `onOtherEvent`, `SQLListener` listens to the following `SparkListenerEvent` events:

- `SparkListenerSQLExecutionStart`
- `SparkListenerSQLExecutionEnd`
- `SparkListenerDriverAccumUpdates`

## Registering Active Execution (`SparkListenerSQLExecutionStart` Event)

```
case class SparkListenerSQLExecutionStart(
    executionId: Long,
    description: String,
    details: String,
    physicalPlanDescription: String,
    sparkPlanInfo: SparkPlanInfo,
    time: Long)
extends SparkListenerEvent
```

`SparkListenerSQLExecutionStart` events starts recording information about the `executionId` SQL query execution.

When a `SparkListenerSQLExecutionStart` event arrives, a new `SQLExecutionUIData` for the `executionId` query execution is created and stored in `activeExecutions` internal registry. It is also stored in `_executionIdToData` lookup table.

## SparkListenerSQLExecutionEnd

```
case class SparkListenerSQLExecutionEnd(
    executionId: Long,
    time: Long)
extends SparkListenerEvent
```

`SparkListenerSQLExecutionEnd` event stops recording information about the `executionId` SQL query execution (tracked as `SQLExecutionUIData`). `SQLListener` saves the input time as `completionTime`.

If there are no other running jobs (registered in `SQLExecutionUIData`), the query execution is removed from the `activeExecutions` internal registry and moved to either `completedExecutions` or `failedExecutions` registry.

This is when `SQLListener` checks the number of `SQLExecutionUIData` entires in either registry — `failedExecutions` or `completedExecutions` — and removes the excess of the old entries beyond `spark.sql.ui.retainedExecutions`.

## SparkListenerDriverAccumUpdates

```
case class SparkListenerDriverAccumUpdates(
    executionId: Long,
    accumUpdates: Seq[(Long, Long)])
extends SparkListenerEvent
```

When `SparkListenerDriverAccumUpdates` comes, `SQLExecutionUIData` for the input `executionId` is looked up (in `_executionIdToData`) and `SQLExecutionUIData.driverAccumUpdates` is updated with the input `accumUpdates`.

## onJobEnd

```
onJobEnd(jobEnd: SparkListenerJobEnd): Unit
```

When called, `onJobEnd` retrieves the `SQLExecutionUIData` for the job and records it either successful or failed depending on the job result.

If it is the last job of the query execution (tracked as `SQLExecutionUIData`), the execution is removed from `activeExecutions` internal registry and moved to either

If the query execution has already been marked as completed (using `completionTime`) and there are no other running jobs (registered in `SQLExecutionUIData`), the query execution is removed from the `activeExecutions` internal registry and moved to either `completedExecutions` or `failedExecutions` registry.

This is when `SQLListener` checks the number of `SQLExecutionUIData` entires in either registry — `failedExecutions` or `completedExecutions` — and removes the excess of the old entries beyond `spark.sql.ui.retainedExecutions`.

## Getting SQL Execution Data (getExecution method)

```
getExecution(executionId: Long): Option[SQLExecutionUIData]
```

## Getting Execution Metrics (getExecutionMetrics method)

```
getExecutionMetrics(executionId: Long): Map[Long, String]
```

`getExecutionMetrics` gets the metrics (aka *accumulator updates*) for `executionId` (by which it collects all the tasks that were used for an execution).

It is exclusively used to render the [ExecutionPage](#) page in web UI.

## mergeAccumulatorUpdates method

`mergeAccumulatorUpdates` is a `private` helper method for...TK

It is used exclusively in [getExecutionMetrics](#) method.

## SQLExecutionUIData

`SQLExecutionUIData` is the data abstraction of `SQLListener` to describe SQL query executions. It is a container for jobs, stages, and accumulator updates for a single query execution.

## Settings

### spark.sql.ui.retainedExecutions

`spark.sql.ui.retainedExecutions` (default: `1000`) is the number of `SQLExecutionUIData` entries to keep in `failedExecutions` and `completedExecutions` internal registries.

When a query execution finishes, the execution is removed from the internal `activeExecutions` registry and stored in `failedExecutions` or `completedExecutions` given the end execution status. It is when `SQLListener` makes sure that the number of `SQLExecutionUIData` entries does not exceed `spark.sql.ui.retainedExecutions` and removes the excess of the old entries.

# JobProgressListener

`JobProgressListener` is the [SparkListener](#) for web UI.

As a `SparkListener` it intercepts [Spark events](#) and collect information about jobs, stages, and tasks that the web UI uses to present the status of a Spark application.

`JobProgressListener` is interested in the following events:

1. A job starts.

Caution

[FIXME](#) What information does `JobProgressListener` track?

## poolToActiveStages

```
poolToActiveStages = HashMap[PoolName, HashMap[StageId, StageInfo]]()
```

`poolToActiveStages` ...

Caution

[FIXME](#)

## Handling SparkListenerJobStart Events (onJobStart method)

```
onJobStart(jobStart: SparkListenerJobStart): Unit
```

When called, `onJobStart` reads the optional Spark Job group id (using `SparkListenerJobStart.properties` and `SparkContext.SPARK_JOB_GROUP_ID` key).

It then creates a [JobUIData](#) (as `jobData`) based on the input `jobStart`. `status` attribute is `JobExecutionStatus.RUNNING`.

The internal `jobGroupToJobIds` is updated with the job group and job ids.

The internal `pendingStages` is updated with `StageInfo` for the stage id (for every `StageInfo` in `SparkListenerJobStart.stageInfos` collection).

`numTasks` attribute in the `jobData` (as `JobUIData` instance created above) is set to the sum of tasks in every stage (from `jobStart.stageInfos`) for which `completionTime` attribute is not set.

The internal `jobIdToDate` and `activeJobs` are updated with `jobData` for the current job.

The internal `stageIdToActiveJobIds` is updated with the stage id and job id (for every stage in the input `jobStart` ).

The internal `stageIdToInfo` is updated with the stage id and `StageInfo` (for every `StageInfo` in `jobStart.stageInfos` ).

A `StageUIData` is added to the internal `stageIdToData` for every `StageInfo` (in `jobStart.stageInfos` ).

Note	<code>onJobStart</code> is a part of <a href="#">SparkListener contract</a> to handle... <a href="#">FIXME</a>
------	--

## stageIdToInfo Registry

```
stageIdToInfo = new HashMap[StageId, StageInfo]
```

## stageIdToActiveJobIds Registry

```
stageIdToActiveJobIds = new HashMap[StageId, HashSet[JobId]]
```

## jobIdToData Registry

```
jobIdToData = new HashMap[JobId, JobUIData]
```

## activeJobs Registry

```
activeJobs = new HashMap[JobId, JobUIData]
```

## pendingStages Registry

```
pendingStages = new HashMap[StageId, StageInfo]
```

Caution	<a href="#">FIXME</a>
---------	-----------------------

## JobUIData

Caution	<a href="#">FIXME</a>
---------	-----------------------

## blockManagerIds method

```
blockManagerIds: Seq[BlockManagerId]
```

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Registries

### stageIdToData Registry

```
stageIdToData = new HashMap[(StageId, StageAttemptId), StageUIData]
```

`stageIdToData` holds `StageUIData` per stage (given the stage and attempt ids).

### StageUIData

Caution	<a href="#">FIXME</a>
---------	-----------------------

### schedulingMode Attribute

`schedulingMode` attribute is used to show the [scheduling mode](#) for the Spark application in [Spark UI](#).

Note	It corresponds to <code>spark.scheduler.mode</code> setting.
------	--

When [SparkListenerEnvironmentUpdate](#) is received, `JobProgressListener` looks up

`spark.scheduler.mode` key in `Spark Properties` map to set the internal `schedulingMode` field.

Note	It is used in Jobs and Stages tabs.
------	-------------------------------------

# spark-submit script

`spark-submit` script allows you to manage your Spark applications. You can [submit your Spark application to a Spark deployment environment](#), [kill](#) or [request status](#) of Spark applications.

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

```
$ ./bin/spark-submit
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://...]
Usage: spark-submit run-example [options] example-class [example args]
...
```

When executed, `spark-submit` script executes `org.apache.spark.deploy.SparkSubmit` class.

## Driver Cores in Cluster Deploy Mode (`--driver-cores` option)

```
--driver-cores NUM
```

`--driver-cores` command-line option sets the number of cores for the [driver](#) in the [cluster deploy mode](#).

Note	<code>--driver-cores</code> switch is only available for cluster mode (for Standalone, Mesos, and YARN).
------	--

Note	It corresponds to <code>spark.driver.cores</code> setting.
------	--

Note	It is printed out to the standard error output in <a href="#">verbose mode</a> .
------	--

## System Properties

`spark-submit` collects system properties for execution in the internal `sysProps`.

Caution	<a href="#">FIXME</a> How is <code>sysProps</code> calculated?
---------	--

## Additional JAR Files to Distribute (`--jars` option)

```
--jars JARS
```

--jars is a comma-separated list of local jars to include on the driver's and executors' classpaths.

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Additional Files to Distribute (--files option)

--files FILES
---------------

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Additional Archives to Distribute (--archives option)

--archives ARCHIVES
---------------------

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Specifying YARN Resource Queue (--queue option)

--queue QUEUE_NAME
--------------------

With --queue you can choose the YARN resource queue to [submit a Spark application to](#). The default queue name is `default`.

Caution	<a href="#">FIXME</a> What is a queue ?
---------	---

Note	It corresponds to <code>spark.yarn.queue</code> Spark's setting.
------	--

Tip	It is printed out to the standard error output in <a href="#">verbose mode</a> .
-----	--

## Actions

### Submitting Applications for Execution (submit method)

The default action of `spark-submit` script is to submit a Spark application to a deployment environment for execution.

**Tip** Use `--verbose` command-line switch to know the main class to be executed, arguments, system properties, and classpath (to ensure that the command-line arguments and switches were processed properly).

When executed, `spark-submit` executes `submit` method.

```
submit(args: SparkSubmitArguments): Unit
```

If `proxyUser` is set it will...[FIXME](#)

<b>Caution</b>	<a href="#">FIXME</a> Review why and when to use <code>proxyUser</code> .
----------------	---

It passes the execution on to [runMain](#).

## Executing Main ([runMain](#) method)

```
runMain(
  childArgs: Seq[String],
  childClasspath: Seq[String],
  sysProps: Map[String, String],
  childMainClass: String,
  verbose: Boolean): Unit
```

`runMain` is an internal method to build execution environment and invoke the main method of the Spark application that has been submitted for execution.

<b>Note</b>	It is exclusively used when <a href="#">submitting applications for execution</a> .
-------------	---

It optionally prints out input parameters with `verbose` input flag enabled (i.e. `true`).

<b>Note</b>	<code>verbose</code> flag corresponds to <a href="#">--verbose switch</a> .
-------------	---

It builds the context classloader depending on `spark.driver.userClassPathFirst` flag.

<b>Caution</b>	<a href="#">FIXME</a> What is <code>spark.driver.userClassPathFirst</code> ?
----------------	--

It adds the local jars specified in `childClasspath` input parameter to the context classloader (that is later responsible for loading the `childMainClass` main class).

<b>Note</b>	<code>childClasspath</code> input parameter corresponds to <a href="#">--jars switch</a> with the primary resource if specified in <a href="#">client deploy mode</a> .
-------------	---

It sets all the system properties specified in `sysProps` input parameter.

<b>Note</b>	It uses Java's <a href="#">System.setProperty</a> to set the system properties.
-------------	---

**Tip**

Read [System Properties](#) about how the process of collecting system properties works.

It creates an instance of `childMainClass` main class.

**Note**

`childMainClass` corresponds to the main class `spark-submit` was invoked with.

**Tip**

You should avoid using `scala.App` trait for main classes in Scala as reported in [SPARK-4170 Closure problems when running Scala app that "extends App"](#)

If you use `scala.App` for the main class, you should see the following WARN message in the logs:

```
WARN Subclasses of scala.App may not work correctly. Use a main() method instead.
```

Finally, it executes the `main` method of the Spark application passing in the `childArgs` arguments.

Any `SparkUserAppException` exceptions lead to `System.exit` while the others are simply re-thrown.

### Adding Local Jars to ClassLoader (addJarToClasspath method)

```
addJarToClasspath(localJar: String, loader: MutableURLClassLoader)
```

`addJarToClasspath` is an internal method to add `file` or `local` jars (as `localJar`) to the `loader` classloader.

Internally, `addJarToClasspath` resolves the URI of `localJar`. If the URI is `file` or `local` and the file denoted by `localJar` exists, `localJar` is added to `loader`. Otherwise, the following warning is printed out to the logs:

```
Warning: Local jar /path/to/fake.jar does not exist, skipping.
```

For all other URIs, the following warning is printed out to the logs:

```
Warning: Skip remote jar hdfs://fake.jar.
```

**Note**

`addJarToClasspath` assumes `file` URI when `localJar` has no URI specified, e.g. `/path/to/local.jar`.

**Caution**

**FIXME** What is a URI fragment? How does this change re YARN distributed cache? See `utils#resolveURI` .

## Killing Applications (`--kill` switch)

`--kill`

## Requesting Application Status (`--status` switch)

`--status`

## Command-line Options

Execute `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://...]
Usage: spark-submit run-example [options] example-class [example args]

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.
                                Comma-separated list of maven coordinates of jars to include
                                on the driver and executor classpaths. Will search the local
                                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 FILE       Comma-separated list of groupId:artifactId, to exclude while
                                resolving the dependencies provided in --packages to avoid
                                dependency conflicts.
  --repositories REPO           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
```

```

e
  --files FILES
g
  --conf PROP=VALUE
  --properties-file FILE
ot
  --driver-memory MEM
  --driver-java-options
  --driver-library-path
  --driver-class-path
t
  --executor-memory MEM
  --proxy-user NAME
  --help, -h
  --verbose, -v
  --version,
Spark standalone with cluster deploy mode only:
  --driver-cores NUM
Spark standalone or Mesos with cluster deploy mode only:
  --supervise
  --kill SUBMISSION_ID
  --status SUBMISSION_ID
Spark standalone and Mesos only:
  --total-executor-cores NUM
Spark standalone and YARN only:
  --executor-cores NUM
YARN-only:
  --driver-cores NUM
  --queue QUEUE_NAME
  --num-executors NUM
  --archives ARCHIVES
e
  --principal PRINCIPAL
on the PYTHONPATH for Python apps.
Comma-separated list of files to be placed in the working directory of each executor.
Arbitrary Spark configuration property.
Path to a file from which to load extra properties. If not specified, this will look for conf/spark-defaults.conf.
Memory for driver (e.g. 1000M, 2G) (Default: 1024M).
Extra Java options to pass to the driver.
Extra library path entries to pass to the driver.
Extra class path entries to pass to the driver. Note that jars added with --jars are automatically included in the classpath.
Memory per executor (e.g. 1000M, 2G) (Default: 1G).
User to impersonate when submitting the application.
This argument does not work with --principal / --keytab.
Show this help message and exit.
Print additional debug output.
Print the version of current Spark.
Cores for driver (Default: 1).
If given, restarts the driver on failure.
If given, kills the driver specified.
If given, requests the status of the driver specified.
Total cores for all executors.
Number of cores per executor. (Default: 1 in YARN mode, or all available cores on the worker in standalone mode)
Number of cores used by the driver, only in cluster mode (Default: 1).
The YARN queue to submit to (Default: "default").
Number of executors to launch (Default: 2).
Comma separated list of archives to be extracted into the working directory of each executor.
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` ([see Deploy Mode](#))
- `--driver-class-path`
- `--driver-cores` ([see Driver Cores in Cluster Deploy Mode](#))
- `--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` (see **Verbose Mode**)
  - `--version` (see **Version**)

## YARN-only options:

- `--archives`
  - `--executor-cores`
  - `--keytab`
  - `--num-executors`
  - `--principal`
  - `--queue` (see [Specifying YARN Resource Queue \(--queue switch\)](#))

## **Version (--version switch)**

### **Verbose Mode (`--verbose` switch)**

When `spark-submit` is executed with `--verbose` command-line switch, it enters **verbose mode**.

In verbose mode, the parsed arguments are printed out to the System error output.

ETXMF

It also prints out `propertiesFile` and the properties from the file.

FIXME

## Deploy Mode (--deploy-mode switch)

You use spark-submit's `--deploy-mode` command-line option to specify the [deploy mode](#) for a Spark application.

## 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?

## Launching SparkSubmit (main method)

**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](#).

**Tip**

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

When executed, `spark-submit` script simply passes the call to `spark-class` with `org.apache.spark.deploy.SparkSubmit` class followed by command-line arguments.

It creates an instance of [SparkSubmitArguments](#).

If in [verbose mode](#), it prints out the application arguments.

It then relays the execution to [action-specific internal methods](#) (with the application arguments):

- When no action was explicitly given, it is assumed `submit` action.
- `kill` (when `--kill` switch is used)
- `requestStatus` (when `--status` switch is used)

**Note**

The action can only have one of the three available values: `SUBMIT` , `KILL` , or `REQUEST_STATUS` .

## SparkSubmitArguments — spark-submit Command-Line Argument Parser

`SparkSubmitArguments` is a `private[deploy]` class to handle the command-line arguments of `spark-submit` script that the [actions](#) use for their execution (possibly with the explicit `env` environment).

```
SparkSubmitArguments(  
  args: Seq[String],  
  env: Map[String, String] = sys.env)
```

**Note**

`SparkSubmitArguments` is created when [launching](#) `spark-submit` script with only `args` passed in and later used for printing the arguments in [verbose mode](#).

## 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 script

`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](#).

Tip	spark-class script loads additional environment settings (see <a href="#">spark-env.sh section in this document</a> ). And then spark-class searches for so-called <b>the Spark assembly jar</b> ( <code>spark-assembly.hadoop..jar</code> ) in <code>SPARK_HOME/lib</code> or <code>SPARK_HOME/assembly/target/scala-\$SPARK_SCALA_VERSION</code> for a binary distribution or Spark built from sources, respectively.
-----	---

Note	Set <code>SPARK_PREPEND_CLASSES</code> to have the Spark launcher classes (from <code>\$SPARK_HOME/launcher/target/scala-\$SPARK_SCALA_VERSION/classes</code> ) 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. The `Main` class programmatically computes the final command to be executed.

## 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.

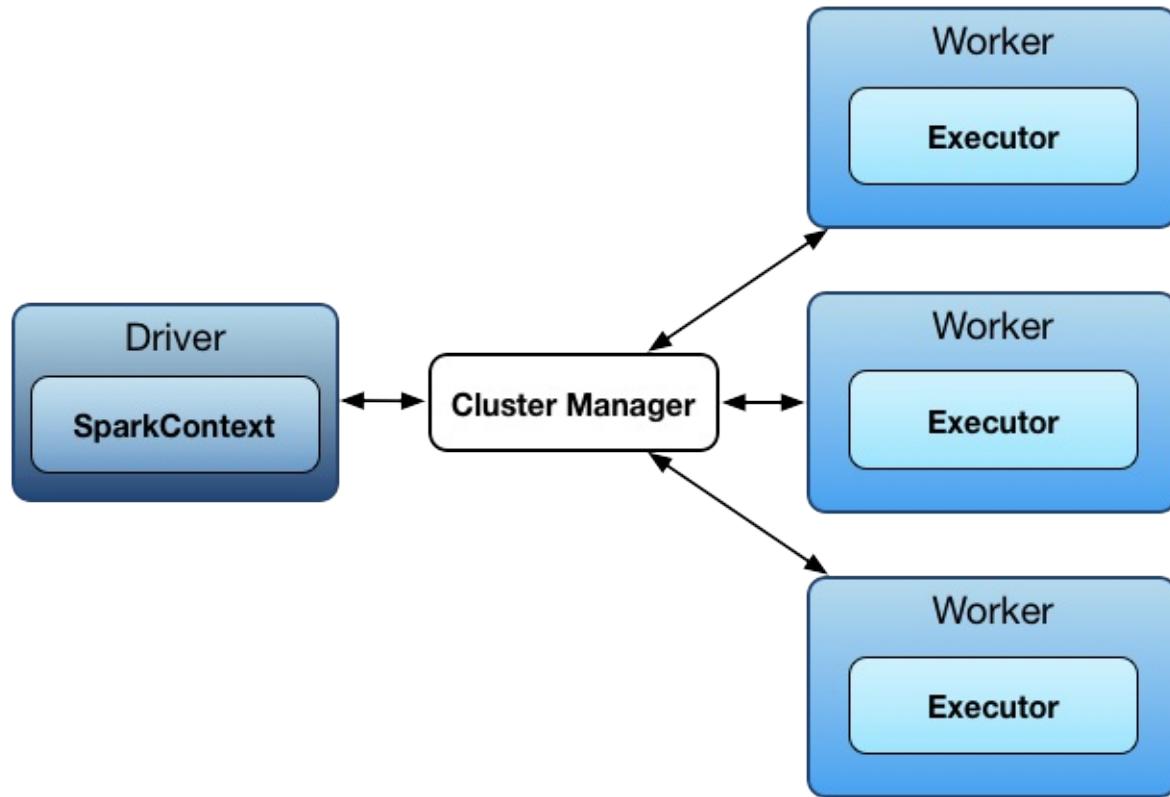


Figure 1. Spark architecture

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.

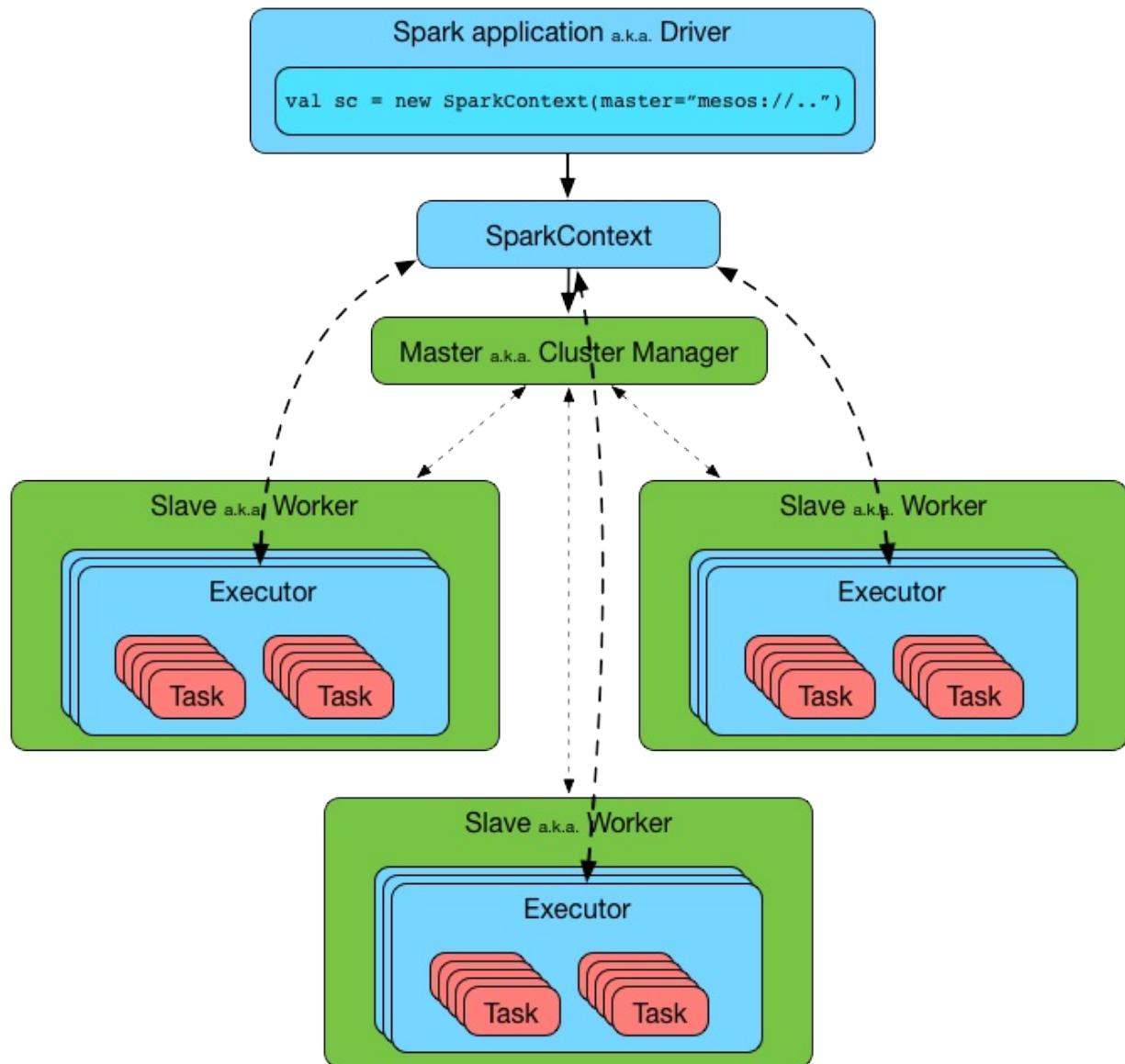


Figure 2. Spark architecture in detail

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

# Driver

A **Spark driver** (aka an application's driver process) is a JVM process that hosts [SparkContext](#) for a Spark application.

It is the cockpit of jobs and tasks execution (using [DAGScheduler](#) and [Task Scheduler](#)). It hosts [Web UI](#) for the environment.

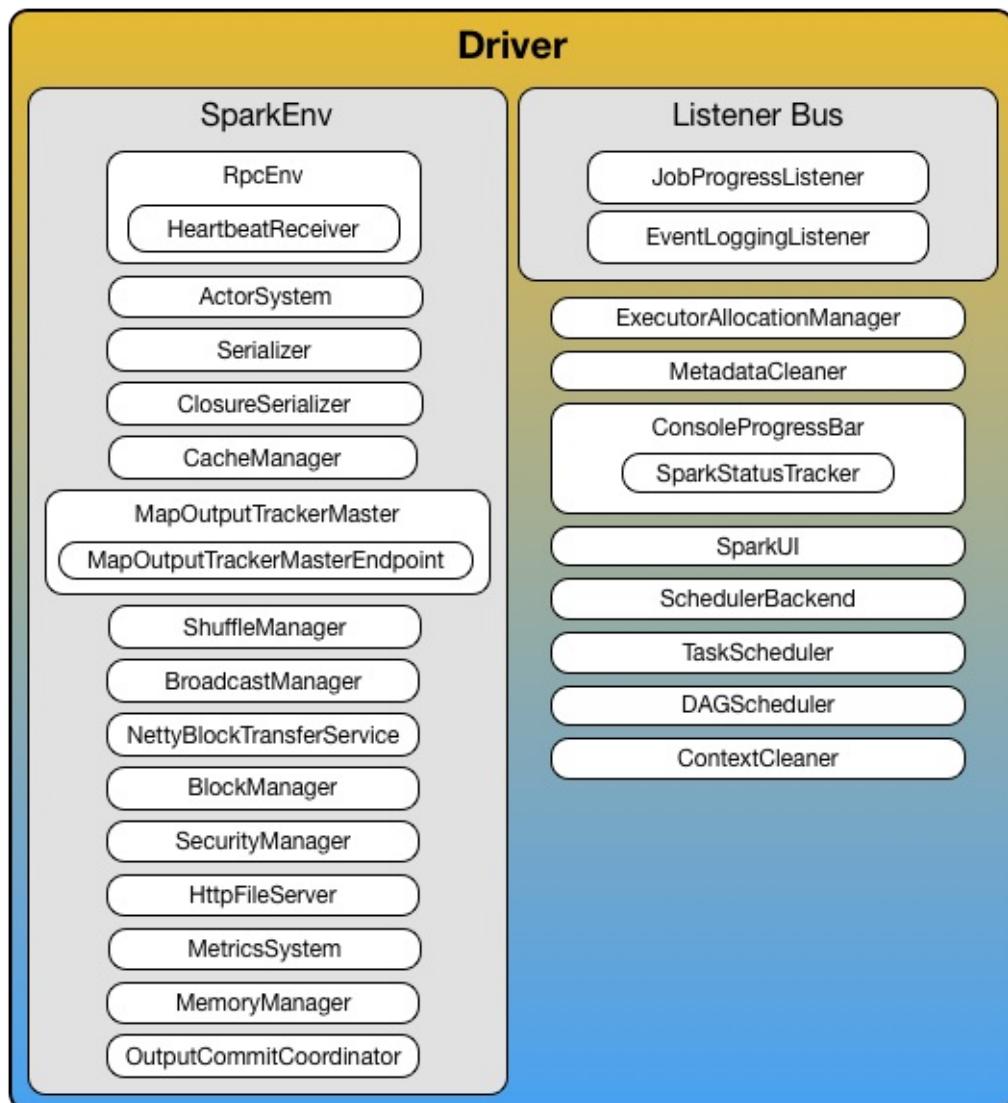


Figure 1. Driver with the services

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.

Note

[Spark shell](#) is a Spark application and the driver. It creates a [SparkContext](#) that is available as `sc`.

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

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

Caution	<a href="#">FIXME</a> Diagram of RpcEnv for a driver (and later executors). Perhaps it should be in the notes about RpcEnv?
---------	---

- 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
- Launches [tasks](#)

## Driver's Memory

It can be set first using [spark-submit](#)'s `--driver-memory` command-line option or [spark.driver.memory](#) and falls back to [SPARK\\_DRIVER\\_MEMORY](#) if not set earlier.

Note	It is printed out to the standard error output in <a href="#">spark-submit</a> 's verbose mode.
------	---

## Driver's Cores

It can be set first using [spark-submit](#)'s `--driver-cores` command-line option for [cluster deploy mode](#).

Note	In <a href="#">client deploy mode</a> the driver's memory corresponds to the memory of the JVM process the Spark application runs on.
------	---

Note	It is printed out to the standard error output in <a href="#">spark-submit</a> 's verbose mode.
------	---

## Settings

### spark.driver.libraryPath

`spark.driver.libraryPath`

### spark.driver.extraLibraryPath

`spark.driver.extraLibraryPath`

### spark.driver.extraJavaOptions

`spark.driver.extraJavaOptions`

### spark.driver.appUIAddress

`spark.driver.appUIAddress` is only used in [Spark on YARN](#). It is set when [YarnClientSchedulerBackend](#) starts to run [ExecutorLauncher](#) (and [register ApplicationMaster](#) for the Spark application).

### spark.driver.extraClassPath

`spark.driver.extraClassPath` is an optional setting that is used to...[FIXME](#)

### spark.driver.cores

`spark.driver.cores` (default: `1`) sets the number of CPU cores assigned for the driver in [cluster deploy mode](#).

Note

When [Client is created](#) (for Spark on YARN in cluster mode only), it sets the number of cores for [ApplicationManager](#) using `spark.driver.cores`.

Read [Driver's Cores](#) for a closer coverage.

### spark.driver.memory

`spark.driver.memory` (default: `1g`) sets the driver's memory size (in MiBs).

Read [Driver's Memory](#) for a closer coverage.

# 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	<a href="#">FIXME</a> Are workers perhaps part of Spark Standalone only?
---------	--

Caution	<a href="#">FIXME</a> 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	<a href="#">FIXME</a> 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	<a href="#">FIXME</a>
---------	-----------------------

## Settings

- `spark.worker.cleanup.enabled` (default: `false`) [Cleanup enabled](#).

# Executors

**Executors** are distributed agents that execute [tasks](#).

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

Note

Executors are managed exclusively 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 can run multiple tasks over its lifetime, both in parallel and sequentially. They track [running tasks](#) (by their task ids in `runningTasks` internal map). Consult [Launching Tasks](#) section.

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

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

Enable `INFO` or `DEBUG` logging level for `org.apache.spark.executor.Executor` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.executor.Executor=INFO
```

Refer to [Logging](#).

## Creating Executor Instance

`Executor` requires `executorId`, `executorHostname`, a `SparkEnv` (as `env`), `userClassPath` and whether it runs in local or non-local mode (as `isLocal` that is non-local by default).

Note	<code>isLocal</code> is enabled exclusively for <a href="#">LocalEndpoint</a> (for <a href="#">Spark in local mode</a> ).
------	---

While an executor is being created 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 sending heartbeats to the driver (using the internal `startDriverHeartbeater` method).

The [BlockManager](#) is initialized (only when in non-local/cluster mode).

Note	The <code>BlockManager</code> for an executor is available in <code>sparkEnv</code> passed to the constructor.
------	--

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

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

Note	A <code>Executor</code> is created when <code>CoarseGrainedExecutorBackend</code> receives <code>RegisteredExecutor</code> message, in <code>MesosExecutorBackend.registered</code> and when <code>LocalEndpoint</code> is created.
------	---

Caution	<a href="#"><b>FIXME</b></a> How many cores are assigned per executor?
---------	--

## Launching Tasks (`launchTask` method)

<pre>launchTask(     context: ExecutorBackend,     taskId: Long,     attemptNumber: Int,     taskName: String,     serializedTask: ByteBuffer): Unit</pre>
--

`launchTask` creates a [TaskRunner](#) object, registers it in the internal `runningTasks` map (by `taskId`), and executes it on [Executor task launch worker Thread Pool](#).

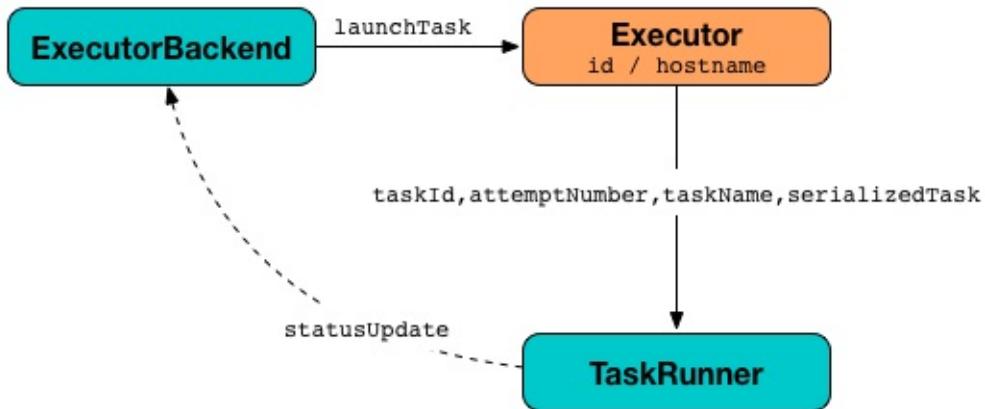


Figure 1. Launching tasks on executor using TaskRunners

Note

`launchTask` is called by [CoarseGrainedExecutorBackend](#) (when it handles [LaunchTask](#) message), [MesosExecutorBackend](#), and [LocalEndpoint](#) that represent different cluster managers.

## Sending Heartbeats and Active Tasks Metrics (startDriverHeartbeater method)

Executors keep sending [metrics for active tasks](#) to the driver every `spark.executor.heartbeatInterval` (defaults to `10s` with some random initial delay so the heartbeats from different executors do not pile up on the driver).

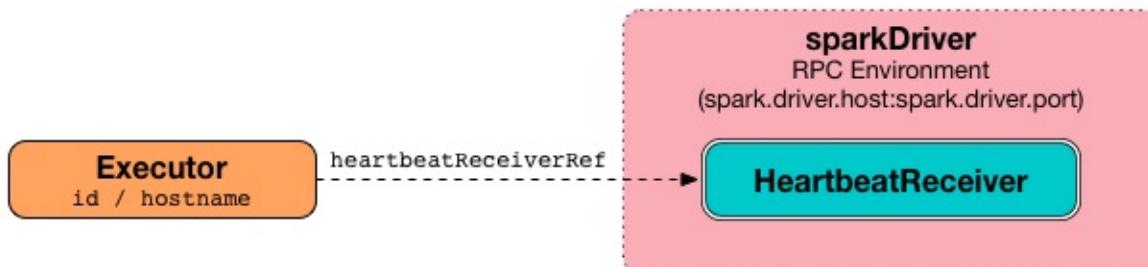


Figure 2. Executors use HeartbeatReceiver endpoint to report task metrics

An executor sends heartbeats using the [internal heartbeater - Heartbeat Sender Thread](#).

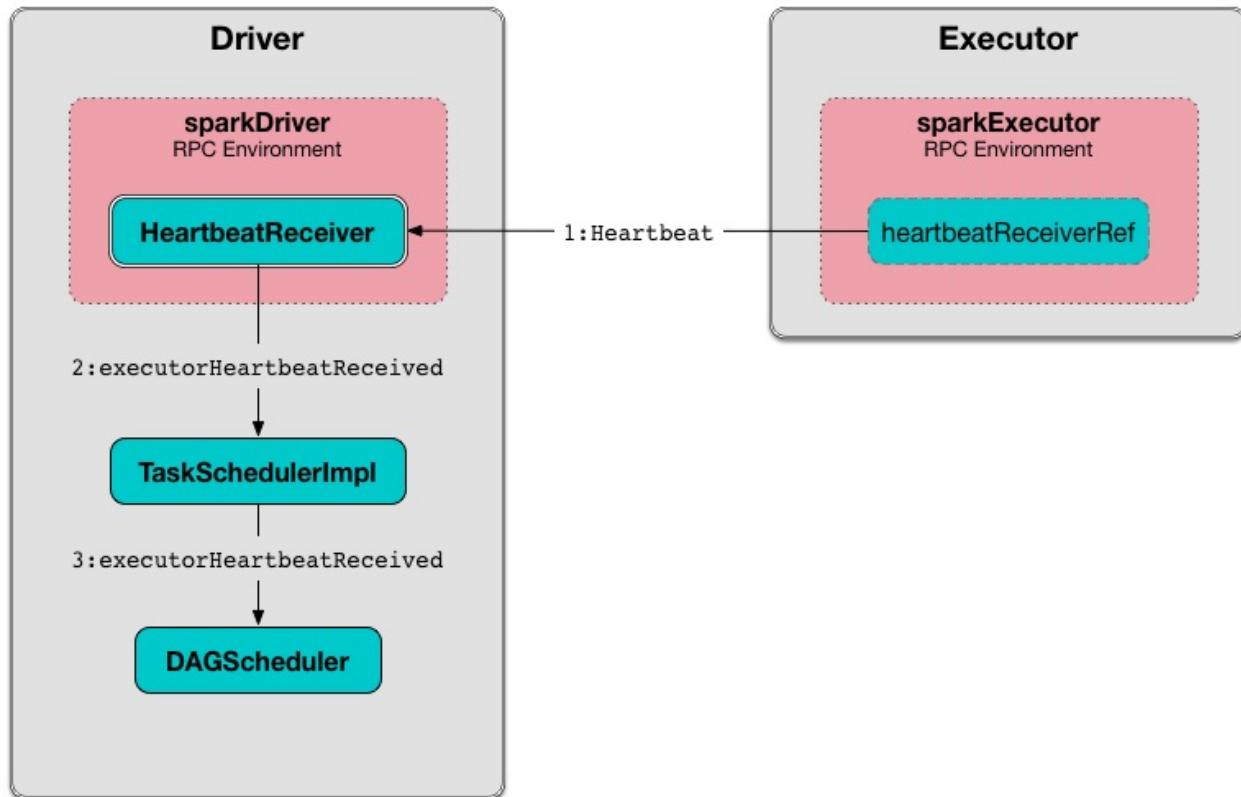


Figure 3. HeartbeatReceiver's Heartbeat Message Handler

For each `task` in `TaskRunner` (in the internal `runningTasks` registry), the task's metrics are computed (i.e. `mergeShuffleReadMetrics` and `setJvmGCTime`) that become part of the heartbeat (with accumulators).

Caution	<b>FIXME</b> How do <code>mergeShuffleReadMetrics</code> and <code>setJvmGCTime</code> influence accumulators ?
---------	---

Note	Executors track the <code>TaskRunner</code> that run <code>tasks</code> . A <code>task</code> might not be assigned to a <code>TaskRunner</code> yet when the executor sends a heartbeat.
------	---

A blocking `Heartbeat` message that holds the executor id, all accumulator updates (per task id), and `BlockManagerId` is sent to `HeartbeatReceiver` `RPC endpoint` (with `spark.executor.heartbeatInterval` timeout).

Caution	<b>FIXME</b> When is <code>heartbeatReceiverRef</code> created?
---------	---

If the response `requests to reregister BlockManager`, you should see the following `INFO` message in the logs:

INFO Executor: Told to re-register on heartbeat
---

The `BlockManager` is reregistered.

The internal `heartbeatFailures` counter is reset (i.e. becomes `0`).

If there are any issues with communicating with the driver, you should see the following WARN message in the logs:

```
WARN Executor: Issue communicating with driver in heartbeater
```

The internal `heartbeatFailures` is incremented and checked to be less than the [acceptable number of failures](#). If the number is greater, the following ERROR is printed out to the logs:

```
ERROR Executor: Exit as unable to send heartbeats to driver more than [HEARTBEAT_MAX_FAILURES] times
```

The executor exits (using `System.exit` and exit code 56).

Tip

Read about `TaskMetrics` in [TaskMetrics](#).

## heartbeater - Heartbeat Sender Thread

`heartbeater` is a daemon [ScheduledThreadPoolExecutor](#) with a single thread.

The name of the thread pool is **driver-heartbeater**.

## Coarse-Grained Executors

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

## 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	<a href="#">FIXME</a> 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 <a href="#">SparkContext is created</a> .
------	---

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 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 4. 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**

## Internal Registries

- `runningTasks` is ...[FIXME](#)
- `heartbeatFailures` is ...[FIXME](#)

## Settings

### **spark.executor.cores**

`spark.executor.cores` - the number of cores for an executor

### **spark.executor.extraClassPath**

`spark.executor.extraClassPath` is a list of URLs representing a user's CLASSPATH.

Each entry is separated by system-dependent path separator, i.e. `:` on Unix/MacOS systems and `;` on Microsoft Windows.

### **spark.executor.extraJavaOptions**

`spark.executor.extraJavaOptions` - extra Java options for executors.

It is used to [prepare the command to launch CoarseGrainedExecutorBackend](#) in a YARN container.

## spark.executor.extraLibraryPath

`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.

It is used to [prepare the command to launch CoarseGrainedExecutorBackend](#) in a YARN container.

## spark.executor.userClassPathFirst

`spark.executor.userClassPathFirst` (default: `false`) controls whether to load classes in user jars before those in Spark jars.

## spark.executor.heartbeatInterval

`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.heartbeat.maxFailures

`spark.executor.heartbeat.maxFailures` (default: `60`) controls how many times an executor will try to send heartbeats to the driver before it gives up and exits (with exit code `56`).

Note

It was introduced in [SPARK-13522 Executor should kill itself when it's unable to heartbeat to the driver more than N times](#)

## spark.executor.id

`spark.executor.id`

## spark.executor.instances

`spark.executor.instances` (default: `0`) sets the number of executors to use.

When greater than `0`, it disables [dynamic allocation](#).

## spark.executor.memory

`spark.executor.memory` (default: `1g`) - 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.

## Others

- `spark.executor.logs.rolling.maxSize`
- `spark.executor.logs.rolling.maxRetainedFiles`
- `spark.executor.logs.rolling.strategy`
- `spark.executor.logs.rolling.time.interval`
- `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.

# TaskRunner

`TaskRunner` is a thread of execution that manages a single individual [task](#). It can be [run](#) or [killed](#) that boils down to [running](#) or [killing the task](#) the `TaskRunner` object manages.

**Tip** Enable `INFO` or `DEBUG` logging level for `org.apache.spark.executor.Executor` logger to see what happens inside `TaskRunner` (since `TaskRunner` is an internal class of `Executor` ).

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

```
log4j.logger.org.apache.spark.executor.Executor=DEBUG
```

Refer to [Logging](#).

## Lifecycle

Caution	<a href="#">FIXME</a> Image with state changes
---------	--

A `TaskRunner` object is created when [an executor is requested to launch a task](#).

It is created with an [ExecutorBackend](#) (to send the task's status updates to), task and attempt ids, task name, and serialized version of the task (as `ByteBuffer` ).

## Running Task (run method)

Note	<code>run</code> is part of <a href="#">java.lang.Runnable</a> contract that <code>TaskRunner</code> follows.
------	---

When `run` is executed, it creates a [TaskMemoryManager](#) object (using the global [MemoryManager](#) and the constructor's `taskId`) to manage the memory allocated for the task's execution.

It starts measuring the time to deserialize a task.

It sets the current context classloader.

Caution	<a href="#">FIXME</a> What is part of the classloader?
---------	--

It creates a new instance of the global [closure Serializer](#).

You should see the following INFO message in the logs:

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

At this point, the task is considered running and the `ExecutorBackend.statusUpdate` is executed (with `taskId` and `TaskState.RUNNING` state).

`run` deserializes the task's environment (from `serializedTask` bytes using `Task.deserializeWithDependencies`) to have the task's files, jars and properties, and the bytes (i.e. the real task's body).

Note	The target task to run is not serialized yet, but only its environment - the files, jars, and properties.
------	---

Caution	<b>FIXME</b> Describe <code>Task.deserializeWithDependencies</code> .
---------	---

`updateDependencies(taskFiles, taskJars)` is called.

Caution	<b>FIXME</b> What does <code>updateDependencies</code> do?
---------	--

This is the moment when the proper `Task` object is serialized (from `taskBytes`) using the earlier-created `Closure.Serializer` object. The local properties (as `localProperties`) are initialized to be the task's properties (from the earlier call to

`Task.deserializeWithDependencies`) and the `TaskMemoryManager` (created earlier in the method) is set to the task.

Note	The task's properties were part of the serialized object passed on to the current <code>TaskRunner</code> object.
------	---

Note	Until <code>run</code> deserializes the task object, it is only available as the <code>serializedTask</code> byte buffer.
------	---

If `kill` method has been called in the meantime, the execution stops by throwing a `TaskKilledException`. Otherwise, `TaskRunner` continues executing the task.

You should see the following DEBUG message in the logs:

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

TaskRunner sends update of the epoch of the task to `MapOutputTracker`.

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

The `taskStart` time which corresponds to the current time is recorded.

The `task runs` (with `taskId`, `attemptNumber`, and the globally-configured `MetricsSystem`). It runs inside a "monitored" block (i.e. `try-finally` block) to clean up after the task's run finishes regardless of the final outcome - the task's value or an exception thrown.

After the task's run finishes (and regardless of an exception thrown or not), `run` always calls `BlockManager.releaseAllLocksForTask` (with the current task's `taskId`).

`run` then always [queries TaskMemoryManager for memory leaks](#). If there is any (i.e. the memory freed after the call is greater than 0) and `spark.unsafe.exceptionOnMemoryLeak` is enabled (it is not by default) with no exception having been thrown while the task was running, a `SparkException` is thrown:

```
Managed memory leak detected; size = [freedMemory] bytes, TID = [taskId]
```

Otherwise, if `spark.unsafe.exceptionOnMemoryLeak` is disabled or an exception was thrown by the task, the following ERROR message is displayed in the logs instead:

```
ERROR Executor: Managed memory leak detected; size = [freedMemory] bytes, TID = [taskId]
```

#### Note

If there is a memory leak detected, it leads to a `SparkException` or ERROR message in the logs.

If there are any `releasedLocks` (after calling `BlockManager.releaseAllLocksForTask` earlier) and `spark.storage.exceptionOnPinLeak` is enabled (it is not by default) with no exception having been thrown while the task was running, a `SparkException` is thrown:

```
[releasedLocks] block locks were not released by TID = [taskId]:  
[releasedLocks separated by comma]
```

Otherwise, if `spark.storage.exceptionOnPinLeak` is disabled or an exception was thrown by the task, the following WARN message is displayed in the logs instead:

```
WARN Executor: [releasedLocks] block locks were not released by TID = [taskId]:  
[releasedLocks separated by comma]
```

#### Note

If there are any `releaseLocks`, they lead to a `SparkException` or WARN message in the logs.

The `taskFinish` time which corresponds to the current time is recorded.

If the `task was killed` a `TaskKilledException` is thrown (and the `TaskRunner` exits).

Caution	<b>FIXME</b> Finish me!
---------	-------------------------

When a task finishes successfully, it returns a value. The value is serialized (using a new instance of `Serializer` from `SparkEnv`, i.e. `serializer` ).

Note	There are two <code>Serializer</code> objects in <code>SparkEnv</code> .
------	--

The time to serialize the task's value is tracked (using `beforeSerialization` and `afterSerialization` ).

The task's metrics are set, i.e. `executorDeserializeTime` , `executorRunTime` , `jvmGCTime` , and `resultSerializationTime` .

Caution	<b>FIXME</b> Describe the metrics in more details. And include a figure to show the metric points.
---------	--

`run` collects the latest values of accumulators (as `accumUpdates` ).

A `DirectTaskResult` object with the serialized result and the latest values of accumulators is created (as `directResult` ). The `DirectTaskResult` object is serialized (using the global closure `Serializer`).

The limit of the buffer for the serialized `DirectTaskResult` object is calculated (as `resultSize` ).

The `serializedResult` is calculated (that soon will be sent to `ExecutorBackend`). It depends on the size of `resultSize` .

If `maxResultSize` is set and the size of the serialized `DirectTaskResult` exceeds it, the following WARN message is displayed in the logs:

```
WARN Executor: Finished [taskName] (TID [taskId]). Result is larger than maxResultSize ([resultSize] > [maxResultSize]), dropping it.
```

Caution	<b>FIXME</b> Describe <code>maxResultSize</code> .
---------	--

```
$ ./bin/spark-shell -c spark.driver.maxResultSize=1m

scala> sc.version
res0: String = 2.0.0-SNAPSHOT

scala> sc.getConf.get("spark.driver.maxResultSize")
res1: String = 1m

scala> sc.range(0, 1024 * 1024 + 10, 1).collect
WARN Executor: Finished task 4.0 in stage 0.0 (TID 4). Result is larger than maxResult
Size (1031.4 KB > 1024.0 KB), dropping it.
...
ERROR TaskSetManager: Total size of serialized results of 1 tasks (1031.4 KB) is bigge
r than spark.driver.maxResultSize (1024.0 KB)
...
org.apache.spark.SparkException: Job aborted due to stage failure: Total size of seria
lized results of 1 tasks (1031.4 KB) is bigger than spark.driver.maxResultSize (1024.0
KB)
    at org.apache.spark.scheduler.DAGScheduler.org$apache$spark$scheduler$DAGScheduler$$
failJobAndIndependentStages(DAGScheduler.scala:1448)
...
...
```

The final `serializedResult` becomes a serialized `IndirectTaskResult` with a `TaskResultBlockId` for the task's `taskId` and `resultSize`.

Otherwise, when `maxResultSize` is not positive or `resultSize` is smaller than `maxResultSize`, but greater than `maxDirectResultSize`, a `TaskResultBlockId` object for the task's `taskId` is created (as `blockId`) and `serializedDirectResult` is stored as a `blockId` block to `BlockManager` with `MEMORY_AND_DISK_SER` storage level.

Caution	<a href="#">FIXME</a> Describe <code>maxDirectResultSize</code> .
---------	---

The following INFO message is printed out to the logs:

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

The final `serializedResult` becomes a serialized `IndirectTaskResult` with a `TaskResultBlockId` for the task's `taskId` and `resultSize`.

Note	The difference between the two cases is that the result is dropped or sent via BlockManager.
------	--

When the two cases above do not hold, the following INFO message is printed out to the logs:

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

The final `serializedResult` becomes the `serializedDirectResult` (that is the `SerializedDirectTaskResult`).

Note	The final <code>serializedResult</code> is either a <code>IndirectTaskResult</code> (with or without <code>BlockManager</code> used) or a <code>DirectTaskResult</code> .
------	---

The `serializedResult` serialized result for the task is sent to the driver using `ExecutorBackend` as `TaskState.FINISHED`.

Caution	<code>FIXME</code> Complete <code>catch</code> block.
---------	---

When the `TaskRunner` finishes, `taskId` is removed from the internal `runningTasks` map of the owning `Executor` (that ultimately cleans up any references to the `TaskRunner`).

Note	<code>TaskRunner</code> is Java's <code>Runnable</code> and the contract requires that once a <code>TaskRunner</code> has completed execution it may not be restarted.
------	--

## Killing Task (kill method)

```
kill(interruptThread: Boolean): Unit
```

`kill` marks the current instance of `TaskRunner` as killed and passes the call to kill a task on to the task itself (if available).

When executed, you should see the following INFO message in the logs:

```
INFO TaskRunner: Executor is trying to kill [taskName] (TID [taskId])
```

Internally, `kill` enables the internal flag `killed` and executes its `Task.kill` method if a task is available.

Note	The internal flag <code>killed</code> is checked in <code>run</code> to stop executing the task. Calling <code>Task.kill</code> method allows for task interruptions later on.
------	--

## Settings

- `spark.unsafe.exceptionOnMemoryLeak` (default: `false`)



# Spark Services

# MemoryManager — Memory Management

`MemoryManager` is an abstract base **memory manager** to manage shared memory for execution and storage.

**Execution memory** is used for computation in shuffles, joins, sorts and aggregations.

**Storage memory** is used for caching and propagating internal data across the nodes in a cluster.

A `MemoryManager` is created when [SparkEnv is created](#) (one per JVM) and can be one of the two possible implementations:

- [UnifiedMemoryManager](#) — the default memory manager since Spark 1.6.
- [StaticMemoryManager](#) (legacy)

Note	<code>org.apache.spark.memory.MemoryManager</code> is a <code>private[spark]</code> Scala trait in Spark.
------	---

## MemoryManager Contract

Every `MemoryManager` obeys the following contract:

- [maxOnHeapStorageMemory](#)
- [acquireStorageMemory](#)

## acquireStorageMemory

```
acquireStorageMemory(blockId: BlockId, numBytes: Long, memoryMode: MemoryMode): Boolean
```

```
acquireStorageMemory
```

Caution	<a href="#">FIXME</a>
---------	-----------------------

`acquireStorageMemory` is used in [MemoryStore](#) to put bytes.

## maxOnHeapStorageMemory

```
maxOnHeapStorageMemory: Long
```

`maxOnHeapStorageMemory` is the total amount of memory available for storage, in bytes. It can vary over time.

Caution	<a href="#">FIXME</a> Where is this used?
---------	---

It is used in [MemoryStore](#) to ??? and [BlockManager](#) to ???

## releaseExecutionMemory

## releaseAllExecutionMemoryForTask

## tungstenMemoryMode

`tungstenMemoryMode` informs others whether Spark works in `OFF_HEAP` or `ON_HEAP` memory mode.

It uses `spark.memory.offHeap.enabled` (default: `false`), `spark.memory.offHeap.size` (default: `0`), and `org.apache.spark.unsafe.Platform.unaligned` before `OFF_HEAP` is assumed.

Caution	<a href="#">FIXME</a> Describe <code>org.apache.spark.unsafe.Platform.unaligned</code> .
---------	--

# UnifiedMemoryManager

Caution	<a href="#">FIXME</a>
---------	-----------------------

`UnifiedMemoryManager` is the default [MemoryManager](#) since Spark 1.6 with `onHeapStorageMemory` being ??? and `onHeapExecutionMemory` being ???

Note	The other now legacy (before Spark 1.6) <code>MemoryManager</code> is <code>StaticMemoryManager</code> .
------	--

## acquireStorageMemory method

Note	<code>acquireStorageMemory</code> is a part of the <a href="#">MemoryManager Contract</a> .
------	---

`acquireStorageMemory` has two modes of operation per `memoryMode`, i.e. `MemoryMode.ON_HEAP` or `MemoryMode.OFF_HEAP`, for execution and storage pools, and the maximum amount of memory to use.

Caution	<a href="#">FIXME</a> Where are they used?
---------	--

In `MemoryMode.ON_HEAP`, `onHeapExecutionMemoryPool`, `onHeapStorageMemoryPool`, and [maxOnHeapStorageMemory](#) are used.

In `MemoryMode.OFF_HEAP`, `offHeapExecutionMemoryPool`, `offHeapStorageMemoryPool`, and [maxOffHeapMemory](#) are used.

Caution	<a href="#">FIXME</a> What is the difference between them?
---------	--

It makes sure that the requested number of bytes `numBytes` (for a block to store) fits the available memory. If it is not the case, you should see the following INFO message in the logs and the method returns `false`.

```
INFO Will not store [blockId] as the required space ([numBytes] bytes) exceeds our memory limit ([maxMemory] bytes)
```

If the requested number of bytes `numBytes` is greater than `memoryFree` in the storage pool, `acquireStorageMemory` will attempt to use the free memory from the execution pool.

Note	The storage pool can use the free memory from the execution pool.
------	---

It will take as much memory as required to fit `numBytes` from `memoryFree` in the execution pool (up to the whole free memory in the pool).

Ultimately, `acquireStorageMemory` requests the storage pool for `numBytes` for `blockId`.

## acquireUnrollMemory method

**Note**

`acquireUnrollMemory` is a part of the [MemoryManager Contract](#).

`acquireUnrollMemory` simply passes calls to [acquireStorageMemory](#).

## maxOnHeapStorageMemory method

**Note**

`maxOnHeapStorageMemory` is a part of the [MemoryManager Contract](#).

## Creating UnifiedMemoryManager Instance

```
private[spark] class UnifiedMemoryManager private[memory] (
  conf: SparkConf,
  val maxHeapMemory: Long,
  onHeapStorageRegionSize: Long,
  numCores: Int)
```

When an instance of `UnifiedMemoryManager` is created, it requires a [SparkConf](#) with the following numbers:

- `maxHeapMemory`
- `onHeapStorageRegionSize`
- `numCores`

It makes sure that the sum of `onHeapExecutionMemoryPool` and `onHeapStorageMemoryPool` pool sizes is exactly the constructor's `maxHeapMemory`.

It also makes sure that the sum of `offHeapExecutionMemoryPool` and `offHeapStorageMemoryPool` pool sizes is exactly `maxOffHeapMemory`.

**Caution**

[\*\*FIXME\*\*](#) Describe the pools

# SparkEnv - Spark Runtime Environment

**Spark Runtime Environment** (`SparkEnv`) is the runtime environment with Spark services that interact with each other to build the entire 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.

**Tip** Enable `INFO` or `DEBUG` logging level for `org.apache.spark.SparkEnv` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.SparkEnv=DEBUG
```

Refer to [Logging](#).

## SparkEnv

**SparkEnv** holds all runtime objects for a running Spark instance, using `SparkEnv.createDriverEnv()` for a driver and `SparkEnv.createExecutorEnv()` for an executor.

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
```

## Creating "Base" SparkEnv (create method)

```
create(  
  conf: SparkConf,  
  executorId: String,  
  hostname: String,  
  port: Int,  
  isDriver: Boolean,  
  isLocal: Boolean,  
  numUsableCores: Int,  
  listenerBus: LiveListenerBus = null,  
  mockOutputCommitCoordinator: Option[OutputCommitCoordinator] = None): SparkEnv
```

`create` is a internal helper method to create a "base" `SparkEnv` regardless of the target environment — be it a driver or an executor.

When executed, `create` creates a `Serializer` (based on `spark.serializer`). You should see the following `DEBUG` message in 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 (with `UnifiedMemoryManager` being the default).

It creates a `NettyBlockTransferService`.

It creates a `BlockManagerMaster` object with the `BlockManagerMaster` RPC endpoint reference (by registering or looking it up by name and `BlockManagerMasterEndpoint`), the input `SparkConf`, and the input `isDriver` flag.

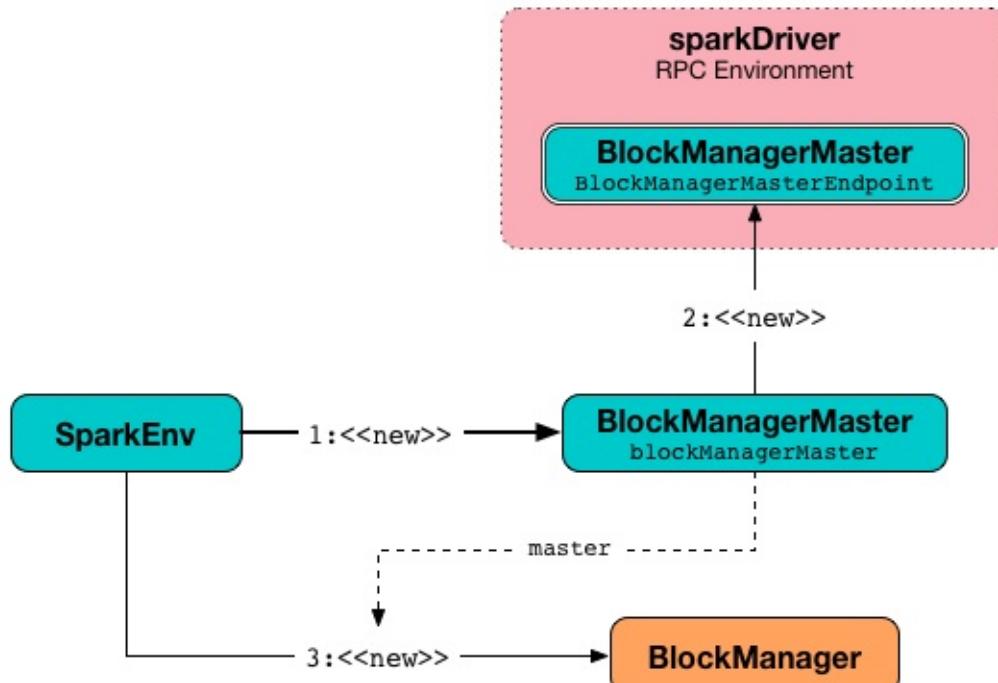


Figure 1. Creating BlockManager for the Driver

Note

`create` registers the `BlockManagerMaster` RPC endpoint for the driver and looks it up for executors.

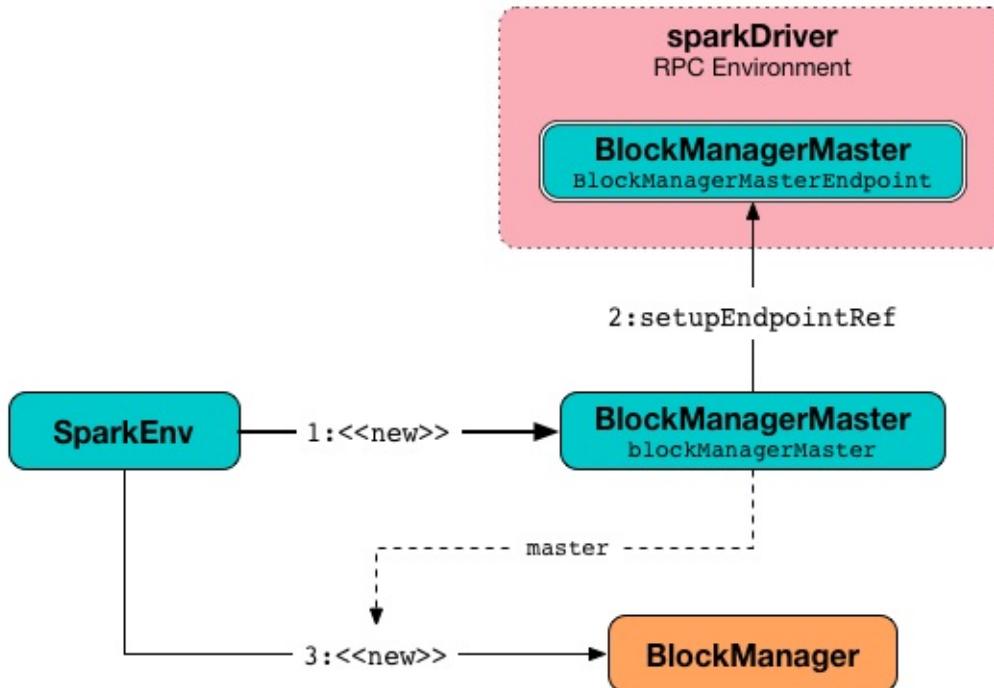


Figure 2. Creating BlockManager for Executor

It creates a `BroadcastManager` (using the above `BlockManagerMaster` object and other services).

It creates a `BroadcastManager`.

It creates a `CacheManager`.

It creates a `MetricsSystem` for a driver and a worker separately.

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.

Note	<code>create</code> is called by <code>createDriverEnv</code> and <code>createExecutorEnv</code> .
------	--

## Registering or Looking up RPC Endpoint by Name (`registerOrLookupEndpoint` method)

```
registerOrLookupEndpoint(name: String, endpointCreator: => RpcEndpoint)
```

`registerOrLookupEndpoint` registers or looks up a RPC endpoint by `name`.

If called from the driver, you should see the following INFO message in the logs:

```
INFO SparkEnv: Registering [name]
```

And the RPC endpoint is registered in the RPC environment.

Otherwise, it obtains a RPC endpoint reference by `name`.

## Creating SparkEnv for Driver (createDriverEnv method)

```
createDriverEnv(  
    conf: SparkConf,  
    isLocal: Boolean,  
    listenerBus: LiveListenerBus,  
    numCores: Int,  
    mockOutputCommitCoordinator: Option[OutputCommitCoordinator] = None): SparkEnv
```

`createDriverEnv` creates a `SparkEnv` execution environment for the driver.

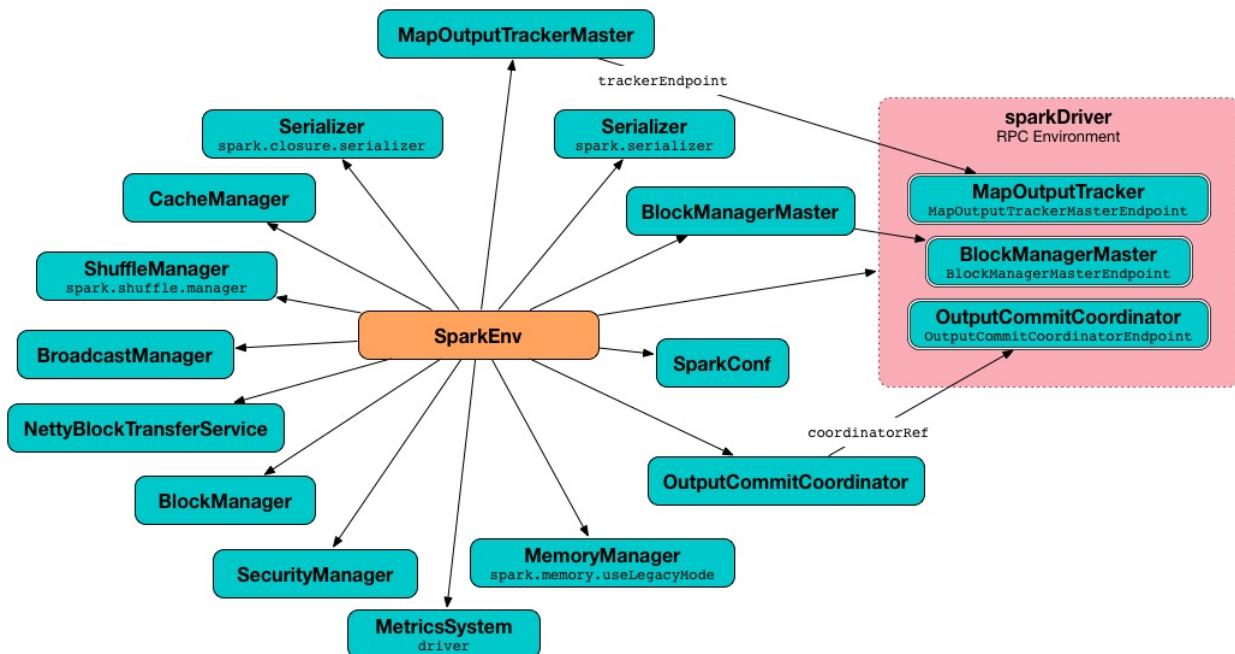


Figure 3. Spark Environment for driver

The method accepts an instance of [SparkConf](#), whether it runs in local mode or not, [LiveListenerBus](#), the number of driver's cores to use for execution in local mode or `0` otherwise, and a [OutputCommitCoordinator](#) (default: none).

`createDriverEnv` ensures that `spark.driver.host` and `spark.driver.port` settings are set in `conf` [SparkConf](#).

It then passes the call straight on to the [create helper method](#) (with `driver` executor id, `isDriver` enabled, and the input parameters).

Note

`createDriverEnv` is exclusively used by [SparkContext](#) to create a `sparkEnv` (while a [SparkContext](#) is being created for the driver).

## Creating SparkEnv for Executor (`createExecutorEnv` method)

`SparkEnv.createExecutorEnv` creates an **executor's (execution) environment** that is the Spark execution environment for an executor.

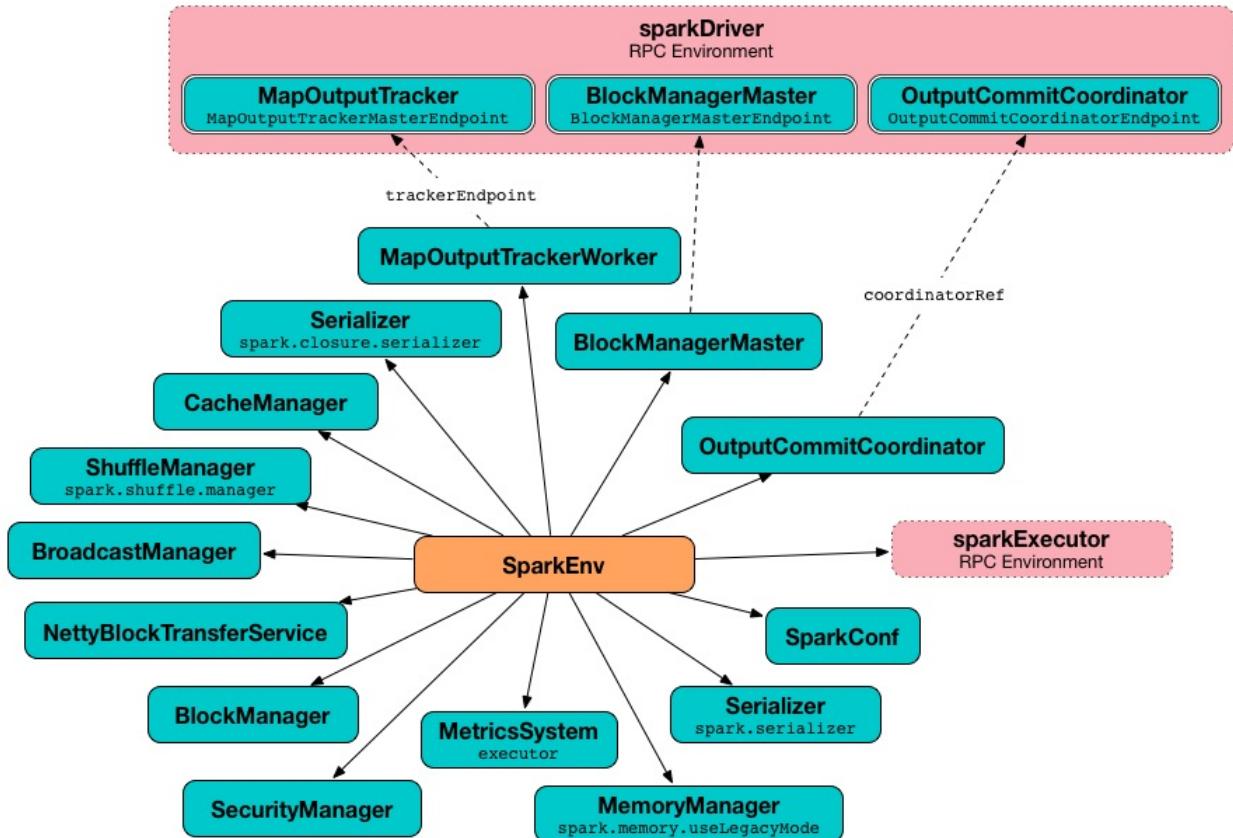


Figure 4. Spark Environment for executor

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.

## serializer

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

## closureSerializer

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

## Settings

### spark.driver.host

`spark.driver.host` is the name of the machine where the driver runs. It is set when [SparkContext is created](#).

### spark.driver.port

`spark.driver.port` is the port the driver listens to. It is first set to `0` in the driver when [SparkContext is initialized](#). It is later set to the port of [RpcEnv](#) of the driver (in [SparkEnv.create](#)).

### spark.serializer

`spark.serializer` (default: `org.apache.spark.serializer.JavaSerializer`) - the Serializer.

### spark.closure.serializer

`spark.closure.serializer` (default: `org.apache.spark.serializer.JavaSerializer`) is the Serializer.

### spark.shuffle.manager

`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

`spark.memory.useLegacyMode` (default: `false`) controls the [MemoryManager](#) in use. It is `StaticMemoryManager` when enabled (`true`) or [UnifiedMemoryManager](#) when disabled (`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](#)).

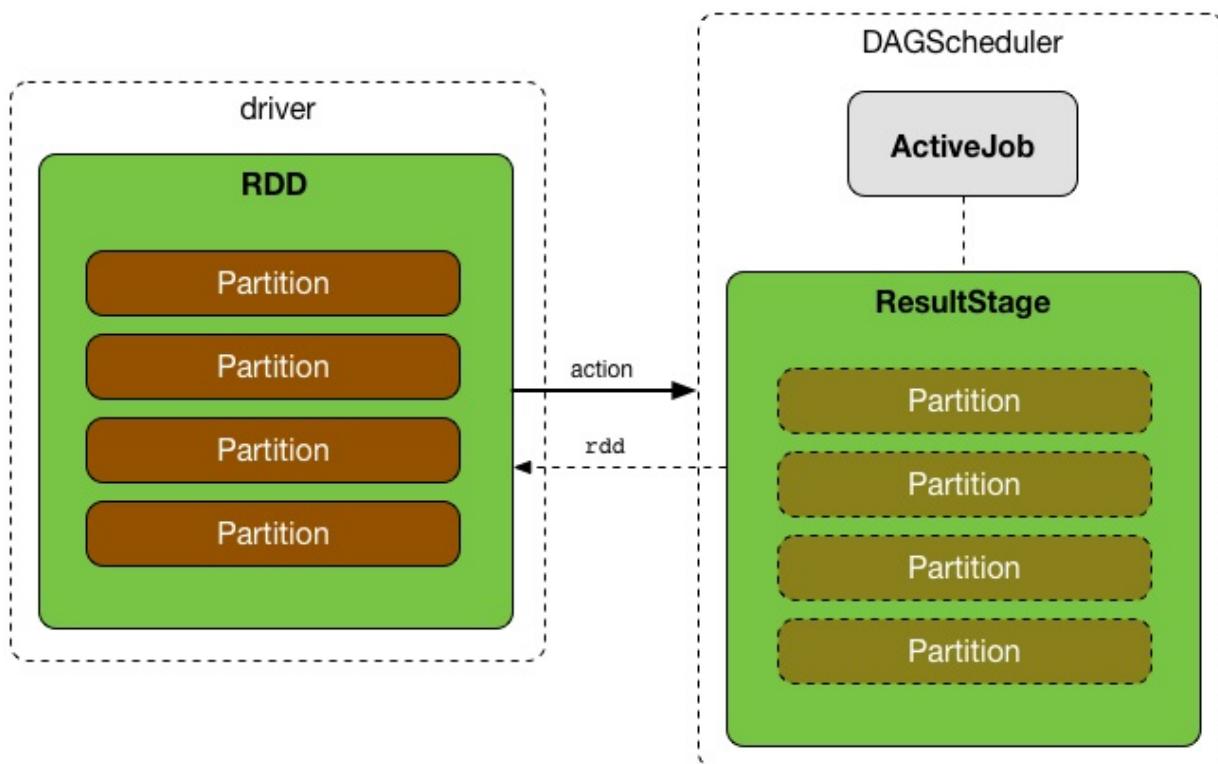


Figure 1. Executing action leads to new ResultStage and ActiveJob in DAGScheduler  
The fundamental concepts of DAGScheduler are **jobs** and **stages** (refer to [Jobs](#) and [Stages](#) respectively).

DAGScheduler works on a driver. It is created as part of [SparkContext's initialization](#), right after [TaskScheduler](#) and [SchedulerBackend](#) are ready.

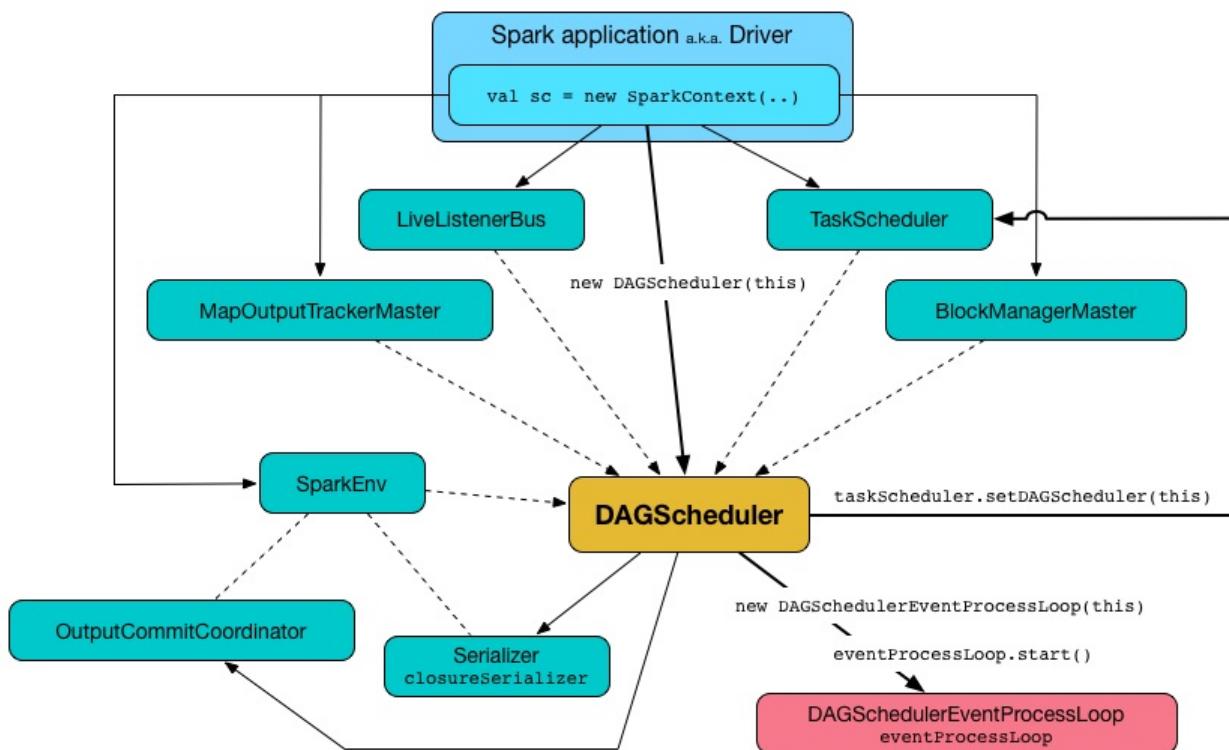


Figure 2. DAGScheduler as created by SparkContext with other services  
 DAGScheduler does three things in Spark (thorough explanations follow):

- 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** Enable `DEBUG` or `TRACE` logging level for `org.apache.spark.scheduler.DAGScheduler` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.scheduler.DAGScheduler=TRACE
```

Refer to [Logging](#).

DAGScheduler needs [SparkContext](#), [Task Scheduler](#), [LiveListenerBus](#), [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.

## Creating DAGScheduler Instance

**Caution**

[FIXME](#)

## 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.

<b>Caution</b>	<b><a href="#">FIXME</a></b> Review <ul style="list-style-type: none"> <li>• <code>cleanupStateForJobAndIndependentStages</code></li> </ul>
----------------	---

## 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`.

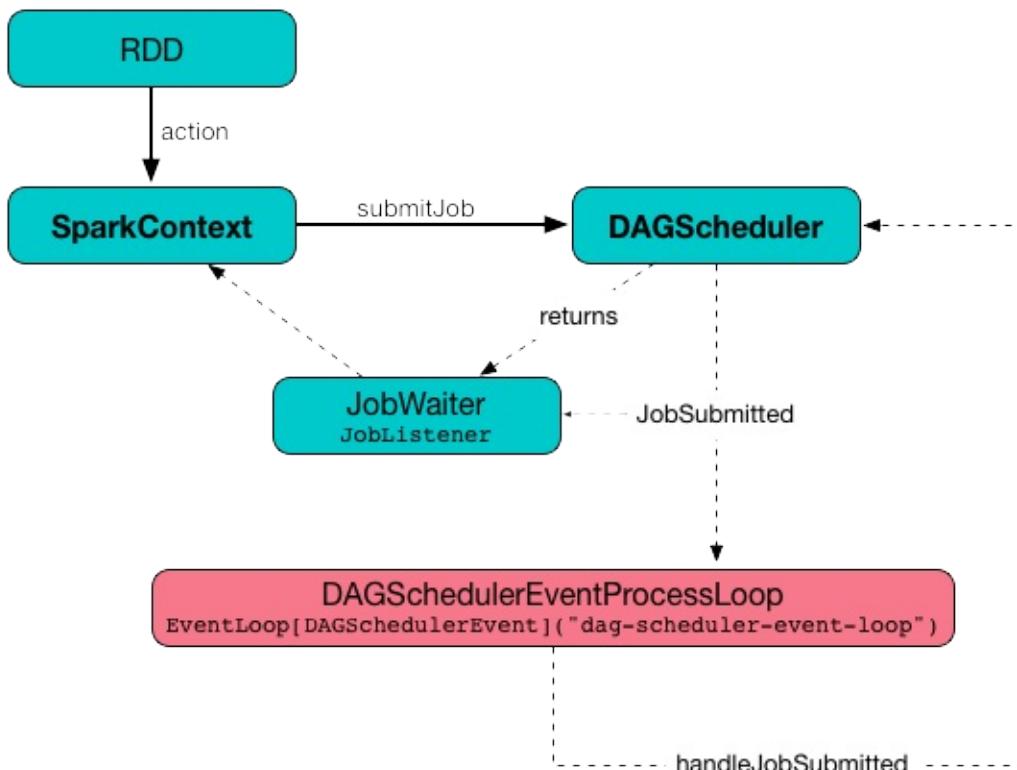


Figure 3. `DAGScheduler.submitJob`

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): Unit
```

`taskEnded` method simply posts a [CompletionEvent](#) event to the [DAGScheduler](#)'s internal [event loop](#).

Note	<code>DAGScheduler.taskEnded</code> method is called by a <a href="#">TaskSetManager</a> to report task completions, failures including.
------	--

Tip	Read about <code>TaskMetrics</code> in <a href="#">TaskMetrics</a> .
-----	--

## failJobAndIndependentStages

The internal `failJobAndIndependentStages` method...[FIXME](#)

Note	It is called by... <a href="#">FIXME</a>
------	--

## 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 waiting on condition [0x00000000125826000]
```

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`

	<b>FIXME</b>
Caution	<ul style="list-style-type: none"> <li>• What is an approximate job (as in <code>DAGScheduler.runApproximateJob</code>)?</li> <li>• <code>statistics?</code> <code>MapOutputStatistics</code> ?</li> </ul>

## 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 <code>SparkContext.cancelJob(jobId: Int)</code> calls <code>DAGScheduler.cancelJob</code> , no feature/code in Spark calls <code>SparkContext.cancelJob(jobId: Int)</code> . 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	<b>FIXME</b>
---------	--------------

## 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	<p><code>handleExecutorLost</code> recognizes two cases (by means of <code>fetchFailed</code>):</p> <ul style="list-style-type: none"> <li>• fetch failures (<code>fetchFailed</code> is <code>true</code>) from executors that are indirectly assumed lost. See &lt;&gt;<code>handleTaskCompletion-FetchFailed</code>, <code>FetchFailed</code> case in <code>handleTaskCompletion</code>&gt;.</li> <li>• lost executors (<code>fetchFailed</code> is <code>false</code>) for executors that did not report being alive in a given timeframe</li> </ul>
------	--

The current epoch number could be provided (as `maybeEpoch`) or it is calculated by requesting it from [MapOutputTrackerMaster](#) (using [MapOutputTrackerMaster.getEpoch](#)).

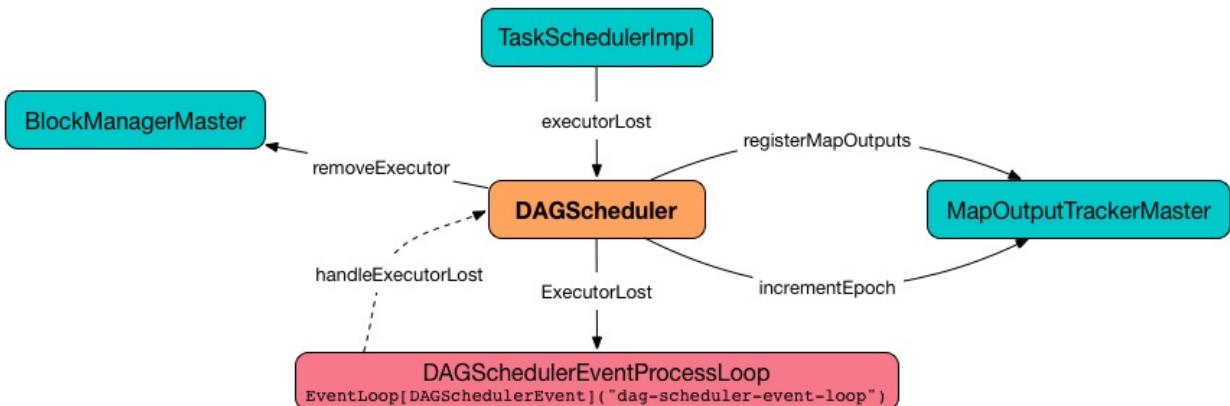


Figure 4. `DAGScheduler.handleExecutorLost`

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])
```

The executor `execId` is removed (from [BlockManagerMaster](#) on the driver).

If the [external shuffle service](#) is not used or the `ExecutorLost` event was for a map output fetch operation, all [ShuffleMapStage](#) (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	<a href="#">FIXME</a> 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 <code>jobIds</code> .
------	--

`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 the internal `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	<a href="#">FIXME</a> 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	<code>FIXME cleanupStateForJobAndIndependentStages</code> code review.
---------	--

Caution	<code>FIXME Where are job.properties assigned to a job?</code>
---------	--

```
"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.

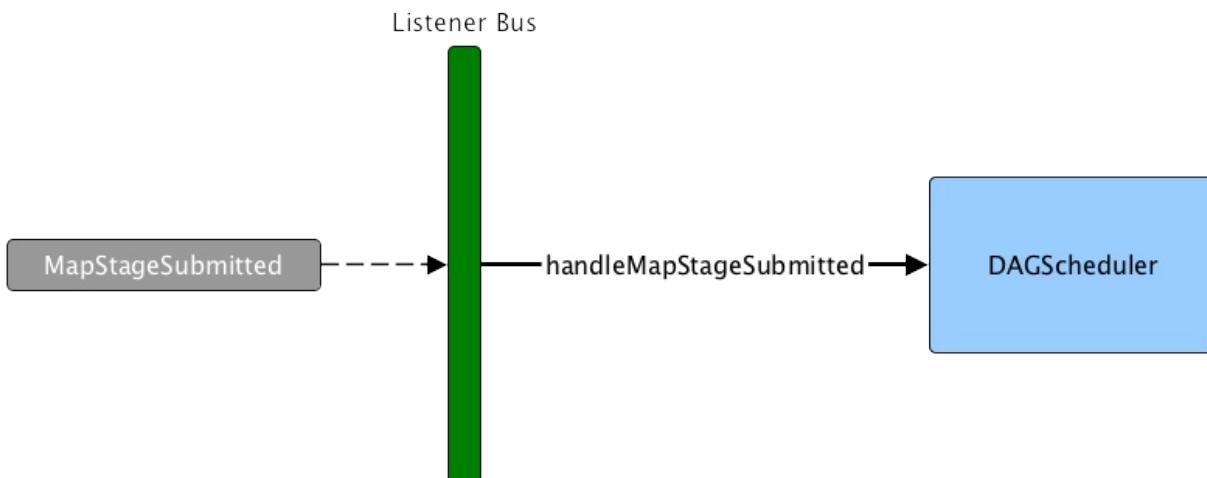


Figure 5. `DAGScheduler.handleMapStageSubmitted` handles `MapStageSubmitted` events. It is called with a job id (for a new job to be created), a `ShuffleDependency`, and a `JobListener`.

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]
```

[SparkListenerJobStart](#) event is posted to [LiveListenerBus](#) (so other event listeners know about the event - not only DAGScheduler).

The execution procedure of MapStageSubmitted events is then exactly ([FIXME](#) ?) as for [JobSubmitted](#).

	<p>The difference between <code>handleMapStageSubmitted</code> and <code>handleJobSubmitted</code>:</p> <ul style="list-style-type: none"> <li>• <code>handleMapStageSubmitted</code> has <code>shuffleDependency</code> among the input parameters while <code>handleJobSubmitted</code> has <code>finalRDD</code>, <code>func</code>, and <code>partitions</code>.</li> <li>• <code>handleMapStageSubmitted</code> initializes <code>finalStage</code> as <code>getShuffleMapStage(dependency, jobId)</code> while <code>handleJobSubmitted</code> as <code>finalStage = newResultStage(finalRDD, func, partitions, jobId, callSite)</code></li> <li>• <code>handleMapStageSubmitted</code> INFO logs <code>Got map stage job %s (%s) with %d output partitions with dependency.rdd.partitions.length</code> while <code>handleJobSubmitted</code> does <code>Got job %s (%s) with %d output partitions with partitions.length</code>.</li> </ul>
Tip	<ul style="list-style-type: none"> <li>• <a href="#">FIXME</a>: Could the above be cut to <code>ActiveJob.numPartitions</code> ?</li> <li>• <code>handleMapStageSubmitted</code> adds a new job with <code>finalStage.addActiveJob(job)</code> while <code>handleJobSubmitted</code> sets with <code>finalStage.setActiveJob(job)</code> .</li> <li>• <code>handleMapStageSubmitted</code> checks if the final stage has already finished, tells the listener and removes it using the code:</li> </ul> <pre style="background-color: #f0f0f0; padding: 10px;"><code>if (finalStage.isAvailable) {     markMapStageJobAsFinished(job, mapOutputTracker.getStatistics(dependency)) }</code></pre>

## JobSubmitted and handleJobSubmitted

When DAGScheduler receives **JobSubmitted** event it calls

`DAGScheduler.handleJobSubmitted` method.

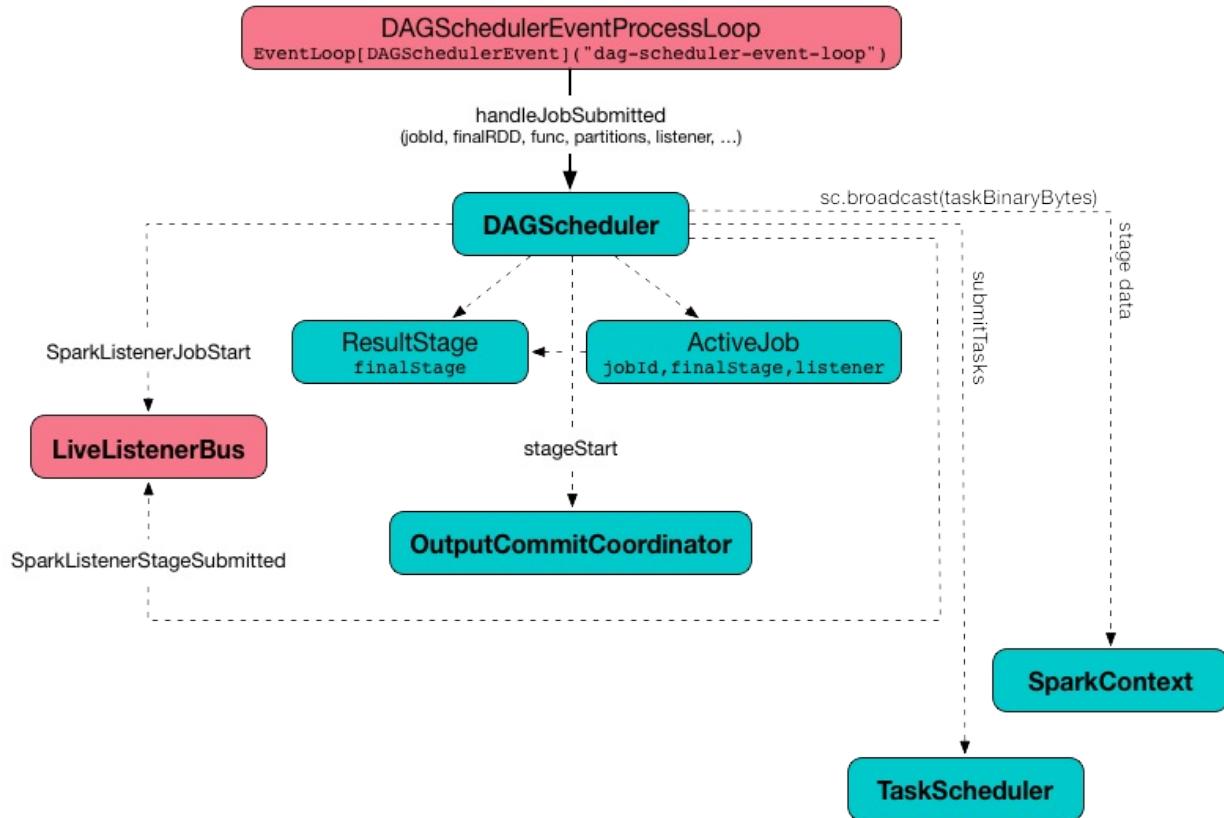


Figure 6. `DAGScheduler.handleJobSubmitted`

`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	<a href="#">FIXME</a> 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 partitions
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](#) message is posted to [LiveListenerBus](#).

Caution	<a href="#">FIXME</a> <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

`CompletionEvent` event informs DAGScheduler about task completions. It is handled by `handleTaskCompletion(event: CompletionEvent)`.

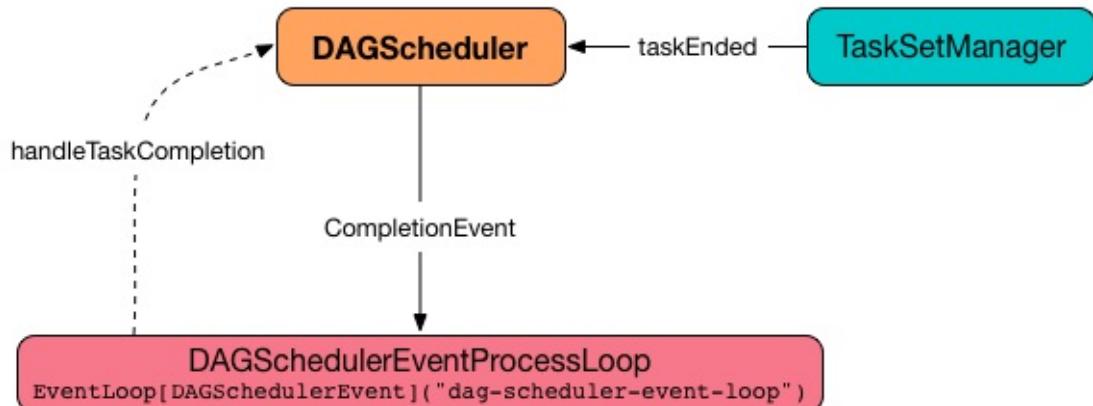


Figure 7. DAGScheduler and CompletionEvent

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 to `LiveListenerBus`. 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 to `LiveListenerBus`.

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	<b><code>FIXME</code></b> 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 to `LiveListenerBus` 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	<b><code>FIXME</code></b> 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 failed: [missingPartitions]
```

`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.stageAttemptId] and there is a more recent attempt for that stage (attempt ID [failedStage.latestInfo.attemptId]) running
```

**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 [mapStage] ([mapStage.name])
```

`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 running
```

When `disallowStageRetryForTest` is set, `abortStage(failedStage, "Fetch failure will not retry stage due to testing config", None)` is called.

Caution	<a href="#">FIXME</a> 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 allowable number of times: ${Stage.MAX_CONSECUTIVE_FETCH_FAILURES}. Most recent failure reason: ${failureMessage}", None)
```

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 to fetch failure
```

And the following code is executed:

```
messageScheduler.schedule(
  new Runnable {
    override def run(): Unit = eventProcessLoop.post(ResubmitFailedStages)
  }, DAGScheduler.RESUBMIT_TIMEOUT, TimeUnit.MILLISECONDS)
```

Caution	<a href="#">FIXME</a> 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	<a href="#">FIXME</a> 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	<a href="#">FIXME</a> 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	<a href="#">FIXME</a> Review <code>getShuffleMapStage</code>
---------	--

Caution	<a href="#">FIXME</a> ...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 spark-BlockManagerMaster.adoc[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 (`submitMissingTasks` method)

```
submitMissingTasks(stage: Stage, jobId: Int): Unit
```

`submitMissingTasks` is a private method that...[FIXME](#)

When executed, it prints the following DEBUG message out to the logs:

```
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	<a href="#">FIXME</a> Review <code>outputCommitCoordinator.stageStart</code>
---------	--

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.

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	<a href="#">FIXME</a> Review <code>taskBinaryBytes</code> .
---------	---

When serializing the stage fails, the stage is removed from the internal `runningStages` set, `abortStage` is called and the method stops.

Caution	<a href="#">FIXME</a> Review <code>abortStage</code> .
---------	--

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	<a href="#">FIXME</a> 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}, available outputs: ${stage.numAvailableOutputs}, partitions: ${stage.numPartitions})
```

For [ResultStage](#):

```
DEBUG DAGScheduler: Stage [stage] is actually done; (partitions: [numPartitions])
```

Note	<code>submitMissingTasks</code> is called when...
------	---

## getPreferredLocs - Computing Preferred Locations for Tasks and Partitions

Caution	<a href="#">FIXME</a> 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.

Caution	<a href="#">FIXME</a> What is <code>DAGSchedulerSource</code> ?
---------	---

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

## Updating Accumulators with Partial Values from Completed Tasks (`updateAccumulators` method)

	<code>updateAccumulators(event: CompletionEvent): Unit</code>
--	---

The private `updateAccumulators` method merges the partial values of accumulators from a completed task into their "source" accumulators on the driver.

Note	It is called by <a href="#">handleTaskCompletion</a> .
------	--

For each [AccumableInfo](#) in the `completionEvent`, a partial value from a task is obtained (from `AccumableInfo.update`) and added to the driver's accumulator (using `Accumable.++=` method).

For named accumulators with the update value being a non-zero value, i.e. not

`Accumable.zero`:

- `stage.latestInfo.accumulables` for the `AccumableInfo.id` is set
- `CompletionEvent.taskInfo.accumulables` has a new [AccumableInfo](#) added.

Caution	<a href="#">FIXME</a> Where are <code>Stage.latestInfo.accumulables</code> and <code>CompletionEvent.taskInfo.accumulables</code> used?
---------	---

## 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](#).

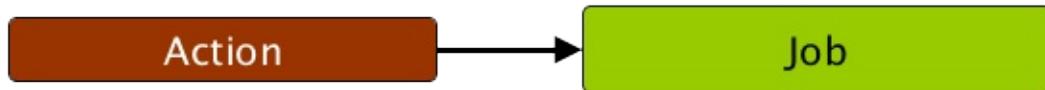


Figure 1. RDD actions submit jobs to DAGScheduler

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](#).

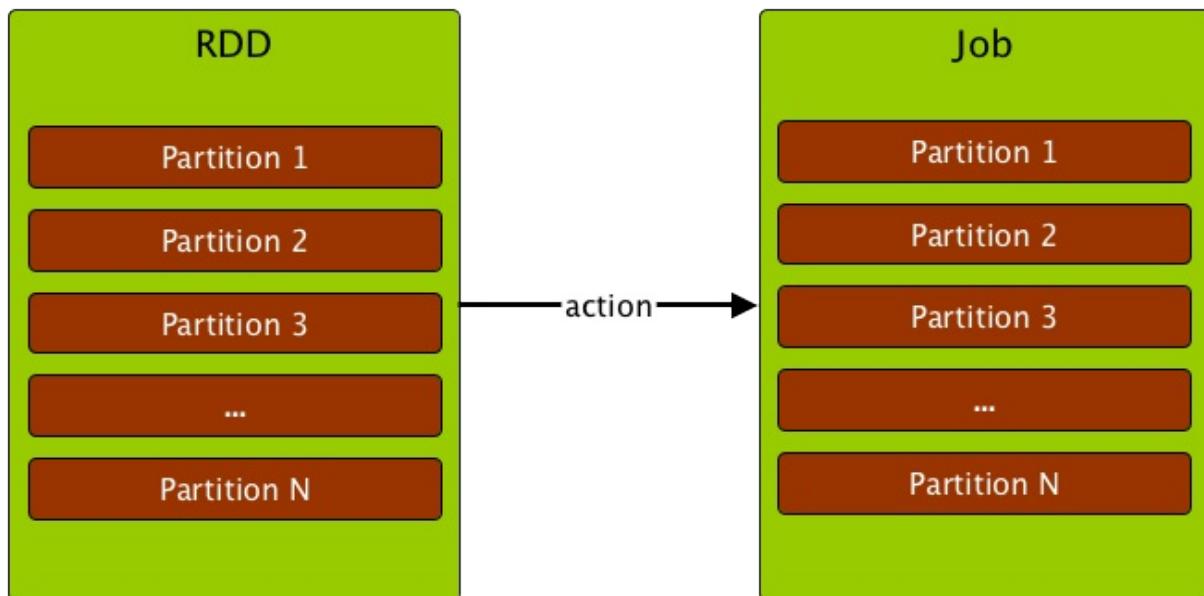


Figure 2. Computing a job is computing the partitions of an RDD

Note	Note that not all partitions have always to be computed for <a href="#">ResultStages</a> 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	<b>FIXME</b> <ul style="list-style-type: none"> <li>Where are instances of ActiveJob used?</li> </ul>
---------	---

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 physical unit of execution. It is a step in a physical execution plan.

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.

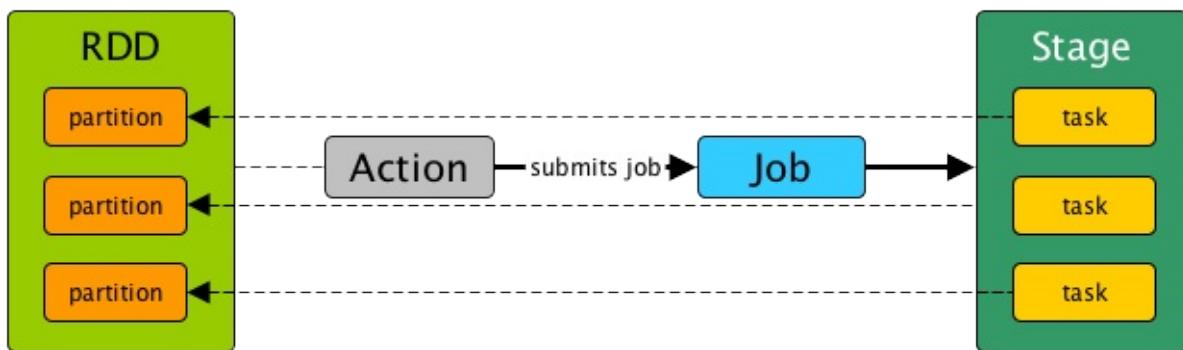


Figure 1. Stage, tasks and submitting a job

In other words, a Spark job is a computation with that computation sliced into stages.

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](#)).

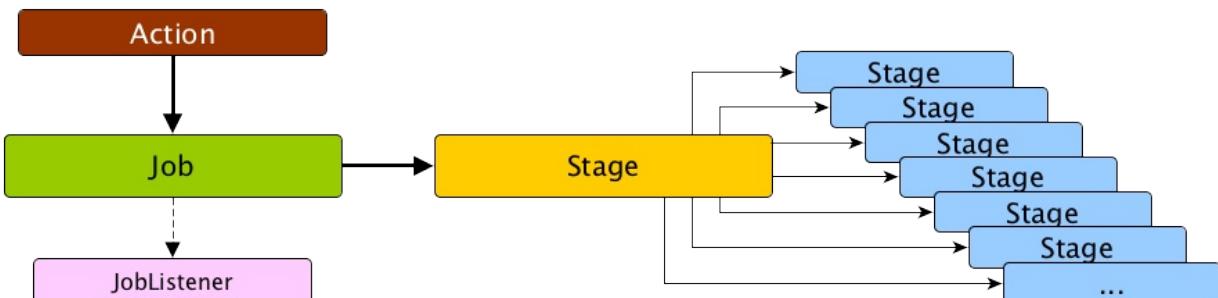


Figure 2. Submitting a job triggers execution of the stage and its parent stages  
Finally, every stage has a `firstJobId` that is the id of the job that submitted the stage.

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 **ShuffleMapStage** linked — they can be created from scratch or linked to, i.e. shared, if other jobs use them already.

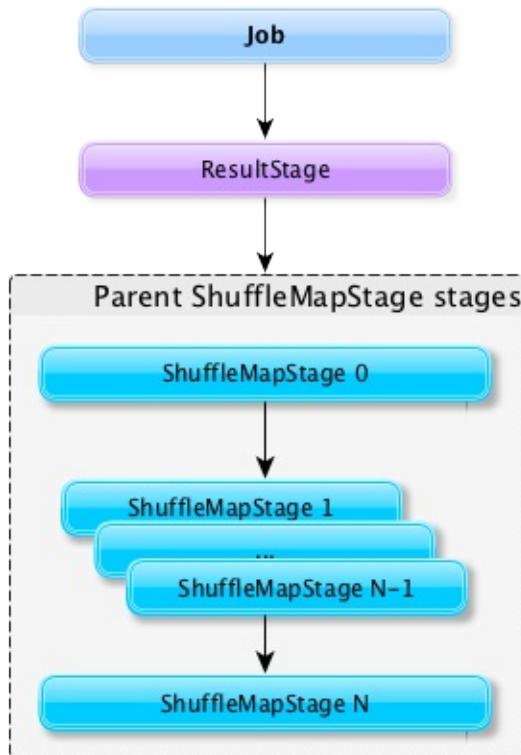


Figure 3. DAGScheduler and Stages for a job

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.

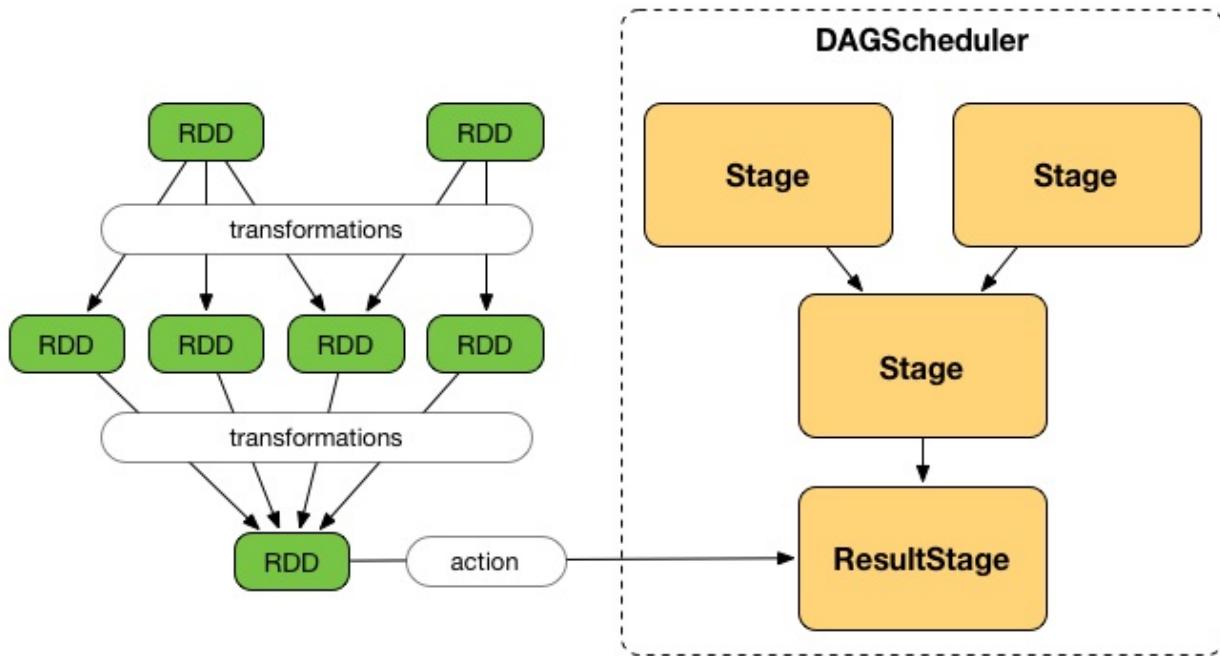


Figure 4. Graph of Stages

Shuffle boundaries introduce a barrier where stages/tasks must wait for the previous stage to finish before they fetch map outputs.

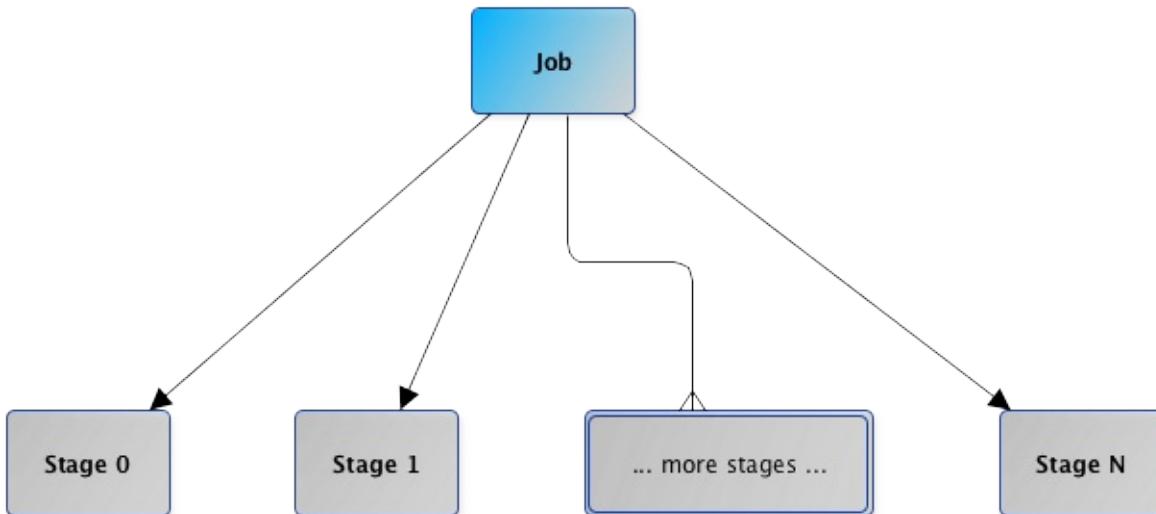


Figure 5. DAGScheduler splits a job into stages

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 reduceByKey at <console>:24)
```

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?

## 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).

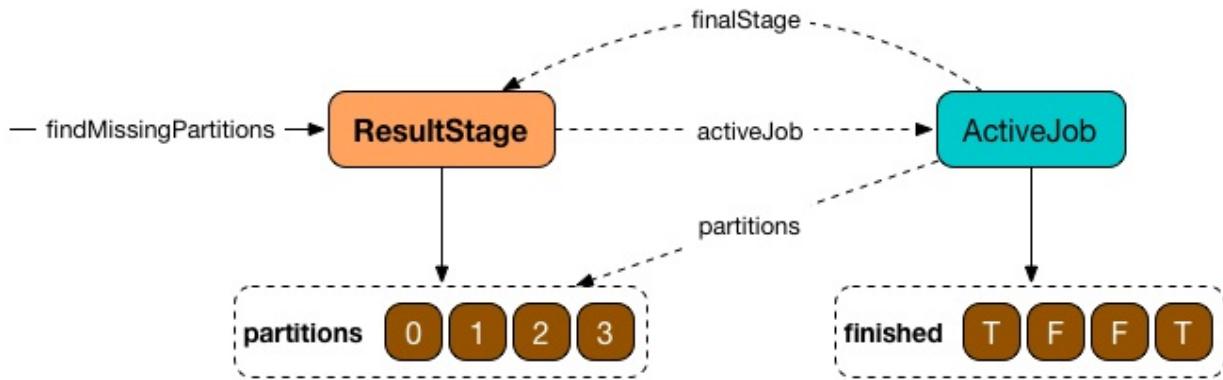


Figure 6. `ResultStage.findMissingPartitions` and `ActiveJob`

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.

# ShuffleMapStage — Intermediate Stage in Job

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](#)

<b>Caution</b>	<a href="#">FIXME</a> <ul style="list-style-type: none"> <li>• What's <code>ShuffleMapStage.outputLocs</code> and <code>MapStatus</code> ?</li> <li>• <code>newShuffleMapStage</code></li> </ul>
----------------	--

## 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

# Stages



## Details for Job 3

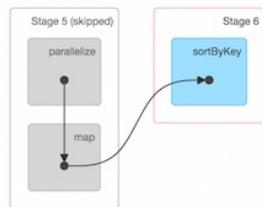
Status: SUCCEEDED

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 1. Skipped Stages are already-computed ShuffleMapStages

# ResultStage — Final Stage in Job

A `ResultStage` is the final stage in a job that applies a function on one or many partitions of the target RDD to compute the result of an action.

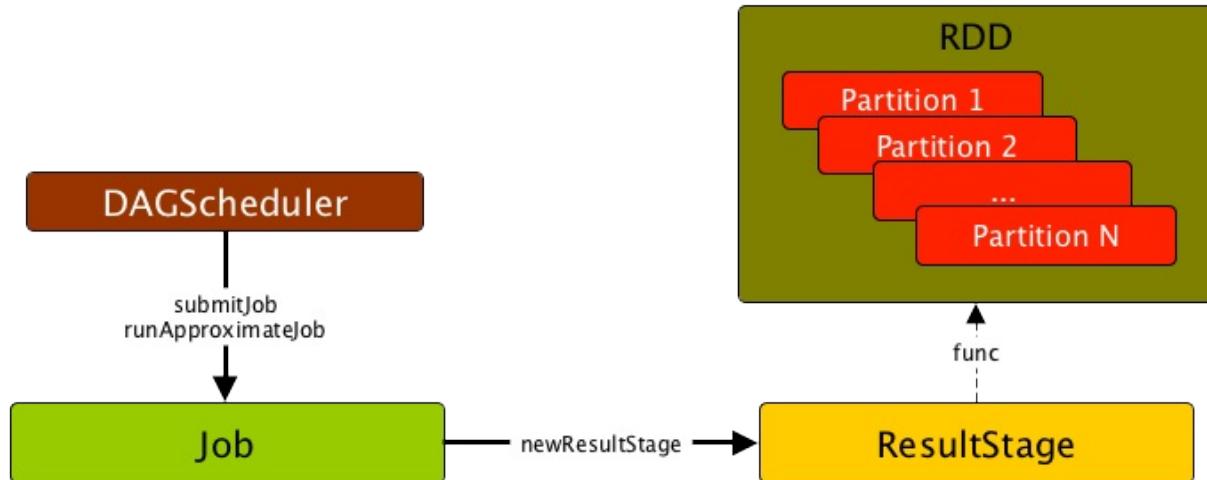


Figure 1. Job creates ResultStage as the first stage

The partitions are given as a collection of partition ids (`partitions`) and the function `func`:  
`(TaskContext, Iterator[_]) ⇒ _`.

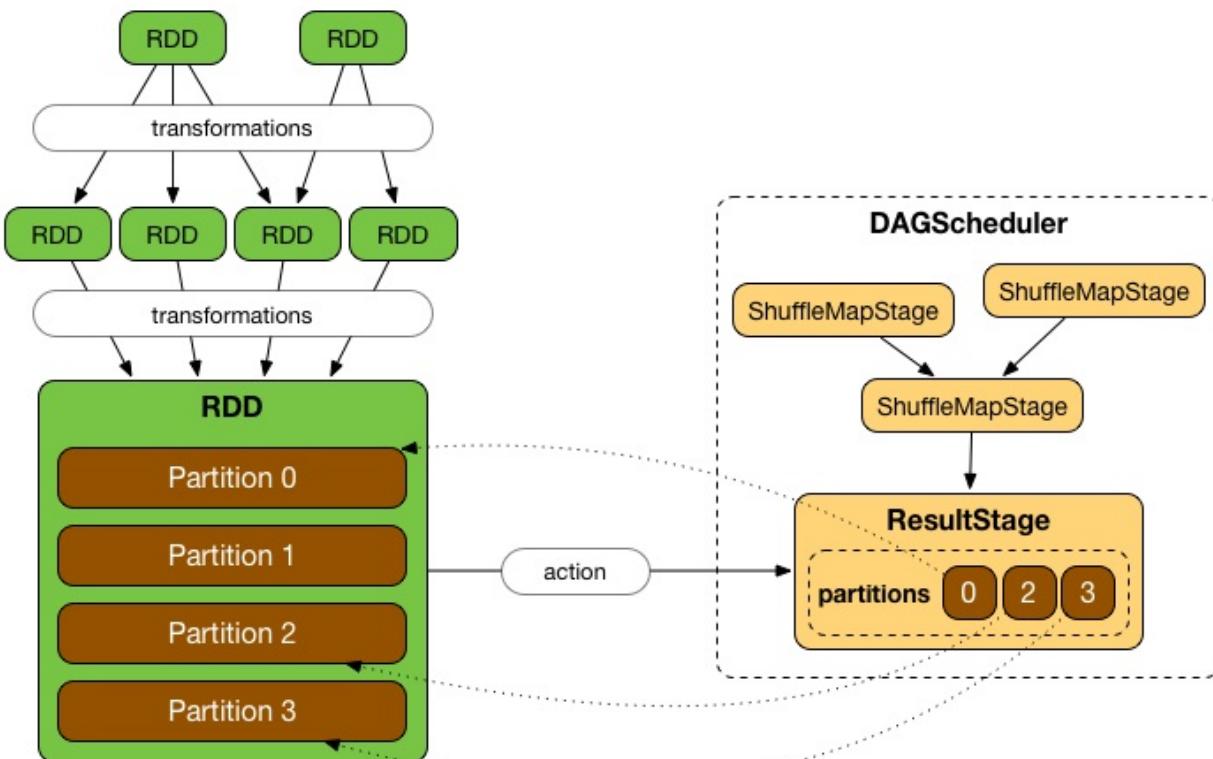


Figure 2. ResultStage and partitions

Tip

Read about `TaskContext` in [TaskContext](#).



# TaskScheduler

A `TaskScheduler` schedules `tasks` for a `single Spark application` according to `scheduling mode`.

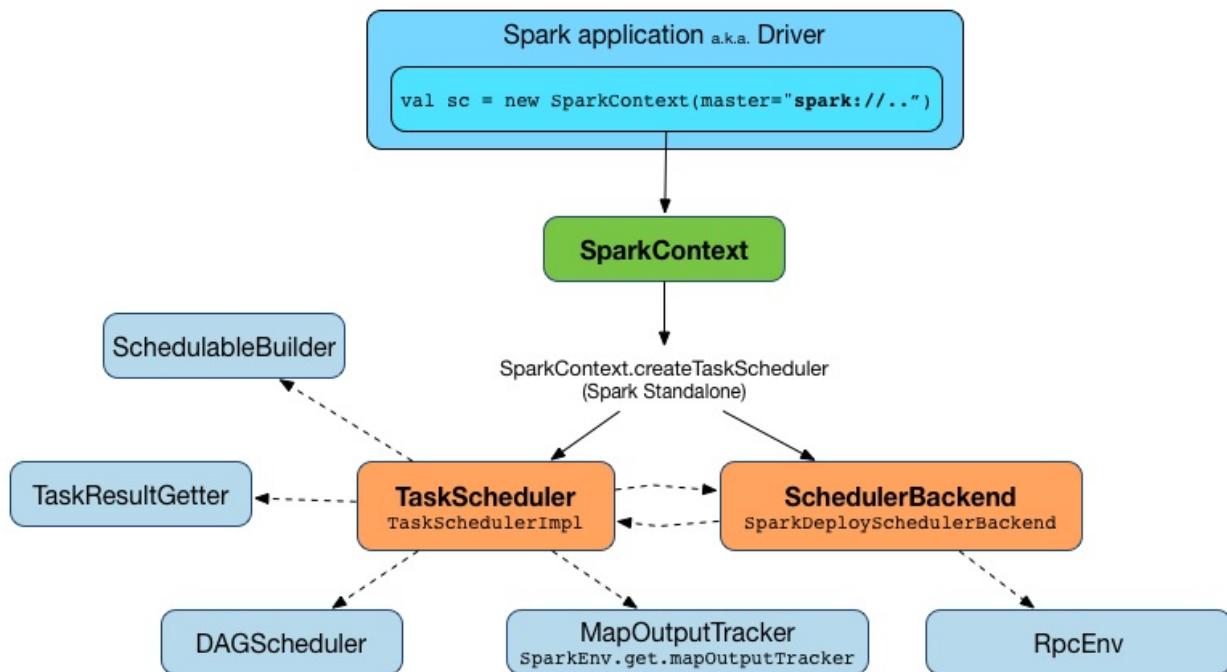


Figure 1. TaskScheduler works for a single SparkContext

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.

**Note**

`TaskScheduler` is a `private[spark]` Scala trait. You can find the sources in [org.apache.spark.scheduler.TaskScheduler](#).

## TaskScheduler Contract

Every `TaskScheduler` follows the following contract:

- It can be `started`.
- It can be `stopped`.
- It can `do post-start initialization` if needed for additional post-start initialization.
- It `submits TaskSets for execution`.
- It can `cancel tasks for a stage`.
- It can `set a custom DAGScheduler`.

- It can calculate the default level of parallelism.
- It returns a custom application id.
- It returns an application attempt id.
- It can handle executor's heartbeats and executor lost events.
- It has a `rootPool` Pool (of `Schedulables`).
- It can put tasks in order according to a scheduling policy (as `schedulingMode`). It is used in `SparkContext.getSchedulerMode`.

Caution

FIXME Have an exercise to create a SchedulerBackend.

## TaskScheduler's Lifecycle

A `TaskScheduler` is created while `SparkContext` is being created (by calling `SparkContext.createTaskScheduler` for a given master URL and deploy mode).

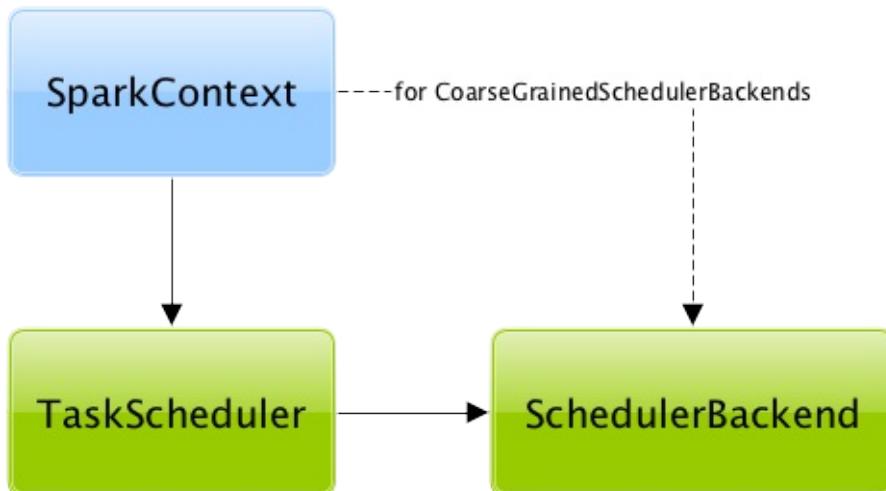


Figure 2. TaskScheduler uses SchedulerBackend to support different clusters

At this point in `SparkContext`'s lifecycle, the internal `_taskScheduler` points at the `TaskScheduler` (and it is "announced" by sending a blocking `TaskSchedulerIsSet` message to `HeartbeatReceiver` RPC endpoint).

The `TaskScheduler` is started right after the blocking `TaskSchedulerIsSet` message receives a response.

The application ID and the application's attempt ID are set at this point (and `sparkContext` uses the application id to set up `spark.app.id`, `SparkUI`, and `BlockManager`).

Caution	<b><a href="#">FIXME</a></b> The application id is described as "associated with the job." in TaskScheduler, but I think it is "associated with the application" and you can have many jobs per application.
---------	--

Right before SparkContext is fully initialized, [TaskScheduler.postStartHook](#) is called.

The internal `_taskScheduler` is cleared (i.e. set to `null`) while [SparkContext](#) is being stopped.

[TaskScheduler](#) is stopped while [DAGScheduler](#) is being stopped.

Warning	<b><a href="#">FIXME</a></b> If it is SparkContext to start a TaskScheduler, shouldn't SparkContext stop it too? Why is this the way it is now?
---------	---

## Starting TaskScheduler

```
start(): Unit
```

`start` is currently called while [SparkContext](#) is being created.

## Stopping TaskScheduler

```
stop(): Unit
```

`stop` is currently called while [DAGScheduler](#) is being stopped.

## Post-Start Initialization (postStartHook method)

```
postStartHook() {}
```

`postStartHook` does nothing by default, but allows custom implementations to do some post-start initialization.

Note	It is currently called right before <a href="#">SparkContext's initialization</a> finishes.
------	---

## Submitting TaskSets for Execution

```
submitTasks(taskSet: TaskSet): Unit
```

`submitTasks` accepts a [TaskSet](#) for execution.

## Note

It is currently called by [DAGScheduler](#) when there are tasks to be executed for a stage.

## Cancelling Tasks for Stage (`cancelTasks` method)

```
cancelTasks(stageId: Int, interruptThread: Boolean): Unit
```

`cancelTasks` cancels all tasks submitted for execution in a stage `stageId`.

## Note

It is currently called by [DAGScheduler](#) when it cancels a stage.

## Setting Custom DAGScheduler

```
setDAGScheduler(dagScheduler: DAGScheduler): Unit
```

`setDAGScheduler` sets a custom `DAGScheduler`.

## Note

It is currently called by [DAGScheduler](#) when it is created.

## Calculating Default Level of Parallelism (`defaultParallelism` method)

```
defaultParallelism(): Int
```

`defaultParallelism` calculates the default level of parallelism to use in a Spark application as the number of partitions in RDDs and also as a hint for sizing jobs.

## Note

It is called by `SparkContext` for its `defaultParallelism`.

## Tip

Read more in [Calculating Default Level of Parallelism \(`defaultParallelism` method\)](#) for the one and only implementation of the `TaskScheduler` contract — `TaskSchedulerImpl`.

## Calculating Application ID (`applicationId` method)

```
applicationId(): String
```

`applicationId` gives the current application's id. It is in the format `spark-application-[System.currentTimeMillis]` by default.

Note	It is currently used in <a href="#">SparkContext</a> while it is being initialized.
------	---

## Calculating Application Attempt ID (`applicationAttemptId` method)

```
applicationAttemptId(): Option[String]
```

`applicationAttemptId` gives the current application's attempt id.

Note	It is currently used in <a href="#">SparkContext</a> while it is being initialized.
------	---

## Handling Executor's Heartbeats (`executorHeartbeatReceived` method)

```
executorHeartbeatReceived(  
    execId: String,  
    accumUpdates: Array[(Long, Seq[AccumulatorV2[_, _]])],  
    blockManagerId: BlockManagerId): Boolean
```

`executorHeartbeatReceived` handles heartbeats from an executor `execId` with the partial values of accumulators and `BlockManagerId`.

It is expected to be positive (i.e. return `true`) when the executor `execId` is managed by the `TaskScheduler`.

Note	It is currently used in <a href="#">HeartbeatReceiver RPC endpoint</a> in <a href="#">SparkContext</a> to handle heartbeats from executors.
------	---

## Handling Executor Lost Events (`executorLost` method)

```
executorLost(executorId: String, reason: ExecutorLossReason): Unit
```

`executorLost` handles events about an executor `executorId` being lost for a given `reason`.

Note	It is currently used in <a href="#">HeartbeatReceiver RPC endpoint</a> in <a href="#">SparkContext</a> to process host expiration events and to remove executors in scheduler backends.
------	---

## Available Implementations

Spark comes with the following task schedulers:

- [TaskSchedulerImpl](#)
- [YarnScheduler](#) - the TaskScheduler for [Spark on YARN](#) in [client deploy mode](#).
- [YarnClusterScheduler](#) - the TaskScheduler for [Spark on YARN](#) in [cluster deploy mode](#).

# Tasks

In Spark, a **task** (aka *command*) is the smallest individual unit of execution that represents a partition in a dataset and that an [executor can execute on a single machine](#).

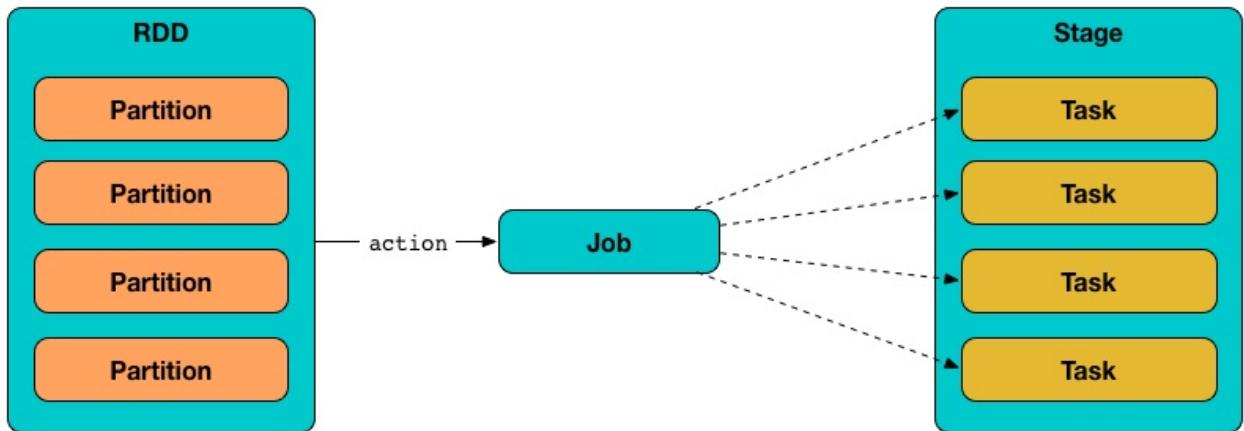


Figure 1. Tasks correspond to partitions in RDD

In other (more technical) words, a task is a computation on a data partition in a stage in a job.

A task can only belong to one stage and operate on a single partition. All tasks in a stage must be completed before the stages that follow can start.

Tasks are spawned one by one for each stage and partition.

**Caution**

[FIXME](#) What are `stageAttemptId` and `taskAttemptId` ?

A task in Spark is represented by the `Task` abstract class with two concrete implementations:

- [ShuffleMapTask](#) that executes a task and divides the task's output to multiple buckets (based on the task's partitioner).
- [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 are a set of [ShuffleMapTasks](#).

## Task Attributes

A `Task` instance is uniquely identified by the following task attributes:

- `stageId` - there can be many stages in a job. Every stage has its own unique `stageId` that the task belongs to.
- `stageAttemptId` - a stage can be re-attempted for execution in case of failure. `stageAttemptId` represents the attempt id of a stage that the task belongs to.
- `partitionId` - a task is a unit of work on a partitioned distributed dataset. Every partition has its own unique `partitionId` that a task processes.
- `metrics` - an instance of [TaskMetrics](#) for the task.
- `localProperties` - local private properties of the task.

## Running Task Thread (run method)

```
run(
    taskAttemptId: Long,
    attemptNumber: Int,
    metricsSystem: MetricsSystem): T
```

`run` registers task attempt id to the executor's BlockManager and creates a [TaskContextImpl](#) that in turn gets set as the thread local [TaskContext](#).

If the task has been killed before the task runs it is [killed](#) (with `interruptThread` flag disabled).

The [task runs](#).

Caution	<a href="#">FIXME</a> Describe <code>catch</code> and <code>finally</code> blocks.
---------	--

Note	When <code>run</code> is called from <a href="#">TaskRunner.run</a> , the <code>Task</code> has just been deserialized from <code>taskBytes</code> that were sent over the wire to an executor. <code>localProperties</code> and <a href="#">TaskMemoryManager</a> are already assigned.
------	--

## Running Task (runTask method)

### Task States

A task can be in one of the following states:

- `LAUNCHING`
- `RUNNING` when the task is being started.
- `FINISHED` when the task finished with the serialized result.

- `FAILED` when the task fails, e.g. 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`.

Note	Task status updates are sent from executors to the driver through <a href="#">ExecutorBackend</a> .
------	---

Task is finished when it is in one of `FINISHED`, `FAILED`, `KILLED`, `LOST`

`LOST` and `FAILED` states are considered failures.

Tip	Task states correspond to <a href="#">org.apache.mesos.Protos.TaskState</a> .
-----	---

## Collect Latest Values of Accumulators (`collectAccumulatorUpdates`)

```
collectAccumulatorUpdates(taskFailed: Boolean = false): Seq[AccumulableInfo]
```

`collectAccumulatorUpdates` collects the latest values of accumulators used in a task (and returns the values as a collection of [AccumulableInfo](#)).

Note	It is used in <a href="#">TaskRunner</a> to send a task's final results with the latest values of accumulators used.
------	--

When `taskFailed` is `true` it filters out accumulators with `countFailedValues` disabled.

Caution	<a href="#">FIXME</a> Why is the check <code>context != null</code> ?
---------	---

Note	It uses <code>context.taskMetrics.accumulatorUpdates()</code> .
------	---

Caution	<a href="#">FIXME</a> What is <code>context.taskMetrics.accumulatorUpdates()</code> doing?
---------	--

## Killing Task (kill method)

```
kill(interruptThread: Boolean)
```

`kill` marks the task to be killed, i.e. it sets the internal `_killed` flag to `true`.

It calls [TaskContextImpl.markInterrupted](#) when `context` is set.

If `interruptThread` is enabled and the internal `taskThread` is available, `kill` interrupts it.

Caution	<a href="#">FIXME</a> When could <code>context</code> and <code>interruptThread</code> not be set?
---------	--

## ShuffleMapTask

A **ShuffleMapTask** divides the elements of an RDD into multiple buckets (based on a partitioner specified in [ShuffleDependency](#)).

## ResultTask

Caution	<a href="#">FIXME</a>
---------	-----------------------

## taskMemoryManager attribute

`taskMemoryManager` is the [TaskMemoryManager](#) that manages the memory allocated by the task.

# 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 [Scheduledable](#)).

The `priority` field is used in [FIFOSchedulingAlgorithm](#) in which equal priorities give stages an advantage (not to say *priority*).

Note	<code>FIFOSchedulingAlgorithm</code> is only used for <code>FIFO</code> scheduling mode in a <a href="#">Pool</a> (i.e. a schedulable collection of <code>Schedulable</code> objects).
------	--

Effectively, the `priority` field is the job's id of the first job this stage was part of (for FIFO scheduling).

# Schedulable

`Schedulable` is a [contract of schedulable entities](#).

Note

`Schedulable` is a `private[spark]` Scala trait. You can find the sources in [org.apache.spark.scheduler.Schedulable](#).

There are currently two types of `Schedulable` entities in Spark:

- [Pool](#)
- [TaskSetManager](#)

## Schedulable Contract

Every `Schedulable` follows the following contract:

- It has a `name`.

```
name: String
```

- It has a `parent` [Pool](#) (of other `Schedulables`).

```
parent: Pool
```

With the `parent` property you could build a tree of `Schedulables`

- It has a `schedulingMode`, `weight`, `minShare`, `runningTasks`, `priority`, `stageId`.

```
schedulingMode: SchedulingMode  
weight: Int  
minShare: Int  
runningTasks: Int  
priority: Int  
stageId: Int
```

- It manages a [collection of Schedulables](#) and can add or remove one.

```
schedulableQueue: ConcurrentLinkedQueue[Schedulable]  
addSchedulable(schedulable: Schedulable): Unit  
removeSchedulable(schedulable: Schedulable): Unit
```

Note	<code>schedulableQueue</code> is <code>java.util.concurrent.ConcurrentLinkedQueue</code> .
------	--

- It can query for a `Schedulable` by name.

```
getSchedulableByName(name: String): Schedulable
```

- It can return a sorted collection of `TaskSetManagers`.
- It can be informed about lost `executors`.

```
executorLost(executorId: String, host: String, reason: ExecutorLossReason): Unit
```

It is called by `TaskSchedulerImpl` to inform `TaskSetManagers` about executors being lost.

- It checks for **speculatable tasks**.

```
checkSpeculatableTasks(): Boolean
```

Caution	<a href="#">FIXME</a> What are speculatable tasks?
---------	--

## getSortedTaskSetQueue

```
getSortedTaskSetQueue: ArrayBuffer[TaskSetManager]
```

`getSortedTaskSetQueue` is used in `TaskSchedulerImpl` to handle resource offers (to let every `TaskSetManager` know about a new executor ready to execute tasks).

## schedulableQueue

```
schedulableQueue: ConcurrentLinkedQueue[Schedulable]
```

`schedulableQueue` is used in `SparkContext.getAllPools`.

# TaskSetManager

A `TaskSetManager` is a [Schedulerable](#) that manages execution of the tasks in a single [TaskSet](#) (after having it been handed over by [TaskScheduler](#)).

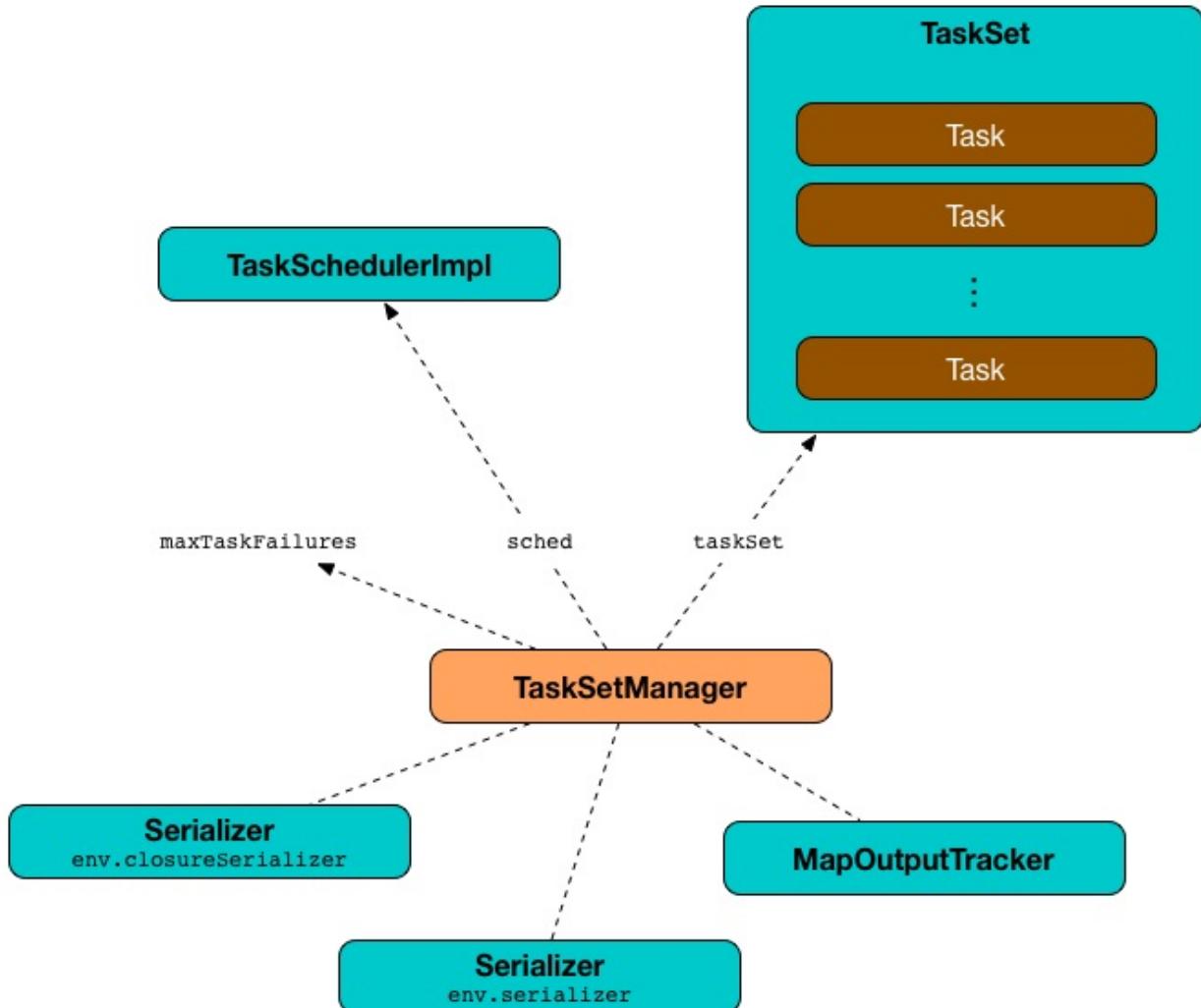


Figure 1. TaskSetManager and its Dependencies

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:

- `name` is `TaskSet_[taskSet.stageId.toString]`
- no `parent` is ever assigned, i.e. it is always `null`.
- It means that it can only be a leaf in the tree of Schedulables (with [Pools](#) being the nodes).
- `schedulingMode` always returns `SchedulingMode.NONE` (since there is nothing to schedule).
- `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 owned [TaskSet](#) (using `taskSet.priority`).
- `stageId` is the stage id of the owned [TaskSet](#) (using `taskSet.stageId`).
- `schedulableQueue` returns no queue, i.e. `null`.
- `addSchedulable` and `removeSchedulable` do nothing.
- `getSchedulableByName` always returns `null`.
- `getSortedTaskSetQueue` returns a one-element collection with the sole element being itself.
- [executorLost](#)
- [checkSpeculatableTasks](#)

## Handling Executor Lost Events (`executorLost` method)

**Note**

`executorLost` is part of the [Schedulable Contract](#) which is called by [TaskSchedulerImpl](#) to inform [TaskSetManagers](#) about executors being lost.

Since [TaskSetManager](#) manages execution of the tasks in a single [TaskSet](#), when an executor gets lost, the affected tasks that have been running on the failed executor need to be re-enqueued. `executorLost` is the mechanism to "announce" the event to all [TaskSetManagers](#).

`executorLost` first checks whether the [Taskset](#) is for a [ShuffleMapStage](#) (in which case all [TaskSet.tasks](#) are instances of [ShuffleMapTask](#)) as well as whether an [external shuffle server](#) is used (that could serve the shuffle outputs in case of failure).

If it is indeed for a failed [ShuffleMapStage](#) and no external shuffle server is enabled, all successfully-completed tasks for the failed executor (using [taskInfos internal registry](#)) get added to the collection of pending tasks and the [DAGScheduler](#) is informed about resubmission (as [Resubmitted end reason](#)).

The [internal registries](#) - [successful](#), [copiesRunning](#), and [tasksSuccessful](#) - are updated.

Regardless of the above check, all currently-running tasks for the failed executor are reported as [failed](#) (with the task state being [FAILED](#)).

[recomputeLocality](#) is called.

## Checking Speculatable Tasks ([checkSpeculatableTasks](#) method)

**Note**

`checkSpeculatableTasks` is part of the [Schedulable Contract](#).

`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 it ran more than [threshold] ms
```

The task gets added to the internal `speculatableTasks` collection. The method responds positively.

## **addPendingTask**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **dequeueSpeculativeTask**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **dequeueTask**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **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**

`TaskLocality` is an enumeration with `PROCESS_LOCAL`, `NODE_LOCAL`, `NO_PREF`, `RACK_LOCAL`, `ANY` 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 `TaskSetManager.resourceoffer` + Does this have anything related to the following section about scheduling tasks?

```
resourceOffer(  
  execId: String,  
  host: String,  
  maxLocality: TaskLocality): Option[TaskDescription]
```

When a `TaskSetManager` is a [zombie](#), `resourceOffer` returns no `TaskDescription` (i.e. `None`).

For a non-zombie `TaskSetManager` , `resourceOffer` ...[FIXME](#)

Caution	<a href="#">FIXME</a>
---------	-----------------------

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, PROCESS_LOCAL, 1997 bytes)
```

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 ([serializedTask.limit / 1024] KB). The maximum recommended task size is 100 KB.
```

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], partition [task.partitionId],[taskLocality], [serializedTask.limit] bytes)
```

For example:

```
INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1, localhost, partition 1, PROCESS_LOCAL, 2054 bytes)
```

## Scheduling Tasks in TaskSet

Caution	<a href="#">FIXME</a>
---------	-----------------------

For each submitted [TaskSet](#), a new `TaskSetManager` is created. The `TaskSetManager` completely and exclusively owns a `TaskSet` submitted for execution.

Caution	<a href="#">FIXME</a> A picture with <code>TaskSetManager</code> owning <code>TaskSet</code>
---------	--

## 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, the `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 `TaskInfo`

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] ms on [info.host] ([tasksSuccessful]/[numTasks])
```

**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] has already completed successfully
```

`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	<a href="#">FIXME</a> Describe <code>TaskInfo</code>
---------	--

The method calculates the failure exception to report per `TaskEndReason`. See below for the possible cases of `TaskEndReason`.

Caution	<a href="#">FIXME</a> Describe <code>TaskEndReason</code> .
---------	---

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.

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.description]; not retrying"
```

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.description]) [duplicate [count]]
```

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 reason unrelated to the task. Not counting this failure towards the maximum number of failures for the task.
```

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) t
hrow new Exception("Partition 2 marked failed") else it).count
...
15/10/27 17:24:56 INFO DAGScheduler: Submitting 2 missing tasks from ResultStage 1 (Ma
pPartitionsRDD[7] at mapPartitionsWithIndex at <console>:25)
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,
partition 0,PROCESS_LOCAL, 2062 bytes)
...
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,
partition 0,PROCESS_LOCAL, 2062 bytes)
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:0+1784
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
localhost: java.lang.Exception (Partition 2 marked failed) [duplicate 4]
15/10/27 17:24:56 ERROR TaskSetManager: Task 0 in stage 1.0 failed 5 times; aborting j
ob
15/10/27 17:24:56 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks have all co
mpleted, from pool
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
.058 s
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 0.08581
0 s
org.apache.spark.SparkException: Job aborted due to stage failure: Task 0 in stage 1.0
 failed 5 times, most recent failure: Lost task 0.4 in stage 1.0 (TID 7, localhost): j
ava.lang.Exception: Partition 2 marked failed

```

## 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 `maybeFinishTaskSet` method called? When is `runningTasks` `0`?

## 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`
- `stageId` (default: `taskSet.stageId`)
- `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`

# Schedulable Pool

`Pool` is a [Schedulable](#) entity that represents a tree of [TaskSetManagers](#), i.e. it contains a collection of `TaskSetManagers` or the `Pools` thereof.

A `Pool` has a mandatory name, a [scheduling mode](#), initial `minShare` and `weight` that are defined when it is created.

Note	An instance of <code>Pool</code> is created when <a href="#">TaskSchedulerImpl</a> is initialized.
------	--

Note	The <a href="#">TaskScheduler Contract</a> and <a href="#">Schedulable Contract</a> both require that their entities have <code>rootPool</code> of type <code>Pool</code> .
------	---

## taskSetSchedulingAlgorithm Attribute

Using the [scheduling mode](#) (given when a `Pool` object is created), `Pool` selects [SchedulingAlgorithm](#) and sets `taskSetSchedulingAlgorithm`:

- [FIFOSchedulingAlgorithm](#) for FIFO scheduling mode.
- [FairSchedulingAlgorithm](#) for FAIR scheduling mode.

It throws an `IllegalArgumentException` when unsupported scheduling mode is passed on:

Unsupported spark.scheduler.mode: [schedulingMode]	
--	--

Tip	Read about the scheduling modes in <a href="#">SchedulingMode</a> .
-----	---

Note	<code>taskSetSchedulingAlgorithm</code> is used in <a href="#">getSortedTaskSetQueue</a> .
------	--

## addSchedulable

Note	<code>addSchedulable</code> is part of the <a href="#">Schedulable Contract</a> .
------	---

`addSchedulable` adds a `Schedulable` to the [schedulableQueue](#) and [schedulableNameToSchedulable](#).

More importantly, it sets the `Schedulable` entity's [parent](#) to itself.

## Getting TaskSetManagers Sorted ([getSortedTaskSetQueue](#) method)

Note	<code>getSortedTaskSetQueue</code> is part of the <a href="#">Schedulable Contract</a> .
------	--

`getSortedTaskSetQueue` sorts all the [Schedulables](#) in `schedulableQueue` queue by a [SchedulingAlgorithm](#) (from the internal `taskSetSchedulingAlgorithm`).

Note	It is called when <code>TaskSchedulerImpl</code> processes executor resource offers.
------	--

## Schedulables by Name (`schedulableNameToSchedulable` registry)

```
schedulableNameToSchedulable = new ConcurrentHashMap[String, Schedulable]
```

`schedulableNameToSchedulable` is a lookup table of [Schedulable](#) objects by their names.

Beside the obvious usage in the housekeeping methods like `addSchedulable`, `removeSchedulable`, `getSchedulableByName` from the [Schedulable Contract](#), it is exclusively used in [SparkContext.getPoolForName](#).

## SchedulingAlgorithm

`SchedulingAlgorithm` is the interface for a sorting algorithm to sort [Schedulables](#).

There are currently two `SchedulingAlgorithms`:

- [FIFOSchedulingAlgorithm](#) for FIFO scheduling mode.
- [FairSchedulingAlgorithm](#) for FAIR scheduling mode.

## FIFOSchedulingAlgorithm

`FIFOSchedulingAlgorithm` is a scheduling algorithm that compares `Schedulables` by their `priority` first and, when equal, by their `stageId`.

Note	<code>priority</code> and <code>stageId</code> are part of <a href="#">Schedulable Contract</a> .
------	---

Caution	<i>FIXME A picture is worth a thousand words.</i> How to picture the algorithm?
---------	---

## FairSchedulingAlgorithm

`FairSchedulingAlgorithm` is a scheduling algorithm that compares `Schedulables` by their `minShare`, `runningTasks`, and `weight`.

Note	<code>minShare</code> , <code>runningTasks</code> , and <code>weight</code> are part of <a href="#">Schedulable Contract</a> .
------	--

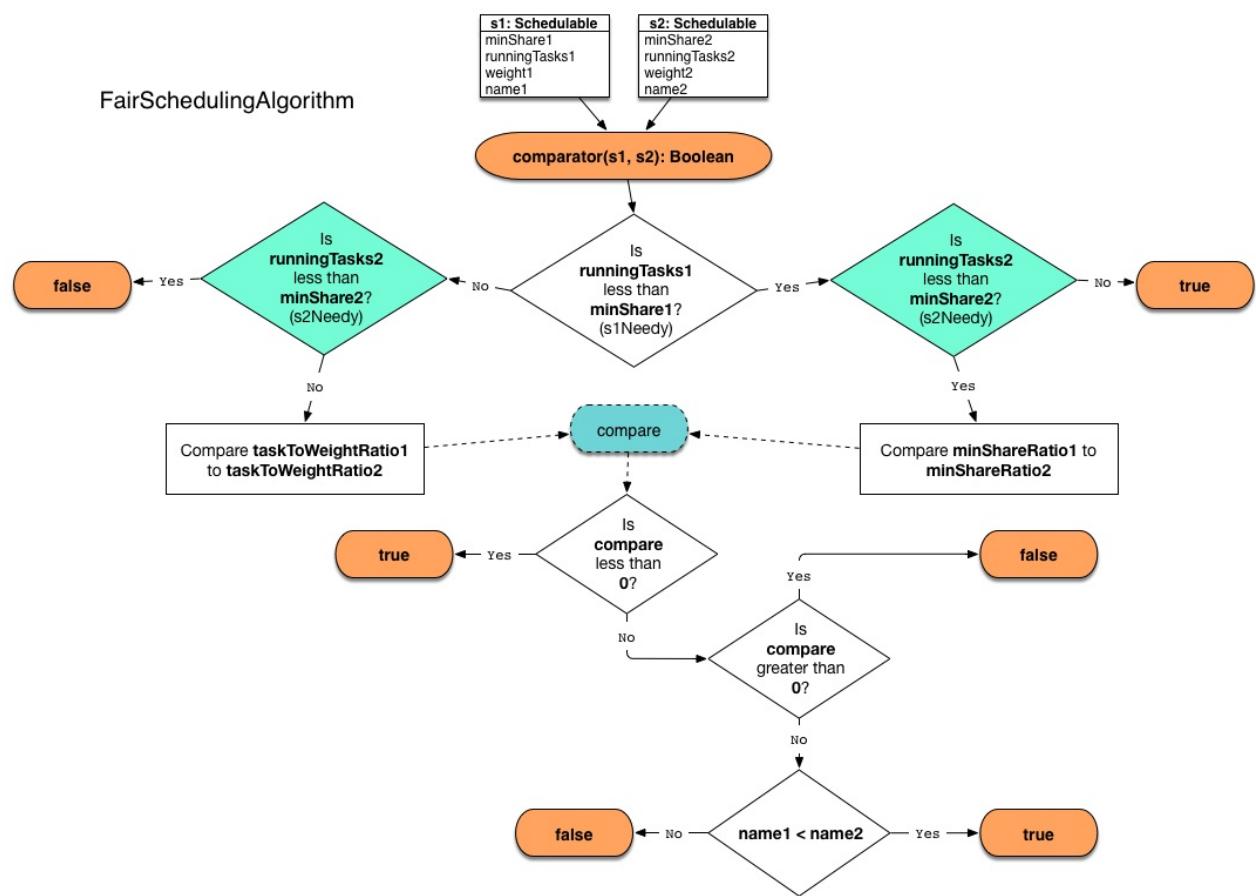


Figure 1. FairSchedulingAlgorithm

For each input `Schedulable`, `minShareRatio` is computed as `runningTasks` by `minShare` (but at least `1`) while `taskToWeightRatio` is `runningTasks` by `weight`.

# Schedulable Builders

`SchedulableBuilder` is a [contract of schedulable builders](#) that operate on a [pool of TaskSetManagers](#) (from an owning [TaskSchedulerImpl](#)).

Schedulable builders can [build pools](#) and [add new Schedulable entities](#) to the pool.

Note	A <code>SchedulableBuilder</code> is created when <code>TaskSchedulerImpl</code> is being initialized. You can select the <code>SchedulableBuilder</code> to use by <code>spark.scheduler.mode</code> setting.
------	--

Spark comes with two implementations of the `SchedulableBuilder` Contract:

- [FIFOSchedulableBuilder](#) - the default `SchedulableBuilder`
- [FairSchedulableBuilder](#)

Note	<code>SchedulableBuilder</code> is a <code>private[spark]</code> Scala trait. You can find the sources in <a href="#">org.apache.spark.scheduler.SchedulableBuilder</a> .
------	---

## SchedulableBuilder Contract

Every `SchedulableBuilder` provides the following services:

- It manages a [root pool](#).
- It can [build pools](#).
- It can [add a Schedulable with properties](#).

## Root Pool (`rootPool` method)

```
rootPool: Pool
```

`rootPool` method returns a [Pool](#) (of [Schedulables](#)).

This is the data structure managed (*aka wrapped*) by `SchedulableBuilders`.

## Build Pools (`buildPools` method)

```
buildPools(): Unit
```

Note	It is exclusively called by <code>TaskSchedulerImpl.initialize</code> .
------	---

## Adding Schedulable (to Pool) (addTaskSetManager method)

```
addTaskSetManager(manager: Schedulable, properties: Properties): Unit
```

`addTaskSetManager` registers the `manager` [Schedulable](#) (with additional `properties`) to the `rootPool`.

**Note**

`addTaskSetManager` is exclusively used by [TaskSchedulerImpl](#) to submit a [TaskSetManager](#) for a stage for execution.

# FIFOSchedulableBuilder - SchedulableBuilder for FIFO Scheduling Mode

`FIFOSchedulableBuilder` is a [SchedulableBuilder](#) that is a *mere* wrapper around a single [Pool](#) (the only constructor parameter).

Note	<code>FIFOSchedulableBuilder</code> is the default <code>SchedulableBuilder</code> for <code>TaskSchedulerImpl</code> (see <a href="#">Creating TaskSchedulerImpl</a> ).
Note	When <code>FIFOSchedulableBuilder</code> is created, the <code>TaskSchedulerImpl</code> passes its own <code>rootPool</code> (that belongs to the <a href="#">TaskScheduler Contract</a> that <code>TaskSchedulerImpl</code> follows).

`FIFOSchedulableBuilder` obeys the [SchedulableBuilder Contract](#) as follows:

- `buildPools` does nothing.
- `addTaskSetManager` [passes the input `Schedulable` to the one and only `rootPool` Pool \(using `addSchedulable`\)](#) and completely disregards the properties of the `Schedulable`.

# FairScheduledBuilder - SchedulableBuilder for FAIR Scheduling Mode

`FairScheduledBuilder` is a `SchedulableBuilder` with the pools configured in an [optional allocations configuration file](#).

It reads the allocations file using the internal `buildFairSchedulerPool` method.

<b>Tip</b>	<p>Enable <code>INFO</code> logging level for <code>org.apache.spark.scheduler.FairScheduledBuilder</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.scheduler.FairScheduledBuilder=INFO</pre> <p>Refer to <a href="#">Logging</a>.</p>
------------	---

## buildPools

`buildPools` builds the `rootPool` based on the allocations configuration file from the optional `spark.scheduler.allocation.file` or `fairscheduler.xml` (on the classpath).

<b>Note</b>	<p><code>buildPools</code> is part of the <a href="#">SchedulableBuilder Contract</a>.</p>
<b>Tip</b>	<p>Spark comes with <code>fairscheduler.xml.template</code> to use as a template for the allocations configuration file to start from.</p>

It then ensures that the default pool is also registered.

## addTaskSetManager

`addTaskSetManager` looks up the default pool (using `Pool.getSchedulableByName`).

<b>Note</b>	<p><code>addTaskSetManager</code> is part of the <a href="#">SchedulableBuilder Contract</a>.</p>
<b>Note</b>	<p>Although the <code>Pool.getSchedulableByName</code> method may return no Schedulable for a name, the default root pool does exist as it is assumed it was registered before.</p>

If `properties` for the `Schedulable` were given, `spark.scheduler.pool` property is looked up and becomes the current pool name (or defaults to `default` ).

**Note**

`spark.scheduler.pool` is the only property supported. Refer to [spark.scheduler.pool](#) later in this document.

If the pool name is not available, it is registered with the pool name, `FIFO` scheduling mode, minimum share `0`, and weight `1`.

After the new pool was registered, you should see the following INFO message in the logs:

```
INFO FairSchedulableBuilder: Created pool [poolName], schedulingMode: FIFO, minShare: 0, weight: 1
```

The `manager` schedulable is registered to the pool (either the one that already existed or was created just now).

You should see the following INFO message in the logs:

```
INFO FairSchedulableBuilder: Added task set [manager.name] to pool [poolName]
```

## spark.scheduler.pool Property

[SparkContext.setLocalProperty](#) allows for setting properties per thread. This mechanism is used by `FairSchedulableBuilder` to watch for `spark.scheduler.pool` property to group jobs from a thread and submit them to a non-default pool.

```
val sc: SparkContext = ???  
sc.setLocalProperty("spark.scheduler.pool", "myPool")
```

**Tip**

See [addTaskSetManager](#) for how this setting is used.

## fairscheduler.xml Allocations Configuration File

The allocations configuration file is an XML file.

The default `conf/fairscheduler.xml.template` looks as follows:

```
<?xml version="1.0"?>
<allocations>
  <pool name="production">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
  </pool>
  <pool name="test">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>3</minShare>
  </pool>
</allocations>
```

**Tip**

The top-level element's name `allocations` can be anything. Spark does not insist on `allocations` and accepts any name.

## Ensure Default Pool is Registered (`buildDefaultPool` method)

`buildDefaultPool` method checks whether `default` was defined already and if not it adds the `default` pool with `FIFO` scheduling mode, minimum share `0`, and weight `1`.

You should see the following INFO message in the logs:

```
INFO FairSchedulableBuilder: Created default pool default, schedulingMode: FIFO, minSh  
are: 0, weight: 1
```

## Build Pools from XML Allocations File (`buildFairSchedulerPool` method)

```
buildFairSchedulerPool(is: InputStream)
```

`buildFairSchedulerPool` reads `Pools` from the allocations configuration file (as `is`).

For each `pool` element, it reads its name (from `name` attribute) and assumes the default pool configuration to be `FIFO` scheduling mode, minimum share `0`, and weight `1` (unless overrode later).

**Caution**

**FIXME** Why is the difference between `minShare 0` and `weight 1` vs `rootPool` in `TaskSchedulerImpl.initialize` - `0` and `0`? It is definitely an inconsistency.

If `schedulingMode` element exists and is not empty for the pool it becomes the current pool's scheduling mode. It is case sensitive, i.e. with all uppercase letters.

If `minShare` element exists and is not empty for the pool it becomes the current pool's `minShare`. It must be an integer number.

If `weight` element exists and is not empty for the pool it becomes the current pool's `weight`. It must be an integer number.

The pool is then [registered to](#) `rootPool`.

If all is successful, you should see the following INFO message in the logs:

```
INFO FairSchedulableBuilder: Created pool [poolName], schedulingMode: [schedulingMode]
, minShare: [minShare], weight: [weight]
```

## Settings

### **spark.scheduler.allocation.file**

`spark.scheduler.allocation.file` is the file path of an optional scheduler configuration file that [FairSchedulableBuilder.buildPools](#) uses to build pools.

# Scheduling Mode — spark.scheduler.mode

**Scheduling Mode** (aka *order task policy* or *scheduling policy* or *scheduling order*) defines a policy to sort tasks in order for execution.

The scheduling mode `schedulingMode` attribute is a part of the [TaskScheduler Contract](#).

The only implementation of the `TaskScheduler` contract in Spark — [TaskSchedulerImpl](#) — uses `spark.scheduler.mode` setting to configure `schedulingMode` that is *merely* used to set up the `rootPool` attribute (with `FIFO` being the default). It happens when [TaskSchedulerImpl is initialized](#).

There are three acceptable scheduling modes:

- `FIFO` with no pools but a single top-level unnamed pool with elements being [TaskSetManager](#) objects; lower priority gets [Schedulable](#) sooner or earlier stage wins.
- `FAIR` with a hierarchy of `Schedulable` (sub)pools with the `rootPool` at the top.
- **NONE** (not used)

**Note** Out of three possible `SchedulingMode` policies only `FIFO` and `FAIR` modes are supported by [TaskSchedulerImpl](#).

**Note** After the root pool is initialized, the scheduling mode is no longer relevant (since the [Schedulable](#) that represents the root pool is fully set up).

The root pool is later used when [TaskSchedulerImpl submits tasks \(as TaskSets \) for execution](#).

**Note** The `root pool` is a `Schedulable`. Refer to [Schedulable](#).

## Monitoring FAIR Scheduling Mode using Spark UI

**Caution**

**FIXME** Describe me...

# TaskSchedulerImpl - Default TaskScheduler

`TaskSchedulerImpl` is the default implementation of [TaskScheduler Contract](#) and extends it to track racks per host and port. It can schedule tasks for multiple types of cluster managers by means of Scheduler Backends.

Using `spark.scheduler.mode` setting you can select the [scheduling policy](#).

It submits tasks using [SchedulableBuilders](#).

When a Spark application starts (and an instance of `SparkContext` is created)

`TaskSchedulerImpl` with a [SchedulerBackend](#) and [DAGScheduler](#) are created and soon started.

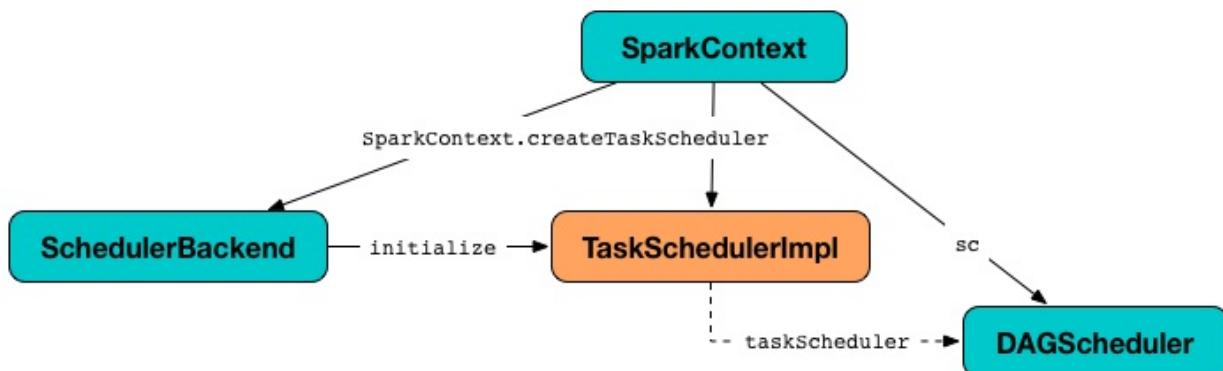


Figure 1. TaskSchedulerImpl and Other Services

Note

`TaskSchedulerImpl` is a `private[spark]` class with the source code in [org.apache.spark.scheduler.TaskSchedulerImpl](#).

Enable `INFO` or `DEBUG` logging level for `org.apache.spark.scheduler.TaskSchedulerImpl` logger to see what happens inside.

Tip

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

```
log4j.logger.org.apache.spark.scheduler.TaskSchedulerImpl=DEBUG
```

## applicationAttemptId method

```
applicationAttemptId(): Option[String]
```

Caution

[FIXME](#)

## schedulableBuilder Attribute

`schedulableBuilder` is a [SchedulableBuilder](#) for the `TaskSchedulerImpl`.

It is set up when a `TaskSchedulerImpl` is initialized and can be one of two available builders:

- [FIFOSchedulableBuilder](#) when scheduling policy is FIFO (which is the default scheduling policy).
- [FairSchedulableBuilder](#) for FAIR scheduling policy.

Note	Use <code>spark.scheduler.mode</code> setting to select the scheduling policy.
------	--

## Tracking Racks per Hosts and Ports (getRackForHost method)

<code>getRackForHost(value: String): Option[String]</code>
--

`getRackForHost` is a method to know about the racks per hosts and ports. By default, it assumes that racks are unknown (i.e. the method returns `None`).

Note	It is overridden by the YARN-specific TaskScheduler <a href="#">YarnScheduler</a> .
------	---

`getRackForHost` is currently used in two places:

- [TaskSchedulerImpl.resourceOffers](#) to track hosts per rack (using the [internal hostsByRack registry](#)) while processing resource offers.
- [TaskSchedulerImpl.removeExecutor](#) to...FIXME
- [TaskSetManager.addPendingTask](#), [TaskSetManager.dequeueTask](#), and [TaskSetManager.dequeueSpeculativeTask](#)

## Creating TaskSchedulerImpl

Creating a `TaskSchedulerImpl` object requires a [SparkContext](#) object, the [acceptable number of task failures](#) (`maxTaskFailures`) and optional [isLocal](#) flag (disabled by default, i.e. `false`).

Note	There is another <code>TaskSchedulerImpl</code> constructor that requires a <a href="#">SparkContext</a> object only and sets <a href="#">maxTaskFailures</a> to <code>spark.task.maxFailures</code> or, if <code>spark.task.maxFailures</code> is not set, defaults to <code>4</code> .
------	--

While being created, it initializes [internal registries](#) to their default values.

It then sets `schedulingMode` to the value of `spark.scheduler.mode` setting or `FIFO`.

Note	<code>schedulingMode</code> is part of <a href="#">TaskScheduler Contract</a> .
------	---

Failure to set `schedulingMode` results in a `SparkException`:

```
Unrecognized spark.scheduler.mode: [schedulingModeConf]
```

It sets `taskResultGetter` as a [TaskResultGetter](#).

Caution	<a href="#">FIXME</a> Where is <code>taskResultGetter</code> used?
---------	--

## Acceptable Number of Task Failures (maxTaskFailures attribute)

The acceptable number of task failures (`maxTaskFailures`) can be explicitly defined when creating [TaskSchedulerImpl](#) instance or based on `spark.task.maxFailures` setting that defaults to 4 failures.

Note	It is exclusively used when <a href="#">submitting tasks</a> through <a href="#">TaskSetManager</a> .
------	---

## Internal Cleanup After Removing Executor (removeExecutor method)

```
removeExecutor(executorId: String, reason: ExecutorLossReason): Unit
```

`removeExecutor` removes the `executorId` executor from the [internal registries](#): `executorIdToTaskCount`, `executorIdToHost`, `executorsByHost`, and `hostsByRack`. If the affected hosts and racks are the last entries in `executorsByHost` and `hostsByRack`, appropriately, they are removed from the registries.

Unless `reason` is `LossReasonPending`, the executor is removed from `executorIdToHost` registry and [TaskSetManagers](#) get notified.

Note	The internal <code>removeExecutor</code> is called as part of <a href="#">statusUpdate</a> and <a href="#">executorLost</a> .
------	---

## Local vs Non-Local Mode (isLocal attribute)

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Initializing TaskSchedulerImpl (initialize method)

```
initialize(backend: SchedulerBackend): Unit
```

`initialize` initializes a `TaskSchedulerImpl` object.

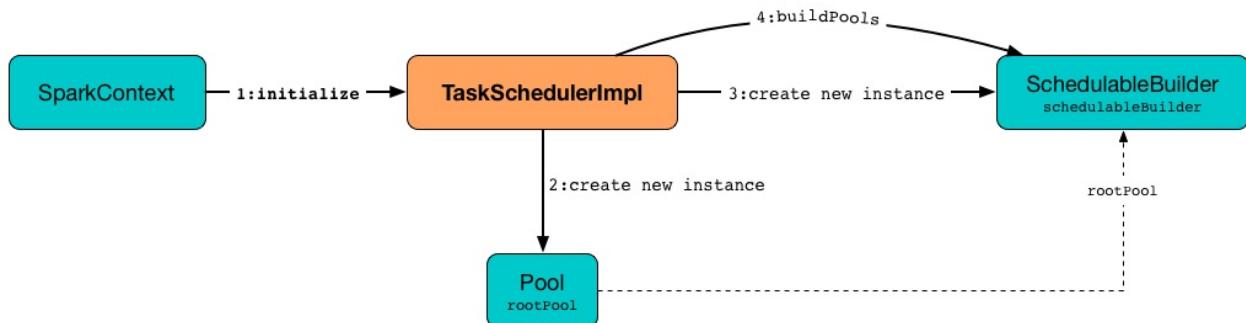


Figure 2. `TaskSchedulerImpl` initialization

Note

`initialize` is called while `SparkContext` is being created and creates `SchedulerBackend` and `TaskScheduler`.

`initialize` saves the reference to the current `SchedulerBackend` (as `backend`) and sets `rootPool` to be an empty-named `Pool` with already-initialized `schedulingMode` (while creating a `TaskSchedulerImpl` object), `initMinShare` and `initWeight` as `0`.

Note

`schedulingMode` and `rootPool` are a part of `TaskScheduler Contract`.

It then creates the internal `SchedulableBuilder` object (as `schedulableBuilder`) based on `schedulingMode`:

- `FIFOSchedulableBuilder` for `FIFO` scheduling mode
- `FairSchedulableBuilder` for `FAIR` scheduling mode

With the `schedulableBuilder` object created, `initialize` requests it to `build pools`.

Caution

**FIXME** Why are `rootPool` and `schedulableBuilder` created only now? What do they need that it is not available when `TaskSchedulerImpl` is created?

## Starting TaskSchedulerImpl (start method)

As part of `initialization of a sparkContext`, `TaskSchedulerImpl` is started (using `start` from the `TaskScheduler Contract`).

```
start(): Unit
```

It starts the [scheduler backend](#) it manages.

Below is a figure of the method calls in Spark Standalone mode.

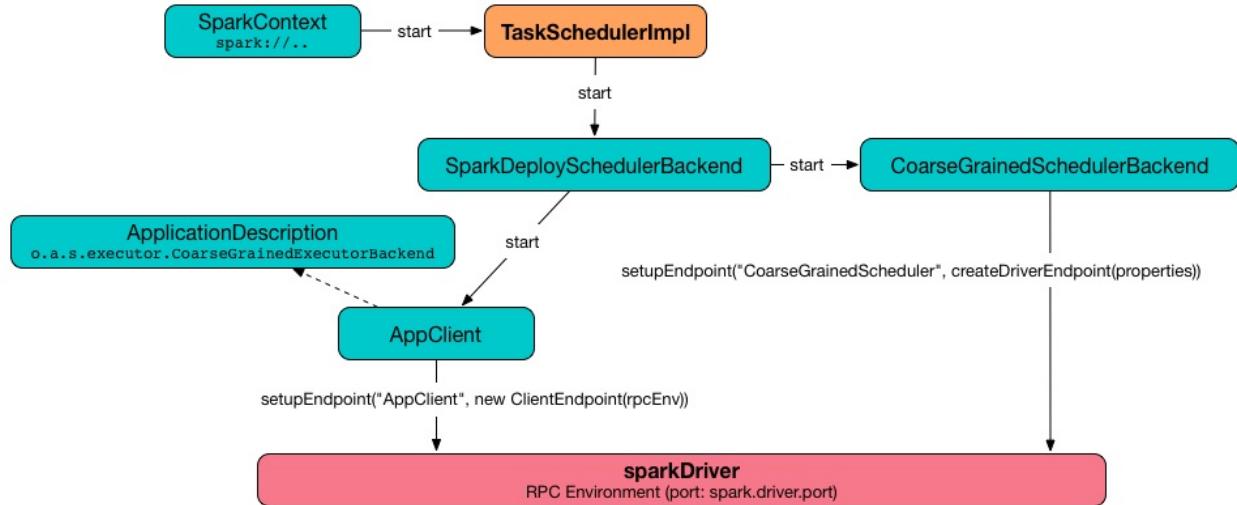


Figure 3. Starting TaskSchedulerImpl in Spark Standalone mode

It also starts the **task-scheduler-speculation** executor pool. See [Speculative Execution of Tasks](#).

## Post-Start Initialization (using postStartHook)

`postStartHook` is a custom implementation of [postStartHook from the TaskScheduler Contract](#) that waits until a scheduler backend is ready (using the internal blocking `waitBackendReady`).

Note

`postStartHook` is used when [SparkContext is created](#) (before it is fully created) and [YarnClusterScheduler.postStartHook](#).

## Waiting Until SchedulerBackend is Ready (waitBackendReady method)

The private `waitBackendReady` method waits until a [SchedulerBackend is ready](#).

It keeps on checking the status every 100 milliseconds until the SchedulerBackend is ready or the [SparkContext is stopped](#).

If the SparkContext happens to be stopped while doing the waiting, a `IllegalStateException` is thrown with the message:

Spark context stopped while waiting for backend

## Stopping TaskSchedulerImpl (stop method)

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	<a href="#">FIXME</a> How does Spark handle repeated results of speculative tasks since there are copies launched?
---------	--

## Calculating Default Level of Parallelism (`defaultParallelism` method)

**Default level of parallelism** is a hint for sizing jobs. It is a part of the [TaskScheduler contract](#) and used by [SparkContext](#) to create RDDs with the right number of partitions when not specified explicitly.

`TaskSchedulerImpl` uses [SchedulerBackend.defaultParallelism\(\)](#) to calculate the value, i.e. it just passes it along to a scheduler backend.

## Submitting Tasks (using submitTasks)

Note	<code>submitTasks</code> is a part of <a href="#">TaskScheduler Contract</a> .
------	--

```
submitTasks(taskSet: TaskSet): Unit
```

`submitTasks` creates a [TaskSetManager](#) for the input `TaskSet` and adds it to the [Schedulable root pool](#).

Note	The <a href="#">root pool</a> can be a single flat linked queue (in <a href="#">FIFO scheduling mode</a> ) or a hierarchy of pools of <a href="#">Schedulables</a> (in <a href="#">FAIR scheduling mode</a> ).
------	--

It makes sure that the requested resources, i.e. CPU and memory, are assigned to the Spark application for a non-local environment before requesting the current [SchedulerBackend](#) to revive offers.

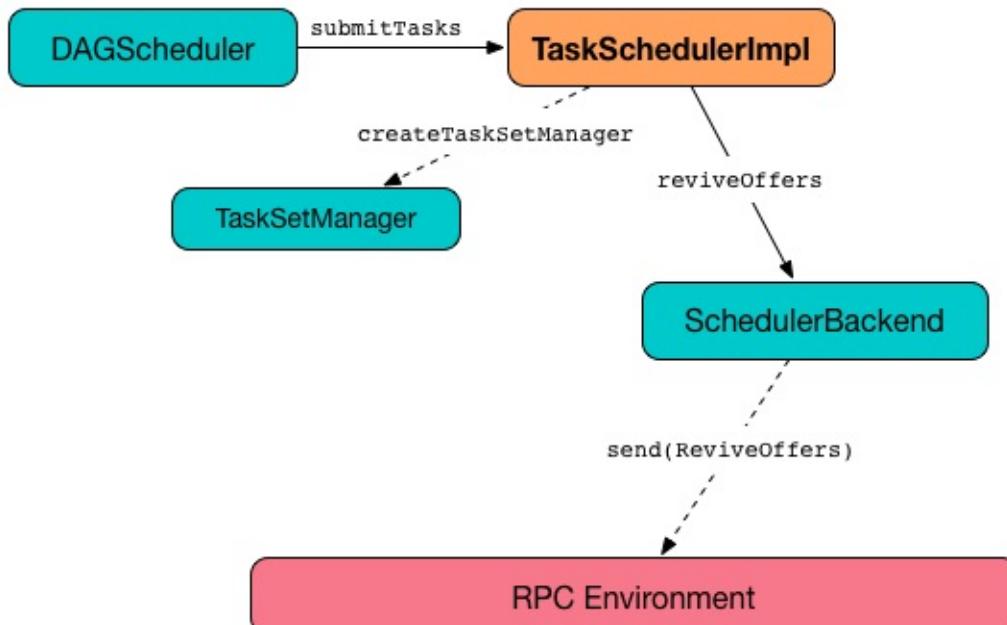


Figure 4. `TaskSchedulerImpl.submitTasks`

Note	If there are tasks to launch for missing partitions in a stage, <code>DAGScheduler</code> executes <code>submitTasks</code> (see <a href="#">submitMissingTasks for Stage and Job</a> ).
------	--

When `submitTasks` 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 input `taskSet` and the [acceptable number of task failures](#).

Note	The acceptable number of task failures is specified when a <a href="#">TaskSchedulerImpl is created</a> .
------	---

Note	A <code>TaskSet</code> knows the tasks to execute (as <code>tasks</code> ) and stage id (as <code>stageId</code> ) the tasks belong to. Read <a href="#">TaskSets</a> .
------	---

The `TaskSet` is registered in the internal [taskSetsByStageIdAndAttempt](#) registry with the [TaskSetManager](#).

If there is more than one active [TaskSetManager](#) for the stage, a `IllegalStateException` is thrown with the message:

```
more than one active taskSet for stage [stage]: [TaskSet ids]
```

Note	<code>TaskSetManager</code> is considered <b>active</b> when it is not a <b>zombie</b> .
------	--

The `TaskSetManager` is [added to the `Schedulable` pool \(via `schedulableBuilder`\)](#).

When the method is called the very first time (`hasReceivedTask` is `false`) in cluster mode only (i.e. `isLocal` of the `TaskSchedulerImpl` is `false`), `starvationTimer` is scheduled to execute after [spark.starvation.timeout](#) to ensure that the requested resources, i.e. CPUs and memory, were assigned by a cluster manager.

Note	After the first <a href="#">spark.starvation.timeout</a> passes, the internal <code>hasReceivedTask</code> flag becomes <code>true</code> .
------	---

Every time the starvation timer thread is executed and `hasLaunchedTask` flag is `false`, the following `WARN` message is printed out to the logs:

```
WARN Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources
```

Otherwise, when the `hasLaunchedTask` flag is `true` the timer thread cancels itself.

Ultimately, `submitTasks` requests the `SchedulerBackend` to [revive offers](#).

Tip	Use <code>dag-scheduler-event-loop</code> thread to step through the code in a debugger.
-----	--

## taskSetsByStageIdAndAttempt Registry

Caution

FIXME

A mapping between stages and a collection of attempt ids and TaskSetManagers.

## Processing Executor Resource Offers (using resourceOffers)

```
resourceOffers(offers: Seq[WorkerOffer]): Seq[Seq[TaskDescription]]
```

`resourceOffers` method is called by [SchedulerBackend](#) (for clustered environments) or [LocalBackend](#) (for local mode) with `WorkerOffer` resource offers that represent cores (CPUs) available on all the active executors with one `WorkerOffer` per active executor.

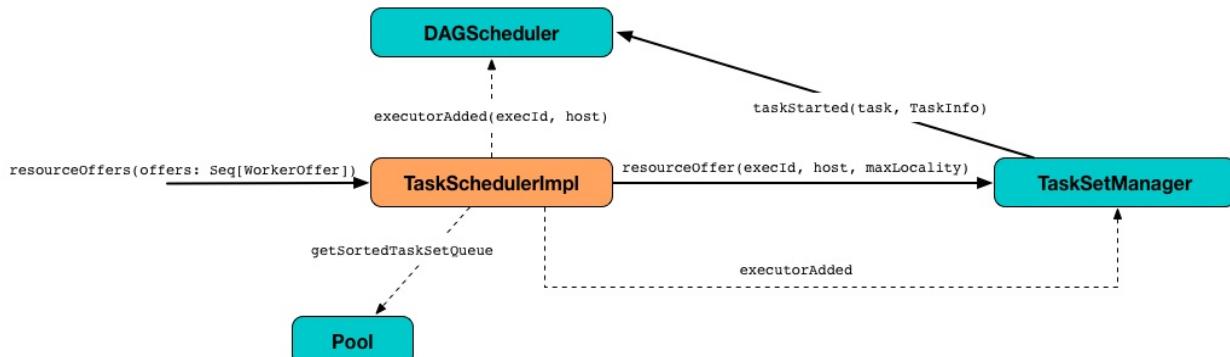


Figure 5. Processing Executor Resource Offers

Note

`resourceOffers` is a mechanism to propagate information about active executors to `TaskSchedulerImpl` with the hosts and racks (if supported by the cluster manager).

A `WorkerOffer` is a 3-tuple with executor id, host, and the number of free cores available.

```
WorkerOffer(executorId: String, host: String, cores: Int)
```

For each `WorkerOffer` (that represents free cores on an executor) `resourceOffers` method records the host per executor id (using the internal `executorIdToHost`) and sets `0` as the number of tasks running on the executor if there are no tasks on the executor (using `executorIdToTaskCount`). It also records hosts (with executors in the internal `executorsByHost` registry).

Warning

FIXME BUG? Why is the executor id **not** added to `executorsByHost` ?

For the offers with a host that has not been recorded yet (in the internal `executorsByHost` registry) the following occurs:

1. The host is recorded in the internal `executorsByHost` registry.
2. `executorAdded` callback is called (with the executor id and the host from the offer).
3. `newExecAvail` flag is enabled (it is later used to inform `TaskSetManagers` about the new executor).

**Caution**

**FIXME** a picture with `executorAdded` call from TaskSchedulerImpl to DAGScheduler.

It shuffles the input `offers` that is supposed to help evenly distributing tasks across executors (that the input `offers` represent) and builds internal structures like `tasks` and `availableCpus`.

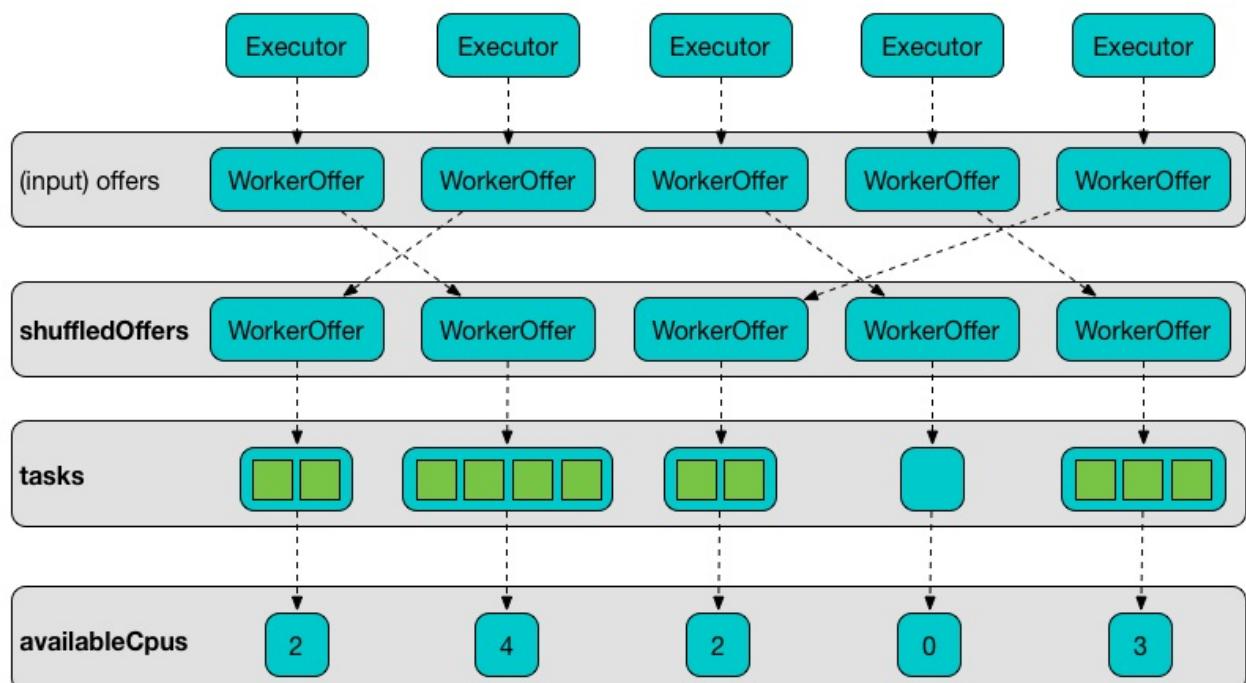


Figure 6. Internal Structures of resourceOffers with 5 WorkerOffers

The root pool is requested for [TaskSetManagers](#) sorted appropriately (according to the [scheduling order](#)).

**Note**

`rootPool` is a part of the [TaskScheduler Contract](#) and is exclusively managed by [SchedulableBuilders](#) (that add `TaskSetManagers` to the root pool).

For every `TaskSetManager` in the `TaskSetManager` sorted queue, the following DEBUG message is printed out to the logs:

```
DEBUG TaskSchedulerImpl: parentName: [taskSet.parent.name], name: [taskSet.name], runningTasks: [taskSet.runningTasks]
```

**Note**

The internal `rootPool` is configured while `TaskSchedulerImpl` is being initialized.

While traversing over the sorted collection of `TaskSetManagers`, if a new host (with an executor) was registered, i.e. the `newExecAvail` flag is enabled, `TaskSetManagers` are informed about the new executor added.

Note	A <code>TaskSetManager</code> will be informed about one or more new executors once per host regardless of the number of executors registered on the host.
------	--

For each `TaskSetManager` (in `sortedTaskSets`) and for each preferred locality level (ascending), `resourceOfferSingleTaskSet` is called until `launchedTask` flag is `false`.

Caution	<code>FIXME</code> <code>resourceOfferSingleTaskSet</code> + the sentence above less code-centric.
---------	--

Check whether the number of cores in an offer is greater than the [number of cores needed for a task](#).

When `resourceOffers` managed to launch a task (i.e. `tasks` collection is not empty), the internal `hasLaunchedTask` flag becomes `true` (that effectively means what the name says "*There were executors and I managed to launch a task*").

`resourceOffers` returns the `tasks` collection.

Note	<code>resourceOffers</code> is called when <code>coarseGrainedSchedulerBackend</code> makes resource offers.
------	--

## resourceOfferSingleTaskSet method

```
resourceOfferSingleTaskSet(
    taskSet: TaskSetManager,
    maxLocality: TaskLocality,
    shuffledOffers: Seq[WorkerOffer],
    availableCpus: Array[Int],
    tasks: Seq[ArrayBuffer[TaskDescription]]): Boolean
```

`resourceOfferSingleTaskSet` is a private helper method that is executed when...

## 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	<code>FIXME</code> Image with the dependencies
---------	--

Tip	Consult <a href="#">Task States</a> in Tasks to learn about the different task states.
Note	The only instance of <code>TaskResultGetter</code> is created while <code>TaskSchedulerImpl</code> is <a href="#">being created</a> (as <code>taskResultGetter</code> ). It requires a <code>SparkEnv</code> and <code>TaskSchedulerImpl</code> . It is stopped when <code>TaskSchedulerImpl</code> 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 the global closure [Serializer](#)).

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	<a href="#">FIXME Review</a> <code>taskSetManager.canFetchMoreResults(serializedData.limit())</code> .
---------	---

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 the global [closure Serializer](#)) and `sparkEnv.blockManager.master.removeBlock(blockId)` is called afterwards.

`TaskSchedulerImpl.handleSuccessfulTask(taskSetManager, tid, result)` is called.

Caution	<a href="#">FIXME</a> What is <code>TaskSchedulerImpl.handleSuccessfulTask</code> 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 the global [closure Serializer](#)).

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	<a href="#">FIXME</a> 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 SchedulerBackend.reviveOffers\(\) called only for lost executors?](#)

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 is likely the result of receiving duplicate task finished status updates)
```

## 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](#).

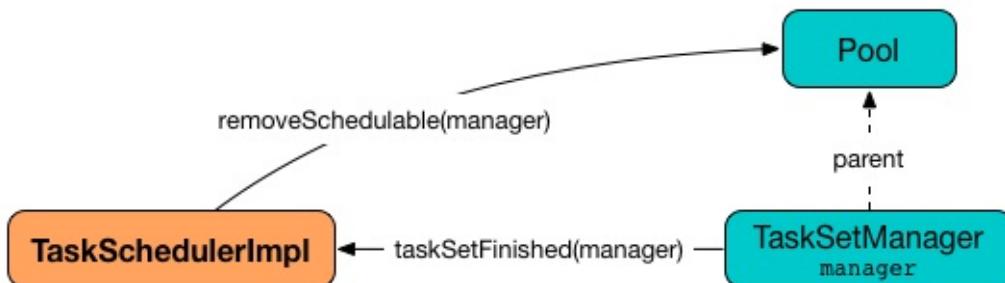
Either way there is an error related to task execution.

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 7. `TaskSchedulerImpl.taskSetFinished` is called when all tasks are finished

Note	<code>taskSetFinished</code> is called by <code>TaskSetManager</code> at the very end of <code>TaskSetManager.handleSuccessfulTask</code> .
------	---

`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 <code>TaskSetManager</code> owns a <code>TaskSet</code> that corresponds to a stage.
------	--

`Pool.removeScheduled(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 [manager.parent.name]
```

## TaskSchedulerImpl.executorAdded

```
executorAdded(execId: String, host: String)
```

`executorAdded` method simply passes the notification on to the `DAGScheduler` (using `DAGScheduler.executorAdded`)

Caution	<a href="#">FIXME</a> Image with a call from <code>TaskSchedulerImpl</code> to <code>DAGScheduler</code> , please.
---------	--

## Internal Registries

Caution	<a href="#">FIXME</a> 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**

`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 a [TaskSetManager](#).

### **spark.task.cpus**

`spark.task.cpus` (default: `1`) sets how many CPUs to request per task.

### **spark.scheduler.mode**

`spark.scheduler.mode` (default: `FIFO`) is a case-insensitive name of the [scheduling mode](#) and can be one of `FAIR`, `FIFO`, or `NONE`.

Note

Only `FAIR` and `FIFO` are supported by `TaskSchedulerImpl`. See [schedulableBuilder](#).

### **spark.speculation.interval**

`spark.speculation.interval` (default: `100ms`) - how often to check for speculative tasks.

### **spark.starvation.timeout**

`spark.starvation.timeout` (default: `15s`) - Threshold above which Spark warns a user that an initial TaskSet may be starved.

## **spark.resultGetter.threads**

`spark.resultGetter.threads` (default: 4) - the number of threads for [TaskResultGetter](#).

# TaskContext

`TaskContext` allows a task to access contextual information about itself as well as register task listeners.

Using `TaskContext` you can access local properties that were set by the driver. You can also access task metrics.

You can access the active `TaskContext` instance using `TaskContext.get` method.

`TaskContext` belongs to `org.apache.spark` package.

```
import org.apache.spark.TaskContext
```

Note

`TaskContext` is serializable.

## Contextual Information

- `stageId` is the id of the stage the task belongs to.
- `partitionId` is the id of the partition computed by the task.
- `attemptNumber` is to denote how many times the task has been attempted (starting from 0).
- `taskAttemptId` is the id of the attempt of the task.
- `isCompleted` returns `true` when a task is completed.
- `isInterrupted` returns `true` when a task was killed.

All these attributes are accessible using appropriate getters, e.g. `getPartitionId` for the partition id.

## Registering Task Listeners

Using `TaskContext` object you can register task listeners for task completion regardless of the final state and task failures only.

### addTaskCompletionListener

`addTaskCompletionListener` registers a `TaskCompletionListener` listener that will be executed on task completion.

**Note**

It will be executed regardless of the final state of a task - success, failure, or cancellation.

```
val rdd = sc.range(0, 5, numSlices = 1)

import org.apache.spark.TaskContext
val printTaskInfo = (tc: TaskContext) => {
    val msg = s"""|-----
                  |partitionId: ${tc.partitionId}
                  |stageId:      ${tc.stageId}
                  |attemptNum:   ${tc.attemptNumber}
                  |taskAttemptId: ${tc.taskAttemptId}
                  |-----""".stripMargin
    println(msg)
}

rdd.foreachPartition { _ =>
    val tc = TaskContext.get
    tc.addTaskCompletionListener(printTaskInfo)
}
```

## addTaskFailureListener

`addTaskFailureListener` registers a `TaskFailureListener` listener that will only be executed on task failure. It can be executed multiple times since a task can be re-attempted when it fails.

```

val rdd = sc.range(0, 2, numSlices = 2)

import org.apache.spark.TaskContext
val printTaskErrorInfo = (tc: TaskContext, error: Throwable) => {
    val msg = s"""|-----
                  |partitionId: ${tc.partitionId}
                  |stageId:      ${tc.stageId}
                  |attemptNum:   ${tc.attemptNumber}
                  |taskAttemptId: ${tc.taskAttemptId}
                  |error:        ${error.toString}
                  |-----""".stripMargin
    println(msg)
}

val throwExceptionForOddNumber = (n: Long) => {
    if (n % 2 == 1) {
        throw new Exception(s"No way it will pass for odd number: $n")
    }
}

// FIXME It won't work.
rdd.map(throwExceptionForOddNumber).foreachPartition { _ =>
    val tc = TaskContext.get
    tc.addTaskFailureListener(printTaskErrorInfo)
}

// Listener registration matters.
rdd.mapPartitions { (it: Iterator[Long]) =>
    val tc = TaskContext.get
    tc.addTaskFailureListener(printTaskErrorInfo)
    it
}.map(throwExceptionForOddNumber).count

```

## Accessing Local Properties (getLocalProperty method)

```
getLocalProperty(key: String): String
```

You can use `getLocalProperty` method to access local properties that were set by the driver using [SparkContext.setLocalProperty](#).

## Task Metrics

```
taskMetrics(): TaskMetrics
```

`taskMetrics` method is part of the Developer API that allows to access the instance of [TaskMetrics](#) for a task.

```
getMetricsSources(sourceName: String): Seq[Source]
```

`getMetricsSources` allows to access all metrics sources for `sourceName` name which are associated with the instance that runs the task.

## Accessing Active TaskContext (TaskContext.get method)

```
get(): TaskContext
```

`TaskContext.get` method returns `TaskContext` instance for the active task (as a `TaskContextImpl` object). There can only be one instance and tasks can use the object to access contextual information about themselves.

```
val rdd = sc.range(0, 3, numSlices = 3)

scala> rdd.partitions.size
res0: Int = 3

rdd.foreach { n =>
    import org.apache.spark.TaskContext
    val tc = TaskContext.get
    val msg = s"""|-----
                  |partitionId: ${tc.partitionId}
                  |stageId: ${tc.stageId}
                  |attemptNum: ${tc.attemptNumber}
                  |taskAttemptId: ${tc.taskAttemptId}
                  |-----""".stripMargin
    println(msg)
}
```

Note

`TaskContext` object uses `ThreadLocal` to keep it thread-local, i.e. to associate state with the thread of a task.

## TaskContextImpl

`TaskContextImpl` is the only implementation of `TaskContext` abstract class.

Caution

FIXME

- stage
- partition
- task attempt

- attempt number
- runningLocally = false
- `taskMemoryManager`

Caution	<a href="#">FIXME</a> Where and how is <code>TaskMemoryManager</code> used?
---------	---

## markInterrupted

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Creating TaskContextImpl Instance

Caution	<a href="#">FIXME</a>
---------	-----------------------

# TaskMemoryManager

`TaskMemoryManager` manages the memory allocated by an [individual task](#).

It assumes that:

- The number of bits to address pages (aka `PAGE_NUMBER_BITS`) is `13`
- The number of bits to encode offsets in data pages (aka `OFFSET_BITS`) is `51` (i.e. `64` bits - `PAGE_NUMBER_BITS`)
- The number of entries in the [page table](#) and [allocated pages](#) (aka `PAGE_TABLE_SIZE`) is `8192` (i.e. `1 << PAGE_NUMBER_BITS`)
- The maximum page size (aka `MAXIMUM_PAGE_SIZE_BYTES`) is `15GB` (i.e. `((1L << 31) - 1) * 8L`)

Note	It is used to create a <a href="#">TaskContextImpl</a> instance.
------	--

Enable `INFO`, `DEBUG` or even `TRACE` logging levels for `org.apache.spark.memory.TaskMemoryManager` logger to see what happens inside.

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

Tip

```
log4j.logger.org.apache.spark.memory.TaskMemoryManager=TRACE
```

Refer to [Logging](#).

Caution	<a href="#">FIXME</a> How to trigger the messages in the logs? What to execute to have them printed out to the logs?
---------	--

[FIXME](#) How to trigger the messages in the logs? What to execute to have them printed out to the logs?

## Creating TaskMemoryManager Instance

```
TaskMemoryManager(MemoryManager memoryManager, long taskAttemptId)
```

A single `TaskMemoryManager` manages the memory of a single task (by the task's `taskAttemptId`).

Note	Although the constructor parameter <code>taskAttemptId</code> refers to a task's attempt id it is really a <code>taskId</code> . It should be changed perhaps?
------	--

Although the constructor parameter `taskAttemptId` refers to a task's attempt id it is really a `taskId`. It should be changed perhaps?

When called, the constructor uses the input `MemoryManager` to know whether it is in **Tungsten memory mode** (disabled by default) and saves the `MemoryManager` and `taskAttemptId` for later use.

It also initializes the internal `consumers` to be empty.

**Note**

When a `TaskRunner` starts running, it creates a new instance of `TaskMemoryManager` for the task by `taskId`. It then assigns the `TaskMemoryManager` to the individual task before it runs.

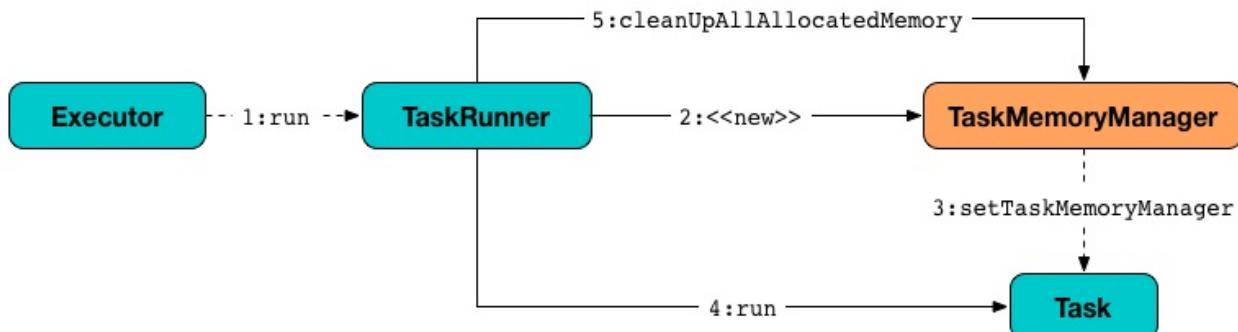


Figure 1. Creating TaskMemoryManager for Task

## Acquire Execution Memory (`acquireExecutionMemory` method)

```
long acquireExecutionMemory(long required, MemoryConsumer consumer)
```

`acquireExecutionMemory` allocates up to `required` size of memory for `consumer`. When no memory could be allocated, it calls `spill` on every consumer, itself including. Finally, it returns the allocated memory.

**Note**

It synchronizes on itself, and so no other calls on the object could be completed.

**Note**

`MemoryConsumer` knows its mode — on- or off-heap.

It first calls `memoryManager.acquireExecutionMemory(required, taskAttemptId, mode)`.

**Tip**

`TaskMemoryManager` is a mere wrapper of `MemoryManager` to track `consumers`?

When the memory obtained is less than requested (by `required`), it requests all `consumers` to `spill` the remaining required memory.

**Note**

It requests memory from consumers that work in the same mode except the requesting one.

You may see the following DEBUG message when `spill` released some memory:

```
DEBUG Task [taskAttemptId] released [bytes] from [consumer] for [consumer]
```

`acquireExecutionMemory` calls `memoryManager.acquireExecutionMemory(required, taskAttemptId, mode)` again (it called it at the beginning).

It does the memory acquisition until it gets enough memory or there are no more consumers to request `spill` from.

You may also see the following ERROR message in the logs when there is an error while requesting `spill` with `outOfMemoryError` followed.

```
ERROR error while calling spill() on [consumer]
```

If the earlier `spill` on the consumers did not work out and there is still not enough memory acquired, `acquireExecutionMemory` calls `spill` on the input `consumer` (that requested more memory!)

If the `consumer` releases some memory, you should see the following DEBUG message in the logs:

```
DEBUG Task [taskAttemptId] released [bytes] from itself ([consumer])
```

`acquireExecutionMemory` calls `memoryManager.acquireExecutionMemory(required, taskAttemptId, mode)` once more.

**Note**

`memoryManager.acquireExecutionMemory(required, taskAttemptId, mode)` could have been called "three" times, i.e. at the very beginning, for each consumer, and on itself.

It records the `consumer` in `consumers` registry.

You should see the following DEBUG message in the logs:

```
DEBUG Task [taskAttemptId] acquired [bytes] for [consumer]
```

**Note**

`acquireExecutionMemory` is called when a `MemoryConsumer` tries to acquires a `memory` and `allocatePage`.

## Getting Page (getPage method)

Caution

FIXME

## Getting Page Offset (getOffsetInPage method)

Caution

FIXME

## Freeing Memory Page (freePage method)

Caution

FIXME

## cleanUpAllAllocatedMemory

It clears [page table](#).

All recorded [consumers](#) are queried for the size of used memory. If the memory used is greater than 0, the following WARN message is printed out to the logs:

```
WARN TaskMemoryManager: leak [bytes] memory from [consumer]
```

The `consumers` collection is then cleared.

[MemoryManager.releaseExecutionMemory](#) is executed to release the memory that is not used by any consumer.

Before `cleanUpAllAllocatedMemory` returns, it calls

[MemoryManager.releaseAllExecutionMemoryForTask](#) that in turn becomes the return value.

Caution

FIXME Image with the interactions to `MemoryManager` .

## Allocating Memory Block for Tungsten Consumers (allocatePage method)

```
MemoryBlock allocatePage(long size, MemoryConsumer consumer)
```

Note

It only handles **Tungsten Consumers**, i.e. [MemoryConsumers](#) in `tungstenMemoryMode` mode.

`allocatePage` allocates a block of memory (aka *page*) smaller than `MAXIMUM_PAGE_SIZE_BYTES` maximum size.

It checks `size` against the internal `MAXIMUM_PAGE_SIZE_BYTES` maximum size. If it is greater than the maximum size, the following `IllegalArgumentException` is thrown:

```
Cannot allocate a page with more than [MAXIMUM_PAGE_SIZE_BYTES] bytes
```

It then [acquires execution memory](#) (for the input `size` and `consumer`).

It finishes by returning `null` when no execution memory could be acquired.

With the execution memory acquired, it finds the smallest unallocated page index and records the page number (using [allocatedPages](#) registry).

If the index is `PAGE_TABLE_SIZE` or higher, [releaseExecutionMemory\(acquired, consumer\)](#) is called and then the following `IllegalStateException` is thrown:

```
Have already allocated a maximum of [PAGE_TABLE_SIZE] pages
```

It then attempts to allocate a `MemoryBlock` from [Tungsten MemoryAllocator](#) (calling `memoryManager.tungstenMemoryAllocator().allocate(acquired)`).

Caution

[FIXME](#) What is `MemoryAllocator` ?

When successful, `MemoryBlock` gets assigned `pageNumber` and it gets added to the internal [pageTable](#) registry.

You should see the following TRACE message in the logs:

```
TRACE Allocate page number [pageNumber] ([acquired] bytes)
```

The `page` is returned.

If a `OutOfMemoryError` is thrown when allocating a `MemoryBlock` page, the following WARN message is printed out to the logs:

```
WARN Failed to allocate a page ([acquired] bytes), try again.
```

And `acquiredButNotUsed` gets `acquired` memory space with the `pageNumber` cleared in [allocatedPages](#) (i.e. the index for `pageNumber` gets `false` ).

Caution

[FIXME](#) Why is the code tracking `acquiredButNotUsed` ?

Another [allocatePage](#) attempt is recursively tried.

Caution

[FIXME](#) Why is there a hope for being able to allocate a page?

## releaseExecutionMemory

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

## Internal Registries

### pageTable

`pageTable` is an internal array of size `PAGE_TABLE_SIZE` with indices being `MemoryBlock` objects.

When [allocating a `MemoryBlock` page for Tungsten consumers](#), the index corresponds to `pageNumber` that points to the `MemoryBlock` page allocated.

### allocatedPages

`allocatedPages` is an internal collection of flags (`true` or `false` values) of size `PAGE_TABLE_SIZE` with all bits initially disabled (i.e. `false`).

Tip	<code>allocatedPages</code> is <a href="#">java.util.BitSet</a> .
-----	---

When [allocatePage](#) is called, it will record the page in the registry by setting the bit at the specified index (that corresponds to the allocated page) to `true`.

### consumers

`consumers` is an internal set of [MemoryConsumers](#).

### acquiredButNotUsed

`acquiredButNotUsed` tracks the size of memory allocated but not used.

### pageSizeBytes method

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

### showMemoryUsage method

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



# MemoryConsumer

`MemoryConsumer` is the contract for memory consumers of `TaskMemoryManager` with support for spilling.

A `MemoryConsumer` basically tracks how much memory is allocated.

Creating a `MemoryConsumer` requires a `TaskMemoryManager` with optional `pageSize` and a `MemoryMode`.

Note

If not specified, `pageSize` defaults to `TaskMemoryManager.pageSizeBytes` and `ON_HEAP` memory mode.

## MemoryConsumer Contract

Caution

`FIXME` the contract

## Memory Allocated (used Registry)

`used` is the amount of memory in use (i.e. allocated) by the `MemoryConsumer`.

### spill method

```
abstract long spill(long size, MemoryConsumer trigger) throws IOException
```

## Deallocate LongArray (freeArray method)

```
void freeArray(LongArray array)
```

`freeArray` deallocates the `LongArray`.

## Deallocate MemoryBlock (freePage method)

```
protected void freePage(MemoryBlock page)
```

`freePage` is a protected method to deallocate the `MemoryBlock`.

Internally, it decrements `used` registry by the size of `page` and frees the page.

## Allocate LongArray (allocateArray method)

```
LongArray allocateArray(long size)
```

allocateArray allocates LongArray of size length.

Internally, it allocates a page for the requested size. The size is recorded in the internal used counter.

However, if it was not possible to allocate the size memory, it shows the current memory usage and a OutOfMemoryError is thrown.

```
Unable to acquire [required] bytes of memory, got [got]
```

## Acquiring Memory (acquireMemory method)

```
long acquireMemory(long size)
```

acquireMemory acquires execution memory of size size. The memory is recorded in used registry.

## TaskMetrics

Caution	<a href="#">FIXME</a>
---------	-----------------------

## incUpdatedBlockStatuses

Caution	<a href="#">FIXME</a>
---------	-----------------------

# 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

`org.apache.spark.scheduler.SchedulerBackend` is a `private[spark]` Scala trait in Spark.

Every `SchedulerBackend` has to follow the following contract:

- Can be started (using `start()`) and stopped (using `stop()`)
- [reviveOffers](#)
- Calculate [default level of parallelism](#)
- [killTask](#)

- Answers `isReady()` to inform whether it is currently started or stopped. 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

## reviveOffers

**Note**

It is used in `TaskSchedulerImpl` using `backend` internal reference when submitting tasks.

There are currently three custom implementations of `reviveoffers` available in Spark for different clustering options:

- For local mode read [Task Submission a.k.a. reviveOffers](#).
- [CoarseGrainedSchedulerBackend](#)
- [MesosFineGrainedSchedulerBackend](#)

## Default Level of Parallelism (`defaultParallelism` method)

```
defaultParallelism(): Int
```

**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.

## killTask

```
killTask(taskId: Long, executorId: String, interruptThread: Boolean)
```

`killTask` throws a `UnsupportedOperationException` by default.

## applicationAttemptId

```
applicationAttemptId(): Option[String] = None
```

`applicationAttemptId` returns the application attempt id of a Spark application.

It is currently only supported by [YARN cluster scheduler backend](#) as the YARN cluster manager supports multiple application attempts.

Note

`applicationAttemptId` is also a part of [TaskScheduler contract](#) and [TaskSchedulerImpl](#) directly calls the SchedulerBackend's `applicationAttemptId`.

## 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](#) (for **client** deploy mode)
    - [YarnClusterSchedulerBackend](#) (for **cluster** deploy mode).
  - [CoarseMesosSchedulerBackend](#)
- [MesosSchedulerBackend](#)



# CoarseGrainedSchedulerBackend

`CoarseGrainedSchedulerBackend` is a [SchedulerBackend](#) and [ExecutorAllocationClient](#).

It is responsible for requesting resources from a cluster manager for executors to be able to launch tasks (on [coarse-grained executors](#)).

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.

When [being created](#), `CoarseGrainedSchedulerBackend` requires a [Task Scheduler](#), and a [RPC Environment](#).

It uses [LiveListenerBus](#).

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`)

**Tip** Enable `INFO` or `DEBUG` logging level for `org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend=DE
```

Refer to [Logging](#).

## Creating CoarseGrainedSchedulerBackend Instance

`CoarseGrainedSchedulerBackend` requires a [task scheduler](#) and a [RPC Environment](#) when being created.

It initializes the following registries:

- `totalCoreCount` to `0`
- `totalRegisteredExecutors` to `0`
- `maxRpcMessageSize` to `spark.rpc.message.maxSize`.
- `_minRegisteredRatio` to `spark.scheduler.minRegisteredResourcesRatio` (between `0` and `1` inclusive).
- `maxRegisteredWaitingTimeMs` to `spark.scheduler.maxRegisteredResourcesWaitingTime`.
- `createTime` to the current time.
- `executorDataMap` to an empty collection.
- `numPendingExecutors` to `0`
- `executorsPendingToRemove` to an empty collection.
- `hostToLocalTaskCount` to an empty collection.
- `localityAwareTasks` to `0`
- `currentExecutorIdCounter` to `0`

It accesses the current [LiveListenerBus](#) and [SparkConf](#) through the constructor's reference to [TaskSchedulerImpl](#).

## Getting Executor Ids (`getExecutorIds` method)

When called, `getExecutorIds` simply returns executor ids from the internal `executorDataMap` registry.

Note

It is called when [SparkContext calculates executor ids](#).

## CoarseGrainedSchedulerBackend Contract

Caution

FIXME

- It can [reset a current internal state to the initial state](#).

## doRequestTotalExecutors

```
doRequestTotalExecutors(requestedTotal: Int): Boolean = false
```

`doRequestTotalExecutors` requests `requestedTotal` executors from a cluster manager. It is a protected method that returns `false` by default (that coarse-grained scheduler backends are supposed to further customize).

## Note

It is called when `coarseGrainedSchedulerBackend` requests additional or total number of executors, or when killing unneeded executors.

In fact, all the aforementioned methods are due to the [ExecutorAllocationClient contract](#) that `CoarseGrainedSchedulerBackend` follows.

## Note

It is customized by the coarse-grained scheduler backends for [YARN](#), [Spark Standalone](#), and [Mesos](#).

## Internal Registries

### currentExecutorIdCounter Counter

`currentExecutorIdCounter` is the last (highest) identifier of all allocated executors.

## Note

It is exclusively used in `YarnSchedulerEndpoint` to respond to `RetrieveLastAllocatedExecutorId` message.

### executorDataMap Registry

```
executorDataMap = new HashMap[String, ExecutorData]
```

`executorDataMap` tracks executor data by executor id.

It uses `ExecutorData` that holds an executor's endpoint reference, address, host, the number of free and total CPU cores, the URL of execution logs.

## Note

A new executor (id, data) pair is added when `DriverEndpoint` receives `RegisterExecutor` message and removed when `DriverEndpoint` receives `RemoveExecutor` message or a remote host (with one or many executors) disconnects.

## numPendingExecutors

Caution	<a href="#">FIXME</a>
---------	-----------------------

## numExistingExecutors

Caution	<a href="#">FIXME</a>
---------	-----------------------

## executorsPendingToRemove

Caution	<a href="#">FIXME</a>
---------	-----------------------

## localityAwareTasks

Caution	<a href="#">FIXME</a>
---------	-----------------------

## hostToLocalTaskCount

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Requesting Additional Executors (`requestExecutors` method)

```
requestExecutors(numAdditionalExecutors: Int): Boolean
```

`requestExecutors` is a "decorator" method that ultimately calls a cluster-specific `doRequestTotalExecutors` method and returns whether the request was acknowledged or not (it is assumed `false` by default).

Note	<code>requestExecutors</code> method is a part of <a href="#">ExecutorAllocationClient Contract</a> that <a href="#">SparkContext</a> uses for requesting additional executors (as a part of a developer API for dynamic allocation of executors).
------	--

When called, you should see the following INFO message followed by DEBUG message in the logs:

```
INFO Requesting [numAdditionalExecutors] additional executor(s) from the cluster manager
DEBUG Number of pending executors is now [numPendingExecutors]
```

The internal `numPendingExecutors` is increased by the input `numAdditionalExecutors`.

`requestExecutors` requests executors from a cluster manager (that reflects the current computation needs). The "new executor total" is a sum of the internal `numExistingExecutors` and `numPendingExecutors` decreased by the number of executors pending to be removed.

If `numAdditionalExecutors` is negative, a `IllegalArgumentException` is thrown:

```
Attempted to request a negative number of additional executor(s) [numAdditionalExecutors] from the cluster manager. Please specify a positive number!
```

Note	It is a final method that no other scheduler backends could customize further.
------	--

Note	The method is a synchronized block that makes multiple concurrent requests be handled in a serial fashion, i.e. one by one.
------	---

## Requesting Exact Number of Executors (`requestTotalExecutors` method)

```
requestTotalExecutors(  
    numExecutors: Int,  
    localityAwareTasks: Int,  
    hostToLocalTaskCount: Map[String, Int]): Boolean
```

`requestTotalExecutors` is a "decorator" method that ultimately calls a cluster-specific `doRequestTotalExecutors` method and returns whether the request was acknowledged or not (it is assumed `false` by default).

Note	<code>requestTotalExecutors</code> is a part of <code>ExecutorAllocationClient Contract</code> that <code>SparkContext</code> uses for requesting the exact number of executors.
------	--

It sets the internal `localityAwareTasks` and `hostToLocalTaskCount` registries. It then calculates the exact number of executors which is the input `numExecutors` and the executors pending removal decreased by the number of already-assigned executors.

If `numExecutors` is negative, a `IllegalArgumentException` is thrown:

```
Attempted to request a negative number of executor(s) [numExecutors] from the cluster manager. Please specify a positive number!
```

Note	It is a final method that no other scheduler backends could customize further.
------	--

Note	The method is a synchronized block that makes multiple concurrent requests be handled in a serial fashion, i.e. one by one.
------	---

## minRegisteredRatio

```
minRegisteredRatio: Double
```

`minRegisteredRatio` returns a ratio between `0` and `1` (inclusive). You can use `spark.scheduler.minRegisteredResourcesRatio` to control the value.

## Starting CoarseGrainedSchedulerBackend (start method)

`start` initializes `CoarseGrainedScheduler` RPC Endpoint.

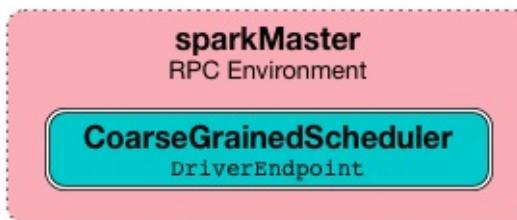


Figure 1. CoarseGrainedScheduler Endpoint

Note	<code>start</code> is part of the <a href="#">SchedulerBackend Contract</a> .
------	---

Note	The RPC Environment is passed on as an constructor parameter.
------	---

## Stopping (stop method)

`stop` method [stops executors](#) and `CoarseGrainedScheduler` RPC endpoint.

Note	<code>stop</code> is part of the <a href="#">SchedulerBackend Contract</a> .
------	--

Note	When called with no <code>driverEndpoint</code> both <code>stop()</code> and <code>stopExecutors()</code> do nothing. <code>driverEndpoint</code> is initialized in <code>start</code> 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.

## Compute Default Level of Parallelism (defaultParallelism method)

The default parallelism is controlled by `spark.default.parallelism` or is at least `2` or `totalCoreCount`.

Note	<code>defaultParallelism</code> is part of the <a href="#">SchedulerBackend Contract</a> .
------	--

## Reviving Offers (`reviveOffers` method)

Note	<code>reviveOffers</code> is part of the <a href="#">SchedulerBackend Contract</a> .
------	--

`reviveOffers` simply sends a [ReviveOffers](#) message to `driverEndpoint` (so it is processed asynchronously, i.e. on a separate thread, later on).

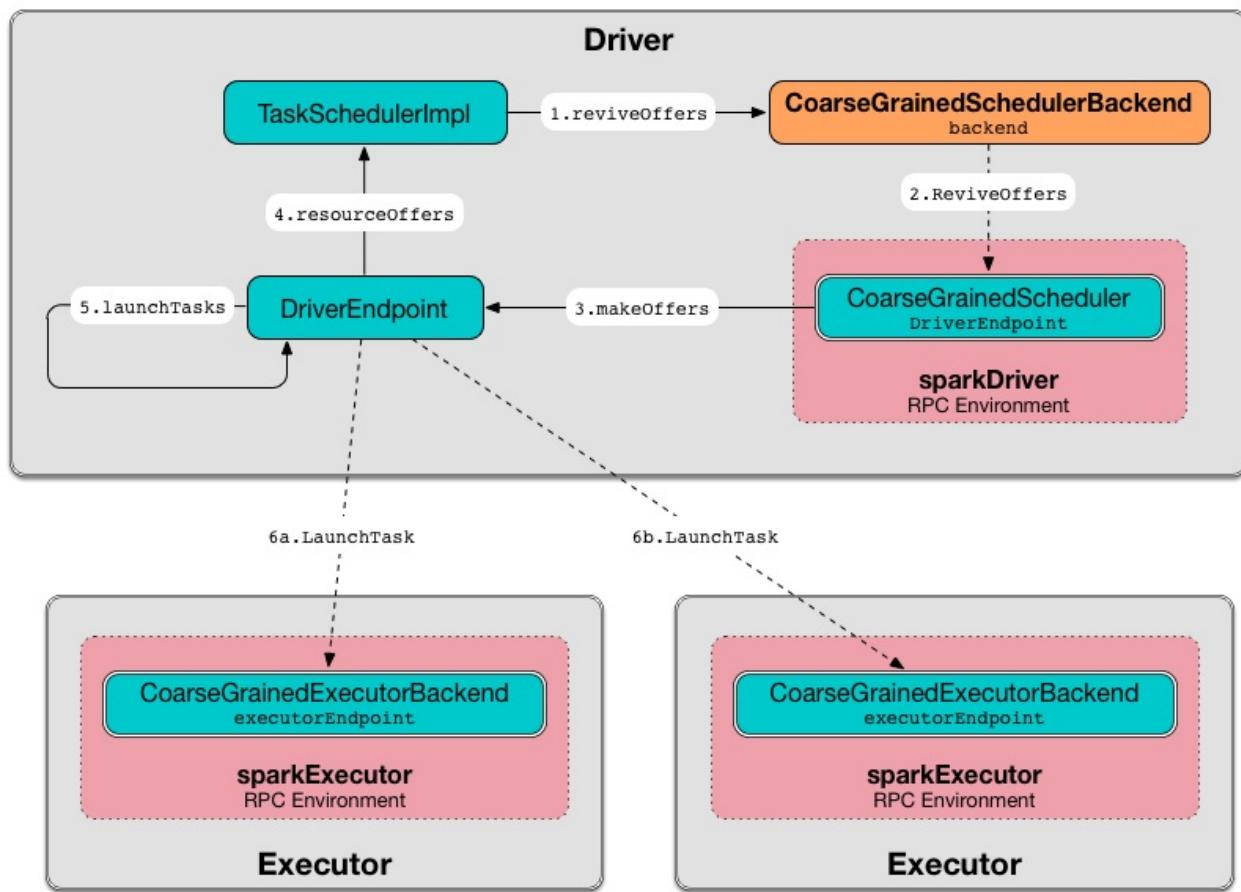


Figure 2. Reviving Offers by CoarseGrainedExecutorBackend

## Killing Task (`killTask` method)

`killTask` simply sends a [KillTask](#) message to `driverEndpoint`.

Caution	<a href="#">FIXME Image</a>
---------	-----------------------------

Note	<code>killTask</code> is part of the <a href="#">SchedulerBackend Contract</a> .
------	--

## Delaying Task Launching (`isReady` method)

`isReady` is a custom implementation of `isReady` from the `SchedulerBackend` Contract that allows to delay task launching until sufficient resources are registered or `spark.scheduler.maxRegisteredResourcesWaitingTime` passes.

Note	<code>isReady</code> is used exclusively by <code>TaskSchedulerImpl.waitBackendReady</code> .
------	---

It starts checking whether there are sufficient resources available (using `sufficientResourcesRegistered` method).

Note	By default <code>sufficientResourcesRegistered</code> always responds that sufficient resources are available.
------	--

If `sufficient resources are available`, you should see the following INFO message in the logs:

INFO SchedulerBackend is ready for scheduling beginning after reached <code>minRegisteredResourcesRatio</code> : [minRegisteredRatio]
---

The method finishes returning `true`.

Note	<code>minRegisteredRatio</code> in the logs above is in the range 0 to 1 (uses <code>spark.scheduler.minRegisteredResourcesRatio</code> ) 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 (as `createTime`) passed `spark.scheduler.maxRegisteredResourcesWaitingTime` to give a way to submit tasks (despite `minRegisteredRatio` not being reached yet).

You should see the following INFO message in the logs:

INFO SchedulerBackend is ready for scheduling beginning after waiting <code>maxRegisteredResourcesWaitingTime</code> : [maxRegisteredWaitingTimeMs] (ms)
---

The method finishes returning `true`.

Otherwise, when `no sufficient resources are available` and `maxRegisteredWaitingTimeMs` has not been passed, it finishes returning `false`.

## sufficientResourcesRegistered

`sufficientResourcesRegistered` always responds that sufficient resources are available.

## Stop All Executors (stopExecutors method)

`stopExecutors` sends a blocking [StopExecutors](#) message to `driverEndpoint` (if already initialized).

Note	It is called exclusively while <code>coarseGrainedSchedulerBackend</code> is <a href="#">being stopped</a> .
------	--

You should see the following INFO message in the logs:

```
INFO CoarseGrainedSchedulerBackend: Shutting down all executors
```

## Reset State (reset method)

`reset` resets the internal state:

1. Sets `numPendingExecutors` to 0
2. Clears `executorsPendingToRemove`
3. Sends a blocking [RemoveExecutor](#) message to `driverEndpoint` for every executor (in the internal `executorDataMap`) to inform it about `SlaveLost` with the message:

```
Stale executor after cluster manager re-registered.
```

`reset` is a method that is defined in `CoarseGrainedSchedulerBackend`, but used and overridden exclusively by [YarnSchedulerBackend](#).

## Remove Executor (removeExecutor method)

```
removeExecutor(executorId: String, reason: ExecutorLossReason)
```

`removeExecutor` sends a blocking [RemoveExecutor](#) message to `driverEndpoint`.

Note	It is called by subclasses <a href="#">SparkDeploySchedulerBackend</a> , <a href="#">CoarseMesosSchedulerBackend</a> , and <a href="#">YarnSchedulerBackend</a> .
------	---

## CoarseGrainedScheduler RPC Endpoint (driverEndpoint)

When [CoarseGrainedSchedulerBackend starts](#), it registers **CoarseGrainedScheduler** RPC endpoint to be the driver's communication endpoint.

Internally, it is a [DriverEndpoint](#) object available as the `driverEndpoint` internal field.

**Note**

`coarseGrainedSchedulerBackend` is created while [SparkContext](#) is being created that in turn lives inside a [Spark driver](#). That explains the name `driverEndpoint` (at least partially).

It is called **standalone scheduler's driver endpoint** internally.

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` simply passes the call on to [makeOffers](#).

**Caution**

[FIXME](#) When is an executor alive? What other states can an executor be in?

### 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, cores, logUrls)
```

Note

`RegisterExecutor` is sent when [CoarseGrainedExecutorBackend \(RPC Endpoint\) starts](#).

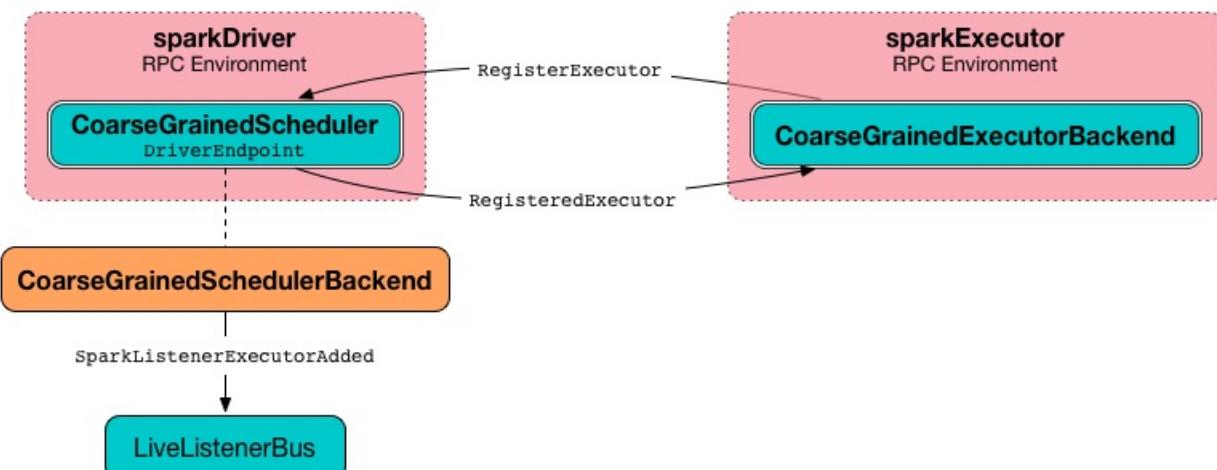


Figure 3. Executor registration (RegisterExecutor RPC message flow)

Only one executor can register under `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](#) to [LiveListenerBus](#).

Ultimately, [makeOffers](#) is called.

## DriverEndpoint

`DriverEndpoint` is a [ThreadSafeRpcEndpoint](#).

### onDisconnected Callback

When called, `onDisconnected` removes the worker from the internal [addressToExecutorId registry](#) (that effectively removes the worker from a cluster).

While removing, it calls [removeExecutor](#) with the reason being `SlaveLost` and message:

Remote RPC client disassociated. Likely due to containers exceeding thresholds, or network issues. Check driver logs for WARN messages.

Note

`onDisconnected` is called when a remote host is lost.

## Making Resource Offers (`makeOffers` method)

```
makeOffers(): Unit
```

`makeOffers` is a private method that takes the active executors (out of the `executorDataMap` internal registry) and creates `WorkerOffer` resource offers for each (one per executor with the executor's id, host and free cores).

Caution

Only free cores are considered in making offers. Memory is not! Why?!

It then requests `TaskSchedulerImpl` to process the resource offers to create a collection of `TaskDescription` collections that it in turn uses to [launch tasks](#).

## Launching Tasks (launchTasks method)

```
launchTasks(tasks: Seq[Seq[TaskDescription]])
```

`launchTasks` is a private helper method that iterates over `TaskDescription` objects in the `tasks` input collection and ...[FIXME](#)

Note

`launchTasks` gets called when `CoarseGrainedSchedulerBackend` is [making resource offers](#).

Internally, it serializes a `TaskDescription` (using the global [closure Serializer](#)) to a serialized task and checks the size of the serialized format of the task so it is less than

`maxRpcMessageSize`.

Caution

[FIXME](#) Describe `maxRpcMessageSize`.

If the serialized task's size is over the maximum RPC message size, the task's `TaskSetManager` [is aborted](#).

Caution

[FIXME](#) At that point, tasks have their executor assigned. When and how did that happen?

If the serialized task's size is correct, the task's executor is looked up in the internal `executorDataMap` registry to record that the task is about to be launched and the number of free cores of the executor is decremented by the `CPUS_PER_TASK` constant (i.e. [spark.task.cpus](#)).

Caution

[FIXME](#) When and how is `spark.task.cpus` set?

Note

`ExecutorData` keeps track of the number of free cores of the executor (as `freeCores`) as well as the `RpcEndpointRef` of the executor to send tasks to launch to (as `executorEndpoint`).

You should see the following INFO in the logs:

```
INFO DriverEndpoint: Launching task [taskId] on executor id: [executorId] hostname: [executorHost].
```

Ultimately, `launchTasks` sends a [LaunchTask](#) message to the executor's RPC endpoint with the serialized task (wrapped in `SerializableBuffer`).

Note	Scheduling in Spark relies on cores only (not memory), i.e. the number of tasks Spark can run on an executor is constrained by the number of cores available only. When submitting Spark application for execution both — memory and cores — can be specified explicitly.
------	---

## Known Implementations

- [StandaloneSchedulerBackend](#)
- [link:spark-mesos-](#)  
[MesosCoarseGrainedSchedulerBackend.adoc](#)[MesosCoarseGrainedSchedulerBackend]

## Settings

### **spark.rpc.message.maxSize**

`spark.rpc.message.maxSize` (default: `128` and not greater than `2047m` - `200k`) for the largest frame size for RPC messages (serialized tasks or task results) in MB.

### **spark.default.parallelism**

`spark.default.parallelism` (default: maximum of `totalCoreCount` and `2`) - [default parallelism](#) for the scheduler backend.

### **spark.scheduler.minRegisteredResourcesRatio**

`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**

`spark.scheduler.maxRegisteredResourcesWaitingTime` (default: `30s`) - the time to wait for sufficient resources available. See [isReady](#).

# Executor Backends

`ExecutorBackend` is a pluggable interface used by `executors` to send status updates about the [different states of a task](#) to a scheduler.

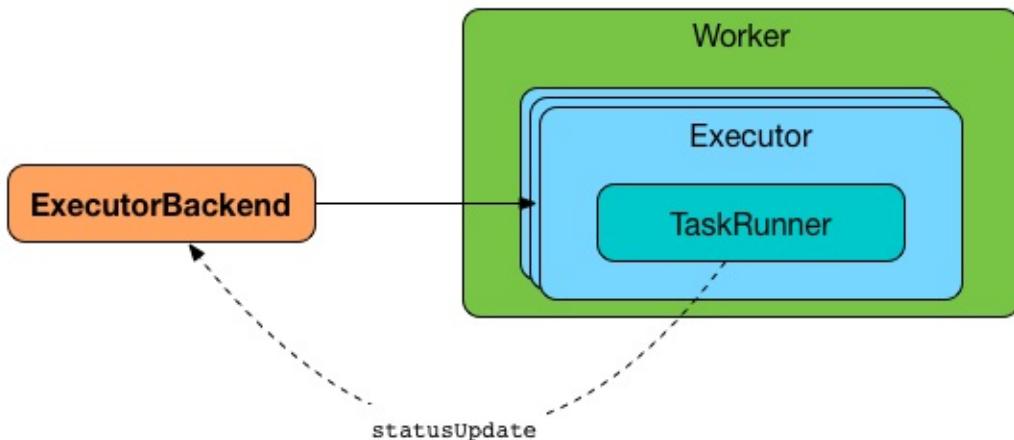


Figure 1. ExecutorBackends work on executors and communicate with driver

Caution

[FIXME](#) What is "a scheduler" in this context?

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` manages a single `executor` object. The internal `executor` object is created after a connection to the driver is established (i.e. after `RegisteredExecutor` has arrived).

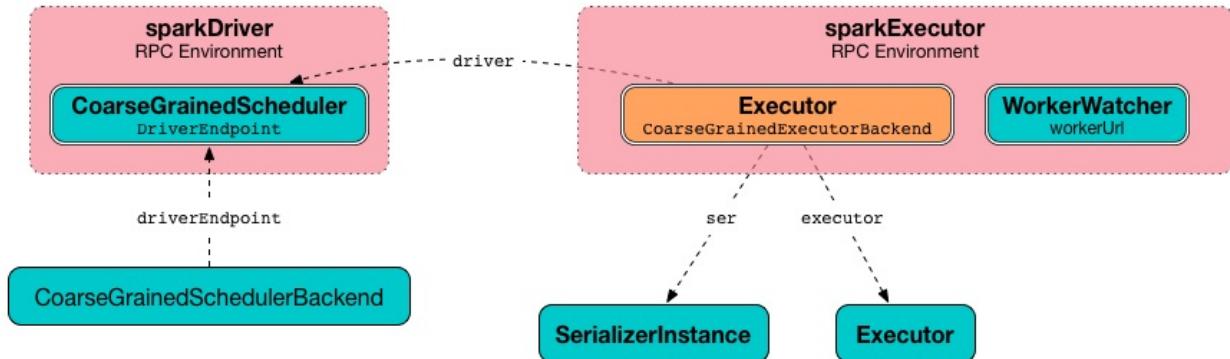


Figure 1. CoarseGrainedExecutorBackend and Others

`CoarseGrainedExecutorBackend` is an `executor backend` for `coarse-grained executors` that live until the executor backend terminates.

`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.

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.
--

All task status updates are sent along to `driverRef` as `StatusUpdate` messages.

Enable `INFO` logging level for `org.apache.spark.executor.CoarseGrainedExecutorBackend` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.executor.CoarseGrainedExecutorBackend=INFO
```

## Starting RpcEndpoint (onStart method)

**Note**

`onStart` is a [RpcEndpoint callback method](#) that is executed before a RPC endpoint starts to handle messages.

When `onStart` is executed, it prints out the following INFO message to the logs:

```
INFO CoarseGrainedExecutorBackend: Connecting to driver: [driverUrl]
```

It then accesses the [RpcEndpointRef](#) for the driver (using the constructor's `driverUrl`) and eventually initializes the internal `driver` that it will send a blocking `RegisterExecutor` message to.

If there is an issue while registering the executor, you should see the following ERROR message in the logs and process exits (with the exit code `1`).

```
ERROR Cannot register with driver: [driverUrl]
```

**Note**

The `RegisterExecutor` message contains `executorId`, the `RpcEndpointRef` to itself, `cores`, and [log URLs](#).

## Extracting Log URLs (extractLogUrls method)

**Caution**

[FIXME](#)

## driver RpcEndpointRef

`driver` is an optional [RpcEndpointRef](#) for the driver.

**Tip**

See [Starting RpcEndpoint \(onStart method\)](#) to learn how it is initialized.

## 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	<a href="#">FIXME</a>
---------	-----------------------

## Usage

Caution	<a href="#">FIXME</a> Where is <code>org.apache.spark.executor.CoarseGrainedExecutorBackend</code> used?
---------	--

It is used in:

- `SparkDeploySchedulerBackend`
- `CoarseMesosSchedulerBackend`
- `SparkClassCommandBuilder` - ???

## start

## stop

## requestTotalExecutors

## executor internal field

`executor` is an `Executor`...[FIXME](#)

Caution	<a href="#">FIXME</a>
---------	-----------------------

## RPC Messages

### RegisteredExecutor

`RegisteredExecutor(hostname)`

When a `RegisteredExecutor` message arrives, you should see the following INFO in the logs:

```
INFO CoarseGrainedExecutorBackend: Successfully registered with driver
```

The internal `executor` is created using `executorId` constructor parameter, with `hostname` that has arrived and others.

**Note**

The message is sent after `CoarseGrainedSchedulerBackend` handles a `RegisterExecutor` message.

## RegisterExecutorFailed

```
RegisterExecutorFailed(message)
```

When a `RegisterExecutorFailed` message arrives, the following ERROR is printed out to the logs:

```
ERROR CoarseGrainedExecutorBackend: Slave registration failed: [message]
```

`CoarseGrainedExecutorBackend` then exits with the exit code `1`.

## LaunchTask

```
LaunchTask(data: SerializableBuffer)
```

The `LaunchTask` handler deserializes `TaskDescription` from `data` (using the global `ClosureSerializer`).

**Note**

`LaunchTask` message is sent by `CoarseGrainedSchedulerBackend.launchTasks`.

```
INFO CoarseGrainedExecutorBackend: Got assigned task [taskId]
```

It then launches the task on the executor (using `Executor.launchTask` method).

If however the internal `executor` field has not been created yet, it prints out the following ERROR to the logs:

```
ERROR CoarseGrainedExecutorBackend: Received LaunchTask command but executor was null
```

And it then exits.

## 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.

# BlockManager

`BlockManager` is a key-value store for blocks of data in Spark. `BlockManager` acts as a local cache that runs on every node in Spark cluster, i.e. the `driver` and `executors`. It provides interface for uploading and fetching blocks both locally and remotely using various stores, i.e. memory, disk, and off-heap. See [Stores](#) in this document.

A `BlockManager` is a [BlockDataManager](#), i.e. manages the storage for blocks that can represent cached RDD partitions, intermediate shuffle outputs, broadcasts, etc. It is also a [BlockEvictionHandler](#) that drops a block from memory and storing it on a disk if applicable.

**Cached blocks** are blocks with non-zero sum of memory and disk sizes.

`BlockManager` is created as a [Spark application starts](#).

A `blockManager` must be [initialized](#) before it is fully operable.

When the [External Shuffle Service](#) is enabled, `BlockManager` uses [ExternalShuffleClient](#) to read other executors' shuffle files.

**Tip** Enable `INFO`, `DEBUG` or `TRACE` logging level for `org.apache.spark.storage.BlockManager` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.storage.BlockManager=TRACE
```

Refer to [Logging](#).

**Tip** You may want to shut off `WARN` messages being printed out about the current state of blocks using the following line to cut the noise:

```
log4j.logger.org.apache.spark.storage.BlockManager=OFF
```

## Using External Shuffle Service (`externalShuffleServiceEnabled` flag)

When the [External Shuffle Service](#) is enabled for a Spark application, `BlockManager` uses [ExternalShuffleClient](#) to read other executors' shuffle files.

**Caution**

**FIXME** How is `shuffleClient` used?

## registerTask

Caution

[FIXME](#)

## 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.

## Storing Block (`putBytes` method)

```
putBytes(
    blockId: BlockId,
    bytes: ChunkedByteBuffer,
    level: StorageLevel,
    tellMaster: Boolean = true): Boolean
```

`putBytes` puts the `blockId` block of `bytes` bytes and `level` storage level to `BlockManager`.

It simply passes the call on to the internal `doPutBytes`.

## doPutBytes

```
def doPutBytes[T](
    blockId: BlockId,
    bytes: ChunkedByteBuffer,
    level: StorageLevel,
    classTag: ClassTag[T],
    tellMaster: Boolean = true,
    keepReadLock: Boolean = false): Boolean
```

`doPutBytes` is an internal method that calls the internal helper `doPut` with `putBody` being a function that accepts a `BlockInfo` and does the uploading.

If the replication storage level is greater than 1, replication starts in a separate thread (using the internal `replicate` method).

**Caution****FIXME** When is replication storage level greater than 1?

For a memory storage level, depending on whether it is a deserialized one or not, `putIteratorAsValues` or `putBytes` of `MemoryStore` are used, respectively. If the put did not succeed and the storage level is also a disk one, you should see the following WARN message in the logs:

```
WARN BlockManager: Persisting block [blockId] to disk instead.
```

`DiskStore.putBytes` is called.

**Note**

`DiskStore` is only used when `MemoryStore` has failed for memory and disk storage levels.

If the storage level is a disk one only, `DiskStore.putBytes` is called.

`doPutBytes` requests `current block status` and if the block was successfully stored, and the driver should know about it (`tellMaster`), it reports current storage status of the block to the driver. The `current TaskContext metrics` are updated with the updated block status.

Regardless of the block being successfully stored or not, you should see the following DEBUG message in the logs:

```
DEBUG BlockManager: Put block [blockId] locally took [time] ms
```

For replication level greater than `1`, `doPutBytes` waits for the earlier asynchronous replication to finish.

The final result of `doPutBytes` is the result of storing the block successful or not (as computed earlier).

## replicate

**Caution****FIXME**

## doPutIterator

**Caution****FIXME**

## doPut

```
doPut[T](
  blockId: BlockId,
  level: StorageLevel,
  classTag: ClassTag[_],
  tellMaster: Boolean,
  keepReadLock: Boolean)(putBody: BlockInfo => Option[T]): Option[T]
```

`doPut` is an internal helper method for `doPutBytes` and `doPutIterator`.

`doPut` executes the input `putBody` function with a `BlockInfo` being a new `BlockInfo` object that `BlockInfoManager` managed to create a lock for writing.

If the block has already been created, the following WARN message is printed out to the logs:

```
WARN Block [blockId] already exists on this machine; not re-adding it
```

It releases the read lock for the block when `keepReadLock` flag is disabled. `doPut` returns `None` immediately.

`putBody` is executed.

If the result of `putBody` is `None` the block is considered saved successfully.

For successful save and `keepReadLock` enabled, `blockInfoManager.downgradeLock(blockId)` is called.

For successful save and `keepReadLock` disabled, `blockInfoManager.unlock(blockId)` is called.

For unsuccessful save, `blockInfoManager.removeBlock(blockId)` is called and the following WARN message is printed out to the logs:

```
WARN Putting block [blockId] failed
```

Ultimately, the following DEBUG message is printed out to the logs:

```
DEBUG Putting block [blockId] [withOrWithout] replication took [usedTime] ms
```

## Removing Block From Memory and Disk (removeBlock method)

```
removeBlock(blockId: BlockId, tellMaster: Boolean = true): Unit
```

`removeBlock` removes the `blockId` block from the [MemoryStore](#) and [DiskStore](#).

When executed, it prints out the following DEBUG message to the logs:

```
DEBUG Removing block [blockId]
```

It requests [BlockInfoManager](#) for lock for writing for the `blockId` block. If it receives none, it prints out the following WARN message to the logs and quits.

```
WARN Asked to remove block [blockId], which does not exist
```

Otherwise, with a write lock for the block, the block is removed from [MemoryStore](#) and [DiskStore](#) (see [Removing Block in MemoryStore](#) and [Removing Block in DiskStore](#) ).

If both removals fail, it prints out the following WARN message:

```
WARN Block [blockId] could not be removed as it was not found in either the disk, memory, or external block store
```

The block is removed from [BlockInfoManager](#).

It then [calculates the current block status](#) that is used to [report the block status to the driver](#) (if the input `tellMaster` and the info's `tellMaster` are both enabled, i.e. `true`) and the [current TaskContext metrics are updated with the change](#).

**Note**

It is used to [remove RDDs](#) and [broadcast](#) as well as in [BlockManagerSlaveEndpoint](#) while handling `RemoveBlock` messages.

## Removing RDD Blocks (`removeRdd` method)

```
removeRdd(rddId: Int): Int
```

`removeRdd` removes all the blocks that belong to the `rddId` RDD.

It prints out the following INFO message to the logs:

```
INFO Removing RDD [rddId]
```

It then requests RDD blocks from [BlockInfoManager](#) and [removes them \(from memory and disk\)](#) (without informing the driver).

The number of blocks removed is the final result.

## Note

It is used by [BlockManagerSlaveEndpoint](#) while handling `RemoveRdd` messages.

## Removing Broadcast Blocks (`removeBroadcast` method)

```
removeBroadcast(broadcastId: Long, tellMaster: Boolean): Int
```

`removeBroadcast` removes all the blocks that belong to the `broadcastId` broadcast.

It prints out the following DEBUG message to the logs:

```
DEBUG Removing broadcast [broadcastId]
```

It then requests all `BroadcastBlockId` objects that belong to the `broadcastId` broadcast from [BlockInfoManager](#) and removes them (from memory and disk).

The number of blocks removed is the final result.

## Note

It is used by [BlockManagerSlaveEndpoint](#) while handling `RemoveBroadcast` messages.

## Getting Block Status (`getStatus` method)

## Caution

## FIXME

## Creating BlockManager Instance

A `BlockManager` needs the following services to be created:

- `executorId` (for the driver and executors)
- [RpcEnv](#)
- [BlockManagerMaster](#)
- `SerializerManager`
- [SparkConf](#)
- [MemoryManager](#)
- [MapOutputTracker](#)
- [ShuffleManager](#)

- [BlockTransferService](#)
- [SecurityManager](#)

**Note** `executorId` is `SparkContext.DRIVER_IDENTIFIER`, i.e. `driver` for the driver and the value of `--executor-id` command-line argument for

[CoarseGrainedExecutorBackend](#) executors or [MesosExecutorBackend](#).

**Caution**

[FIXME](#) Elaborate on the executor backends and executor ids.

When a `BlockManager` instance is created it sets the internal `externalShuffleServiceEnabled` flag to the value of `spark.shuffle.service.enabled` setting.

It creates an instance of [DiskBlockManager](#) (requesting `deleteFilesOnStop` when an external shuffle service is not in use).

It creates an instance of [BlockInfoManager](#) (as `blockInfoManager` ).

It creates **block-manager-future** daemon cached thread pool with 128 threads maximum (as `futureExecutionContext` ).

It creates a [MemoryStore](#) and [DiskStore](#).

[MemoryManager](#) gets the [MemoryStore](#) object assigned.

It requests the current maximum memory from `MemoryManager` (using `maxOnHeapStorageMemory` as `maxMemory` ).

It calculates the port used by the external shuffle service (as `externalShuffleServicePort` ).

**Note**

It is computed specially in Spark on YARN.

**Caution**

[FIXME](#) Describe the YARN-specific part.

It creates a client to read other executors' shuffle files (as `shuffleClient` ). If the external shuffle service is used an [ExternalShuffleClient](#) is created or the input [BlockTransferService](#) is used.

It sets [the maximum number of failures before this block manager refreshes the block locations from the driver](#) (as `maxFailuresBeforeLocationRefresh` ).

It registers [BlockManagerSlaveEndpoint](#) with the input [RpcEnv](#), itself, and [MapOutputTracker](#) (as `slaveEndpoint` ).

**Note**

A `BlockManager` instance is created while [SparkEnv](#) is being created.

## shuffleClient

Caution

FIXME

(that is assumed to be a [ExternalShuffleClient](#))

## shuffleServerId

Caution

FIXME

### Initializing BlockManager (initialize method)

```
initialize(appId: String): Unit
```

`initialize` method is called to initialize the `BlockManager` instance on the driver and executors (see [Creating SparkContext Instance](#) and [Creating Executor Instance](#), respectively).

Note

The method must be called before a `BlockManager` can be considered fully operable.

It does the following:

1. It initializes [BlockTransferService](#).
2. It initializes a shuffle client, be it [ExternalShuffleClient](#) or [BlockTransferService](#).
3. It sets `shuffleServerId` to an instance of `BlockManagerId` given an executor id, host name and port for [BlockTransferService](#).
4. It creates the address of the server that serves this executor's shuffle files (using `shuffleServerId`)

Caution

[FIXME](#) Describe `shuffleServerId`. Where is it used?

If the [External Shuffle Service](#) is used, the following INFO appears in the logs:

```
INFO external shuffle service port = [externalShuffleServicePort]
```

It [registers itself to the driver's BlockManagerMaster](#) passing the `BlockManagerId`, the maximum memory (as `maxMemory`), and the `BlockManagerSlaveEndpoint`.

Ultimately, if the initialization happens on an executor and the [External Shuffle Service](#) is used, it [registers to the shuffle service](#).

## Note

The method is called when the driver is launched (and `SparkContext` is created) and when an Executor is launched.

## Registering Executor's BlockManager with External Shuffle Server (`registerWithExternalShuffleServer` method)

```
registerWithExternalShuffleServer(): Unit
```

`registerWithExternalShuffleServer` is an internal helper method to register the `BlockManager` for an executor with an external shuffle server.

## Note

It is executed when a `BlockManager` is initialized on an executor and an external shuffle service is used.

When executed, you should see the following INFO message in the logs:

```
INFO Registering executor with local external shuffle service.
```

It uses `shuffleClient` to register the block manager using `shuffleServerId` (i.e. the host, the port and the executorId) and a `ExecutorShuffleInfo`.

## Note

The `ExecutorShuffleInfo` uses `localDirs` and `subDirsPerLocalDir` from `DiskBlockManager` and the class name of the constructor `ShuffleManager`.

It tries to register at most 3 times with 5-second sleeps in-between.

## Note

The maximum number of attempts and the sleep time in-between are hard-coded, i.e. they are not configured.

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 waiting 5 seconds...
```

## Re-registering Blocks to Driver (reregister method)

```
reregister(): Unit
```

When is called, you should see the following INFO in the logs:

```
INFO BlockManager: BlockManager re-registering with master
```

It registers itself to the driver's [BlockManagerMaster](#) (just as it was when [BlockManager](#) was initializing). It passes the [BlockManagerId](#), the maximum memory (as `maxMemory`), and the [BlockManagerSlaveEndpoint](#).

**Caution**

**FIXME** Where is `maxMemory` used once passed to the driver?

`reregister` will then report all the local blocks to the [BlockManagerMaster](#).

You should see the following INFO message in the logs:

```
INFO BlockManager: Reporting [blockInfoManager.size] blocks to the master.
```

For each block metadata (in [BlockInfoManager](#)) it gets block current status and tries to send it to the [BlockManagerMaster](#).

If there is an issue communicating to the [BlockManagerMaster](#), you should see the following ERROR message in the logs:

```
ERROR BlockManager: Failed to report [blockId] to master; giving up.
```

After the ERROR message, `reregister` stops reporting.

**Note**

`reregister` is called by [Executor](#) when it was told to re-register while sending heartbeats.

## Calculate Current Block Status ([getCurrentBlockStatus](#) method)

```
getCurrentBlockStatus(blockId: BlockId, info: BlockInfo): BlockStatus
```

`getCurrentBlockStatus` returns the current `BlockStatus` of the `BlockId` block (with the block's current [StorageLevel](#), memory and disk sizes). It uses [MemoryStore](#) and [DiskStore](#) for size and other information.

**Note**

Most of the information to build `BlockStatus` is already in `BlockInfo` except that it may not necessarily reflect the current state per [MemoryStore](#) and [DiskStore](#).

Internally, it uses the input `BlockInfo` to know about the block's storage level. If the storage level is not set (i.e. `null`), the returned `BlockStatus` assumes the default `NONE` storage level and the memory and disk sizes being `0`.

If however the storage level is set, `getCurrentBlockStatus` uses `MemoryStore` or `DiskStore` to check whether the block is stored in the storages or not and request for their sizes in the storages respectively (using their `getSize` or assume `0`).

**Note**

It is acceptable that the `BlockInfo` says to use memory or disk yet the block is not in the storages (yet or anymore). The method will give current status.

**Note**

`getCurrentBlockStatus` is used when executor's `BlockManager` is requested to report the current status of the local blocks to the master, saving a block to a storage or removing a block from memory only or both, i.e. from memory and disk.

## Removing Blocks From Memory Only (`dropFromMemory` method)

```
dropFromMemory(  
    blockId: BlockId,  
    data: () => Either[Array[T], ChunkedByteBuffer]): StorageLevel
```

When `dropFromMemory` is executed, you should see the following INFO message in the logs:

```
INFO BlockManager: Dropping block [blockId] from memory
```

It then asserts that the `blockId` block is `locked for writing`.

If the block's `StorageLevel` uses disks and the internal `DiskStore` object (`diskStore`) does not contain the block, it is saved then. You should see the following INFO message in the logs:

```
INFO BlockManager: Writing block [blockId] to disk
```

**Caution**

**FIXME** Describe the case with saving a block to disk.

The block's memory size is fetched and recorded (using `MemoryStore.getSize`).

The block is `removed from memory` if exists. If not, you should see the following WARN message in the logs:

```
WARN BlockManager: Block [blockId] could not be dropped from memory as it does not exist
```

It then [calculates the current storage status of the block](#) and [reports it to the driver](#). It only happens when `info.tellMaster`.

**Caution**

[\*\*FIXME\*\*](#) When would `info.tellMaster` be `true`?

A block is considered updated when it was written to disk or removed from memory or both. If either happened, the [current TaskContext metrics are updated with the change](#).

Ultimately, `dropFromMemory` returns the current storage level of the block.

**Note**

`dropFromMemory` is part of the single-method [BlockEvictionHandler](#) interface.

## Reporting Current Storage Status of Block to Driver (`reportBlockStatus` method)

```
reportBlockStatus(  
    blockId: BlockId,  
    info: BlockInfo,  
    status: BlockStatus,  
    droppedMemorySize: Long = 0L): Unit
```

`reportBlockStatus` is an internal method for [reporting a block status to the driver](#) and if told to re-register it prints out the following INFO message to the logs:

```
INFO BlockManager: Got told to re-register updating block [blockId]
```

It does asynchronous reregistration (using `asyncReregister`).

In either case, it prints out the following DEBUG message to the logs:

```
DEBUG BlockManager: Told master about block [blockId]
```

**Note**

`reportBlockStatus` is called by [doPutBytes](#), [doPutIterator](#), [dropFromMemory](#), and [removeBlock](#).

## tryToReportBlockStatus

```
def tryToReportBlockStatus(
    blockId: BlockId,
    info: BlockInfo,
    status: BlockStatus,
    droppedMemorySize: Long = 0L): Boolean
```

`tryToReportBlockStatus` is an internal method to report block status to the driver.

It executes `BlockManagerMaster.updateBlockInfo` only if the state changes should be reported to the driver (i.e. `info.tellMaster` is enabled).

It returns `true` or `BlockManagerMaster.updateBlockInfo`'s response.

## BlockEvictionHandler

`BlockEvictionHandler` is a `private[storage]` Scala trait with a single method [dropFromMemory](#).

```
dropFromMemory(
    blockId: BlockId,
    data: () => Either[Array[T], ChunkedByteBuffer]): StorageLevel
```

Note

A `BlockManager` is a `BlockEvictionHandler`.

Note

`dropFromMemory` is called when `MemoryStore` evicts blocks from memory to free space.

## BlockManagerSlaveEndpoint

`BlockManagerSlaveEndpoint` is a [thread-safe RPC endpoint](#) for remote communication between executors and the driver.

Caution

[FIXME](#) the intro needs more love.

While a `BlockManager` is being created so is the `BlockManagerSlaveEndpoint` RPC endpoint with the name **BlockManagerEndpoint[randomId]** to handle [RPC messages](#).

Enable `DEBUG` logging level for `org.apache.spark.storage.BlockManagerSlaveEndpoint` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.storage.BlockManagerSlaveEndpoint=DEBUG
```

Refer to [Logging](#).

## RemoveBlock Message

```
RemoveBlock(blockId: BlockId)
```

When a `RemoveBlock` message comes in, you should see the following DEBUG message in the logs:

```
DEBUG BlockManagerSlaveEndpoint: removing block [blockId]
```

It then calls [BlockManager](#) to remove `blockId` `block`.

**Note** Handling `RemoveBlock` messages happens on a separate thread. See [BlockManagerSlaveEndpoint Thread Pool](#).

When the computation is successful, you should see the following DEBUG in the logs:

```
DEBUG BlockManagerSlaveEndpoint: Done removing block [blockId], response is [response]
```

And `true` `response` is sent back. You should see the following DEBUG in the logs:

```
DEBUG BlockManagerSlaveEndpoint: Sent response: true to [senderAddress]
```

In case of failure, you should see the following ERROR in the logs and the stack trace.

```
ERROR BlockManagerSlaveEndpoint: Error in removing block [blockId]
```

## RemoveRdd Message

```
RemoveRdd(rddId: Int)
```

When a `RemoveRdd` message comes in, you should see the following DEBUG message in the logs:

```
DEBUG BlockManagerSlaveEndpoint: removing RDD [rddId]
```

It then calls [BlockManager](#) to remove `rddId` [RDD](#).

Note

Handling `RemoveRdd` messages happens on a separate thread. See [BlockManagerSlaveEndpoint Thread Pool](#).

When the computation is successful, you should see the following DEBUG in the logs:

```
DEBUG BlockManagerSlaveEndpoint: Done removing RDD [rddId], response is [response]
```

And the number of blocks removed is sent back. You should see the following DEBUG in the logs:

```
DEBUG BlockManagerSlaveEndpoint: Sent response: [#blocks] to [senderAddress]
```

In case of failure, you should see the following ERROR in the logs and the stack trace.

```
ERROR BlockManagerSlaveEndpoint: Error in removing RDD [rddId]
```

## RemoveShuffle Message

```
RemoveShuffle(shuffleId: Int)
```

When a `RemoveShuffle` message comes in, you should see the following DEBUG message in the logs:

```
DEBUG BlockManagerSlaveEndpoint: removing shuffle [shuffleId]
```

If [MapOutputTracker](#) was given (when the RPC endpoint was created), it calls [MapOutputTracker](#) to [unregister](#) the `shuffleId` [shuffle](#).

It then calls [ShuffleManager](#) to [unregister](#) the `shuffleId` [shuffle](#).

Note

Handling `RemoveShuffle` messages happens on a separate thread. See [BlockManagerSlaveEndpoint Thread Pool](#).

When the computation is successful, you should see the following DEBUG in the logs:

```
DEBUG BlockManagerSlaveEndpoint: Done removing shuffle [shuffleId], response is [response]
```

And the result is sent back. You should see the following DEBUG in the logs:

```
DEBUG BlockManagerSlaveEndpoint: Sent response: [response] to [senderAddress]
```

In case of failure, you should see the following ERROR in the logs and the stack trace.

```
ERROR BlockManagerSlaveEndpoint: Error in removing shuffle [shuffleId]
```

## RemoveBroadcast Message

```
RemoveBroadcast(broadcastId: Long)
```

When a `RemoveBroadcast` message comes in, you should see the following DEBUG message in the logs:

```
DEBUG BlockManagerSlaveEndpoint: removing broadcast [broadcastId]
```

It then calls `BlockManager` to remove the `broadcastId` broadcast.

Note

Handling `RemoveBroadcast` messages happens on a separate thread. See [BlockManagerSlaveEndpoint Thread Pool](#).

When the computation is successful, you should see the following DEBUG in the logs:

```
DEBUG BlockManagerSlaveEndpoint: Done removing broadcast [broadcastId], response is [response]
```

And the result is sent back. You should see the following DEBUG in the logs:

```
DEBUG BlockManagerSlaveEndpoint: Sent response: [response] to [senderAddress]
```

In case of failure, you should see the following ERROR in the logs and the stack trace.

```
ERROR BlockManagerSlaveEndpoint: Error in removing broadcast [broadcastId]
```

## GetBlockStatus Message

```
GetBlockStatus(blockId: BlockId)
```

When a `GetBlockStatus` message comes in, it responds with the result of [calling BlockManager about the status of `blockId`](#).

## GetMatchingBlockIds Message

```
GetMatchingBlockIds(filter: BlockId => Boolean)
```

When a `GetMatchingBlockIds` message comes in, it responds with the result of [calling BlockManager for matching blocks for `filter`](#).

## TriggerThreadDump Message

When a `TriggerThreadDump` message comes in, a thread dump is generated and sent back.

## BlockManagerSlaveEndpoint Thread Pool

`BlockManagerSlaveEndpoint` uses **block-manager-slave-async-thread-pool** daemon thread pool (`asyncThreadPool1`) for some messages to talk to other Spark services, i.e. `BlockManager`, [MapOutputTracker](#), [ShuffleManager](#) in a non-blocking, asynchronous way.

The reason for the async thread pool is that the block-related operations might take quite some time and to release the main RPC thread other threads are spawned to talk to the external services and pass responses on to the clients.

Note	<code>BlockManagerSlaveEndpoint</code> uses Java's <a href="#">java.util.concurrent.ThreadPoolExecutor</a> .
------	--

## 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]
```

## 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 `BlockManager` object for the `spark.app.id` id.

## Settings

- `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.
- `spark.block.failures.beforeLocationRefresh` (default: `5`).

# MemoryStore

`MemoryStore` manages blocks (in the internal `entries` registry).

`MemoryStore` requires `SparkConf`, `BlockInfoManager`, `SerializerManager`, `MemoryManager` and `BlockEvictionHandler` to be created.

Caution	<code>FIXME</code> Where are these dependencies used?
---------	---

Caution	<code>FIXME</code> Where is the <code>MemoryStore</code> created? What params provided?
---------	---

Note	<code>MemoryStore</code> is a <code>private[spark]</code> class.
------	--

Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.storage.memory.MemoryStore</code> logger to see what happens inside.</p>
-----	---

Tip	<p>Add the following line to <code>conf/log4j.properties</code> :</p>
-----	---

Tip	<code>log4j.logger.org.apache.spark.storage.memory.MemoryStore=DEBUG</code>
-----	---

Tip	<p>Refer to <a href="#">Logging</a>.</p>
-----	--

## entries Registry

`entries` is Java's `LinkedHashMap` with the initial capacity of `32`, the load factor of `0.75` and `access-order` ordering mode (i.e. iteration is in the order in which its entries were last accessed, from least-recently accessed to most-recently).

Note	<code>entries</code> is Java's <a href="#">java.util.LinkedHashMap</a> .
------	--

## putBytes

```
putBytes[T: ClassTag](
  blockId: BlockId,
  size: Long,
  memoryMode: MemoryMode,
  _bytes: () => ChunkedByteBuffer): Boolean
```

`putBytes` requests `size` memory for the `blockId` block from the current `MemoryManager`. If successful, it registers a `SerializedMemoryEntry` (with the input `_bytes` and `memoryMode`) for `blockId` in the internal `entries` registry.

You should see the following INFO message in the logs:

```
INFO Block [blockId] stored as bytes in memory (estimated size [size], free [bytes])
```

`putBytes` returns `true` after `putBytes` stored `blockId`.

## Evicting Blocks to Free Space

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Removing Block

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Settings

### **spark.storage.unrollMemoryThreshold**

`spark.storage.unrollMemoryThreshold` (default: `1024 * 1024`) controls...

# DiskStore

Caution	<a href="#">FIXME</a>
---------	-----------------------

## putBytes

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Removing Block

Caution	<a href="#">FIXME</a>
---------	-----------------------

# BlockDataManager - Block Storage Management API

`BlockDataManager` is a pluggable [interface](#) to manage storage for blocks (aka *block storage management API*). Blocks are identified by `BlockId` and stored as [ManagedBuffer](#).

Note

`BlockManager` is currently the only available implementation of `BlockDataManager`.

Note

`org.apache.spark.network.BlockDataManager` is a `private[spark]` Scala trait in Spark.

## BlockDataManager Contract

Every `BlockDataManager` offers the following services:

- `getBlockData` to fetch a local block data by `blockId`.

```
getBlockData(blockId: BlockId): ManagedBuffer
```

- `putBlockData` to upload a block data locally by `blockId`. The return value says whether the operation has succeeded (`true`) or failed (`false`).

```
putBlockData(
  blockId: BlockId,
  data: ManagedBuffer,
  level: StorageLevel,
  classTag: ClassTag[_]): Boolean
```

- `releaseLock` is a release lock for `getBlockData` and `putBlockData` operations.

```
releaseLock(blockId: BlockId): Unit
```

## 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`

## ManagedBuffer

# ShuffleClient

ShuffleClient is an interface ( abstract class ) for reading shuffle files.

Note

BlockTransferService, ExternalShuffleClient, MesosExternalShuffleClient are the current implementations of ShuffleClient Contract.

## ShuffleClient Contract

Every ShuffleClient can do the following:

- It can be init . The default implementation does nothing by default.

```
public void init(String appId)
```

- fetchBlocks fetches a sequence of blocks from a remote node asynchronously.

```
public abstract void fetchBlocks(  
    String host,  
    int port,  
    String execId,  
    String[] blockIds,  
    BlockFetchingListener listener);
```

## ExternalShuffleClient

Caution

FIXME

## Register Block Manager with Shuffle Server (registerWithShuffleServer method)

Caution

FIXME

# BlockTransferService

`BlockTransferService` is a contract for specialized [ShuffleClient](#) objects that can [fetch](#) and [upload blocks in synchronously and asynchronously](#).

Note

`BlockTransferService` is a `private[spark]` abstract class .

Note

[NettyBlockTransferService](#) is the only available implementation of [BlockTransferService Contract](#).

## BlockTransferService Contract

Every `BlockTransferService` offers the following:

- `init` that accepts [BlockDataManager](#) for storing or fetching blocks. It is assumed that the method is called before the service is considered fully operational.

```
init(blockDataManager: BlockDataManager): Unit
```

- `port` the service listens to.

```
port: Int
```

- `hostName` the service listens to.

```
hostName: String
```

- `uploadBlock` to upload a block (of `ManagedBuffer` identified by `blockId`) to a remote `hostname` and `port` .

```
uploadBlock(
  hostname: String,
  port: Int,
  execId: String,
  blockId: BlockId,
  blockData: ManagedBuffer,
  level: StorageLevel,
  classTag: ClassTag[_]): Future[Unit]
```

- Synchronous (and hence blocking) `fetchBlockSync` to fetch one block `blockId` (that corresponds to the [ShuffleClient](#) parent's asynchronous [fetchBlocks](#)).

```
fetchBlockSync(  
    host: String,  
    port: Int,  
    execId: String,  
    blockId: String): ManagedBuffer
```

`fetchBlockSync` is a mere wrapper around [fetchBlocks](#) to fetch one `blockId` block that waits until the fetch finishes.

- Synchronous (and hence blocking) `uploadBlockSync` to upload a block (of `ManagedBuffer` identified by `BlockId`) to a remote `hostname` and `port`.

```
uploadBlockSync(  
    hostname: String,  
    port: Int,  
    execId: String,  
    blockId: BlockId,  
    blockData: ManagedBuffer,  
    level: StorageLevel,  
    classTag: ClassTag[_]): Unit
```

`uploadBlockSync` is a mere wrapper around [uploadBlock](#) that waits until the upload finishes.

## NettyBlockTransferService - Netty-Based BlockTransferService

Caution	<a href="#">FIXME</a>
---------	-----------------------

# BlockManagerMaster - BlockManager for Driver

`BlockManagerMaster` runs on the driver and executors.

`BlockManagerMaster` uses `BlockManagerMasterEndpoint` registered under `BlockManagerMaster` RPC endpoint name on the driver (with the endpoint references on executors) to allow executors for sending block status updates to it and hence keep track of block statuses.

Note	An instance of <code>BlockManagerMaster</code> is created in <code>SparkEnv</code> (for the driver and executors), and immediately used to create their <code>BlockManagers</code> .
------	--

	Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.storage.BlockManagerMaster</code> logger to see what happens inside.
--	--

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

Tip	<pre>log4j.logger.org.apache.spark.storage.BlockManagerMaster=INFO</pre>
-----	--

Refer to [Logging](#).

## Creating BlockManagerMaster Instance

An instance of `BlockManagerMaster` requires a `BlockManagerMaster` RPC endpoint reference, `SparkConf`, and the `isDriver` flag to control whether it is created for the driver or executors.

Note	An instance of <code>BlockManagerMaster</code> is created as part of <a href="#">creating an instance of SparkEnv</a> for the driver and executors.
------	---

## Removing Executor (removeExecutor method)

	<code>removeExecutor(execId: String): Unit</code>
--	---

`removeExecutor` posts `RemoveExecutor(execId)` to `BlockManagerMaster` RPC endpoint and waits for a response.

If `false` in response comes in, a `SparkException` is thrown with the following message:

	<code>BlockManagerMasterEndpoint returned false, expected true.</code>
--	--

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

```
INFO BlockManagerMaster: Removed executor [execId]
```

## Removing Block (removeBlock method)

```
removeBlock(blockId: BlockId)
```

```
removeBlock removes blockId block ...FIXME
```

It posts a `RemoveBlock` message to [BlockManagerMaster RPC endpoint](#) and waits for a response.

## Removing RDD Blocks (removeRdd method)

```
removeRdd(rddId: Int, blocking: Boolean)
```

```
removeRdd removes all the blocks of rddId RDD, possibly in a blocking fashion.
```

It posts a `RemoveRdd(rddId)` message to [BlockManagerMaster RPC endpoint](#) on a separate thread.

If there is an issue, you should see the following WARN message in the logs and the entire exception:

```
WARN Failed to remove RDD [rddId] - [exception]
```

If it is a `blocking` operation, it waits for a result for [spark.rpc.askTimeout](#), [spark.network.timeout](#) or `120` secs.

## Removing Shuffle Blocks (removeShuffle method)

```
removeShuffle(shuffleId: Int, blocking: Boolean)
```

```
removeShuffle removes all the blocks of shuffleId shuffle, possibly in a blocking fashion.
```

It posts a `RemoveShuffle(shuffleId)` message to [BlockManagerMaster RPC endpoint](#) on a separate thread.

If there is an issue, you should see the following WARN message in the logs and the entire exception:

```
WARN Failed to remove shuffle [shuffleId] - [exception]
```

If it is a `blocking` operation, it waits for the result for `spark.rpc.askTimeout`, `spark.network.timeout` or `120` secs.

## Removing Broadcast Blocks (removeBroadcast method)

```
removeBroadcast(broadcastId: Long, removeFromMaster: Boolean, blocking: Boolean)
```

`removeBroadcast` removes all the blocks of `broadcastId` broadcast, possibly in a `blocking` fashion.

It posts a `RemoveBroadcast(broadcastId, removeFromMaster)` message to [BlockManagerMaster RPC endpoint](#) on a separate thread.

If there is an issue, you should see the following WARN message in the logs and the entire exception:

```
WARN Failed to remove broadcast [broadcastId] with removeFromMaster = [removeFromMaster] - [exception]
```

If it is a `blocking` operation, it waits for the result for `spark.rpc.askTimeout`, `spark.network.timeout` or `120` secs.

## Stopping BlockManagerMaster (stop method)

```
stop(): Unit
```

`stop` sends a `StopBlockManagerMaster` message to [BlockManagerMaster RPC endpoint](#) and waits for a response.

Note	It is only executed for the driver.
------	-------------------------------------

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

```
INFO BlockManagerMaster: BlockManagerMaster stopped
```

Otherwise, a `SparkException` is thrown.

```
BlockManagerMasterEndpoint returned false, expected true.
```

## Registering BlockManager to Driver (registerBlockManager method)

```
registerBlockManager(  
    blockManagerId: BlockManagerId,  
    maxMemSize: Long,  
    slaveEndpoint: RpcEndpointRef): Unit
```

When `registerBlockManager` runs, you should see the following INFO message in the logs:

```
INFO BlockManagerMaster: Trying to register BlockManager
```

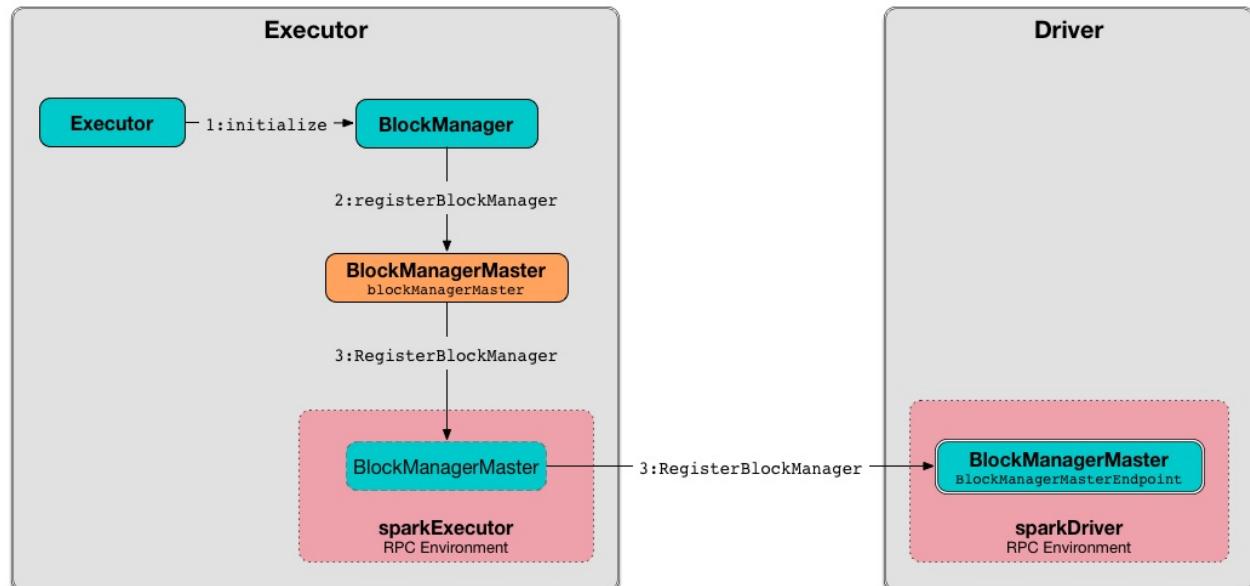


Figure 1. Registering BlockManager with the Driver

It then informs the driver about the new `BlockManager` by sending `RegisterBlockManager` to `BlockManagerMaster RPC endpoint` and waiting for a response.

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

```
INFO BlockManagerMaster: Registered BlockManager
```

Otherwise, a `SparkException` is thrown.

```
BlockManagerMasterEndpoint returned false, expected true.
```

**Note**

`registerBlockManager` is called while [BlockManager is being initialized](#) (on the driver and executors) and while [re-registering blocks to the driver](#).

## Sending UpdateBlockInfo to Driver (updateBlockInfo method)

```
updateBlockInfo(  
    blockManagerId: BlockManagerId,  
    blockId: BlockId,  
    storageLevel: StorageLevel,  
    memSize: Long,  
    diskSize: Long): Boolean
```

`updateBlockInfo` sends a `UpdateBlockInfo` message to [BlockManagerMaster RPC endpoint](#) and waits for a response.

You should see the following DEBUG message in the logs:

```
DEBUG BlockManagerMaster: Updated info of block [blockId]
```

The response from the BlockManagerMaster RPC endpoint is returned.

## Get Block Locations of One Block (getLocations method)

```
getLocations(blockId: BlockId): Seq[BlockManagerId]
```

`getLocations` posts `GetLocations(blockId)` message to [BlockManagerMaster RPC endpoint](#) and waits for a response which becomes the return value.

## Get Block Locations for Multiple Blocks (getLocations method)

```
getLocations(blockIds: Array[BlockId]): IndexedSeq[Seq[BlockManagerId]]
```

`getLocations` posts `GetLocationsMultipleBlockIds(blockIds)` message to [BlockManagerMaster RPC endpoint](#) and waits for a response which becomes the return value.

## getPeers

```
getPeers(blockManagerId: BlockManagerId): Seq[BlockManagerId]
```

`getPeers` posts `GetPeers(blockManagerId)` message [BlockManagerMaster RPC endpoint](#) and waits for a response which becomes the return value.

## getExecutorEndpointRef

```
getExecutorEndpointRef(executorId: String): Option[RpcEndpointRef]
```

`getExecutorEndpointRef` posts `GetExecutorEndpointRef(executorId)` message to [BlockManagerMaster RPC endpoint](#) and waits for a response which becomes the return value.

## getMemoryStatus

```
getMemoryStatus: Map[BlockManagerId, (Long, Long)]
```

`getMemoryStatus` posts a `GetMemoryStatus` message [BlockManagerMaster RPC endpoint](#) and waits for a response which becomes the return value.

## getStorageStatus

```
getStorageStatus: Array[StorageStatus]
```

`getStorageStatus` posts a `GetStorageStatus` message to [BlockManagerMaster RPC endpoint](#) and waits for a response which becomes the return value.

## getBlockStatus

```
getBlockStatus(  
  blockId: BlockId,  
  askSlaves: Boolean = true): Map[BlockManagerId, BlockStatus]
```

`getBlockStatus` posts a `GetBlockStatus(blockId, askSlaves)` message to [BlockManagerMaster RPC endpoint](#) and waits for a response (of type `Map[BlockManagerId, Future[Option[BlockStatus]]]`).

It then builds a sequence of future results that are `BlockStatus` statuses and waits for a result for `spark.rpc.askTimeout`, `spark.network.timeout` or `120` secs.

No result leads to a `SparkException` with the following message:

```
BlockManager returned null for BlockStatus query: [blockId]
```

## getMatchingBlockIds

```
getMatchingBlockIds(  
    filter: BlockId => Boolean,  
    askSlaves: Boolean): Seq[BlockId]
```

`getMatchingBlockIds` posts a `GetMatchingBlockIds(filter, askSlaves)` message to [BlockManagerMaster RPC endpoint](#) and waits for a response which becomes the result for `spark.rpc.askTimeout`, `spark.network.timeout` or `120` secs.

## hasCachedBlocks

```
hasCachedBlocks(executorId: String): Boolean
```

`hasCachedBlocks` posts a `HasCachedBlocks(executorId)` message to [BlockManagerMaster RPC endpoint](#) and waits for a response which becomes the result.

## BlockManagerMasterEndpoint - BlockManagerMaster RPC Endpoint

`BlockManagerMasterEndpoint` is the RPC endpoint for [BlockManagerMaster](#) on the driver (aka master node) to track statuses of the block managers on executors.

Note

It is used to register the `BlockManagerMaster` RPC endpoint when [creating SparkEnv](#).

Enable `INFO` logging level for `org.apache.spark.storage.BlockManagerMasterEndpoint` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.storage.BlockManagerMasterEndpoint=INFO
```

Refer to [Logging](#).

## Internal Registries

### blockLocations

`blockLocations` is a collection of `BlockId` and its locations (as `BlockManagerId`).

**Note** It is used in `removeRdd` to remove blocks for a RDD, `removeBlockManager` to remove blocks after a BlockManager gets removed, `removeBlockFromWorkers`, `updateBlockInfo`, and [getLocations](#).

## RemoveExecutor

```
RemoveExecutor(execId: String)
```

When `RemoveExecutor` is received, `executor` `execId` is removed and the response `true` sent back.

## GetLocations

```
GetLocations(blockId: BlockId)
```

When `GetLocations` comes in, the internal [getLocations](#) method is executed and the result becomes the response sent back.

**Note** `GetLocations` is used to get the block locations of a single block.

## RegisterBlockManager

```
RegisterBlockManager(
    blockManagerId: BlockManagerId,
    maxMemSize: Long,
    sender: RpcEndpointRef)
```

When `RegisterBlockManager` is received, the internal `register` method is executed.

<b>Note</b>	<code>RegisterBlockManager</code> is used to register a <code>BlockManager</code> to the driver.
-------------	--

## register

```
register(id: BlockManagerId, maxMemSize: Long, slaveEndpoint: RpcEndpointRef): Unit
```

`register` records the current time and registers `BlockManager` by `id` if it has not been already registered (using the internal `blockManagerInfo` registry).

Registering a `BlockManager` can only happen once for an executor (identified by `BlockManagerId.executorId` using the internal `blockManagerIdByExecutor` registry).

If another `BlockManager` has earlier been registered for the executor, you should see the following ERROR message in the logs:

```
ERROR Got two different block manager registrations on same executor - will replace old one [oldId] with new one [id]
```

And then `executor is removed`.

You should see the following INFO message in the logs:

```
INFO Registering block manager [hostPort] with [bytes] RAM, [id]
```

The `BlockManager` is recorded in the internal registries: `blockManagerIdByExecutor` and `blockManagerInfo`.

<b>Caution</b>	<b>FIXME</b> Why does <code>blockManagerInfo</code> require a new <code>System.currentTimeMillis()</code> since time was already recorded?
----------------	--

In either case, `SparkListenerBlockManagerAdded(time, id, maxMemSize)` is posted to `listenerBus`.

<b>Note</b>	The method can only be executed on the driver where <code>listenerBus</code> is available.
-------------	--

Caution

[FIXME](#) Describe `listenerBus` + omnigraffle it.

## Other RPC Messages

- `UpdateBlockInfo`
- `GetLocationsMultipleBlockIds`
- `GetPeers`
- `GetRpcHostPortForExecutor`
- `GetMemoryStatus`
- `GetStorageStatus`
- `GetBlockStatus`
- `GetMatchingBlockIds`
- `RemoveRdd`
- `RemoveShuffle`
- `RemoveBroadcast`
- `RemoveBlock`
- `StopBlockManagerMaster`
- `BlockManagerHeartbeat`
- `HasCachedBlocks`

## Removing Executor (`removeExecutor` method)

```
removeExecutor(execId: String)
```

When executed, `removeExecutor` prints the following INFO message to the logs:

```
INFO BlockManagerMasterEndpoint: Trying to remove executor [execId] from BlockManagerMaster.
```

If the `execId` executor is found in the internal `blockManagerIdByExecutor` registry, [the BlockManager for the executor is removed](#).

## Removing BlockManager (removeBlockManager method)

```
removeBlockManager(blockManagerId: BlockManagerId)
```

When executed, `removeBlockManager` looks up `blockManagerId` and removes the executor it was working on from the internal `blockManagerIdByExecutor` as well as from `blockManagerInfo`.

Note	It is a private helper method that is exclusively used while <a href="#">removing an executor</a> .
------	---

It then goes over all the blocks for the `BlockManager`, and removes the executor for each block from `blockLocations` registry.

`SparkListenerBlockManagerRemoved(System.currentTimeMillis(), blockManagerId)` is posted to `listenerBus`.

You should then see the following INFO message in the logs:

```
INFO BlockManagerMasterEndpoint: Removing block manager [blockManagerId]
```

## Get Block Locations (getLocations method)

```
getLocations(blockId: BlockId): Seq[BlockManagerId]
```

When executed, `getLocations` looks up `blockId` in the `blockLocations` internal registry and returns the locations (as a collection of `BlockManagerId`) or an empty collection.

# BlockInfoManager

`BlockInfoManager` manages **memory blocks** (aka *memory pages*). It controls concurrent access to memory blocks by **read** and **write** locks (for existing and **new ones**).

Note

**Locks** are the mechanism to control concurrent access to data and prevent destructive interaction between operations that use the same resource.

Note

`BlockInfoManager` is a `private[storage]` class that belongs to `org.apache.spark.storage` package.

Tip

Enable `TRACE` logging level for `org.apache.spark.storage.BlockInfoManager` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.storage.BlockInfoManager=TRACE
```

Refer to [Logging](#).

## Obtaining Read Lock (`lockForReading` method)

```
lockForReading(  
    blockId: BlockId,  
    blocking: Boolean = true): Option[BlockInfo]
```

`lockForReading` locks `blockId` memory block for reading when the block was registered earlier and no writer tasks use it.

When executed, `lockForReading` prints out the following `TRACE` message to the logs:

```
TRACE BlockInfoManager: Task [currentTaskAttemptId] trying to acquire read lock for [b  
lockId]
```

It looks up the metadata (in `infos` registry).

If no metadata could be found, it returns `None` which means that the block does not exist or was removed (and anybody could acquire a write lock).

Otherwise, when the metadata was found, i.e. registered, it checks so-called `writerTask`.

Only when the block has no writer tasks, a read lock can be acquired (i.e.

`BlockInfo.writerTask` is `BlockInfo.NO_WRITER`). If so, the `readerCount` of the block

metadata is incremented and the block is recorded in the internal `readLocksByTask` registry. You should see the following TRACE message in the logs:

```
TRACE BlockInfoManager: Task [taskAttemptId] acquired read lock for [blockId]
```

The `BlockInfo` for the `blockId` block is returned.

**Note**

`-1024` is a special `taskAttemptId` used to mark a non-task thread, e.g. by a driver thread or by unit test code.

For blocks with `writerTask` other than `NO_WRITER`, when `blocking` is enabled, `lockForReading` waits (until another thread invokes the `Object.notify` method or the `Object.notifyAll` methods for this object).

With `blocking` enabled, it will repeat the waiting-for-read-lock sequence until either `None` or the lock is obtained.

When `blocking` is disabled and the lock could not be obtained, `None` is returned immediately.

**Note**

`lockForReading` is a `synchronized` method, i.e. no two objects can use this and other instance methods.

## Obtaining Write Lock (`lockForWriting` method)

```
lockForWriting(  
    blockId: BlockId,  
    blocking: Boolean = true): Option[BlockInfo]
```

When executed, `lockForWriting` prints out the following TRACE message to the logs:

```
TRACE Task [currentTaskAttemptId] trying to acquire write lock for [blockId]
```

It looks up `blockId` in the internal `infos` registry. When no `BlockInfo` could be found, `None` is returned. Otherwise, `BlockInfo` is checked for `writerTask` to be `BlockInfo.NO_WRITER` with no readers (i.e. `readerCount` is `0`) and only then the lock is returned.

When the write lock can be returned, `BlockInfo.writerTask` is set to `currentTaskAttemptId` and a new binding is added to the internal `writeLocksByTask` registry. You should see the following TRACE message in the logs:

```
TRACE Task [currentTaskAttemptId] acquired write lock for [blockId]
```

If, for some reason, `blockId` has a writer (i.e. `info.writerTask` is not `BlockInfo.NO_WRITER`) or the number of readers is positive (i.e. `BlockInfo.readerCount` is greater than `0`), the method will wait (based on the input `blocking` flag) and attempt the write lock acquisition process until it finishes with a write lock.

**Note**

(deadlock possible) The method is `synchronized` and can block, i.e. `wait` that causes the current thread to wait until another thread invokes `Object.notify` or `Object.notifyAll` methods for this object.

`lockForWriting` return `None` for no `blockId` in the internal `infos` registry or when `blocking` flag is disabled and the write lock could not be acquired.

## Obtaining Write Lock for New Block (`lockNewBlockForWriting` method)

```
lockNewBlockForWriting(  
    blockId: BlockId,  
    newBlockInfo: BlockInfo): Boolean
```

`lockNewBlockForWriting` obtains a write lock for `blockId` but only when the method could register the block.

**Note**

`lockNewBlockForWriting` is similar to `lockForWriting` method but for brand new blocks.

When executed, `lockNewBlockForWriting` prints out the following TRACE message to the logs:

```
TRACE Task [currentTaskAttemptId] trying to put [blockId]
```

If [some other thread has already created the block](#), it finishes returning `false`. Otherwise, when the block does not exist, `newBlockInfo` is recorded in the internal `infos` registry and [the block is locked for this client for writing](#). It then returns `true`.

**Note**

`lockNewBlockForWriting` executes itself in `synchronized` block so once the `BlockInfoManager` is locked the other internal registries should be available only for the currently-executing thread.

## Unlocking Memory Block (`unlock` method)

**Caution****FIXME**

## Releasing All Locks Obtained by Task (`releaseAllLocksForTask` method)

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Removing Memory Block (`removeBlock` method)

Caution	<a href="#">FIXME</a>
---------	-----------------------

## `assertBlockIsLockedForWriting`

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Internal Registries

### infos

`infos` is used to track [BlockInfo](#) per block (identified by [BlockId](#)).

### `readLocksByTask`

`readLocksByTask` is used to track tasks (by [TaskAttemptId](#)) and the blocks they locked for reading (identified by [BlockId](#))

### `writeLocksByTask`

`writeLocksByTask` is used to track tasks (by [TaskAttemptId](#)) and the blocks they locked for writing (identified by [BlockId](#)).

# BlockInfo — Metadata of Memory Block

`BlockInfo` is a metadata of [memory block](#) (aka *memory page*) — the memory block's [size](#), the [number of readers](#) and the [writer task's id](#).

It has a [StorageLevel](#), [ClassTag](#) and [tellMaster](#) flag.

## Size (size attribute)

`size` attribute is the size of the memory block. It starts with `0`.

It represents the number of bytes that [BlockManager saved](#) or [BlockManager.doPutIterator](#).

## Reader Count (readerCount attribute)

`readerCount` attribute is the number of readers of the memory block. It starts with `0`.

It is incremented when a [read lock is acquired](#) and decreases when the following happens:

- The [memory block is unlocked](#)
- All locks for the memory block obtained by a task are released.
- The [memory block is removed](#)
- Clearing the current state of [BlockInfoManager](#).

## Writer Task (writerTask attribute)

`writerTask` attribute is the task that owns the write lock for the memory block.

A writer task can be one of the three possible identifiers:

- `NO_WRITER` (i.e. `-1`) to denote no writers and hence no write lock in use.
- `NON_TASK_WRITER` (i.e. `-1024`) for non-task threads, e.g. by a driver thread or by unit test code.
- the task attempt id of the task which currently holds the write lock for this block.

The writer task is assigned in the following scenarios:

- A [write lock is requested for a memory block \(with no writer and readers\)](#)
- A [memory block is unlocked](#)

- All locks obtained by a task are released
- A memory block is removed
- Clearing the current state of `BlockInfoManager`

# Dynamic Allocation (of Executors)

Tip

See the excellent slide deck [Dynamic Allocation in Spark](#) from Databricks.

**Dynamic Allocation** is a Spark feature to add or remove executors dynamically based on the workload.

It is enabled by `spark.dynamicAllocation.enabled` setting (with no `--num-executors` command-line option or `spark.executor.instances` setting set to zero). When enabled it requires that the [External Shuffle Service](#) is also used (by default it is not as controlled by `spark.shuffle.service.enabled`).

`ExecutorAllocationManager` is the class responsible for dynamic allocation. When `enabled`, it is [started when the Spark context is initialized](#).

Dynamic allocation reports the current state using `ExecutorAllocationManager` [metric source](#).

- 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

## Programmable Dynamic Allocation

`SparkContext` offers a [developer API](#) to scale executors up or down.

## Utils.isDynamicAllocationEnabled method

```
isDynamicAllocationEnabled(conf: SparkConf): Boolean
```

`isDynamicAllocationEnabled` returns `true` if all the following conditions hold:

1. `spark.executor.instances` is `0`
2. `spark.dynamicAllocation.enabled` is enabled
3. Spark on cluster is used (`spark.master` is non-`local`)

Otherwise, it returns `false`.

Note	<code>isDynamicAllocationEnabled</code> returns <code>true</code> , i.e. dynamic allocation is enabled, in Spark local (pseudo-cluster) for testing only (with <code>spark.dynamicAllocation.testing</code> enabled).
------	---

Internally, `isDynamicAllocationEnabled` reads `spark.executor.instances` (assumes `0`) and `spark.dynamicAllocation.enabled` setting (assumes `false`).

If the value of `spark.executor.instances` is not `0` and `spark.dynamicAllocation.enabled` is enabled, `isDynamicAllocationEnabled` prints the following WARN message to the logs:

	WARN Utils: Dynamic Allocation and num executors both set, thus dynamic allocation disabled.
--	--

Note	<code>isDynamicAllocationEnabled</code> is used when Spark calculates the initial number of executors for <b>coarse-grained scheduler backends</b> for <b>YARN</b> , <b>Spark Standalone</b> , and <b>Mesos</b> . It is also used for <b>Spark Streaming</b> .
------	--

Tip	<p>Enable <code>WARN</code> logging level for <code>org.apache.spark.util.Utils</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.util.Utils=WARN</pre> <p>Refer to <a href="#">Logging</a>.</p>
-----	---

## Validating Setting (`validateSettings` method)

<code>validateSettings(): Unit</code>
---------------------------------------

`validateSettings` is an internal method to ensure that the `settings` for dynamic allocation are correct.

It validates the following and throws a `SparkException` if set incorrectly.

1. `spark.dynamicAllocation.minExecutors` must be positive.

2. `spark.dynamicAllocation.minExecutors` must be less than or equal to `spark.dynamicAllocation.maxExecutors`.
3. `spark.dynamicAllocation.maxExecutors`,  
`spark.dynamicAllocation.schedulerBacklogTimeout`,  
`spark.dynamicAllocation.sustainedSchedulerBacklogTimeout`, and  
`spark.dynamicAllocation.executorIdleTimeout` must all be greater than `0`.
4. `spark.shuffle.service.enabled` must be enabled.
5. `spark.executor.cores` must not be less than `spark.task.cpus`.

## Settings

### **spark.dynamicAllocation.enabled**

`spark.dynamicAllocation.enabled` (default: `false`) controls whether dynamic allocation is enabled. It requires that `spark.executor.instances` is `0` (which is the default value).

### **spark.dynamicAllocation.minExecutors**

`spark.dynamicAllocation.minExecutors` (default: `0`) sets the minimum number of executors for dynamic allocation.

It must be positive and less than or equal to `spark.dynamicAllocation.maxExecutors`.

### **spark.dynamicAllocation.maxExecutors**

`spark.dynamicAllocation.maxExecutors` (default: `Integer.MAX_VALUE`) sets the maximum number of executors for dynamic allocation.

It must be greater than `0` and greater than or equal to `spark.dynamicAllocation.minExecutors`.

### **spark.dynamicAllocation.initialExecutors**

`spark.dynamicAllocation.initialExecutors` sets the initial number of executors for dynamic allocation.

### **spark.dynamicAllocation.schedulerBacklogTimeout**

`spark.dynamicAllocation.schedulerBacklogTimeout` (default: `1s`) sets...[FIXME](#)

It must be greater than `0`.

## **spark.dynamicAllocation.sustainedSchedulerBacklogTimeout**

`spark.dynamicAllocation.sustainedSchedulerBacklogTimeout` (default: `spark.dynamicAllocation.schedulerBacklogTimeout`) sets...[FIXME](#)

It must be greater than `0`.

## **spark.dynamicAllocation.executorIdleTimeout**

`spark.dynamicAllocation.executorIdleTimeout` (default: `60s`) sets...[FIXME](#)

It must be greater than `0`.

## **spark.dynamicAllocation.cachedExecutorIdleTimeout**

`spark.dynamicAllocation.cachedExecutorIdleTimeout` (default: `Integer.MAX_VALUE`) sets...[FIXME](#)

## **spark.dynamicAllocation.testing**

`spark.dynamicAllocation.testing` is...[FIXME](#)

## **Future**

- SPARK-4922
- SPARK-4751
- SPARK-7955

# ExecutorAllocationManager — Allocation Manager for Spark Core

`ExecutorAllocationManager` is responsible for dynamically allocating and removing executors based on the workload.

It intercepts Spark events using the internal `ExecutorAllocationListener` that keeps track of the workload (changing the `internal registries` that the allocation manager uses for executors management).

It uses `ExecutorAllocationClient`, `LiveListenerBus`, and `SparkConf` (that are all passed in when `ExecutorAllocationManager` is created).

## addExecutors

Caution	<a href="#">FIXME</a>
---------	-----------------------

## removeExecutor

Caution	<a href="#">FIXME</a>
---------	-----------------------

## maxNumExecutorsNeeded method

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Starting ExecutorAllocationManager (start method)

<code>start(): Unit</code>
----------------------------

`start` registers `ExecutorAllocationListener` (with `LiveListenerBus`) to monitor scheduler events and make decisions when to add and remove executors. It then immediately starts `spark-dynamic-executor-allocation allocation executor` that is responsible for the `scheduling` every `100` milliseconds.

Note	<code>100</code> milliseconds for the period between successive <code>scheduling</code> is fixed, i.e. not configurable.
------	--

It `requests executors` using the input `ExecutorAllocationClient`. It `requests spark.dynamicAllocation.initialExecutors`.

**Note**

`start` is called while `SparkContext` is being created (with dynamic allocation enabled).

## Scheduling Executors (schedule method)

```
schedule(): Unit
```

`schedule` calls `updateAndSyncNumExecutorsTarget` to...[FIXME](#)

It then go over `removeTimes` to remove expired executors, i.e. executors for which expiration time has elapsed.

## updateAndSyncNumExecutorsTarget

```
updateAndSyncNumExecutorsTarget(now: Long): Int
```

`updateAndSyncNumExecutorsTarget` ...[FIXME](#)

If `ExecutorAllocationManager` is `initializing` it returns `0`.

## initializing flag

`initializing` flag starts enabled (i.e. `true`).

## Resetting (reset method)

```
reset(): Unit
```

`reset` resets `ExecutorAllocationManager` to its initial state, i.e.

1. `initializing` is enabled (i.e. `true`).
2. The `currently-desired number of executors` is set to `the initial value`.
3. The `<>numExecutorsToAdd, ???>` is set to `1`.
4. All `executor pending to remove` are cleared.
5. All `???` are cleared.

## initialNumExecutors attribute

Caution	<a href="#">FIXME</a>
---------	-----------------------

## numExecutorsTarget attribute

Caution	<a href="#">FIXME</a>
---------	-----------------------

## numExecutorsToAdd attribute

`numExecutorsToAdd` attribute controls...[FIXME](#)

## Stopping (stop method)

`stop(): Unit`

`stop` shuts down [spark-dynamic-executor-allocation allocation executor](#).

Note	It waits 10 seconds for the complete termination.
------	---

## Internal Registries

### executorsPendingToRemove registry

Caution	<a href="#">FIXME</a>
---------	-----------------------

### removeTimes registry

`removeTimes` keeps track of executors and their...[FIXME](#)

## executorIds

Caution	<a href="#">FIXME</a>
---------	-----------------------

## spark-dynamic-executor-allocation Allocation Executor

`spark-dynamic-executor-allocation` allocation executor is a...[FIXME](#)

It is started...

It is stopped...



# ExecutorAllocationClient

`ExecutorAllocationClient` is a contract for clients to communicate with a cluster manager to request or kill executors.

## Getting Executor Ids (`getExecutorIds` method)

```
getExecutorIds(): Seq[String]
```

`getExecutorIds` is a `private[spark]` method to calculate the identifiers of the executors in use.

Note

It is used when `SparkContext` calculates the executors in use and also when `Spark Streaming` manages executors.

## Requesting Exact Number of Executors (`requestTotalExecutors` method)

```
requestTotalExecutors(  
    numExecutors: Int,  
    localityAwareTasks: Int,  
    hostToLocalTaskCount: Map[String, Int]): Boolean
```

`requestTotalExecutors` is a `private[spark]` method to update the cluster manager with the exact number of executors desired. It returns whether the request has been acknowledged by the cluster manager (`true`) or not (`false`).

Note

It is used when:

1. `SparkContext` requests executors (for coarse-grained scheduler backends only).
2. `ExecutorAllocationManager` starts, does `updateAndSyncNumExecutorsTarget`, and `addExecutors`.
3. `Streaming` `ExecutorAllocationManager` requests executors.
4. `YarnSchedulerBackend` stops.

## Requesting Additional Executors (`requestExecutors` method)

```
requestExecutors(numAdditionalExecutors: Int): Boolean
```

`requestExecutors` requests additional executors from a cluster manager and returns whether the request has been acknowledged by the cluster manager ( `true` ) or not ( `false` ).

**Note**

It is used when `SparkContext` requests additional executors (for coarse-grained scheduler backends only).

## Requesting to Kill Single Executor (`killExecutor` method)

```
killExecutor(executorId: String): Boolean
```

`killExecutor` requests that a cluster manager to kill a single executor that is no longer in use and returns whether the request has been acknowledged by the cluster manager ( `true` ) or not ( `false` ).

**Note**

The default implementation simply calls `killExecutors` (with a single-element collection of executors to kill).

**Note**

It is used in:

1. `ExecutorAllocationManager` to [remove executor](#).
2. `SparkContext` to [request to kill executors](#).
3. `Streaming` `ExecutorAllocationManager` to [request to kill executors](#).

## Requesting to Kill Executors (`killExecutors` method)

```
killExecutors(executorIds: Seq[String]): Boolean
```

`killExecutors` requests that a cluster manager to kill one or many executors that are no longer in use and returns whether the request has been acknowledged by the cluster manager ( `true` ) or not ( `false` ).

**Note**

*Interestingly*, it is only used for `killExecutor`.

# ExecutorAllocationListener

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

`ExecutorAllocationListener` is a [SparkListener](#) that intercepts events about stages, tasks, and executors, i.e. `onStageSubmitted`, `onStageCompleted`, `onTaskStart`, `onTaskEnd`, `onExecutorAdded`, and `onExecutorRemoved`. Using the events [ExecutorAllocationManager](#) can manage the pool of dynamically managed executors.

Note	<code>ExecutorAllocationListener</code> is an internal class of <a href="#">ExecutorAllocationManager</a> with full access to its internal registries.
------	--

# ExecutorAllocationManagerSource — Metric Source for Dynamic Allocation

`ExecutorAllocationManagerSource` is a [metric source](#) for [dynamic allocation](#) with name `ExecutorAllocationManager` and the following gauges:

- `executors/numberExecutorsToAdd` which exposes [numExecutorsToAdd](#).
- `executors/numberExecutorsPendingToRemove` which corresponds to the number of elements in [executorsPendingToRemove](#).
- `executors/numberAllExecutors` which corresponds to the number of elements in [executorIds](#).
- `executors/numberTargetExecutors` which is [numExecutorsTarget](#).
- `executors/numberMaxNeededExecutors` which simply calls [maxNumExecutorsNeeded](#).

Note	Spark uses <a href="#">Metrics</a> Java library to expose internal state of its services to measure.
------	--

Spark uses [Metrics](#) Java library to expose internal state of its services to measure.

# 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	<a href="#">org.apache.spark.shuffle.ShuffleManager</a> is a <code>private[spark]</code> Scala trait.
------	---

Every `shuffleManager` offers 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	<a href="#">FIXME</a> Exercise for a custom implementation of Shuffle Manager using <code>private[spark] ShuffleManager</code> trait.
---------	---

## SortShuffleManager

`SortShuffleManager` is a shuffle manager with the short name being `sort`.

It uses `IndexShuffleBlockResolver` as the `shuffleBlockResolver`.

## Settings

### spark.shuffle.manager

`spark.shuffle.manager` (default: `sort`) sets the default shuffle manager by a short name or the fully-qualified class name of a custom implementation.

### spark.shuffle.spill

`spark.shuffle.spill` (default: `true`) - no longer used, and when `false` the following WARNING shows in the logs:

```
WARN SortShuffleManager: spark.shuffle.spill was set to false, but this configuration  
is ignored as of Spark 1.6+. Shuffle will continue to spill to disk when necessary.
```

## Further reading or watching

1. (slides) [Spark shuffle introduction](#) by [Raymond Liu](#) (aka *colorant*).



# ExternalShuffleService

`ExternalShuffleService` is an **external shuffle service** that serves shuffle blocks from outside an [Executor](#) process. It runs as a standalone application and manages shuffle output files so they are available for executors at all time. As the shuffle output files are managed externally to the executors it offers an uninterrupted access to the shuffle output files regardless of executors being killed or down.

You start `ExternalShuffleService` using [`start-shuffle-service.sh`](#) shell script and enable its use by the driver and executors using [`spark.shuffle.service.enabled`](#).

**Note**

There is a custom external shuffle service for Spark on YARN — [YarnShuffleService](#).

**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
```

Refer to [Logging](#).

## start-shuffle-service.sh shell script

```
start-shuffle-service.sh
```

`start-shuffle-service.sh` shell script allows you to launch `ExternalShuffleService`. The script is under `sbin` directory.

When executed, it runs `sbin/spark-config.sh` and `bin/load-spark-env.sh` shell scripts. It then executes `sbin/spark-daemon.sh` with `start` command and the parameters:

```
org.apache.spark.deploy.ExternalShuffleService and 1.
```

```
$ ./sbin/start-shuffle-service.sh
starting org.apache.spark.deploy.ExternalShuffleService, logging
to ...logs/spark-jacek-
org.apache.spark.deploy.ExternalShuffleService-1-
japila.local.out

$ tail -f ...logs/spark-jacek-
org.apache.spark.deploy.ExternalShuffleService-1-
japila.local.out
Spark Command:
/Library/Java/JavaVirtualMachines/Current/Contents/Home/bin/java
-cp
/Users/jacek/dev/oss/spark/conf/:/Users/jacek/dev/oss/spark/asse
mblly/target/scala-2.11/jars/* -Xmx1g
org.apache.spark.deploy.ExternalShuffleService
=====
Using Spark's default log4j profile: org/apache/spark/log4j-
defaults.properties
16/06/07 08:02:02 INFO ExternalShuffleService: Started daemon
with process name: 42918@japila.local
16/06/07 08:02:03 INFO ExternalShuffleService: Starting shuffle
service on port 7337 with useSasl = false
```

**Tip**

You can also use [spark-class](#) to launch `ExternalShuffleService`.

```
spark-class org.apache.spark.deploy.ExternalShuffleService
```

## Launching ExternalShuffleService (main method)

When started, it executes `utils.initDaemon(log)`.

**Caution**

[FIXME](#) `utils.initDaemon(log)`? See [spark-submit](#).

It loads default Spark properties and creates a `SecurityManager`.

It sets `spark.shuffle.service.enabled` to `true` (as later it is checked whether it is enabled or not).

A `ExternalShuffleService` is created and [started](#).

A shutdown hook is registered so when `ExternalShuffleService` is shut down, it prints the following INFO message to the logs and the `stop` method is executed.

```
INFO ExternalShuffleService: Shutting down shuffle service.
```

**Tip**

Enable `DEBUG` logging level for `org.apache.spark.network.shuffle.ExternalShuffleBlockResolver` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.network.shuffle.ExternalShuffleBlockResolver=DEBUG
```

Refer to [Logging](#).

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 = [cleanupLocalDirs]
INFO ExternalShuffleBlockResolver: Cleaning up executor [AppExecId]'s [executor.localDirs.length] local dirs
DEBUG ExternalShuffleBlockResolver: Successfully cleaned up directory: [localDir]
```

## Creating ExternalShuffleService Instance

`ExternalShuffleService` requires a [SparkConf](#) and [SecurityManager](#).

When created, it reads `spark.shuffle.service.enabled` (disabled by default) and `spark.shuffle.service.port` (defaults to `7337`) configuration settings. It also checks whether authentication is enabled.

**Caution**

[FIXME](#) Review `securityManager.isAuthenticationEnabled()`

It then creates a [TransportConf](#) (as `transportConf`).

It creates a [ExternalShuffleBlockHandler](#) (as `blockHandler`) and [TransportContext](#) (as `transportContext`).

**Caution**

[FIXME](#) `TransportContext?`

No internal `TransportServer` (as `server`) is created.

## Starting ExternalShuffleService (start method)

```
start(): Unit
```

`start` starts a `ExternalShuffleService`.

When `start` is executed, you should see the following INFO message in the logs:

```
INFO ExternalShuffleService: Starting shuffle service on port [port] with useSasl = [useSasl]
```

If `useSasl` is enabled, a `SaslServerBootstrap` is created.

Caution	<a href="#">FIXME</a> <code>SaslServerBootstrap</code> ?
---------	--

The internal `server` reference (a `TransportServer`) is created (which will attempt to bind to `port`).

Note	<code>port</code> is set up by <code>spark.shuffle.service.port</code> or defaults to <code>7337</code> when <code>ExternalShuffleService</code> is created.
------	--

## Stopping ExternalShuffleService (stop method)

```
stop(): Unit
```

`stop` closes the internal `server` reference and clears it (i.e. sets it to `null`).

## ExternalShuffleBlockHandler

`ExternalShuffleBlockHandler` is a `RpcHandler` (i.e. a handler for `sendRPC()` messages sent by `TransportClients`).

When created, `ExternalShuffleBlockHandler` requires a `OneForOneStreamManager` and `TransportConf` with a `registeredExecutorFile` to create a `ExternalShuffleBlockResolver`.

It handles two `BlockTransferMessage` messages: `OpenBlocks` and `RegisterExecutor`.

Enable `TRACE` logging level for `org.apache.spark.network.shuffle.ExternalShuffleBlockHandler` logger to see what happens inside.

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

Tip

```
log4j.logger.org.apache.spark.network.shuffle.ExternalShuffleBlockHandler=TRACE
```

Refer to [Logging](#).

## handleMessage method

```
handleMessage(  
    BlockTransferMessage msgObj,  
    TransportClient client,  
    RpcResponseCallback callback)
```

`handleMessage` handles two types of `BlockTransferMessage` messages:

- [OpenBlocks](#)
- [RegisterExecutor](#)

For any other `BlockTransferMessage` message it throws a `UnsupportedOperationException` :

```
Unexpected message: [msgObj]
```

## OpenBlocks

```
OpenBlocks(String appId, String execId, String[] blockIds)
```

When `OpenBlocks` is received, `handleMessage` authorizes the `client`.

Caution	<a href="#">FIXME</a> <code>checkAuth</code> ?
---------	--

It then [gets block data](#) for each block id in `blockIds` (using `ExternalShuffleBlockResolver`).

Finally, it [registers a stream](#) and does `callback.onSuccess` with a serialized byte buffer (for the `streamId` and the number of blocks in `msg` ).

Caution	<a href="#">FIXME</a> <code>callback.onSuccess</code> ?
---------	---

You should see the following TRACE message in the logs:

```
TRACE Registered streamId [streamId] with [length] buffers for client [clientId] from host [remoteAddress]
```

## RegisterExecutor

```
RegisterExecutor(String appId, String execId, ExecutorShuffleInfo executorInfo)
```

RegisterExecutor

## ExternalShuffleBlockResolver

Caution

[FIXME](#)

### getBlockData method

```
ManagedBuffer getBlockData(String appId, String execId, String blockId)
```

`getBlockData` parses `blockId` (in the format of `shuffle_[shuffleId]_[mapId]_[reduceId]`) and returns the `FileSegmentManagedBuffer` that corresponds to `shuffle_[shuffleId]_[mapId]_0.data`.

`getBlockData` splits `blockId` to 4 parts using `_` (underscore). It works exclusively with `shuffle` block ids with the other three parts being `shuffleId`, `mapId`, and `reduceId`.

It looks up an executor (i.e. a `ExecutorShuffleInfo` in `executors` private registry) for `appId` and `execId` to search for a [ManagedBuffer](#).

The `ManagedBuffer` is indexed using a binary file `shuffle_[shuffleId]_[mapId]_0.index` (that contains offset and length of the buffer) with a data file being `shuffle_[shuffleId]_[mapId]_0.data` (that is returned as `FileSegmentManagedBuffer`).

It throws a `IllegalArgumentException` for block ids with less than four parts:

```
Unexpected block id format: [blockId]
```

or for non- `shuffle` block ids:

```
Expected shuffle block id, got: [blockId]
```

It throws a `RuntimeException` when no `ExecutorShuffleInfo` could be found.

```
Executor is not registered (appId=[appId], execId=[execId])"
```

## OneForOneStreamManager

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

### registerStream method

```
long registerStream(String appId, Iterator<ManagedBuffer> buffers)
```

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

## Settings

### spark.shuffle.service.enabled

`spark.shuffle.service.enabled` flag (default: `false`) controls whether the [External Shuffle Service](#) is used or not. When enabled (`true`), the driver registers with the shuffle service.

`spark.shuffle.service.enabled` has to be enabled for [dynamic allocation of executors](#).

It is used in [CoarseMesosSchedulerBackend](#) to instantiate `MesosExternalShuffleClient`.

It is explicitly disabled for `LocalSparkCluster` (and *any* attempts to set it will fall short).

### spark.shuffle.service.port

`spark.shuffle.service.port` (default: `7337`)

# ExternalClusterManager

`ExternalClusterManager` is a contract for pluggable cluster managers.

**Note** It was introduced in [SPARK-13904 Add support for pluggable cluster manager](#).

It is assumed that `ExternalClusterManager` implementations are available as Java services (with service markers under `META-INF/services` directory).

**Note** `ExternalClusterManager` is a `private[spark]` trait in `org.apache.spark.scheduler` package.

**Note** The only known implementation of the [ExternalClusterManager contract](#) in Spark is [YarnClusterManager](#).

**Note** `SparkContext` finds a `ExternalClusterManager` for a master URL using an internal `getClusterManager`.

## ExternalClusterManager Contract

### initialize

```
initialize(scheduler: TaskScheduler, backend: SchedulerBackend): Unit
```

### canCreate

```
canCreate(masterURL: String): Boolean
```

**Note** It is used when [finding the external cluster manager for a master URL \(in `SparkContext`\)](#).

### createTaskScheduler

```
createTaskScheduler(sc: SparkContext, masterURL: String): TaskScheduler
```

### createSchedulerBackend

```
createSchedulerBackend(sc: SparkContext,  
                      masterURL: String,  
                      scheduler: TaskScheduler): SchedulerBackend
```

# HTTP File Server

It is started on a [driver](#).

Caution	<a href="#">FIXME</a> 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.

Broadcast using a BitTorrent-like implementation

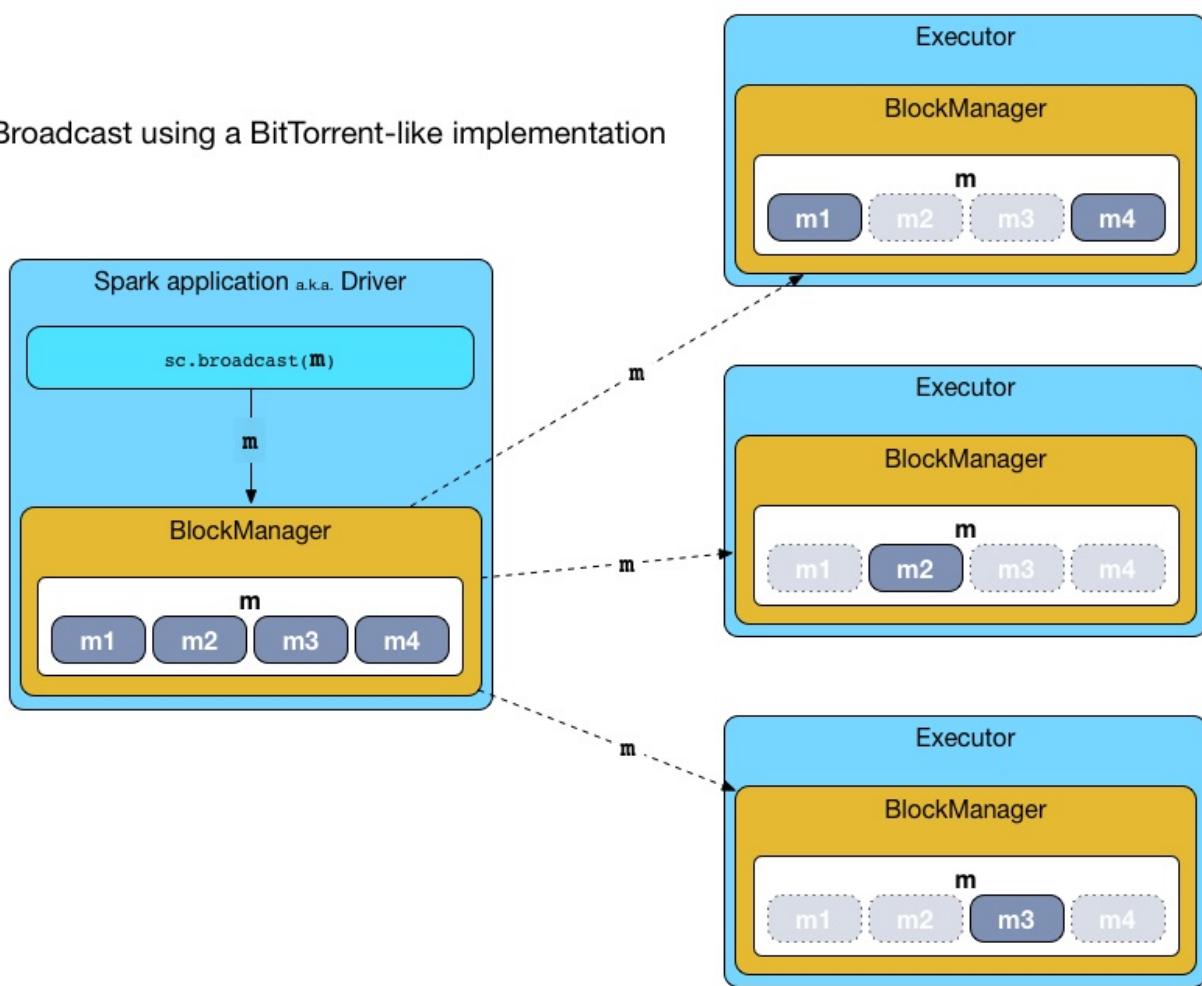


Figure 1. TorrentBroadcast - broadcasting using BitTorrent

`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](#).

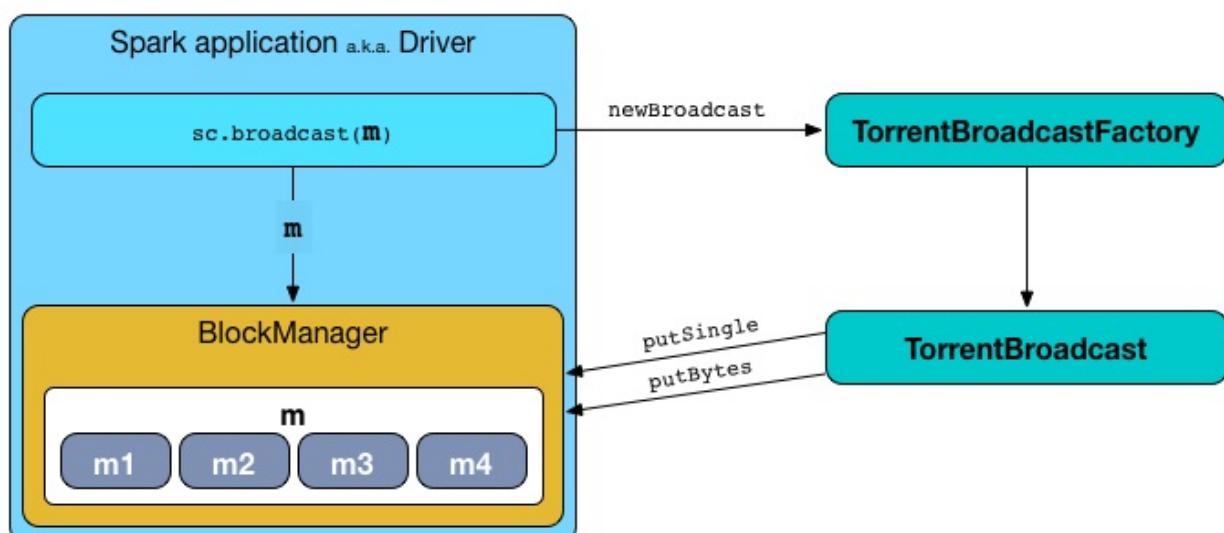


Figure 2. TorrentBroadcast puts broadcast chunks to driver's BlockManager

## 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

# 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.

In [Spark on YARN](#) Spark tries to place tasks alongside HDFS blocks.

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	<a href="#">FIXME</a> Review the <code>CacheManager</code> class.
---------	---

In the code, the current instance of Cache Manager is available under  
`SparkEnv.get.cacheManager`.

## Caching Query (`cacheQuery` method)

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Uncaching Query (`uncacheQuery` method)

Caution	<a href="#">FIXME</a>
---------	-----------------------

# 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.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 only implementation of RPC Environment is [Netty-based implementation](#). 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**.

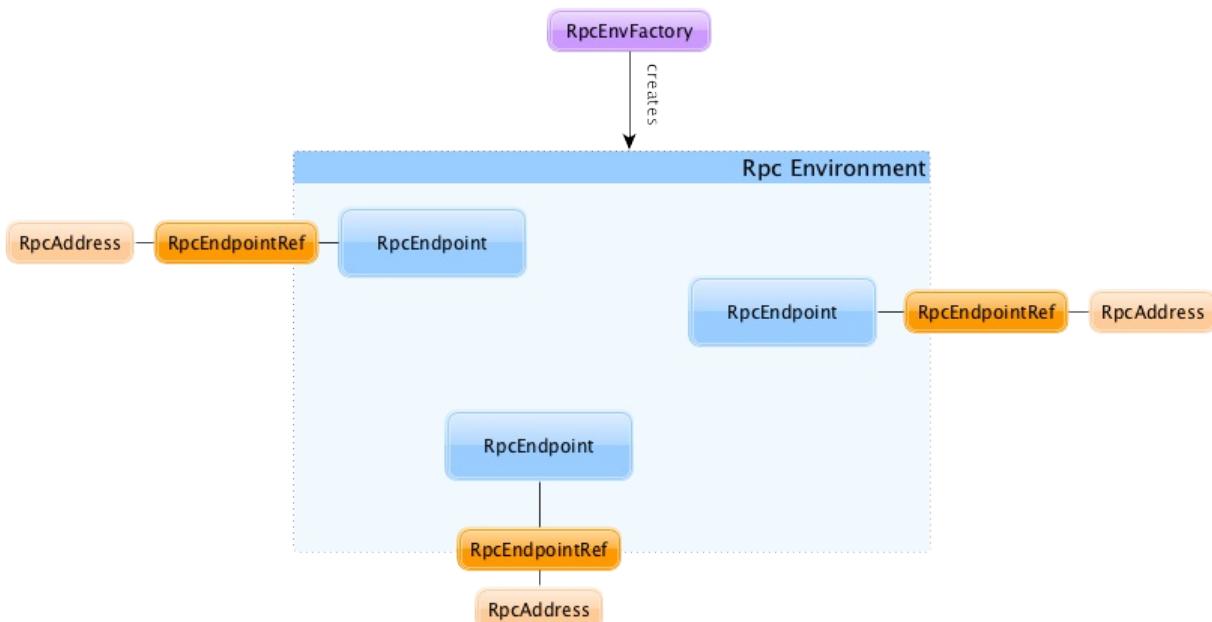


Figure 1. RpcEnvironment with RpcEndpoints and RpcEndpointRefs

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

**RpcEndpoint** defines 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.

Tip	If you want <code>receive</code> to be thread-safe, use <a href="#">ThreadSafeRpcEndpoint</a> .
-----	---

`onError` method is called for any exception thrown.

## ThreadSafeRpcEndpoint

`ThreadSafeRpcEndpoint` is a marker [RpcEndpoint](#) that does nothing by itself but tells...

Caution

[FIXME](#) What is marker?

Note

`ThreadSafeRpcEndpoint` is a `private[spark]` trait .

## 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 = !isDriver)
```

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

`spark.rpc` (default: `netty` since Spark 1.6.0-SNAPSHOT) - the RPC implementation to use. See [RpcEnvFactory](#).

### spark.rpc.lookupTimeout

`spark.rpc.lookupTimeout` (default: `120s`) - the default timeout to use for RPC remote endpoint lookup. Refer to [Endpoint Lookup Timeout](#).

### spark.network.timeout

`spark.network.timeout` (default: `120s`) - the default network timeout to use for RPC remote endpoint lookup.

It is used as a fallback value for [spark.rpc.askTimeout](#).

## Other

- `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
waiting on condition [0x0000000123397000]
```

### 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=0x71  
03 waiting on condition [0x0000000011f78b000]
```

# ContextCleaner

It does cleanup of shuffles, RDDs and broadcasts.

Caution	<a href="#">FIXME</a> What does the above sentence <b>really</b> mean?
---------	--

It uses a daemon **Spark Context Cleaner** thread that cleans RDD, shuffle, and broadcast states (using `keepCleaning` method).

Caution	<a href="#">FIXME</a> Review <code>keepCleaning</code>
---------	--

[ShuffleDependencies](#) register themselves for cleanup using

`ContextCleaner.registerShuffleForCleanup` method.

ContextCleaner uses a Spark context.

## registerRDDForCleanup

Caution	<a href="#">FIXME</a>
---------	-----------------------

## registerAccumulatorForCleanup

Caution	<a href="#">FIXME</a>
---------	-----------------------

## 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.

## unregisterShuffle

Caution

[FIXME](#)

## 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	<a href="#">FIXME</a> What exactly is 2000? Is this the number of tasks in a job?
---------	---

Caution	<a href="#">FIXME</a> Review ShuffleManager
---------	---

## Epoch Number

Caution	<a href="#">FIXME</a>
---------	-----------------------

## 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	<a href="#">FIXME</a>
---------	-----------------------

## MapOutputTrackerMaster.getStatistics

Caution	<a href="#">FIXME</a>
---------	-----------------------

## MapOutputTrackerMaster.unregisterMapOutput

Caution	<a href="#">FIXME</a>
---------	-----------------------

## MapOutputTrackerMaster.registerMapOutputs

Caution	<a href="#">FIXME</a>
---------	-----------------------

## MapOutputTrackerMaster.incrementEpoch

Caution	<a href="#">FIXME</a>
---------	-----------------------

## 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.

Tip	Enable <code>DEBUG</code> logging level for <code>org.apache.spark.util.TimeStampedHashMap</code> logger to see what happens in <code>TimeStampedHashMap</code> . Add the following line to <code>conf/log4j.properties</code> :  <code>log4j.logger.org.apache.spark.util.TimeStampedHashMap=DEBUG</code>
-----	---

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 <code>java.util.concurrent.ConcurrentHashMap</code> .
------	---

# Deployment Environments — Run Modes

Spark Deployment Environments (aka Run Modes):

- [local](#)
- [clustered](#)
  - [Spark Standalone](#)
  - [Spark on Apache Mesos](#)
  - [Spark on Hadoop 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 \(client mode\)](#)

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 (pseudo-cluster)

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 single 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.

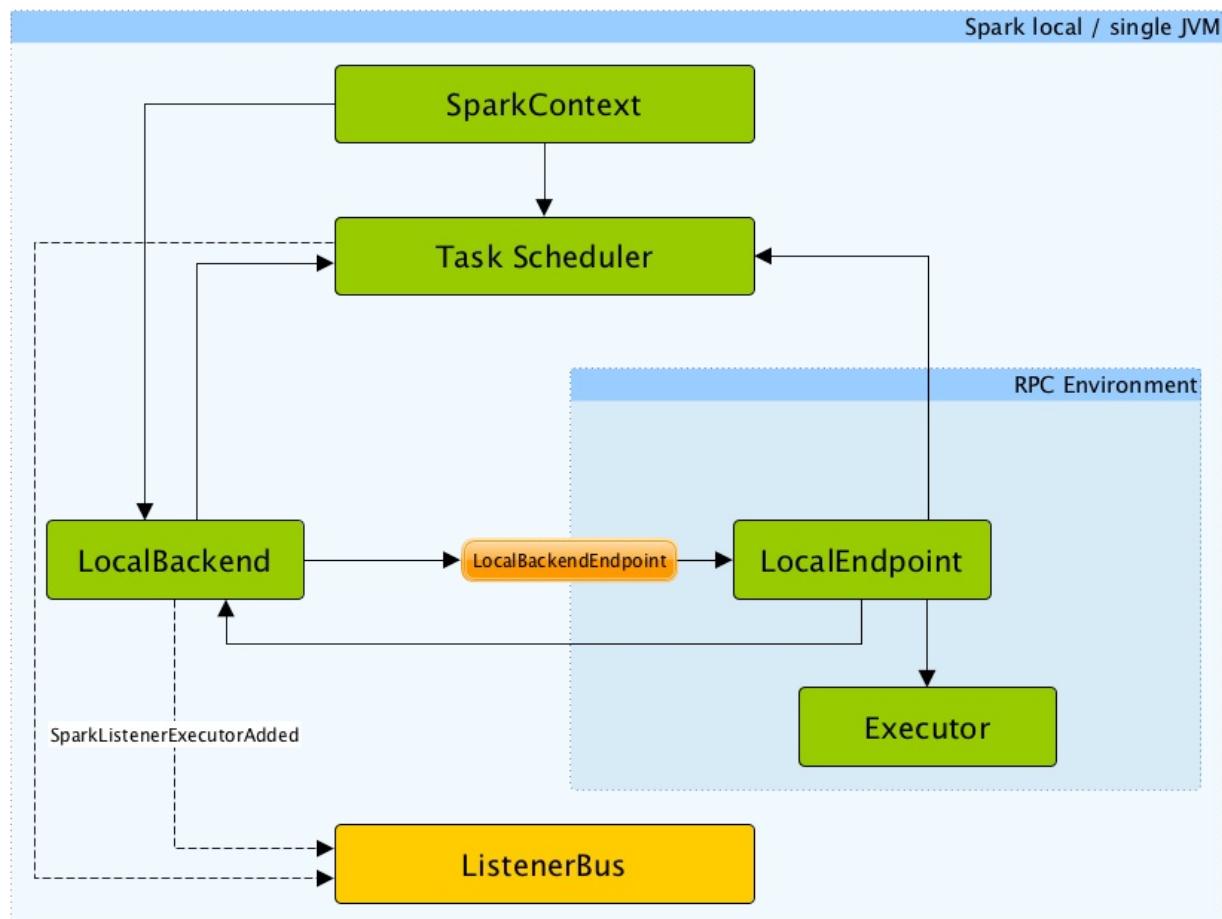


Figure 1. Architecture of Spark local

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`

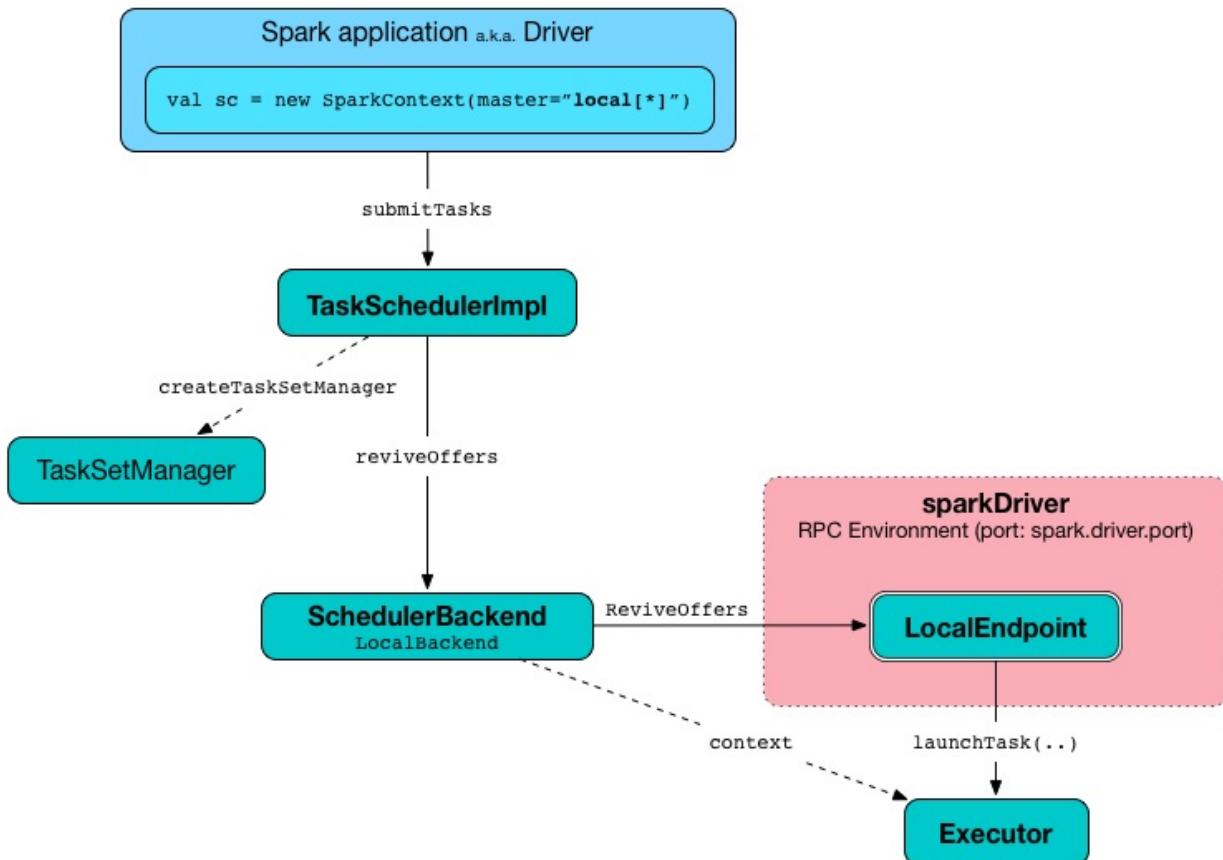


Figure 2. `TaskSchedulerImpl.submitTasks` in local mode

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	<a href="#">FIXME</a> 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`.

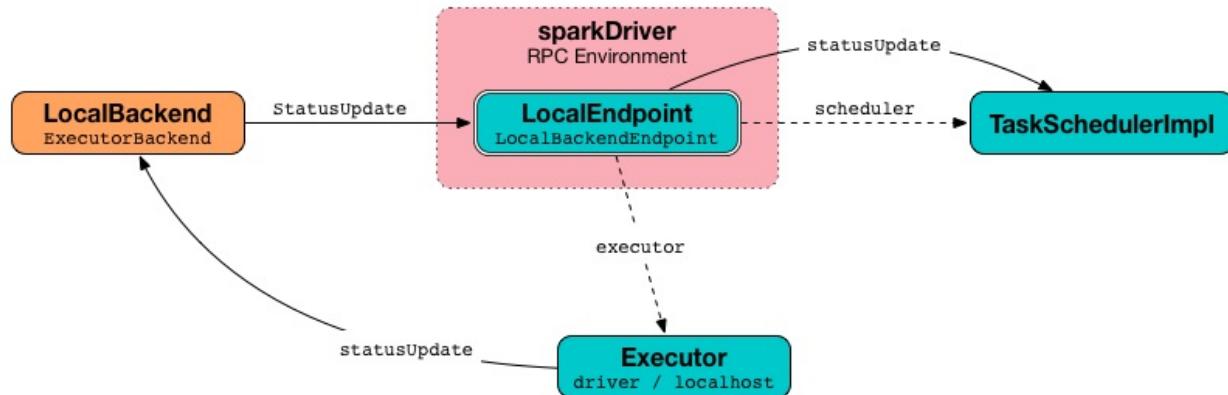


Figure 3. Task status updates flow in local mode

When LocalBackend starts up, it registers a new [RPC Endpoint](#) called **LocalBackendEndpoint** that is backed by [LocalEndpoint](#). This is announced on [LiveListenerBus](#) as `driver` (using [SparkListenerExecutorAdded](#) message).

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

```

## Creating LocalEndpoint Instance

Caution	<a href="#">FIXME</a>
---------	-----------------------

## 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.

# Spark Clustered

Spark can be run in distributed mode on a cluster. The following (open source) **cluster managers** (*aka task schedulers aka resource managers*) are currently supported:

- Spark's own built-in Standalone cluster manager
- Hadoop YARN
- Apache Mesos

Here is a very brief list of pros and cons of using one cluster manager versus the other options supported by Spark:

1. Spark Standalone is included in the official distribution of Apache Spark.
2. Hadoop YARN has a very good support for HDFS with data locality.
3. Apache Mesos makes resource offers that a framework can accept or reject. It is Spark (as a Mesos framework) to decide what resources to accept. It is a *push-based* resource management model.
4. Hadoop YARN responds to a YARN framework's resource requests. Spark (as a YARN framework) requests CPU and memory from YARN. It is a *pull-based* resource management model.
5. Hadoop YARN supports Kerberos for a secured HDFS.

Running Spark on a cluster requires workload and resource management on distributed systems.

Spark driver requests resources from a cluster manager. Currently only CPU and memory are requested resources. It is a cluster manager's responsibility to spawn Spark executors in the cluster (on its workers).

## FIXME

- |         |  |
|---------|--|
| Caution | <ul style="list-style-type: none"><li>• Spark execution in cluster - Diagram of the communication between driver, cluster manager, workers with executors and tasks. See <a href="#">Cluster Mode Overview</a>.</li><li>◦ Show Spark's driver with the main code in Scala in the box</li><li>◦ Nodes with executors with tasks</li><li>• Hosts drivers</li><li>• Manages a cluster</li></ul> |
|---------|--|

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 <b>Sean Owen</b> in <a href="#">What is the relationship between workers, worker instances, and executors?</a>
------	--

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	<a href="#">Review core/src/main/scala/org/apache/spark/deploy/master/Master.scala</a>
---------	--

## 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	<a href="#">FIXME Figure</a>
---------	------------------------------

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 on YARN

You can submit Spark applications to a Hadoop YARN cluster using `yarn master URL`.

```
spark-submit --master yarn mySparkApp.jar
```

Note	Since Spark <b>2.0.0</b> , <code>yarn master URL</code> is the only proper master URL and you can use <code>--deploy-mode</code> to choose between <code>client</code> (default) or <code>cluster</code> modes.
------	---

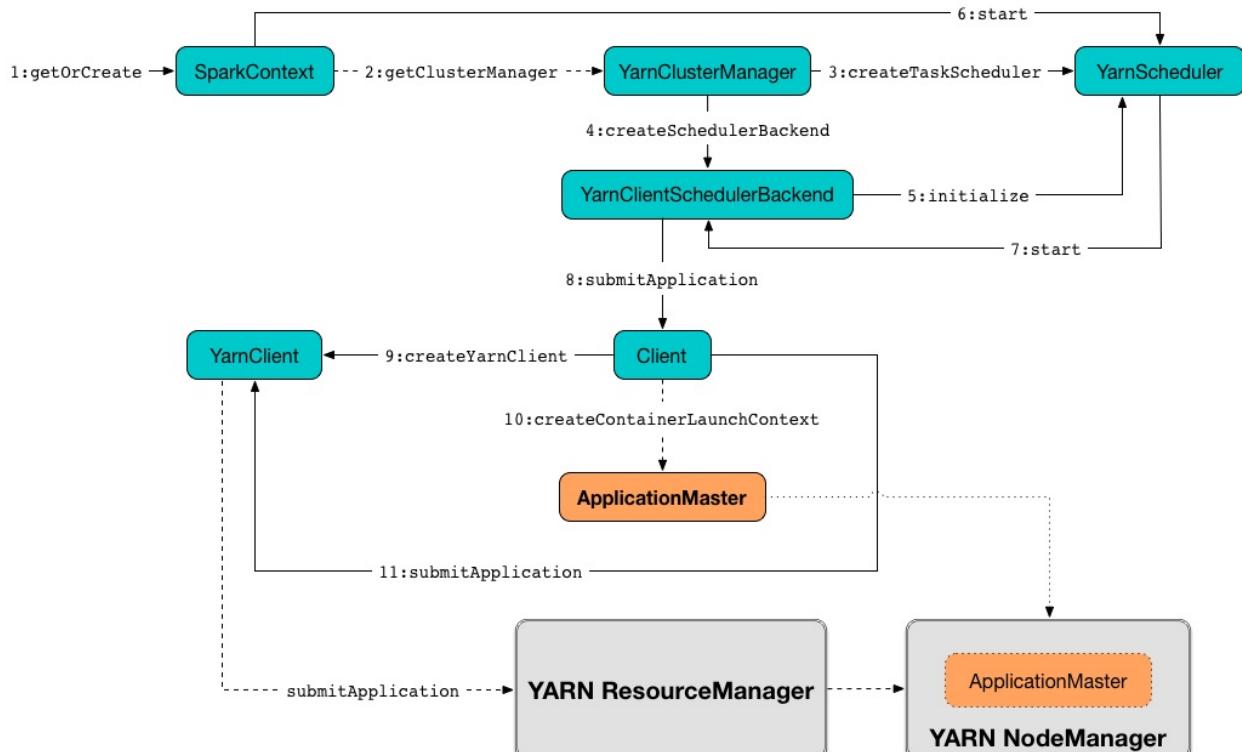


Figure 1. Submitting Spark Application to YARN Cluster (aka Creating SparkContext with `yarn Master URL` and client Deploy Mode)

Without specifying the `deploy mode`, it is assumed `client`.

```
spark-submit --master yarn --deploy-mode client mySparkApp.jar
```

There are two deploy modes for YARN — `client` (default) or `cluster`.

Tip	Deploy modes are all about where the <code>Spark driver</code> runs.
-----	--

In `client` mode the Spark driver (and `SparkContext`) runs on a client node outside a YARN cluster whereas in `cluster` mode it runs inside a YARN cluster, i.e. inside a YARN container alongside `ApplicationMaster` (that acts as the Spark application in YARN).

```
spark-submit --master yarn --deploy-mode cluster mySparkApp.jar
```

In that sense, a Spark application deployed to YARN is a YARN-compatible execution framework that can be deployed to a YARN cluster (alongside other Hadoop workloads). On YARN, a Spark executor maps to a single YARN container.

**Note**

In order to deploy applications to YARN clusters, you need to [use Spark with YARN support](#).

Spark on YARN supports [multiple application attempts](#) and supports [data locality](#) for data in [HDFS](#). You can also take advantage of Hadoop's security and run Spark in a [secure Hadoop environment using Kerberos authentication](#) (aka *Kerberized clusters*).

There are few settings that are specific to YARN (see [Settings](#)). Among them, you can particularly like the [support for YARN resource queues](#) (to divide cluster resources and allocate shares to different teams and users based on advanced policies).

**Tip**

You can start [spark-submit](#) with `--verbose` command-line option to have some settings displayed, including YARN-specific. See [spark-submit and YARN options](#).

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](#).

## Multiple Application Attempts

Spark on YARN supports [multiple application attempts](#) in [cluster mode](#).

See [YarnRMClient.getMaxRegAttempts](#).

**Caution****FIXME**

## spark-submit and YARN options

When you submit your Spark applications using [spark-submit](#) you can use the following YARN-specific command-line options:

- `--archives`
- `--executor-cores`
- `--keytab`

- `--num-executors`
- `--principal`
- `--queue`

**Tip**Read about the corresponding settings in [Settings](#) in this document.

## Memory Requirements

When `Client` submits a Spark application to a YARN cluster, it makes sure that the application will not request more than the maximum memory capability of the YARN cluster.

The memory for `ApplicationMaster` is controlled by custom settings per [deploy mode](#).

For [client deploy mode](#) it is a sum of `spark.yarn.am.memory` (default: `512m`) with an optional overhead as `spark.yarn.am.memoryOverhead`.

For [cluster deploy mode](#) it is a sum of `spark.driver.memory` (default: `1g`) with an optional overhead as `spark.yarn.driver.memoryOverhead`.

If the optional overhead is not set, it is computed as [10%](#) of the main memory (`spark.yarn.am.memory` for client mode or `spark.driver.memory` for cluster mode) or `384m` whatever is larger.

## Spark with YARN support

You need to have Spark that has been compiled with YARN support, i.e. the class `org.apache.spark.deploy.yarn.Client` must be on the CLASSPATH.

Otherwise, you will see the following error in the logs and Spark will exit.

```
Error: Could not load YARN classes. This copy of Spark may not have been compiled with
YARN support.
```

## Master URL

Since Spark **2.0.0**, the only proper master URL is `yarn`.

```
./bin/spark-submit --master yarn ...
```

Before Spark 2.0.0, you could have used `yarn-client` or `yarn-cluster`, but it is now deprecated. When you use the deprecated master URLs, you should see the following warning in the logs:

Warning: Master yarn-client is deprecated since 2.0. Please use master "yarn" with specified deploy mode instead.

## Keytab

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

When a principal is specified a keytab must be specified, too.

The settings `spark.yarn.principal` and `spark.yarn.principal` will be set to respective values and `UserGroupInformation.loginUserFromKeytab` will be called with their values as input arguments.

## Environment Variables

### **SPARK\_DIST\_CLASSPATH**

`SPARK_DIST_CLASSPATH` is a distribution-defined CLASSPATH to add to processes.

It is used to [populate CLASSPATH for ApplicationMaster and executors](#).

## Settings

Caution	FIXME Where and how are they used?
---------	------------------------------------

## Further reading or watching

- (video) [Spark on YARN: a Deep Dive — Sandy Ryza \(Cloudera\)](#)
- (video) [Spark on YARN: The Road Ahead — Marcelo Vanzin \(Cloudera\)](#) from Spark Summit 2015

# YarnShuffleService — ExternalShuffleService on YARN

`YarnShuffleService` is an external shuffle service for [Spark on YARN](#). It is YARN NodeManager's auxiliary service that implements `org.apache.hadoop.yarn.server.api.AuxiliaryService`.

Note

There is the [ExternalShuffleService](#) for Spark and despite their names they don't share code.

Caution

[FIXME](#) What happens when the `spark.shuffle.service.enabled` flag is enabled?

`YarnShuffleService` is [configured in `yarn-site.xml`](#) configuration file and is initialized on each YARN NodeManager node when the node(s) starts.

After the external shuffle service is configured in YARN you enable it in a Spark application using `spark.shuffle.service.enabled` flag.

Note

`YarnShuffleService` was introduced in [SPARK-3797](#).

Enable `INFO` logging level for `org.apache.spark.network.yarn.YarnShuffleService` logger in YARN logging system to see what happens inside.

```
log4j.logger.org.apache.spark.network.yarn.YarnShuffleService=INFO
```

Tip

YARN saves logs in `/usr/local/Cellar/hadoop/2.7.2/libexec/logs` directory on Mac OS X with brew, e.g. `/usr/local/Cellar/hadoop/2.7.2/libexec/logs/yarn-jacek-nodemanager-japila.local.log`.

## Advantages

The advantages of using the YARN Shuffle Service:

- With dynamic allocation enabled executors can be discarded and a Spark application could still get at the shuffle data the executors wrote out.
- It allows individual executors to go into GC pause (or even crash) and still allow other Executors to read shuffle data and make progress.

## Creating YarnShuffleService Instance

When `YarnShuffleService` is created, it calls YARN's `AuxiliaryService` with `spark_shuffle` service name.

You should see the following INFO message in the logs:

```
INFO org.apache.spark.network.yarn.YarnShuffleService: Initializing YARN shuffle service for Spark
INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.AuxServices: Adding auxiliary service spark_shuffle, "spark_shuffle"
```

## getRecoveryPath

Caution	<a href="#">FIXME</a>
---------	-----------------------

## serviceStop

```
void serviceStop()
```

`serviceStop` is a part of YARN's `AuxiliaryService` contract and is called when...[FIXME](#)

Caution	<a href="#">FIXME</a> The contract
---------	------------------------------------

Caution	<a href="#">FIXME</a> What are <code>shuffleServer</code> and <code>blockHandler</code> ? What's their lifecycle?
---------	---

```
ERROR org.apache.spark.network.yarn.YarnShuffleService: Exception when stopping service
```

## stopContainer

```
void stopContainer(ContainerTerminationContext context)
```

`stopContainer` is a part of YARN's `AuxiliaryService` contract and is called when...[FIXME](#)

Caution	<a href="#">FIXME</a> The contract
---------	------------------------------------

When called, `stopContainer` simply prints out the following INFO message in the logs and exits.

```
INFO org.apache.spark.network.yarn.YarnShuffleService: Stopping container [containerId]
```

It obtains the `containerId` from `context` using `getContainerId` method.

## initializeContainer

```
void initializeContainer(ContainerInitializationContext context)
```

`initializeContainer` is a part of YARN's `AuxiliaryService` contract and is called when...[FIXME](#)

Caution	<a href="#">FIXME</a> The contract
---------	------------------------------------

When called, `initializeContainer` simply prints out the following INFO message in the logs and exits.

```
INFO org.apache.spark.network.yarn.YarnShuffleService: Initializing container [containerId]
```

It obtains the `containerId` from `context` using `getContainerId` method.

## stopApplication

```
void stopApplication(ApplicationTerminationContext context)
```

`stopApplication` is a part of YARN's `AuxiliaryService` contract and is called when...[FIXME](#)

Caution	<a href="#">FIXME</a> The contract
---------	------------------------------------

`stopApplication` requests the `ShuffleSecretManager` to `unregisterApp` when authentication is enabled and `ExternalShuffleBlockHandler` to `applicationRemoved`.

When called, `stopApplication` obtains YARN's `ApplicationId` for the application (using the input `context`).

You should see the following INFO message in the logs:

```
INFO org.apache.spark.network.yarn.YarnShuffleService: Stopping application [appId]
```

If `isAuthenticationEnabled`, `secretManager.unregisterApp` is executed for the application id.

It requests `ExternalShuffleBlockHandler` to `applicationRemoved` (with `cleanupLocalDirs` flag disabled).

**Caution**

**FIXME** What does `ExternalShuffleBlockHandler#applicationRemoved` do?

When an exception occurs, you should see the following ERROR message in the logs:

```
ERROR org.apache.spark.network.yarn.YarnShuffleService: Exception when stopping application [appId]
```

## initializeApplication

```
void initializeApplication(ApplicationInitializationContext context)
```

`initializeApplication` is a part of YARN's `AuxiliaryService` contract and is called when...**FIXME**

**Caution**

**FIXME** The contract

`initializeApplication` requests the `ShuffleSecretManager` to `registerApp` when authentication is enabled.

When called, `initializeApplication` obtains YARN's `ApplicationId` for the application (using the input `context`) and calls `context.getApplicationDataForService` for `shuffleSecret`.

You should see the following INFO message in the logs:

```
INFO org.apache.spark.network.yarn.YarnShuffleService: Initializing application [appId]
```

If `isAuthenticationEnabled`, `secretManager.registerApp` is executed for the application id and `shuffleSecret`.

When an exception occurs, you should see the following ERROR message in the logs:

```
ERROR org.apache.spark.network.yarn.YarnShuffleService: Exception when initializing application [appId]
```

## serviceInit

```
void serviceInit(Configuration conf)
```

`serviceInit` is a part of YARN's `AuxiliaryService` contract and is called when...[FIXME](#)

Caution

[FIXME](#)

When called, `serviceInit` creates a `TransportConf` for the `shuffle` module that is used to create `ExternalShuffleBlockHandler` (as `blockHandler`).

It checks `spark.authenticate` key in the configuration (defaults to `false`) and if only authentication is enabled, it sets up a `SaslServerBootstrap` with a `ShuffleSecretManager` and adds it to a collection of `TransportServerBootstraps`.

It creates a `TransportServer` as `shuffleServer` to listen to `spark.shuffle.service.port` (default: `7337`). It reads `spark.shuffle.service.port` key in the configuration.

You should see the following INFO message in the logs:

```
INFO org.apache.spark.network.yarn.YarnShuffleService: Started YARN shuffle service for Spark on port [port]. Authentication is [authEnabled]. Registered executor file is [registeredExecutorFile]
```

## Installation

### YARN Shuffle Service Plugin

Add the YARN Shuffle Service plugin from the `common/network-yarn` module to YARN NodeManager's CLASSPATH.

Tip

Use `yarn classpath` command to know YARN's CLASSPATH.

```
cp common/network-yarn/target/scala-2.11/spark-2.0.0-SNAPSHOT-yarn-shuffle.jar \
/usr/local/Cellar/hadoop/2.7.2/libexec/share/hadoop/yarn/lib/
```

### yarn-site.xml — NodeManager Configuration File

If `external shuffle service is enabled`, you need to add `spark_shuffle` to `yarn.nodemanager.aux-services` in the `yarn-site.xml` file on all nodes.

`yarn-site.xml` — NodeManager Configuration properties

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>spark_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services.spark_shuffle.class</name>
    <value>org.apache.spark.network.yarn.YarnShuffleService</value>
  </property>
  <!-- optional -->
  <property>
    <name>spark.shuffle.service.port</name>
    <value>10000</value>
  </property>
  <property>
    <name>spark.authenticate</name>
    <value>true</value>
  </property>
</configuration>
```

`yarn.nodemanager.aux-services` property is for the auxiliary service name being `spark_shuffle` with `yarn.nodemanager.aux-services.spark_shuffle.class` property being `org.apache.spark.network.yarn.YarnShuffleService`.

## Exception — Attempting to Use External Shuffle Service in Spark Application in Spark on YARN

When you [enable an external shuffle service in a Spark application](#) when using [Spark on YARN](#) but do not [install YARN Shuffle Service](#) you will see the following exception in the logs:

```
Exception in thread "ContainerLauncher-0" java.lang.Error: org.apache.spark.SparkException: Exception while starting container container_1465448245611_0002_01_000002 on host 192.168.99.1
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:148)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
Caused by: org.apache.spark.SparkException: Exception while starting container container_1465448245611_0002_01_000002 on host 192.168.99.1
        at org.apache.spark.deploy.yarn.ExecutorRunnable.startContainer(ExecutorRunnable.scala:126)
        at org.apache.spark.deploy.yarn.ExecutorRunnable.run(ExecutorRunnable.scala:71)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:142)
        ... 2 more
Caused by: org.apache.hadoop.yarn.exceptions.InvalidAuxServiceException: The auxService:spark_shuffle does not exist
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
        at java.lang.reflect.Constructor.newInstance(Constructor.java:423)
        at org.apache.hadoop.yarn.api.records.impl.pb.SerializedExceptionPBImpl.instantiateException(SerializedExceptionPBImpl.java:168)
        at org.apache.hadoop.yarn.api.records.impl.pb.SerializedExceptionPBImpl.deserialize(SerializedExceptionPBImpl.java:106)
        at org.apache.hadoop.yarn.client.api.impl.NMClientImpl.startContainer(NMClientImpl.java:207)
        at org.apache.spark.deploy.yarn.ExecutorRunnable.startContainer(ExecutorRunnable.scala:123)
        ... 4 more
```

# ExecutorRunnable

`ExecutorRunnable` starts a YARN container with `CoarseGrainedExecutorBackend` application. If external shuffle service is used, it is set in the `ContainerLaunchContext` context as a service data using `spark_shuffle`.

Note	Despite the name <code>ExecutorRunnable</code> is not a <code>java.lang.Runnable</code> anymore after <a href="#">SPARK-12447</a> .
Tip	<p>Enable <code>INFO</code> logging level for <code>org.apache.spark.deploy.yarn.ExecutorRunnable</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.deploy.yarn.ExecutorRunnable=INFO</pre> <p>Refer to <a href="#">Logging</a>.</p>

## Creating ExecutorRunnable Instance

```
ExecutorRunnable(
    container: Container,
    conf: Configuration,
    sparkConf: SparkConf,
    masterAddress: String,
    slaveId: String,
    hostname: String,
    executorMemory: Int,
    executorCores: Int,
    appId: String,
    securityMgr: SecurityManager,
    localResources: Map[String, LocalResource])
```

`YarnAllocator` creates an instance of `ExecutorRunnable` when launching Spark executors in allocated YARN containers.

A single `ExecutorRunnable` is created with the YARN container to run a Spark executor in.

The input `conf` (Hadoop's Configuration), `sparkConf`, `masterAddress` directly correspond to the constructor arguments of `YarnAllocator`.

The input `slaveId` is from the internal counter in `YarnAllocator`.

The input `hostname` is the host of the YARN container.

The input `executorMemory` and `executorCores` are from `YarnAllocator`, but come from `spark.executor.memory` and `spark.executor.cores` configuration settings.

The input `appId`, `securityMgr`, and `localResources` are the same as `YarnAllocator` was created for.

## prepareEnvironment

Caution

FIXME

## Running ExecutorRunnable (run method)

When called, you should see the following INFO message in the logs:

```
INFO ExecutorRunnable: Starting Executor Container
```

It creates a YARN `NMClient`, inits it with `yarnConf` and starts it.

It ultimately starts `CoarseGrainedExecutorBackend` in the container.

## Starting CoarseGrainedExecutorBackend in Container (startContainer method)

```
startContainer(): java.util.Map[String, ByteBuffer]
```

`startContainer` uses the `NMClient` API to start a `CoarseGrainedExecutorBackend` in a YARN container.

When `startContainer` is executed, you should see the following INFO message in the logs:

```
INFO ExecutorRunnable: Setting up ContainerLaunchContext
```

It then creates a YARN `ContainerLaunchContext` (which represents all of the information for the YARN NodeManager to launch a container) with the local resources and environment being the `localResources` and `env`, respectively, passed in to the `ExecutorRunnable` when it was created. It also sets security tokens.

It prepares the command to launch `coarseGrainedExecutorBackend` with all the details as provided when the `ExecutorRunnable` was created.

You should see the following INFO message in the logs:

```
INFO ExecutorRunnable:
=====
YARN executor launch context:
  env:
    [key] -> [value]
    ...
  command:
    [commands]
=====
```

The command is set to the just-created `ContainerLaunchContext`.

It sets application ACLs using [YarnSparkHadoopUtil.getApplicationAclsForYarn](#).

If [external shuffle service is used](#), it registers with the YARN shuffle service already started on the NodeManager. The external shuffle service is set in the `ContainerLaunchContext` context as a service data using `spark_shuffle`.

Ultimately, `startContainer` requests the YARN NodeManager to start the YARN container for a Spark executor (as passed in when [the `ExecutorRunnable` was created](#)) with the `ContainerLaunchContext` context.

If any exception happens, a `SparkException` is thrown.

```
Exception while starting container [containerId] on host [hostname]
```

Note	<code>startContainer</code> is exclusively called as a part of <a href="#">running <code>ExecutorRunnable</code></a> .
------	--

## Preparing Command to Launch CoarseGrainedExecutorBackend (prepareCommand method)

```
prepareCommand(
  masterAddress: String,
  slaveId: String,
  hostname: String,
  executorMemory: Int,
  executorCores: Int,
  appId: String): List[String]
```

`prepareCommand` is a private method to prepare the command that is used to [start `org.apache.spark.executor.CoarseGrainedExecutorBackend` application in a YARN container](#). All the input parameters of `prepareCommand` become the [command-line arguments](#) of

`CoarseGrainedExecutorBackend` application.

The input `executorMemory` is in `m` and becomes `-Xmx` in the JVM options.

It uses the optional `spark.executor.extraJavaOptions` for the JVM options.

If the optional `SPARK_JAVA_OPTS` environment variable is defined, it is added to the JVM options.

It uses the optional `spark.executor.extraLibraryPath` to set `prefixEnv`. It uses `client.getClusterPath`.

Caution	<code>FIXME Client.getClusterPath ?</code>
---------	--

It sets `-Dspark.yarn.app.container.log.dir=<LOG_DIR>` It sets the user classpath (using `client.getUserClasspath`).

Caution	<code>FIXME Client.getUserClasspath ?</code>
---------	--

Finally, it creates the entire command to start `org.apache.spark.executor.CoarseGrainedExecutorBackend` with the following arguments:

- `--driver-url` being the input `masterAddress`
- `--executor-id` being the input `slaveId`
- `--hostname` being the input `hostname`
- `--cores` being the input `executorCores`
- `--app-id` being the input `appId`

## Internal Registries

### yarnConf

`yarnConf` is an instance of YARN's `YarnConfiguration`. It is created when `ExecutorRunnable` is.

# Client

`Client` (package: `org.apache.spark.deploy.yarn`) is a handle to a YARN cluster to deploy [ApplicationMaster](#) (for a Spark application being deployed to a YARN cluster).

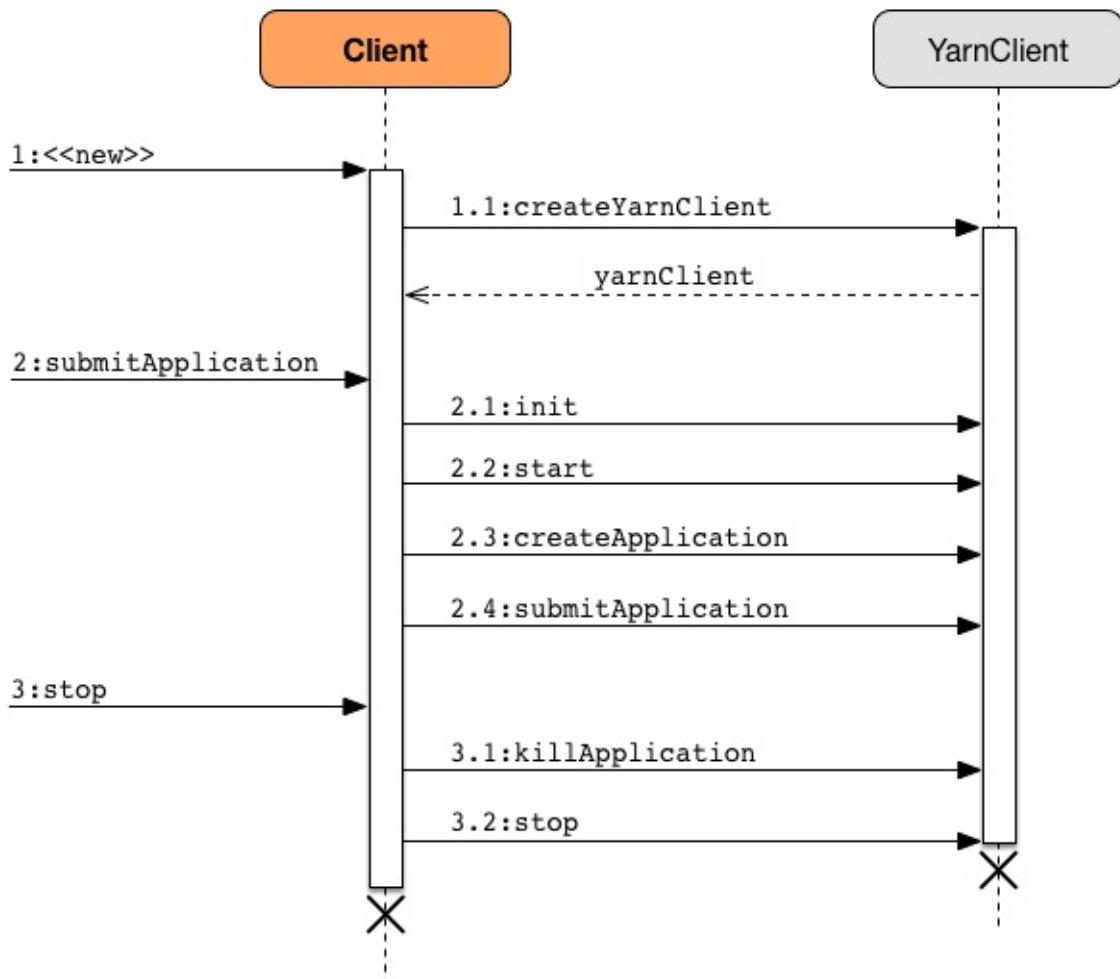


Figure 1. Client and Hadoop's YarnClient Interactions

Depending on the [deploy mode](#) it uses [ApplicationMaster](#) or ApplicationMaster's wrapper [ExecutorLauncher](#) by their class names in a [ContainerLaunchContext](#) (that represents all of the information needed by the YARN NodeManager to launch a container).

It was initially used as a [standalone application](#) to [submit Spark applications](#) to a YARN cluster, but is currently considered obsolete.

Enable `INFO` or `DEBUG` logging level for `org.apache.spark.deploy.yarn.Client` logger to see what happens inside.

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

Tip

```
log4j.logger.org.apache.spark.deploy.yarn.Client=DEBUG
```

Refer to [Logging](#).

## Creating Client Instance

Creating an instance of `Client` does the following:

- Creates an internal instance of `YarnClient` (using `YarnClient.createYarnClient`) that becomes `yarnClient`.
- Creates an internal instance of `YarnConfiguration` (using `YarnConfiguration` and the input `hadoopConf`) that becomes `yarnConf`.
- Sets the internal `isClusterMode` that says whether `spark.submit.deployMode` is `cluster deploy mode`.
- Sets the internal `amMemory` to `spark.driver.memory` when `isClusterMode` is enabled or `spark.yarn.am.memory` otherwise.
- Sets the internal `amMemoryOverhead` to `spark.yarn.driver.memoryOverhead` when `isClusterMode` is enabled or `spark.yarn.am.memoryOverhead` otherwise. If neither is available, the maximum of 10% of `amMemory` and `384` is chosen.
- Sets the internal `amCores` to `spark.driver.cores` when `isClusterMode` is enabled or `spark.yarn.am.cores` otherwise.
- Sets the internal `executorMemory` to `spark.executor.memory`.
- Sets the internal `executorMemoryOverhead` to `spark.yarn.executor.memoryOverhead`. If unavailable, it is set to the maximum of 10% of `executorMemory` and `384`.
- Creates an internal instance of `ClientDistributedCacheManager` (as `distCacheMgr`).
- Sets the variables: `loginFromKeytab` to `false` with `principal`, `keytab`, and `credentials` to `null`.
- Creates an internal instance of `LauncherBackend` (as `launcherBackend`).
- Sets the internal `fireAndForget` flag to the result of `isClusterMode` and not `spark.yarn.submit.waitAppCompletion`.

- Sets the internal variable `appId` to `null`.
- Sets the internal `appStagingBaseDir` to `spark.yarn.stagingDir` or the home directory of Hadoop.

## Submitting Spark Application to YARN (submitApplication method)

When `YarnClientSchedulerBackend` starts, it creates a new instance of `Client` and executes `submitApplication`.

```
submitApplication(): ApplicationId
```

`submitApplication` submits a Spark application (represented by `ApplicationMaster`) to a YARN cluster (i.e. to the `YARN ResourceManager`) and returns the application's `ApplicationId`.

Note	<code>submitApplication</code> is also used in the currently-deprecated <code>Client.run</code> .
------	---

Internally, it executes `LauncherBackend.connect` first and then executes `Client.setupCredentials` to set up credentials for future calls.

It then inits the internal `yarnClient` (with the internal `yarnConf`) and starts it. All this happens using Hadoop API.

Caution	<code>FIXME How to configure YarnClient ? What is YARN's YarnClient.getYarnClusterMetrics ?</code>
---------	--

You should see the following INFO in the logs:

```
INFO Client: Requesting a new application from cluster with [count] NodeManagers
```

It then `YarnClient.createApplication()` to create a new application in YARN and obtains the application id.

The `LauncherBackend` instance changes state to SUBMITTED with the application id.

Caution	<code>FIXME Why is this important?</code>
---------	---

`submitApplication` verifies whether the cluster has resources for the `ApplicationManager` (using `verifyClusterResources`).

It then creates YARN `ContainerLaunchContext` followed by creating YARN `ApplicationSubmissionContext`.

You should see the following INFO message in the logs:

```
INFO Client: Submitting application [appId] to ResourceManager
```

`submitApplication` submits the new YARN `ApplicationSubmissionContext` for [ApplicationMaster](#) to YARN (using Hadoop's `YarnClient.submitApplication`).

It returns the YARN [ApplicationId](#) for the Spark application (represented by [ApplicationMaster](#)).

## loginFromKeytab

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

## Creating YARN ApplicationSubmissionContext (`createApplicationSubmissionContext` method)

```
createApplicationSubmissionContext(  
    newApp: YarnClientApplication,  
    containerContext: ContainerLaunchContext): ApplicationSubmissionContext
```

`createApplicationSubmissionContext` creates YARN's [ApplicationSubmissionContext](#).

Note	YARN's <code>ApplicationSubmissionContext</code> represents all of the information needed by the <a href="#">YARN ResourceManager</a> to launch the <a href="#">ApplicationMaster</a> for a Spark application.
------	--

`createApplicationSubmissionContext` uses YARN's [YarnClientApplication](#) (as the input `newApp`) to create a `ApplicationSubmissionContext`.

`createApplicationSubmissionContext` sets the following information in the `ApplicationSubmissionContext`:

The name of the Spark application	<code>spark.app.name</code> configuration setting or <code>Spark</code> if not set
Queue (to which the Spark application is submitted)	<code>spark.yarn.queue</code> configuration setting
<code>ContainerLaunchContext</code> (that describes the <code>Container</code> with which the <code>ApplicationMaster</code> for the Spark application is launched)	the input <code>containerContext</code>
Type of the Spark application	<code>SPARK</code>
Tags for the Spark application	<code>spark.yarn.tags</code> configuration setting
Number of max attempts of the Spark application to be submitted.	<code>spark.yarn.maxAppAttempts</code> configuration setting
The <code>attemptFailuresValidityInterval</code> in milliseconds for the Spark application	<code>spark.yarn.am.attemptFailuresValidityInterval</code> configuration setting
Resource Capabilities for <code>ApplicationMaster</code> for the Spark application	See <a href="#">Resource Capabilities for ApplicationMaster — Memory and Virtual CPU Cores</a> section below
Rolled Log Aggregation for the Spark application	See <a href="#">Rolled Log Aggregation Configuration for Spark Application</a> section below

You will see the DEBUG message in the logs when the setting is not set:

```
DEBUG spark.yarn.maxAppAttempts is not set. Cluster's default value will be used.
```

## Resource Capabilities for ApplicationMaster — Memory and Virtual CPU Cores

Note	YARN's <a href="#">Resource</a> models a set of computer resources in the cluster. Currently, YARN supports resources with memory and virtual CPU cores capabilities only.
------	--

The requested YARN's `Resource` for the `ApplicationMaster` for a Spark application is the sum of `amMemory` and `amMemoryOverhead` for the memory and `amCores` for the virtual CPU cores.

Besides, if `spark.yarn.am.nodeLabelExpression` is set, a new YARN `ResourceRequest` is created (for the `ApplicationMaster` container) that includes:

Resource Name	* (star) that represents no locality.
Priority	0
Capability	The resource capabilities as defined above.
Number of containers	1
Node label expression	<a href="#">spark.yarn.am.nodeLabelExpression</a> configuration setting
ResourceRequest of AM container	<a href="#">spark.yarn.am.nodeLabelExpression</a> configuration setting

It sets the resource request to this new YARN `ResourceRequest` detailed in the table above.

## Rolled Log Aggregation for Spark Application

Note	YARN's <a href="#">LogAggregationContext</a> represents all of the information needed by the <a href="#">YARN NodeManager</a> to handle the logs for an application.
------	--

If `spark.yarn.rolledLog.includePattern` is defined, it creates a YARN [LogAggregationContext](#) with the following patterns:

Include Pattern	<a href="#">spark.yarn.rolledLog.includePattern</a> configuration setting
Exclude Pattern	<a href="#">spark.yarn.rolledLog.excludePattern</a> configuration setting

## Verifying Maximum Memory Capability of YARN Cluster (`verifyClusterResources` private method)

```
verifyClusterResources(newAppResponse: GetNewApplicationResponse): Unit
```

`verifyClusterResources` is a private helper method that `submitApplication` uses to ensure that the Spark application (as a set of [ApplicationMaster](#) and executors) is not going to request more than the maximum memory capability of the YARN cluster. If so, it throws an `IllegalArgumentException`.

`verifyClusterResources` queries the input `GetNewApplicationResponse` (as `newAppResponse`) for the maximum memory.

```
INFO Client: Verifying our application has not requested more than the maximum memory capability of the cluster ([maximumMemory] MB per container)
```

If the maximum memory capability is above the required executor or `ApplicationMaster` memory, you should see the following INFO message in the logs:

```
INFO Client: Will allocate AM container, with [amMem] MB memory including [amMemoryOverhead] MB overhead
```

If however the executor memory (as a sum of `spark.executor.memory` and `spark.yarn.executor.memoryOverhead` settings) is more than the maximum memory capability, `verifyClusterResources` throws an `IllegalArgumentException` with the following message:

```
Required executor memory ([executorMemory]+[executorMemoryOverhead] MB) is above the max threshold ([maximumMemory] MB) of this cluster! Please check the values of 'yarn.scheduler.maximum-allocation-mb' and/or 'yarn.nodemanager.resource.memory-mb'.
```

If the required memory for `ApplicationMaster` is more than the maximum memory capability, `verifyClusterResources` throws an `IllegalArgumentException` with the following message:

```
Required AM memory ([amMemory]+[amMemoryOverhead] MB) is above the max threshold ([maximumMemory] MB) of this cluster! Please increase the value of 'yarn.scheduler.maximum-allocation-mb'.
```

## Creating Hadoop YARN's ContainerLaunchContext for Launching ApplicationMaster (`createContainerLaunchContext` private method)

When a Spark application is submitted to YARN, it calls the private helper method `createContainerLaunchContext` that creates a YARN `ContainerLaunchContext` request for YARN NodeManager to launch `ApplicationMaster` (in a container).

```
createContainerLaunchContext(newAppResponse: GetNewApplicationResponse): ContainerLaunchContext
```

**Note** The input `newAppResponse` is Hadoop YARN's [GetNewApplicationResponse](#).

When called, you should see the following INFO message in the logs:

```
INFO Setting up container launch context for our AM
```

It gets at the application id (from the input `newAppResponse` ).

It calculates the path of the application's staging directory.

**Caution** [FIXME](#) What's `appStagingBaseDir` ?

It does a *custom* step for a Python application.

It [sets up an environment to launch ApplicationMaster container](#) and [prepareLocalResources](#). A `ContainerLaunchContext` record is created with the environment and the local resources.

The JVM options are calculated as follows:

- `-Xmx` (that [was calculated when the Client was created](#))
- `-Djava.io.tmpdir=` - [FIXME](#): `tmpDir`

**Caution** [FIXME](#) `tmpDir` ?

- Using `useConcMarkSweepGC` when `SPARK_USE_CONC_INCR_GC` is enabled.

**Caution** [FIXME](#) `SPARK_USE_CONC_INCR_GC` ?

- In cluster deploy mode, ...[FIXME](#)
- In client deploy mode, ...[FIXME](#)

**Caution** [FIXME](#)

- `-Dspark.yarn.app.container.log.dir=` ...[FIXME](#)
- Perm gen size option...[FIXME](#)

`--class` is set if in cluster mode based on `--class` command-line argument.

**Caution** [FIXME](#)

If `--jar` command-line argument was specified, it is set as `--jar`.

In cluster deploy mode, `org.apache.spark.deploy.yarn.ApplicationMaster` is created while in client deploy mode it is `org.apache.spark.deploy.yarn.ExecutorLauncher`.

If `--arg` command-line argument was specified, it is set as `--arg`.

The path for `--properties-file` is built based on

```
YarnSparkHadoopUtil.expandEnvironment(Environment.PWD), LOCALIZED_CONF_DIR,
SPARK_CONF_FILE
```

The entire `ApplicationMaster` argument line (as `amArgs`) is of the form:

```
[amClassName] --class [userClass] --jar [userJar] --arg [userArgs] --properties-file [propFile]
```

The entire command line is of the form:

Caution	<code>FIXME prefixEnv ? How is path calculated?</code> <code>ApplicationConstants.LOG_DIR_EXPANSION_VAR ?</code>
---------	---

```
[JAVA_HOME]/bin/java -server [java0pts] [amArgs] 1> [LOG_DIR]/stdout 2> [LOG_DIR]/stderr
```

The command line to launch a `ApplicationMaster` is set to the `ContainerLaunchContext` record (using `setCommands`).

You should see the following DEBUG messages in the logs:

```
DEBUG Client: =====
=====
DEBUG Client: YARN AM launch context:
DEBUG Client:   user class: N/A
DEBUG Client:   env:
DEBUG Client:     [launchEnv]
DEBUG Client:   resources:
DEBUG Client:     [localResources]
DEBUG Client:   command:
DEBUG Client:     [commands]
DEBUG Client: =====
=====
```

A `SecurityManager` is created and set as the application's ACLs.

Caution	<code>FIXME setApplicationACLs ? Set up security tokens?</code>
---------	---

## prepareLocalResources method

Caution

[FIXME](#)

```
prepareLocalResources(  
    destDir: Path,  
    pySparkArchives: Seq[String]): HashMap[String, LocalResource]
```

`prepareLocalResources` is...[FIXME](#)

Caution

[FIXME](#) Describe `credentialManager`

When called, `prepareLocalResources` prints out the following INFO message to the logs:

```
INFO Client: Preparing resources for our AM container
```

Caution

[FIXME](#) What's a delegation token?

`prepareLocalResources` then obtains security tokens from credential providers and gets the nearest time of the next renewal (for renewable credentials).

After all the security delegation tokens are obtained and only when there are any, you should see the following DEBUG message in the logs:

```
DEBUG Client: [token1]  
DEBUG Client: [token2]  
...  
DEBUG Client: [tokenN]
```

Caution

[FIXME](#) Where is `credentials` assigned?

If a keytab is used to log in and the nearest time of the next renewal is in the future, `prepareLocalResources` sets the internal `spark.yarn.credentials.renewalTime` and `spark.yarn.credentials.updateTime` times for renewal and update security tokens.

It gets the replication factor (using `spark.yarn.submit.file.replication` setting) or falls back to the default value for the input `destDir`.

Note

The replication factor is only used for `copyFileToRemote` later. Perhaps it should not be mentioned here (?)

It creates the input `destDir` (on a HDFS-compatible file system) with `0700` permission (`rwx-----`), i.e. inaccessible to all but its owner and the superuser so the owner only can read, write and execute. It uses Hadoop's `Path.getFileSystem` to access Hadoop's

[FileSystem](#) that owns `destDir` (using the constructor's `hadoopConf` — Hadoop's Configuration).

Tip	See <a href="#">org.apache.hadoop.fs.FileSystem</a> to know a list of HDFS-compatible file systems, e.g. <a href="#">Amazon S3</a> or <a href="#">Windows Azure</a> .
-----	---

If a keytab is used to log in, ...[FIXME](#)

Caution	<a href="#">FIXME</a> if ( <code>loginFromKeytab</code> )
---------	---

If the [location of the single archive containing Spark jars \(spark.yarn.archive\)](#) is set, it is [distributed](#) (as ARCHIVE) to `spark_libs`.

Else if the [location of the Spark jars \(spark.yarn.jars\)](#) is set, ...[FIXME](#)

Caution	<a href="#">FIXME</a> Describe <code>case Some(jars)</code>
---------	---

If neither `spark.yarn.archive` nor `spark.yarn.jars` is set, you should see the following WARN message in the logs:

```
WARN Client: Neither spark.yarn.jars nor spark.yarn.archive is set, falling back to up
loading libraries under SPARK_HOME.
```

It then finds the directory with jar files under `SPARK_HOME` (using `YarnCommandBuilderUtils.findJarsDir`).

Caution	<a href="#">FIXME</a> <code>YarnCommandBuilderUtils.findJarsDir</code>
---------	--

And all the jars are zipped to a temporary archive, e.g. `spark_libs2944590295025097383.zip` that is `distribute as ARCHIVE to spark_libs` (only when they differ).

If a user jar ( `--jar` ) was specified on command line, the jar is `distribute as FILE to app.jar`.

It then [distributes](#) additional resources specified in SparkConf for the application, i.e. jars (under `spark.yarn.dist.jars`), files (under `spark.yarn.dist.files`), and archives (under `spark.yarn.dist.archives`).

Note	The additional files to distribute can be defined using <a href="#">spark-submit</a> using command-line options <code>--jars</code> , <code>--files</code> , and <code>--archives</code> .
------	--

Caution	<a href="#">FIXME</a> Describe <code>distribute</code>
---------	--

It sets `spark.yarn.secondary.jars` for the jars that have localized path (non-local paths) or their path (for local paths).

It updates Spark configuration (with internal configuration settings using the internal `distCacheMgr` reference).

**Caution**

**FIXME** Where are they used? It appears they are required for `ApplicationMaster` when it prepares local resources, but what is the sequence of calls to lead to `ApplicationMaster`?

It uploads `spark_conf.zip` to the input `destDir` and sets `spark.yarn.cache.confArchive`

It creates configuration archive and `copyFileToRemote(destDir, localConfArchive, replication, force = true, destName = Some(LOCALIZED_CONF_ARCHIVE))`.

**Caution**

**FIXME** `copyFileToRemote(destDir, localConfArchive, replication, force = true, destName = Some(LOCALIZED_CONF_ARCHIVE))`?

It adds a cache-related resource (using the internal `distCacheMgr`).

**Caution**

**FIXME** What resources? Where? Why is this needed?

Ultimately, it clears the cache-related internal configuration settings — `spark.yarn.cache.filenames`, `spark.yarn.cache.sizes`, `spark.yarn.cache.timestamps`, `spark.yarn.cache.visibilities`, `spark.yarn.cache.types`, `spark.yarn.cache.confArchive` — from the `SparkConf` configuration since they are internal and should not "pollute" the web UI's environment page.

The `localResources` are returned.

**Caution**

**FIXME** How is `localResources` calculated?

**Note**

It is exclusively used when Client creates a `ContainerLaunchContext` to launch a `ApplicationMaster` container.

## Creating \_\_spark\_conf\_\_.zip Archive With Configuration Files and Spark Configuration (createConfArchive private method)

`createConfArchive(): File`

`createConfArchive` is a private helper method that `prepareLocalResources` uses to create an archive with the local config files — `log4j.properties` and `metrics.properties` (before distributing it and the other files for `ApplicationMaster` and executors to use on a YARN cluster).

The archive will also contain all the files under `HADOOP_CONF_DIR` and `YARN_CONF_DIR` environment variables (if defined).

Additionally, the archive contains a `spark_conf.properties` with the current [Spark configuration](#).

The archive is a temporary file with the `spark_conf` prefix and `.zip` extension with the files above.

## Copying File to Remote File System (`copyFileToRemote` helper method)

```
copyFileToRemote(
  destDir: Path,
  srcPath: Path,
  replication: Short,
  force: Boolean = false,
  destName: Option[String] = None): Path
```

`copyFileToRemote` is a `private[yarn]` method to copy `srcPath` to the remote file system `destDir` (if needed) and return the destination path resolved following symlinks and mount points.

Note	It is exclusively used in <a href="#">prepareLocalResources</a> .
------	---

Unless `force` is enabled (it is disabled by default), `copyFileToRemote` will only copy `srcPath` when the source (of `srcPath`) and target (of `destDir`) file systems are the same.

You should see the following INFO message in the logs:

```
INFO Client: Uploading resource [srcPath] -> [destPath]
```

`copyFileToRemote` copies `srcPath` to `destDir` and sets `644` permissions, i.e. world-wide readable and owner writable.

If `force` is disabled or the files are the same, `copyFileToRemote` will only print out the following INFO message to the logs:

```
INFO Client: Source and destination file systems are the same. Not copying [srcPath]
```

Ultimately, `copyFileToRemote` returns the destination path resolved following symlinks and mount points.

## Populating CLASSPATH for ApplicationMaster and Executors (populateClasspath method)

```
populateClasspath(
    args: ClientArguments,
    conf: Configuration,
    sparkConf: SparkConf,
    env: HashMap[String, String],
    extraClassPath: Option[String] = None): Unit
```

`populateClasspath` is a `private[yarn]` helper method that populates the CLASSPATH (for `ApplicationMaster` and `executors`).

**Note**

The input `args` is `null` when preparing environment for `ExecutorRunnable` and the constructor's `args` for `client`.

It merely adds the following entries to the CLASSPATH key in the input `env`:

1. The optional `extraClassPath` (which is first changed to include paths on YARN cluster machines).

**Note**

`extraClassPath` corresponds to `spark.driver.extraClassPath` for the driver and `spark.executor.extraClassPath` for executors.

2. YARN's own `Environment.PWD`
3. `__spark_conf__` directory under YARN's `Environment.PWD`
4. If the deprecated `spark.yarn.user.classpath.first` is set, ...FIXME

**Caution**

FIXME

5. `__spark_libs__/*` under YARN's `Environment.PWD`
6. (unless the optional `spark.yarn.archive` is defined) All the `local` jars in `spark.yarn.jars` (which are first changed to be paths on YARN cluster machines).
7. All the entries from YARN's `yarn.application.classpath` or  
`YarnConfiguration.DEFAULT_YARN_APPLICATION_CLASSPATH` (if `yarn.application.classpath` is not set)
8. All the entries from YARN's `mapreduce.application.classpath` or  
`MRJobConfig.DEFAULT_MAPREDUCE_APPLICATION_CLASSPATH` (if `mapreduce.application.classpath` not set).
9. `SPARK_DIST_CLASSPATH` (which is first changed to include paths on YARN cluster machines).

**Tip**

You should see the result of executing `populateClasspath` when you enable `DEBUG`

```
DEBUG Client:      env:
DEBUG Client:      CLASSPATH -> <CPS>/__spark_conf__<CPS>/__spark_libs__/*<CP:
```

## Changing Path to be YARN NodeManager-aware (`getClusterPath` method)

```
getClusterPath(conf: SparkConf, path: String): String
```

`getClusterPath` replaces any occurrences of `spark.yarn.config.gatewayPath` in `path` to the value of `spark.yarn.config.replacementPath`.

## Adding CLASSPATH Entry to Environment (`addClasspathEntry` method)

```
addClasspathEntry(path: String, env: HashMap[String, String]): Unit
```

`addClasspathEntry` is a private helper method to add the input `path` to `CLASSPATH` key in the input `env`.

## Distributing Files to Remote File System (`distribute` private method)

```
distribute(
  path: String,
  resType: LocalResourceType = LocalResourceType.FILE,
  destName: Option[String] = None,
  targetDir: Option[String] = None,
  appMasterOnly: Boolean = false): (Boolean, String)
```

`distribute` is an internal helper method that `prepareLocalResources` uses to find out whether the input `path` is of `local:` URI scheme and return a localized path for a non-`local` path, or simply the input `path` for a local one.

`distribute` returns a pair with the first element being a flag for the input `path` being local or non-local, and the other element for the local or localized path.

For local `path` that was not distributed already, `distribute` copies the input `path` to remote file system (if needed) and adds `path` to the application's distributed cache.

## Joining Path Components using Path.SEPARATOR (buildPath method)

```
buildPath(components: String*): String
```

`buildPath` is a helper method to join all the path `components` using the directory separator, i.e. `org.apache.hadoop.fs.Path.SEPARATOR`.

## isClusterMode Internal Flag

`isClusterMode` is an internal flag that says whether the Spark application runs in `cluster` or `client` deploy mode. The flag is enabled for `cluster` deploy mode, i.e. `true`.

Note	Since a Spark application requires different settings per deploy mode, <code>isClusterMode</code> flag effectively "splits" <code>Client</code> on two parts per deploy mode—one responsible for <code>client</code> and the other for <code>cluster</code> deploy mode.
------	--

Caution	<code>FIXME</code> Replace the internal fields used below with their true meanings.
---------	---

Table 1. Internal Attributes of `Client` per Deploy Mode (`isClusterMode`)

Internal attribute	cluster deploy mode	client deploy mode
<code>amMemory</code>	<code>spark.driver.memory</code>	<code>spark.yarn.am.memory</code>
<code>amMemoryOverhead</code>	<code>spark.yarn.driver.memoryOverhead</code>	<code>spark.yarn.am.memoryOverhead</code>
<code>amCores</code>	<code>spark.driver.cores</code>	<code>spark.yarn.am.cores</code>
<code>javaOpts</code>	<code>spark.driver.extraJavaOptions</code>	<code>spark.yarn.am.extraJavaOptions</code>
<code>libraryPaths</code>	<code>spark.driver.extraLibraryPath</code> and <code>spark.driver.libraryPath</code>	<code>spark.yarn.am.extraLibraryPath</code> and <code>spark.yarn.libraryPath</code>
<code>--class</code> <code>command-line</code> <code>argument for</code> <code>ApplicationMaster</code>	<code>args.userClass</code>	
Application master class	<code>org.apache.spark.deploy.yarn.ApplicationMaster</code>	<code>org.apache.spark.deploy.yarn.Client</code>

When the `isClusterMode` flag is enabled, the internal reference to YARN's `YarnClient` is used to stop application.

When the `isClusterMode` flag is enabled (and `spark.yarn.submit.waitAppCompletion` is disabled), so is `fireAndForget` internal flag.

## launcherBackend value

`launcherBackend` ...[FIXME](#)

## SPARK\_YARN\_MODE Flag

`SPARK_YARN_MODE` is a flag that says whether...[FIXME](#).

Note	Any environment variable with the <code>SPARK_</code> prefix is propagated to all (remote) processes.
------	---

Caution	<a href="#">FIXME</a> Where is <code>SPARK_</code> prefix rule enforced?
---------	--

Note	<code>SPARK_YARN_MODE</code> is a system property (i.e. available using <code>System.getProperty()</code> ) and a environment variable (i.e. available using <code>System.getenv</code> or <code>sys.env</code> ). See <a href="#">YarnSparkHadoopUtil</a> .
------	--

It is enabled (i.e. `true`) when [SparkContext](#) is created for Spark on YARN in [client deploy mode](#), when `client` sets up an environment to launch `ApplicationMaster` container (and, what is currently considered deprecated, a [Spark application was deployed to a YARN cluster](#)).

Caution	<a href="#">FIXME</a> Why is this needed? <code>git blame</code> it.
---------	--

`SPARK_YARN_MODE` flag is checked when [YarnSparkHadoopUtil](#) or [SparkHadoopUtil](#) are accessed.

It is cleared later when [Client is requested to stop](#).

## Setting Up Environment to Launch ApplicationMaster Container (`setupLaunchEnv` method)

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Internal LauncherBackend (launcherBackend value)

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Internal Hadoop's YarnClient (yarnClient value)

```
val yarnClient = YarnClient.createYarnClient
```

`yarnClient` is a private internal reference to Hadoop's [YarnClient](#) that `Client` uses to [create and submit a YARN application](#) (for your Spark application), [killApplication](#).

`yarnClient` is initied and started when `Client` submits a Spark application to a YARN cluster.

`yarnClient` is stopped when `Client` stops.

## main

`main` method is invoked while a Spark application is being deployed to a YARN cluster.

Note	It is executed by <a href="#">spark-submit</a> with <code>--master yarn</code> command-line argument.
------	---

Note	When you start the <code>main</code> method when starting the <code>Client</code> standalone application <code>org.apache.spark.deploy.yarn.Client</code> , you will see the following WARN message in
------	--

`WARN Client: WARNING: This client is deprecated and will be removed in a future`

`main` turns [SPARK\\_YARN\\_MODE](#) flag on.

It then instantiates [SparkConf](#), parses command-line arguments (using [ClientArguments](#)) and passes the call on to [Client.run](#) method.

## stop

```
stop(): Unit
```

`stop` closes the internal [LauncherBackend](#) and stops the internal `yarnClient`.

It also clears [SPARK\\_YARN\\_MODE](#) flag (to allow switching between cluster types).

## run

`run` submits a Spark application to a YARN ResourceManager (RM).

If `LauncherBackend` is not connected to a RM, i.e. `LauncherBackend.isConnected` returns `false`, and `fireAndForget` is enabled, ...[FIXME](#)

**Caution**

**FIXME** When could `LauncherBackend` lost the connection since it was connected in `submitApplication`?

**Caution**

**FIXME** What is `fireAndForget` ?

Otherwise, when `LauncherBackend` is connected or `fireAndForget` is disabled, `monitorApplication` is called. It returns a pair of `yarnApplicationState` and `finalApplicationStatus` that is checked against three different state pairs and throw a `SparkException`:

- `YarnApplicationState.KILLED` or `FinalApplicationStatus.KILLED` lead to `SparkException` with the message "Application [appId] is killed".
- `YarnApplicationState.FAILED` or `FinalApplicationStatus.FAILED` lead to `SparkException` with the message "Application [appId] finished with failed status".
- `FinalApplicationStatus.UNDEFINED` leads to `SparkException` with the message "The final status of application [appId] is undefined".

**Caution**

**FIXME** What are `YarnApplicationState` and `FinalApplicationStatus` statuses?

## monitorApplication

```
monitorApplication(
    appId: ApplicationId,
    returnOnRunning: Boolean = false,
    logApplicationReport: Boolean = true): (YarnApplicationState, FinalApplicationStatus)
```

`monitorApplication` continuously reports the status of a Spark application `appId` every `spark.yarn.report.interval` until the application state is one of the following `YarnApplicationState`:

- `RUNNING` (when `returnOnRunning` is enabled)
- `FINISHED`
- `FAILED`
- `KILLED`

**Note**

It is used in `run`, `YarnClientSchedulerBackend.waitForApplication` and `MonitorThread.run`.

It gets the application's report from the [YARN ResourceManager](#) to obtain [YarnApplicationState](#) of the [ApplicationMaster](#).

**Tip**

It uses Hadoop's `YarnClient.getApplicationReport(appId)`.

Unless `logApplicationReport` is disabled, it prints the following INFO message to the logs:

```
INFO Client: Application report for [appId] (state: [state])
```

If `logApplicationReport` and DEBUG log level are enabled, it prints report details every time interval to the logs:

```
16/04/23 13:21:36 INFO Client:
  client token: N/A
  diagnostics: N/A
  ApplicationMaster host: N/A
  ApplicationMaster RPC port: -1
  queue: default
  start time: 1461410495109
  final status: UNDEFINED
  tracking URL: http://japila.local:8088/proxy/application_1461410200840_0001/
  user: jacek
```

For INFO log level it prints report details only when the application state changes.

When the application state changes, `LauncherBackend` is notified (using `LauncherBackend.setState`).

**Note**

The application state is an instance of Hadoop's [YarnApplicationState](#).

For states `FINISHED`, `FAILED` or `KILLED`, [cleanupStagingDir](#) is called and the method finishes by returning a pair of the current state and the final application status.

If `returnOnRunning` is enabled (it is disabled by default) and the application state turns `RUNNING`, the method returns a pair of the current state `RUNNING` and the final application status.

**Note**

[cleanupStagingDir](#) won't be called when `returnOnRunning` is enabled and an application turns `RUNNING`. I guess it is likely a left-over since the Client is deprecated now.

The current state is recorded for future checks (in the loop).

## cleanupStagingDir

`cleanupStagingDir` clears the staging directory of an application.

**Note**

It is used in [submitApplication](#) when there is an exception and [monitorApplication](#) when an application finishes and the method quits.

It uses `spark.yarn.stagingDir` setting or falls back to a user's home directory for the staging directory. If [cleanup is enabled](#), it deletes the entire staging directory for the application.

You should see the following INFO message in the logs:

```
INFO Deleting staging directory [stagingDirPath]
```

## reportLauncherState

```
reportLauncherState(state: SparkAppHandle.State): Unit
```

`reportLauncherState` merely passes the call on to `LauncherBackend.setState`.

**Caution**

What does `setState` do?

## ClientArguments

# YarnRMClient

`YarnRMClient` is responsible for [registering](#) and [unregistering](#) a Spark application (in the form of [ApplicationMaster](#)) with [YARN ResourceManager](#) (and hence *RM* in the name). It is a mere wrapper for [AMRMClient\[ContainerRequest\]](#) that is started when [registering ApplicationMaster](#) (and never stopped explicitly!).

Besides being responsible for [registration](#) and [unregistration](#), it also knows the [application attempt identifiers](#) and [tracks the maximum number of attempts to register ApplicationMaster](#).

**Tip** Enable `INFO` logging level for `org.apache.spark.deploy.yarn.YarnRMClient` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.deploy.yarn.YarnRMClient=INFO
```

Refer to [Logging](#).

## Registering ApplicationMaster with YARN ResourceManager (register method)

To [register ApplicationMaster](#) (for a Spark application) with the YARN ResourceManager, Spark uses `register`.

```
register(
  driverUrl: String,
  driverRef: RpcEndpointRef,
  conf: YarnConfiguration,
  sparkConf: SparkConf,
  uiAddress: String,
  uiHistoryAddress: String,
  securityMgr: SecurityManager,
  localResources: Map[String, LocalResource]): YarnAllocator
```

`register` instantiates YARN's [AMRMClient](#), initializes it (using `conf` input parameter) and starts immediately. It saves `uiHistoryAddress` input parameter internally for later use.

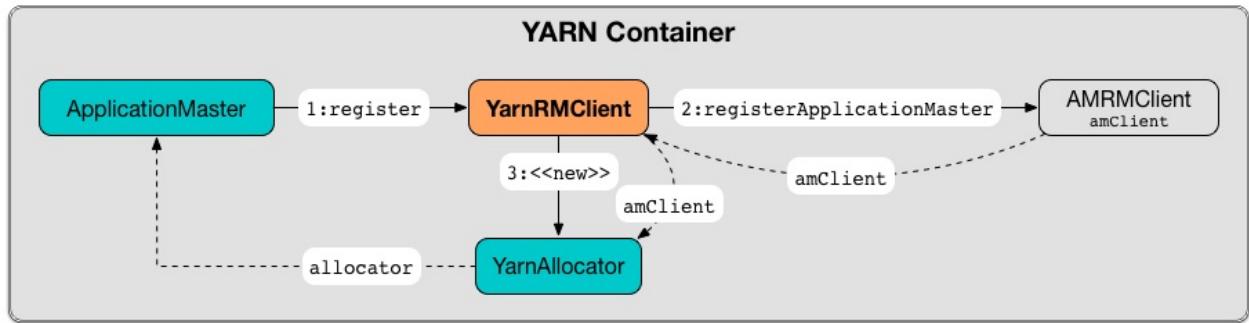


Figure 1. Registering ApplicationMaster with YARN ResourceManager

You should see the following INFO message in the logs (in stderr in YARN):

```
INFO YarnRMClient: Registering the ApplicationMaster
```

It then registers the application master on the local host with port `0` and `uiAddress` input parameter for the URL at which the master info can be seen.

The internal `registered` flag is enabled.

Ultimately, it creates a new `YarnAllocator` with the input parameters of `register` passed in and the just-created YARN `AMRMClient`.

## Unregistering ApplicationMaster from YARN ResourceManager (unregister method)

```
unregister(status: FinalApplicationStatus, diagnostics: String = ""): Unit
```

`unregister` unregisters the ApplicationMaster of a Spark application.

It basically checks that `ApplicationMaster` is registered and only when it requests the internal `AMRMClient` to unregister.

`unregister` is called when `ApplicationMaster` wants to unregister.

## Maximum Number of Attempts to Register ApplicationMaster (getMaxRegAttempts method)

```
getMaxRegAttempts(sparkConf: SparkConf, yarnConf: YarnConfiguration): Int
```

`getMaxRegAttempts` uses `SparkConf` and YARN's `YarnConfiguration` to read configuration settings and return the maximum number of application attempts before `ApplicationMaster` registration with YARN is considered unsuccessful (and so the Spark application).

It reads YARN's `yarn.resourcemanager.am.max-attempts` (available as [YarnConfiguration.RM\\_AM\\_MAX\\_ATTEMPTS](#)) or falls back to [YarnConfiguration.DEFAULT\\_RM\\_AM\\_MAX\\_ATTEMPTS](#) (which is `2`).

The return value is the minimum of the configuration settings of YARN and Spark.

## Getting ApplicationAttemptId of Spark Application (`getAttemptId` method)

```
getAttemptId(): ApplicationAttemptId
```

`getAttemptId` returns YARN's `ApplicationAttemptId` (of the Spark application to which the container was assigned).

Internally, it uses YARN-specific methods like [ConverterUtils.toContainerId](#) and [ContainerId.getApplicationAttemptId](#).

## getAmIpFilterParams

Caution	<a href="#">FIXME</a>
---------	-----------------------

# ApplicationMaster (aka ExecutorLauncher)

`ApplicationMaster` class acts as the [YARN ApplicationMaster](#) for a Spark application running on a YARN cluster (which is commonly called [Spark on YARN](#)).

It uses [YarnAllocator](#) to manage YARN containers for executors.

`ApplicationMaster` is a [standalone application](#) that [YARN NodeManager](#) runs inside a YARN resource container and is responsible for the execution of a Spark application on YARN.

When [created](#) `ApplicationMaster` class is given a [YarnRMClient](#) (which is responsible for registering and unregistering a Spark application).

	<p><code>ExecutorLauncher</code> is a custom <code>ApplicationMaster</code> for <a href="#">client deploy mode</a> only for 1 executor.</p> <pre>\$ jps -lm 71253 org.apache.spark.deploy.yarn.ExecutorLauncher --arg 1 dir/usercache/jacek/appcache/application_1468961163409_0001 70631 org.apache.hadoop.yarn.server.resourcemanager.Resource 70934 org.apache.spark.deploy.SparkSubmit --master yarn --c 71320 sun.tools.jps.Jps -lm 70731 org.apache.hadoop.yarn.server.nodemanager.NodeManager</pre>
Note	

`ApplicationMaster` (and `ExecutorLauncher`) is launched as a result of [Client](#) creating a [ContainerLaunchContext](#) to launch Spark on YARN.

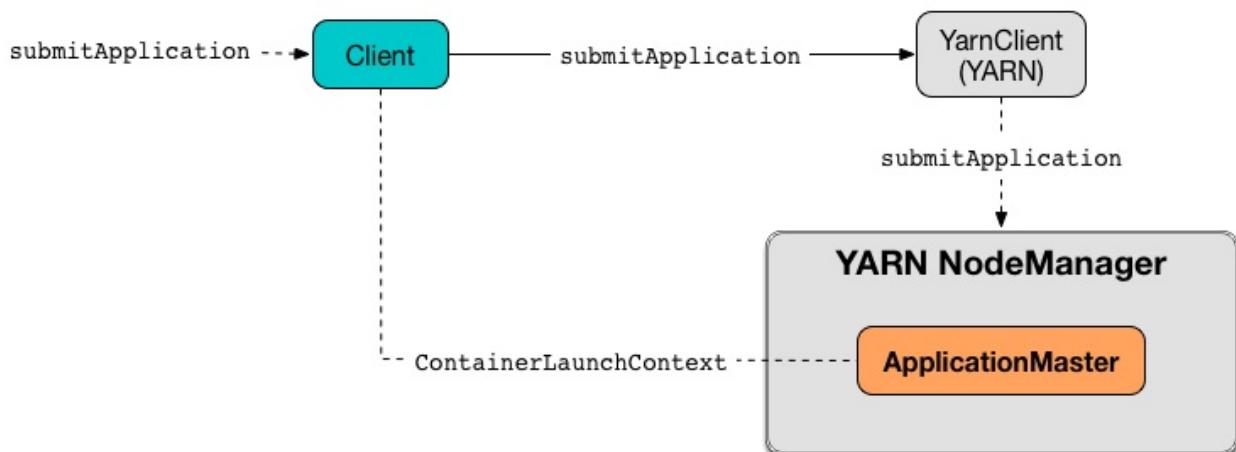


Figure 1. Launching ApplicationMaster

Note	<a href="#">ContainerLaunchContext</a> represents all of the information needed by the YARN NodeManager to launch a container.
------	--

## client Internal Reference to YarnRMClient

`client` is the internal reference to [YarnRMClient](#) that `ApplicationMaster` is given when [created](#).

`client` is primarily used to [register the ApplicationMaster](#) and request containers for executors from YARN and later [unregister ApplicationMaster](#) from YARN [ResourceManager](#).

Besides, it helps obtaining [an application attempt id](#) and [the allowed number of attempts](#) to register `ApplicationMaster`. It also [gets filter parameters](#) to secure ApplicationMaster's UI.

## allocator Internal Reference to YarnAllocator

`allocator` is the internal reference to [YarnAllocator](#) that `ApplicationMaster` uses to request new or release outstanding containers for executors.

It is [created](#) when `ApplicationMaster` is [registered](#) (using the internal [YarnRMClient reference](#)).

## main

`ApplicationMaster` is started as a standalone command-line application inside a YARN container on a node.

Note	The command-line application is executed as a result of sending a <code>containerLaunchContext</code> request to launch <code>ApplicationMaster</code> to YARN <a href="#">ResourceManager</a> (after <a href="#">creating the request for ApplicationMaster</a> )
------	--

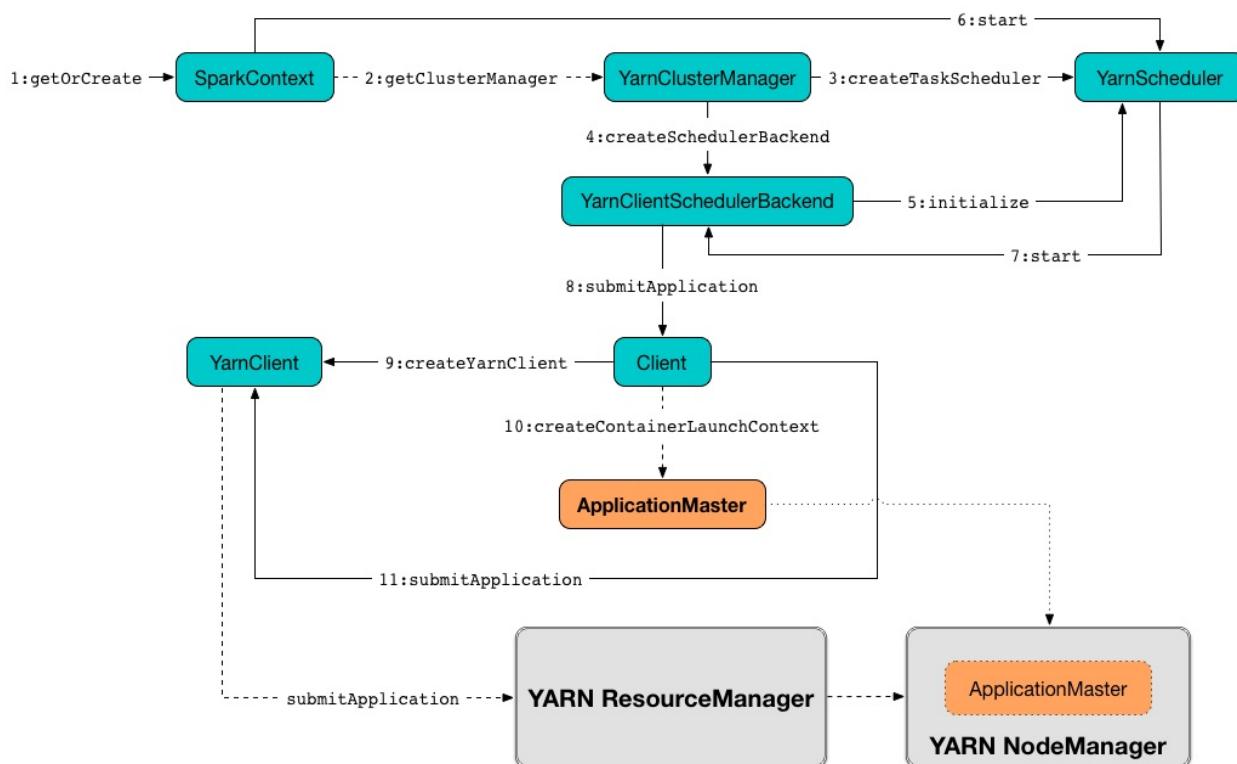


Figure 2. Submitting ApplicationMaster to YARN NodeManager

When executed, `main` first parses [command-line parameters](#) and then uses `SparkHadoopUtil.runAsSparkUser` to run the main code with a Hadoop `UserGroupInformation` as a thread local variable (distributed to child threads) for authenticating HDFS and YARN calls.

<b>Tip</b>	<p>Enable <code>DEBUG</code> logging level for <code>org.apache.spark.deploy.SparkHadoopUtil</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.deploy.SparkHadoopUtil=DEBUG</pre> <p>Refer to <a href="#">Logging</a>.</p>
------------	---

You should see the following message in the logs:

```
DEBUG running as user: [user]
```

`SparkHadoopUtil.runAsSparkUser` function executes a block that [creates a `ApplicationMaster`](#) (passing the `ApplicationMasterArguments` instance and a brand new `YarnRMClient`) and then [runs](#) it.

## Command-Line Parameters (`ApplicationMasterArguments` class)

`ApplicationMaster` uses `ApplicationMasterArguments` class to handle command-line parameters.

`ApplicationMasterArguments` is created right after `main` method has been executed for `args` command-line parameters.

It accepts the following command-line parameters:

- `--jar JAR_PATH` — the path to the Spark application's JAR file
- `--class CLASS_NAME` — the name of the Spark application's main class
- `--arg ARG` — an argument to be passed to the Spark application's main class. There can be multiple `--arg` arguments that are passed in order.
- `--properties-file FILE` — the path to a custom Spark properties file.
- `--primary-py-file FILE` — the main Python file to run.
- `--primary-r-file FILE` — the main R file to run.

When an unsupported parameter is found the following message is printed out to standard error output and `ApplicationMaster` exits with the exit code `1`.

```
Unknown/unsupported param [unknownParam]

Usage: org.apache.spark.deploy.yarn.ApplicationMaster [options]
Options:
  --jar JAR_PATH      Path to your application's JAR file
  --class CLASS_NAME  Name of your application's main class
  --primary-py-file   A main Python file
  --primary-r-file   A main R file
  --arg ARG           Argument to be passed to your application's main class.
                      Multiple invocations are possible, each will be passed in order
  .
  --properties-file FILE Path to a custom Spark properties file.
```

## Registering ApplicationMaster with YARN ResourceManager and Requesting Resources (registerAM method)

When `runDriver` or `runExecutorLauncher` are executed, they use the private helper procedure `registerAM` to register the `ApplicationMaster` (with the `YARN ResourceManager`) and `request resources` (given hints about where to allocate containers to be as close to the data as possible).

```
registerAM(
    _rpcEnv: RpcEnv,
    driverRef: RpcEndpointRef,
    uiAddress: String,
    securityMgr: SecurityManager): Unit
```

Internally, it first reads `spark.yarn.historyServer.address` setting and substitute Hadoop variables to create a complete address of the History Server, i.e.

`[address]/history/[appId]/[attemptId]` .

Caution

**FIXME** substitute Hadoop variables?

Then, `registerAM` creates a `RpcEndpointAddress` for `CoarseGrainedScheduler` RPC `Endpoint` on the driver available on `spark.driver.host` and `spark.driver.port`.

It registers the `ApplicationMaster` with the `YARN ResourceManager` and request resources (given hints about where to allocate containers to be as close to the data as possible).

Ultimately, `registerAM` launches reporter thread.

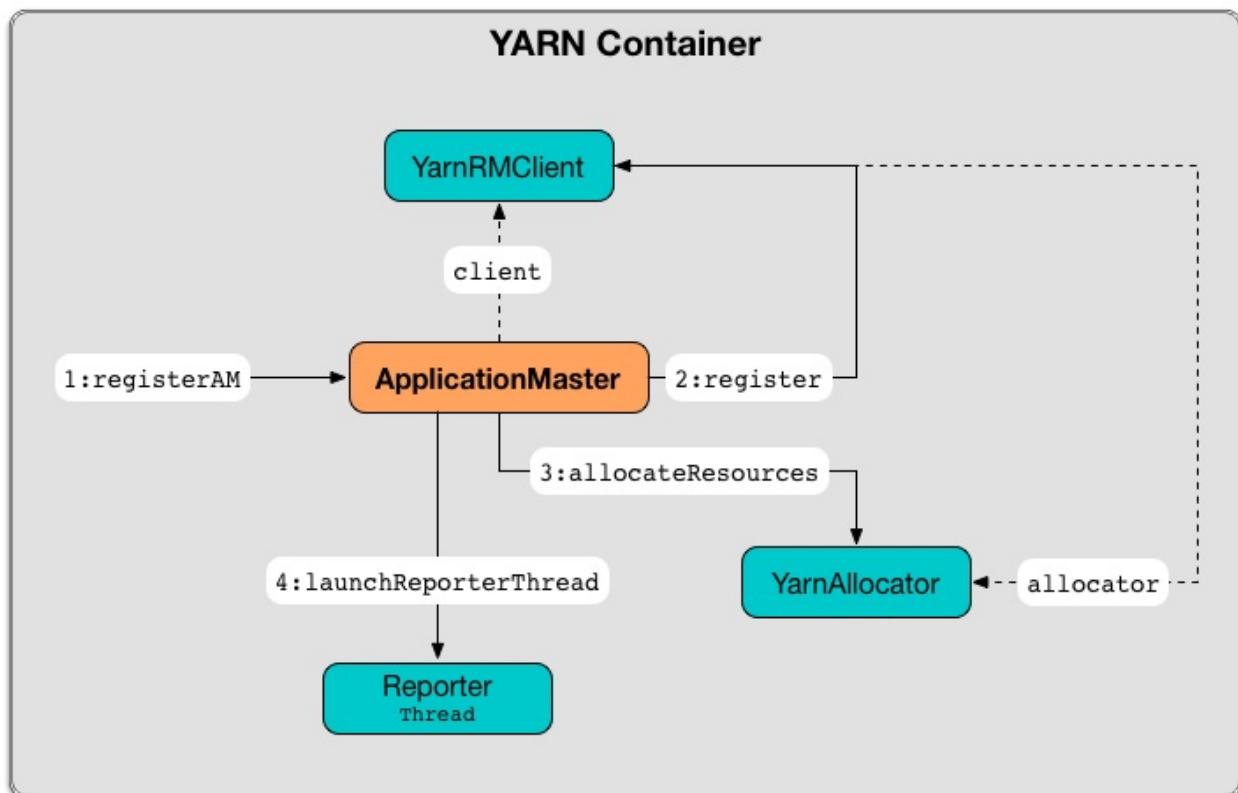


Figure 3. Registering ApplicationMaster with YARN ResourceManager

## Running Driver in Cluster Mode (runDriver method)

```
runDriver(securityMgr: SecurityManager): Unit
```

`runDriver` is a private procedure to...???

It starts by registering Web UI security filters.

Caution

[FIXME](#) Why is this needed? `addAMIpFilter`

It then starts the user class (with the driver) in a separate thread. You should see the following INFO message in the logs:

```
INFO Starting the user application in a separate Thread
```

Caution

[FIXME](#) Review `startUserApplication`.

You should see the following INFO message in the logs:

```
INFO Waiting for spark context initialization
```

Caution

[FIXME](#) Review `waitForSparkContextInitialized`

Caution

[FIXME](#) Finish...

## Running Executor Launcher (`runExecutorLauncher` method)

```
runExecutorLauncher(securityMgr: SecurityManager): Unit
```

`runExecutorLauncher` reads `spark.yarn.am.port` (or assume 0) and starts the `sparkYarnAM` RPC Environment (in client mode).

Caution

[FIXME](#) What's client mode?

It then waits for the driver to be available.

Caution

[FIXME](#) Review `waitForSparkDriver`

It registers Web UI security filters.

Caution

[FIXME](#) Why is this needed? `addAMIpFilter`

Ultimately, `runExecutorLauncher` registers the `ApplicationMaster` and requests resources and waits until the `reporterThread` dies.

Caution

[FIXME](#) Describe `registerAM`

## reporterThread

Caution

FIXME

## launchReporterThread

Caution

FIXME

## Setting Internal SparkContext Reference (sparkContextInitialized methods)

```
sparkContextInitialized(sc: SparkContext): Unit
```

`sparkContextInitialized` passes the call on to the `ApplicationMaster.sparkContextInitialized` that sets the internal `sparkContextRef` reference (to be `sc` ).

## Clearing Internal SparkContext Reference (sparkContextStopped methods)

```
sparkContextStopped(sc: SparkContext): Boolean
```

`sparkContextStopped` passes the call on to the `ApplicationMaster.sparkContextStopped` that clears the internal `sparkContextRef` reference (i.e. sets it to `null` ).

## Creating ApplicationMaster Instance

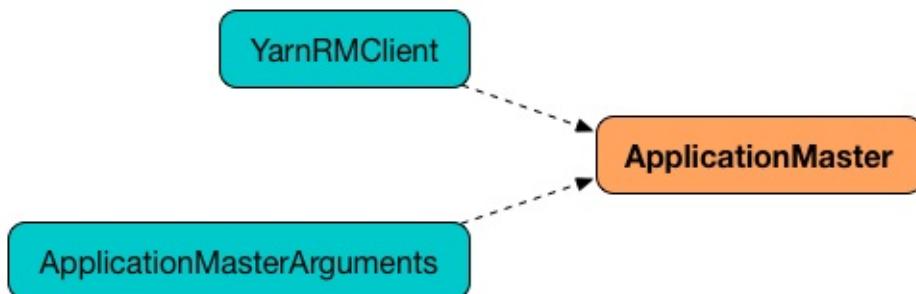


Figure 4. ApplicationMaster's Dependencies

When creating an instance of `ApplicationMaster` it requires `ApplicationMasterArguments` and `YarnRMClient`.

It instantiates `SparkConf` and Hadoop's `YarnConfiguration` (using `SparkHadoopUtil.newConfiguration`).

It assumes [cluster deploy mode](#) when `--class` was specified.

It computes the internal `maxNumExecutorFailures` using the optional `spark.yarn.max.executor.failures` if set. Otherwise, it is twice `spark.executor.instances` or `spark.dynamicAllocation.maxExecutors` (with dynamic allocation enabled) with the minimum of `3`.

It reads `yarn.am.liveness-monitor.expiry-interval-ms` (default: `120000`) from YARN to set the heartbeat interval. It is set to the minimum of the half of the YARN setting or `spark.yarn.scheduler.heartbeat.interval-ms` with the minimum of `0`.

`initialAllocationInterval` is set to the minimum of the heartbeat interval or `spark.yarn.scheduler.initial-allocation.interval`.

It then [loads the localized files](#) (as set by the client).

Caution

[\*\*FIXME\*\*](#) Who's the client?

## localResources attribute

When `ApplicationMaster` is instantiated, it computes internal `localResources` collection of YARN's `LocalResource` by name based on the internal `spark.yarn.cache.*` configuration settings.

```
localResources: Map[String, LocalResource]
```

You should see the following INFO message in the logs:

```
INFO ApplicationMaster: Preparing Local resources
```

It starts by reading the internal Spark configuration settings (that were earlier set when `client` prepared local resources to distribute):

- `spark.yarn.cache.filenames`
- `spark.yarn.cache.sizes`
- `spark.yarn.cache.timestamps`
- `spark.yarn.cache.visibilities`
- `spark.yarn.cache.types`

For each file name in `spark.yarn.cache.filenames` it maps `spark.yarn.cache.types` to an appropriate YARN's `LocalResourceType` and creates a new YARN `LocalResource`.

Note	<code>LocalResource</code> represents a local resource required to run a container.
------	---

If `spark.yarn.cache.confArchive` is set, it is added to `localResources` as `ARCHIVE` resource type and `PRIVATE` visibility.

Note	<code>spark.yarn.cache.confArchive</code> is set when <code>client</code> prepares local resources.
------	---

Note	<code>ARCHIVE</code> is an archive file that is automatically unarchived by the NodeManager.
------	--

Note	<code>PRIVATE</code> visibility means to share a resource among all applications of the same user on the node.
------	--

Ultimately, it removes the cache-related settings from the [Spark configuration](#) and system properties.

You should see the following INFO message in the logs:

```
INFO ApplicationMaster: Prepared Local resources [resources]
```

## Running ApplicationMaster (run method)

When `ApplicationMaster` is started as a standalone command-line application (in a YARN container on a node in a YARN cluster), ultimately `run` is executed.

```
run(): Int
```

The result of calling `run` is the final result of the `ApplicationMaster` command-line application.

`run` sets [cluster mode settings](#), registers the [cleanup shutdown hook](#), schedules `AMDelegationTokenRenewer` and finally registers `ApplicationMaster` for the Spark application (either calling `runDriver` for cluster mode or `runExecutorLauncher` for client mode).

After the [cluster mode settings](#) are set, `run` prints the following INFO message out to the logs:

```
INFO ApplicationAttemptId: [appAttemptId]
```

The `appAttemptId` is the [current application attempt id](#) (using the constructor's `YarnRMClient` as `client`).

The cleanup shutdown hook is registered with shutdown priority lower than that of `SparkContext` (so it is executed after `SparkContext` ).

`SecurityManager` is instantiated with the internal `Spark configuration`. If the `credentials file config` (as `spark.yarn.credentials.file`) is present, a `AMDelegationTokenRenewer` is started.

Caution

`FIXME` Describe `AMDelegationTokenRenewer#scheduleLoginFromKeytab`

It finally runs `ApplicationMaster` for the Spark application (either calling `runDriver` when in cluster mode or `runExecutorLauncher` otherwise).

It exits with `0` exit code.

In case of an exception, `run` prints the following ERROR message out to the logs:

```
ERROR Uncaught exception: [exception]
```

And the application run attempt is `finished` with `FAILED` status and `EXIT_UNCAUGHT_EXCEPTION` (10) exit code.

## Cluster Mode Settings

When in `cluster mode`, `ApplicationMaster` sets the following system properties (in `run`):

- `spark.ui.port` as `0`
- `spark.master` as `yarn`
- `spark.submit.deployMode` as `cluster`
- `spark.yarn.app.id` as application id

Caution

`FIXME` Why are the system properties required? Who's expecting them?

## isClusterMode Internal Flag

Caution

`FIXME` Since `org.apache.spark.deploy.yarn.ExecutorLauncher` is used for client deploy mode, the `isClusterMode` flag could be set there (not depending on `--class` which is correct yet not very obvious).

`isClusterMode` is an internal flag that is enabled (i.e. `true`) for `cluster mode`.

Specifically, it says whether the main class of the Spark application (through `--class` command-line argument) was specified or not. That is how the developers decided to inform `ApplicationMaster` about being run in `cluster mode` when `Client` creates YARN's `ContainerLaunchContext` (for launching `ApplicationMaster` ).

It is used to set additional system properties in `run` and `runDriver` (the flag is enabled) or `runExecutorLauncher` (when disabled).

Besides, it controls the default final status of a Spark application being `FinalApplicationStatus.FAILED` (when the flag is enabled) or `FinalApplicationStatus.UNDEFINED`.

The flag also controls whether to set system properties in `addAmIpFilter` (when the flag is enabled) or send a `AddWebUIFilter` instead.

## Unregistering ApplicationMaster from YARN ResourceManager (unregister method)

`unregister` unregisters the `ApplicationMaster` for the Spark application from the [YARN ResourceManager](#).

```
unregister(status: FinalApplicationStatus, diagnostics: String = null): Unit
```

Note	It is called from the <a href="#">cleanup shutdown hook</a> (that was registered in <code>ApplicationMaster</code> when it <a href="#">started running</a> ) and only when the application's final result is successful or it was the last attempt to run the application.
------	--

It first checks that the `ApplicationMaster` has not already been unregistered (using the internal `unregistered` flag). If so, you should see the following INFO message in the logs:

```
INFO ApplicationMaster: Unregistering ApplicationMaster with [status]
```

There can also be an optional diagnostic message in the logs:

```
(diag message: [msg])
```

The internal `unregistered` flag is set to be enabled, i.e. `true`.

It then requests `YarnRMClient` to [unregister](#).

## Cleanup Shutdown Hook

When `ApplicationMaster` starts running, it registers a shutdown hook that [unregisters the Spark application from the YARN ResourceManager](#) and [cleans up the staging directory](#).

Internally, it checks the internal `finished` flag, and if it is disabled, it [marks the Spark application as failed with `EXIT\_EARLY`](#).

If the internal `unregistered` flag is disabled, it [unregisters the Spark application](#) and [cleans up the staging directory](#) afterwards only when the final status of the ApplicationMaster's registration is `FinalApplicationStatus.SUCCEEDED` or the [number of application attempts is more than allowed](#).

The shutdown hook runs after the `SparkContext` is shut down, i.e. the shutdown priority is one less than `SparkContext`'s.

The shutdown hook is registered using `Spark`'s own `shutdownHookManager.addShutdownHook`.

## finish

Caution	<a href="#">FIXME</a>
---------	-----------------------

## ExecutorLauncher

`ExecutorLauncher` comes with no extra functionality when compared to `ApplicationMaster`. It serves as a helper class to run `ApplicationMaster` under another class name in [client deploy mode](#).

With the two different class names (pointing at the same class `ApplicationMaster`) you should be more successful to distinguish between `ExecutorLauncher` (which is really a `ApplicationMaster`) in [client deploy mode](#) and the `ApplicationMaster` in [cluster deploy mode](#) using tools like `ps` or `jps`.

Note	Consider <code>ExecutorLauncher</code> a <code>ApplicationMaster</code> for client deploy mode.
------	---

## Obtain Application Attempt Id (`getAttemptId` method)

<code>getAttemptId(): ApplicationAttemptId</code>
---

`getAttemptId` returns YARN's `ApplicationAttemptId` (of the Spark application to which the container was assigned).

Internally, it queries YARN by means of [YarnRMClient](#).

## addAmIpFilter helper method

<code>addAmIpFilter(): Unit</code>
------------------------------------

`addAmIpFilter` is a helper method that ...???

It starts by reading Hadoop's environmental variable `ApplicationConstants.APPLICATION_WEB_PROXY_BASE_ENV` that it passes to `YarnRMClient` to compute the configuration for the `AmIpFilter` for web UI.

In cluster deploy mode (when `ApplicationMaster` runs with web UI), it sets `spark.ui.filters` system property as `org.apache.hadoop.yarn.server.webproxy.amfilter.AmIpFilter`. It also sets system properties from the key-value configuration of `AmIpFilter` (computed earlier) as `spark.org.apache.hadoop.yarn.server.webproxy.amfilter.AmIpFilter.param.[key]` being `[value]`.

In client deploy mode (when `ApplicationMaster` runs on another JVM or even host than web UI), it simply sends a `AddWebUIFilter` to `ApplicationMaster` (namely to [AMEndpoint RPC Endpoint](#)).

# AMEndpoint — ApplicationMaster RPC Endpoint

## onStart Callback

When `onstart` is called, `AMEndpoint` communicates with the driver (the `driver` remote RPC Endpoint reference) by sending a one-way `RegisterClusterManager` message with a reference to itself.

After `RegisterClusterManager` has been sent (and received by `YarnSchedulerEndpoint`) the communication between the RPC endpoints of `ApplicationMaster` (YARN) and `YarnSchedulerBackend` (the Spark driver) is considered established.

## RPC Messages

### AddWebUIFilter

```
AddWebUIFilter(  
    filterName: String,  
    filterParams: Map[String, String],  
    proxyBase: String)
```

When `AddWebUIFilter` arrives, you should see the following INFO message in the logs:

```
INFO ApplicationMaster$AMEndpoint: Add WebUI Filter. [addWebUIFilter]
```

It then passes the `AddWebUIFilter` message on to the driver's scheduler backend (through [YarnScheduler RPC Endpoint](#)).

### RequestExecutors

```
RequestExecutors(  
    requestedTotal: Int,  
    localityAwareTasks: Int,  
    hostToLocalTaskCount: Map[String, Int])
```

When `RequestExecutors` arrives, `AMEndpoint` requests [YarnAllocator](#) for executors given [locality preferences](#).

If the `requestedTotal` number of executors is different than the current number, [resetAllocatorInterval](#) is executed.

In case when `YarnAllocator` is not available yet, you should see the following WARN message in the logs:

```
WARN Container allocator is not ready to request executors yet.
```

The response is `false` then.

## resetAllocatorInterval

When [RequestExecutors](#) message arrives, it calls `resetAllocatorInterval` procedure.

```
resetAllocatorInterval(): Unit
```

`resetAllocatorInterval` requests `allocatorLock` monitor lock and sets the internal `nextAllocationInterval` attribute to be `initialAllocationInterval` internal attribute. It then wakes up all threads waiting on `allocatorLock`.

Note	A thread waits on a monitor by calling one of the <code>object.wait</code> methods.
------	---

# YarnClusterManager — ExternalClusterManager for YARN

`YarnClusterManager` is the only currently known [ExternalClusterManager](#) in Spark. It creates a `TaskScheduler` and a `SchedulerBackend` for YARN.

## canCreate method

`YarnClusterManager` can handle the `yarn` master URL only.

## createTaskScheduler method

`createTaskScheduler` creates a [YarnClusterScheduler](#) for `cluster` deploy mode and a [YarnScheduler](#) for `client` deploy mode.

It throws a `SparkException` for unknown deploy modes.

```
Unknown deploy mode '[deployMode]' for Yarn
```

## createSchedulerBackend method

`createSchedulerBackend` creates a [YarnClusterSchedulerBackend](#) for `cluster` deploy mode and a [YarnClientSchedulerBackend](#) for `client` deploy mode.

It throws a `SparkException` for unknown deploy modes.

```
Unknown deploy mode '[deployMode]' for Yarn
```

## initialize method

`initialize` simply initializes the input `TaskSchedulerImpl`.

# TaskSchedulers for YARN

There are currently two [TaskSchedulers](#) for Spark on YARN per [deploy mode](#):

- [YarnScheduler](#) for **client** deploy mode
- [YarnClusterScheduler](#) for **cluster** deploy mode

# YarnScheduler - TaskScheduler for Client Deploy Mode

`YarnScheduler` is the [TaskScheduler for Spark on YARN](#) in [client deploy mode](#).

It is a custom [TaskSchedulerImpl](#) with ability to compute racks per hosts, i.e. it comes with a specialized [getRackForHost](#).

It also sets `org.apache.hadoop.yarn.util.RackResolver` logger to `WARN` if not set already.

## Tracking Racks per Hosts and Ports (getRackForHost method)

`getRackForHost` attempts to compute the rack for a host.

Note	<code>getRackForHost</code> overrides the <a href="#">parent TaskSchedulerImpl's getRackForHost</a>
------	---

It simply uses Hadoop's `org.apache.hadoop.yarn.util.RackResolver` to resolve a hostname to its network location, i.e. a rack.

# YarnClusterScheduler - TaskScheduler for Cluster Deploy Mode

`YarnClusterScheduler` is the [TaskScheduler](#) for [Spark on YARN](#) in [cluster deploy mode](#).

It is a custom [YarnScheduler](#) that makes sure that appropriate initialization of [ApplicationMaster](#) is performed, i.e. [SparkContext](#) is initialized and [stopped](#).

While being created, you should see the following INFO message in the logs:

```
INFO YarnClusterScheduler: Created YarnClusterScheduler
```

**Tip** Enable `INFO` logging level for `org.apache.spark.scheduler.cluster.YarnClusterScheduler` to see what happens inside `YarnClusterScheduler`.

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

```
log4j.logger.org.apache.spark.scheduler.cluster.YarnClusterScheduler=INFO
```

Refer to [Logging](#).

## postStartHook

`postStartHook` calls [ApplicationMaster.sparkContextInitialized](#) before the parent's `postStartHook`.

You should see the following INFO message in the logs:

```
INFO YarnClusterScheduler: YarnClusterScheduler.postStartHook done
```

## Stopping YarnClusterScheduler (stop method)

`stop` calls the parent's `stop` followed by [ApplicationMaster.sparkContextStopped](#).

## SchedulerBackends for YARN

There are currently two [SchedulerBackends](#) for [Spark on YARN](#) per [deploy mode](#):

- [YarnClientSchedulerBackend](#) for **client** deploy mode
- [YarnSchedulerBackend](#) for **cluster** deploy mode

They are concrete [YarnSchedulerBackends](#).

# YarnSchedulerBackend — Coarse-Grained Scheduler Backend for YARN

`YarnSchedulerBackend` is an abstract `CoarseGrainedSchedulerBackend` for YARN that contains common logic for the `client` and `cluster` YARN scheduler backends, i.e. `YarnClientSchedulerBackend` and `YarnClusterSchedulerBackend` respectively.

`YarnSchedulerBackend` is available in the RPC Environment as `YarnScheduler` `RPC Endpoint` (or `yarnSchedulerEndpointRef` internally).

`YarnSchedulerBackend` expects `TaskSchedulerImpl` and `SparkContext` to initialize itself.

It works for a single Spark application (as `appId` of type `ApplicationId`)

Caution	<code>FIXME</code> It may be a note for scheduler backends in general.
---------	--

## attemptId Internal Attribute

<pre>attemptId: Option[ApplicationAttemptId] = None</pre>
---

`attemptId` is the application attempt ID for this run of a Spark application. It is only available for `cluster deploy mode`.

It is explicitly set to `None` when `YarnClientSchedulerBackend starts` (and `bindToYarn` is called).

It is set to the current attempt id (using YARN API's `ApplicationMaster.getAttemptId`) when `YarnClusterSchedulerBackend starts` (and `bindToYarn` is called).

Note	<code>attemptId</code> is exposed using <code>applicationAttemptId</code> which is a part of <code>SchedulerBackend Contract</code> .
------	---

## applicationAttemptId

Note	<code>applicationAttemptId</code> is a part of <code>SchedulerBackend Contract</code> .
------	---

<pre>applicationAttemptId(): Option[String]</pre>
---

`applicationAttemptId` returns the `application attempt id` of a Spark application.

## Resetting YarnSchedulerBackend

**Note** `reset` is a part of [CoarseGrainedSchedulerBackend Contract](#).

`reset` [resets](#) the parent `CoarseGrainedSchedulerBackend` scheduler backend and `ExecutorAllocationManager` (accessible by `SparkContext.executorAllocationManager` ).

## doRequestTotalExecutors

```
def doRequestTotalExecutors(requestedTotal: Int): Boolean
```

**Note** `doRequestTotalExecutors` is a part of the [CoarseGrainedSchedulerBackend Contract](#).

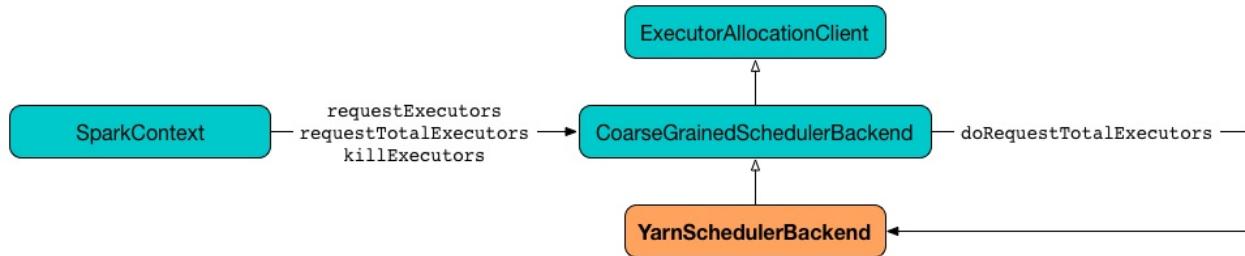


Figure 1. Requesting Total Executors in YarnSchedulerBackend (`doRequestTotalExecutors` method)

`doRequestTotalExecutors` simply sends a blocking [RequestExecutors](#) message to [YarnScheduler RPC Endpoint](#) with the input `requestedTotal` and the internal `localityAwareTasks` and `hostToLocalTaskCount` attributes.

**Caution** [\*\*FIXME\*\*](#) The internal attributes are already set. When and how?

## Reference to YarnScheduler RPC Endpoint (`yarnSchedulerEndpointRef` attribute)

`yarnSchedulerEndpointRef` is the reference to [YarnScheduler RPC Endpoint](#).

## totalExpectedExecutors

`totalExpectedExecutors` is a value that is `0` [initially when a `YarnSchedulerBackend` instance is created](#) but later changes when Spark on YARN starts (in [client mode](#) or [cluster mode](#)).

**Note** After Spark on YARN is started, `totalExpectedExecutors` is initialized to a proper value.

It is used in [sufficientResourcesRegistered](#).

Caution

**FIXME** Where is this used?

## Creating YarnSchedulerBackend Instance

When created, `YarnSchedulerBackend` sets the internal `minRegisteredRatio` which is `0.8` when `spark.scheduler.minRegisteredResourcesRatio` is *not* set or the parent's `minRegisteredRatio`.

`totalExpectedExecutors` is set to `0`.

It creates a `YarnSchedulerEndpoint` (as `yarnSchedulerEndpoint`) and registers it as **YarnScheduler** with the **RPC Environment**.

It sets the internal `askTimeout` **Spark timeout for RPC ask operations using the `SparkContext` constructor parameter.**

It sets optional `appId` (of type `ApplicationId`), `attemptId` (for cluster mode only and of type `ApplicationAttemptId`).

It also creates `SchedulerExtensionServices` object (as `services`).

Caution

**FIXME** What is `SchedulerExtensionServices`?

The internal `shouldResetOnAmRegister` flag is turned off.

## sufficientResourcesRegistered

`sufficientResourcesRegistered` checks whether `totalRegisteredExecutors` is greater than or equals to `totalExpectedExecutors` multiplied by `minRegisteredRatio`.

Note

It overrides the parent's `CoarseGrainedSchedulerBackend.sufficientResourcesRegistered`.

Caution

**FIXME** Where's this used?

## minRegisteredRatio

`minRegisteredRatio` is set when `YarnSchedulerBackend` is created.

It is used in `sufficientResourcesRegistered`.

## Starting the Backend (start method)

`start` creates a `SchedulerExtensionServiceBinding` object (using `SparkContext`, `appId`, and `attemptId`) and starts it (using `SchedulerExtensionServices.start(binding)`).

**Note**

A `SchedulerExtensionServices` object is created when `YarnSchedulerBackend` is initialized and available as `services`.

Ultimately, it calls the parent's `CoarseGrainedSchedulerBackend.start`.

**Note**

`start` throws `IllegalArgumentException` when the internal `appId` has not been set yet.

```
java.lang.IllegalArgumentException: requirement failed: application ID unset
```

## Stopping the Backend (stop method)

`stop` calls the parent's `CoarseGrainedSchedulerBackend.requestTotalExecutors` (using `(0, 0, Map.empty)` parameters).

**Caution**

`FIXME` Explain what `0, 0, Map.empty` means after the method's described for the parent.

It calls the parent's `CoarseGrainedSchedulerBackend.stop`.

Ultimately, it stops the internal `SchedulerExtensionServiceBinding` object (using `services.stop()`).

**Caution**

`FIXME` Link the description of `services.stop()` here.

## Recording Application and Attempt Ids (bindToYarn method)

```
bindToYarn(appId: ApplicationId, attemptId: Option[ApplicationAttemptId]): Unit
```

`bindToYarn` sets the internal `appId` and `attemptId` to the value of the input parameters, `appId` and `attemptId`, respectively.

**Note**

`start` requires `appId`.

## Internal Registries

### shouldResetOnAmRegister flag

When `YarnSchedulerBackend` is created, `shouldResetOnAmRegister` is disabled (i.e. `false`).

`shouldResetOnAmRegister` controls whether to reset `YarnSchedulerBackend` when another `RegisterClusterManager` RPC message arrives.

It allows resetting internal state after the initial ApplicationManager failed and a new one was registered.

Note	It can only happen in <a href="#">client deploy mode</a> .
------	--

## Settings

### **spark.scheduler.minRegisteredResourcesRatio**

`spark.scheduler.minRegisteredResourcesRatio` (default: `0.8`)

# YarnClientSchedulerBackend — SchedulerBackend for YARN in Client Deploy Mode

`YarnClientSchedulerBackend` is the [SchedulerBackend](#) for [Spark on YARN](#) for client deploy mode.

Note	<code>client</code> deploy mode is the default deploy mode of Spark on YARN.
------	--

`YarnClientSchedulerBackend` is a [YarnSchedulerBackend](#) that comes with just two custom implementations of the methods from the [SchedulerBackend Contract](#):

- [start](#)
- [stop](#)

`YarnClientSchedulerBackend` uses `client` internal attribute to submit a Spark application when it [starts up](#) and [waits for the Spark application](#) until it has exited, either successfully or due to some failure.

In order to initialize a `YarnClientSchedulerBackend` Spark passes a [TaskSchedulerImpl](#) and [SparkContext](#) (but only `SparkContext` is used in this object with `TaskSchedulerImpl` being passed on to the supertype — [YarnSchedulerBackend](#)).

`YarnClientSchedulerBackend` belongs to `org.apache.spark.scheduler.cluster` package.

<span style="font-size: 1.5em;">Tip</span>	<p>Enable <code>DEBUG</code> logging level for <code>org.apache.spark.scheduler.cluster.YarnClientSchedulerBackend</code> logger to see what happens inside <code>YarnClientSchedulerBackend</code>.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.scheduler.cluster.YarnClientSchedulerBackend=DEBUG</pre> <p>Refer to <a href="#">Logging</a>.</p>
--	---

## client Internal Attribute

`client` private attribute is an instance of [Client](#) that `YarnClientSchedulerBackend` creates an instance of when it [starts](#) and uses to [submit the Spark application](#).

`client` is also used to [monitor the Spark application](#) when `YarnClientSchedulerBackend` [waits for the application](#).

`client` is stopped when `YarnClientSchedulerBackend` stops.

## Starting YarnClientSchedulerBackend (start method)

`start` is part of the [SchedulerBackend Contract](#). It is executed when `TaskSchedulerImpl` starts.

```
start(): Unit
```

It creates the internal `client` object and submits the Spark application to [YARN ResourceManager](#). After the application is deployed to YARN and running, it starts the internal `monitorThread` state monitor thread. In the meantime it also calls the supertype's `start`.

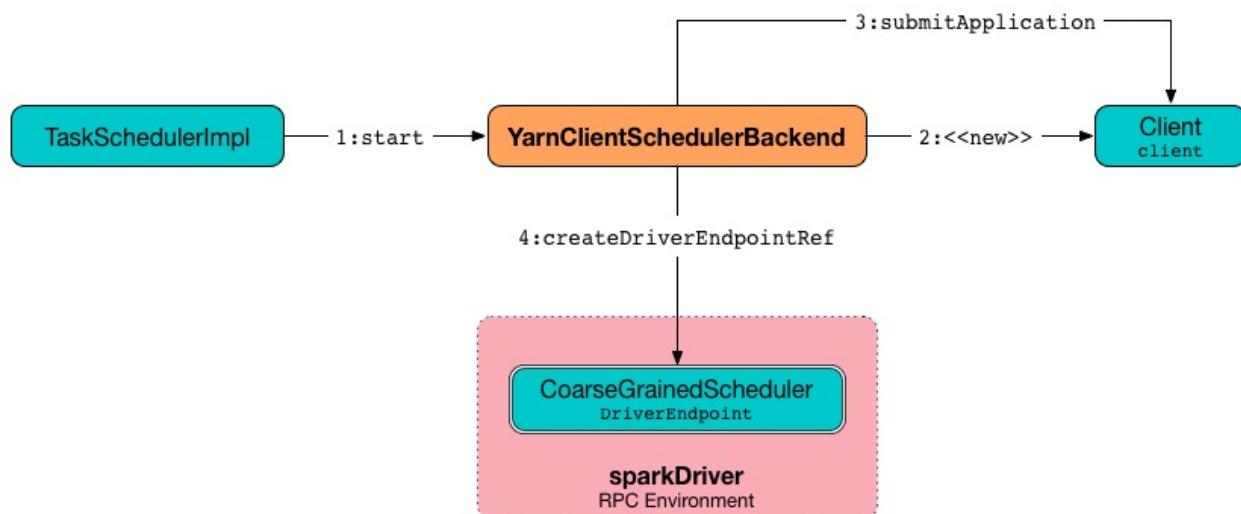


Figure 1. Starting YarnClientSchedulerBackend

`start` sets `spark.driver.appUIAddress` to be `SparkUI.appUIAddress` (only if [Spark's web UI is enabled](#)).

With DEBUG log level enabled you should see the following DEBUG message in the logs:

```
DEBUG YarnClientSchedulerBackend: ClientArguments called with: --arg [hostport]
```

Note

`hostport` is `spark.driver.host` and `spark.driver.port` separated by `:`, e.g. `192.168.99.1:64905`.

It then creates an instance of `ClientArguments` (using `--arg [hostport]` arguments).

It sets the parent's `totalExpectedExecutors` to the initial number of executors.

Caution

**FIXME** Why is this part of subtypes since they both set it to the same value?

It creates a [Client](#) object using the instance of `ClientArguments` and `SparkConf`.

The parent's [YarnSchedulerBackend.bindToYarn](#) method is called with the current application id (being the result of calling [Client.submitApplication](#)) and `None` for the optional `attemptId`.

The parent's [YarnSchedulerBackend.start](#) is called.

[waitForApplication](#) is executed that blocks until the application is running or an [SparkException](#) is thrown.

If `spark.yarn.credentials.file` is defined,

[YarnSparkHadoopUtil.get.startExecutorDelegationTokenRenewer\(conf\)](#) is called.

Caution	<a href="#">FIXME Why? What does <code>startExecutorDelegationTokenRenewer</code> do?</a>
---------	---

A [MonitorThread](#) object is created (using `asyncMonitorApplication`) and started to asynchronously monitor the currently running application.

## stop

`stop` is part of the [SchedulerBackend Contract](#).

It stops the internal helper objects, i.e. `monitorThread` and `client` as well as "announces" the stop to other services through `client.reportLauncherState`. In the meantime it also calls the supertype's `stop`.

`stop` makes sure that the internal `client` has already been created (i.e. it is not `null`), but not necessarily started.

`stop` stops the internal `monitorThread` using `MonitorThread.stopMonitor` method.

It then "announces" the stop using [Client.reportLauncherState\(SparkAppHandle.State.FINISHED\)](#).

Later, it passes the call on to the supertype's `stop` and, once the supertype's `stop` has finished, it calls [YarnSparkHadoopUtil.stopExecutorDelegationTokenRenewer](#) followed by [stopping the internal client](#).

Eventually, when all went fine, you should see the following INFO message in the logs:

```
INFO YarnClientSchedulerBackend: Stopped
```

## Waiting For Spark Application ([waitForApplication](#) method)

```
waitForApplication(): Unit
```

`waitForApplication` is an internal (private) method that waits until the current application is running (using [Client.monitorApplication](#)).

If the application has `FINISHED`, `FAILED`, or has been `KILLED`, a `SparkException` is thrown with the following message:

```
Yarn application has already ended! It might have been killed or unable to launch application master.
```

You should see the following INFO message in the logs for `RUNNING` state:

```
INFO YarnClientSchedulerBackend: Application [appId] has started running.
```

## asyncMonitorApplication

```
asyncMonitorApplication(): MonitorThread
```

`asyncMonitorApplication` internal method creates a separate daemon [MonitorThread](#) thread called "Yarn application state monitor".

Note

`asyncMonitorApplication` does not start the daemon thread.

## MonitorThread

`MonitorThread` internal class is to monitor a Spark application deployed to YARN in client mode.

When started, it calls the blocking [Client.monitorApplication](#) (with no application reports printed out to the console, i.e. `logApplicationReport` is disabled).

Note

`Client.monitorApplication` is a blocking operation and hence it is wrapped in `MonitorThread` to be executed in a separate thread.

When the call to `client.monitorApplication` has finished, it is assumed that the application has exited. You should see the following ERROR message in the logs:

```
ERROR Yarn application has already exited with state [state]!
```

That leads to stopping the current `SparkContext` (using [SparkContext.stop](#)).



# YarnClusterSchedulerBackend - SchedulerBackend for YARN in Cluster Deploy Mode

`YarnClusterSchedulerBackend` is a custom [YarnSchedulerBackend](#) for Spark on YARN in [cluster deploy mode](#).

This is a 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?).

## Note

`YarnClusterSchedulerBackend` is a `private[spark]` Scala class. You can find the sources in [org.apache.spark.scheduler.cluster.YarnClusterSchedulerBackend](#).

## Tip

Enable `DEBUG` logging level for `org.apache.spark.scheduler.cluster.YarnClusterSchedulerBackend` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.scheduler.cluster.YarnClusterSchedulerBackend=DEBUG
```

Refer to [Logging](#).

## Creating YarnClusterSchedulerBackend

Creating a `YarnClusterSchedulerBackend` object requires a [TaskSchedulerImpl](#) and [SparkContext](#) objects.

## Starting YarnClusterSchedulerBackend (start method)

`YarnClusterSchedulerBackend` comes with a custom `start` method.

## Note

`start` is part of the [SchedulerBackend Contract](#).

Internally, it first [queries ApplicationMaster](#) for `attemptId` and records the application and attempt ids.

It then calls the parent's `start` and sets the parent's `totalExpectedExecutors` to the initial number of executors.

## Calculating Driver Log URLs (getDriverLogUrls method)

`getDriverLogUrls` in `YarnClusterSchedulerBackend` calculates the URLs for the driver's logs  
- standard output (stdout) and standard error (stderr).

Note	<code>getDriverLogUrls</code> is part of the <a href="#">SchedulerBackend Contract</a> .
------	--

Internally, it retrieves the `container id` and through environment variables computes the base URL.

You should see the following DEBUG in the logs:

```
DEBUG Base URL for logs: [baseUrl]
```

# YarnSchedulerEndpoint RPC Endpoint

`YarnSchedulerEndpoint` is a [thread-safe RPC endpoint](#) for communication between `YarnSchedulerBackend` on the driver and `ApplicationMaster` on YARN (inside a YARN container).

Caution

[FIXME](#) Picture it.

It uses the [reference to the remote ApplicationMaster RPC Endpoint](#) to send messages to.

**Tip** Enable `INFO` logging level for `org.apache.spark.scheduler.cluster.YarnSchedulerBackend$YarnSchedulerEndpoint` log happens inside.

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

```
log4j.logger.org.apache.spark.scheduler.cluster.YarnSchedulerBackend$YarnSchedulerEndpoint=INFO
```

Refer to [Logging](#).

## RPC Messages

### RequestExecutors

```
RequestExecutors(  
    requestedTotal: Int,  
    localityAwareTasks: Int,  
    hostToLocalTaskCount: Map[String, Int])  
  extends CoarseGrainedClusterMessage
```

`RequestExecutors` is to inform `ApplicationMaster` about the current requirements for the total number of executors (as `requestedTotal`), including already pending and running executors.

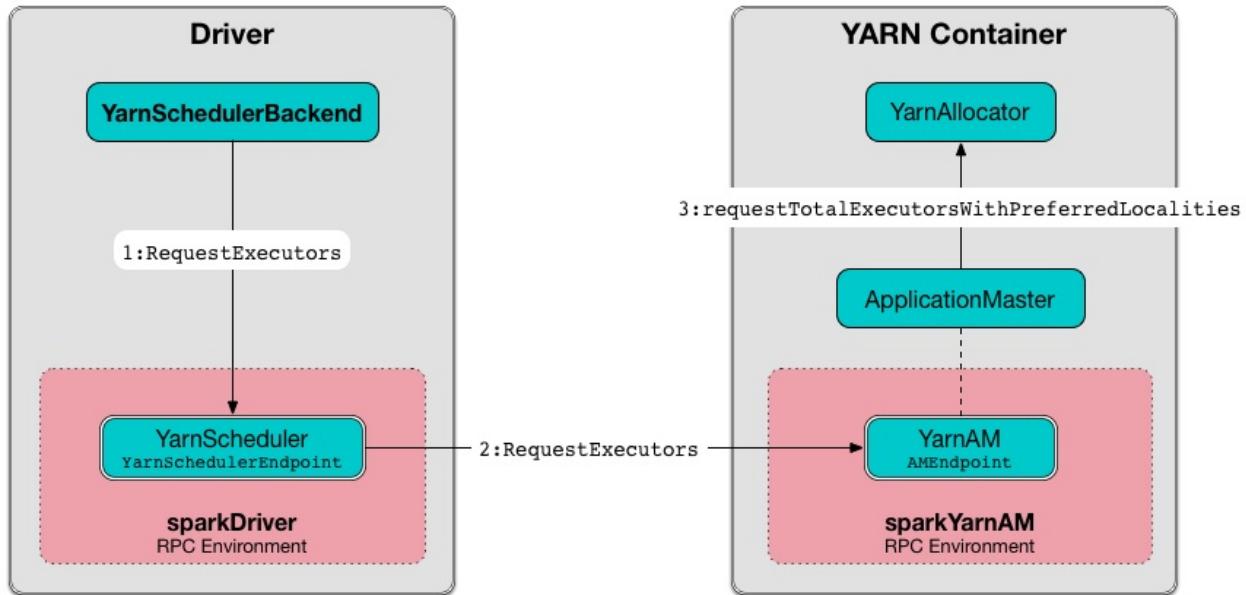


Figure 1. RequestExecutors Message Flow (client deploy mode)

When a `RequestExecutors` arrives, `YarnSchedulerEndpoint` simply passes it on to `ApplicationMaster` (via the [internal RPC endpoint reference](#)). The result of the forward call is sent back in response.

Any issues communicating with the remote `ApplicationMaster` RPC endpoint are reported as ERROR messages in the logs:

```
ERROR Sending RequestExecutors to AM was unsuccessful
```

## RemoveExecutor

## KillExecutors

## AddWebUIFilter

```
AddWebUIFilter(
  filterName: String,
  filterParams: Map[String, String],
  proxyBase: String)
```

`AddWebUIFilter` triggers setting `spark.ui.proxyBase` system property and adding the `filterName` filter to web UI.

`AddWebUIFilter` is sent by `ApplicationMaster` when it adds `AmIpFilter` to web UI.

It firstly sets `spark.ui.proxyBase` system property to the input `proxyBase` (if not empty).

If it defines a filter, i.e. the input `filterName` and `filterParams` are both not empty, you should see the following INFO message in the logs:

```
INFO Add WebUI Filter. [filterName], [filterParams], [proxyBase]
```

It then sets `spark.ui.filters` to be the input `filterName` in the internal `conf` `SparkConf` attribute.

All the `filterParams` are also set as `spark.[filterName].param.[key]` and `[value]`.

The filter is added to web UI using `JettyUtils.addFilters(ui.getHandlers, conf)`.

**Caution**

[FIXME Review](#) `JettyUtils.addFilters(ui.getHandlers, conf)`.

## RegisterClusterManager Message

```
RegisterClusterManager(am: RpcEndpointRef)
```

When `RegisterClusterManager` message arrives, the following INFO message is printed out to the logs:

```
INFO YarnSchedulerBackend$YarnSchedulerEndpoint: ApplicationMaster registered as [am]
```

The internal reference to the remote ApplicationMaster RPC Endpoint is set (to `am`).

If the internal `shouldResetOnAmRegister` flag is enabled, `YarnSchedulerBackend` is reset. It is disabled initially, so `shouldResetOnAmRegister` is enabled.

**Note**

`shouldResetOnAmRegister` controls whether to reset `YarnSchedulerBackend` when another `RegisterClusterManager` RPC message arrives that could be because the ApplicationManager failed and a new one was registered.

## RetrieveLastAllocatedExecutorId

When `RetrieveLastAllocatedExecutorId` is received, `YarnSchedulerEndpoint` responds with the current value of `currentExecutorIdCounter`.

**Note**

It is used by `YarnAllocator` to initialize the internal `executorIdCounter` (so it gives proper identifiers for new executors when `ApplicationMaster` restarts)

## onDisconnected Callback

`onDisconnected` clears the [internal reference to the remote ApplicationMaster RPC Endpoint](#) (i.e. it sets it to `None`) if the remote address matches the reference's.

Note	It is a callback method to be called when... <a href="#">FIXME</a>
------	--

You should see the following WARN message in the logs if that happens:

```
WARN ApplicationMaster has disassociated: [remoteAddress]
```

## onStop Callback

`onStop` shuts [askAmThreadPool](#) down immediately.

Note	<code>onstop</code> is a callback method to be called when... <a href="#">FIXME</a>
------	---

## Internal Reference to ApplicationMaster RPC Endpoint (amEndpoint variable)

`amEndpoint` is a reference to a remote [ApplicationMaster RPC Endpoint](#).

It is set to the current [ApplicationMaster RPC Endpoint](#) when [RegisterClusterManager](#) arrives and cleared when [the connection to the endpoint disconnects](#).

## askAmThreadPool Thread Pool

`askAmThreadPool` is a thread pool called **yarn-scheduler-ask-am-thread-pool** that creates new threads as needed and reuses previously constructed threads when they are available.

# YarnAllocator — Container Allocator

`YarnAllocator` allocates resource containers from `YARN ResourceManager` to run Spark executors on and releases them when the Spark application no longer needs them.

It talks directly to YARN ResourceManager through the `amClient` reference (of YARN's `AMRMClient[ContainerRequest]` type) that it gets when created (from `YarnRMClient` when it registers the ApplicationMaster for a Spark application).

Caution

`FIXME` Image for YarnAllocator uses amClient Reference to YARN ResourceManager

`YarnAllocator` is a part of the internal state of `ApplicationMaster` (via the internal `allocator` reference).

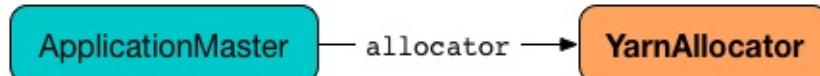


Figure 1. ApplicationMaster uses YarnAllocator (via allocator attribute)

When `YarnAllocator` is created, it requires `driverUrl`, Hadoop's Configuration, a Spark configuration, YARN's `ApplicationAttemptId`, a `SecurityManager`, and a collection of Hadoop's `LocalResources` by their name. The parameters are later used for launching Spark executors in allocated YARN containers.

Caution

`FIXME` An image with YarnAllocator and multiple ExecutorRunnables.

Enable `INFO` or `DEBUG` logging level for `org.apache.spark.deploy.yarn.YarnAllocator` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.deploy.yarn.YarnAllocator=DEBUG
```

Tip

Refer to [Logging](#).

## Creating YarnAllocator Instance

When `YarnRMClient` registers `ApplicationMaster` for a Spark application (with YARN ResourceManager) it creates a new `YarnAllocator` instance.

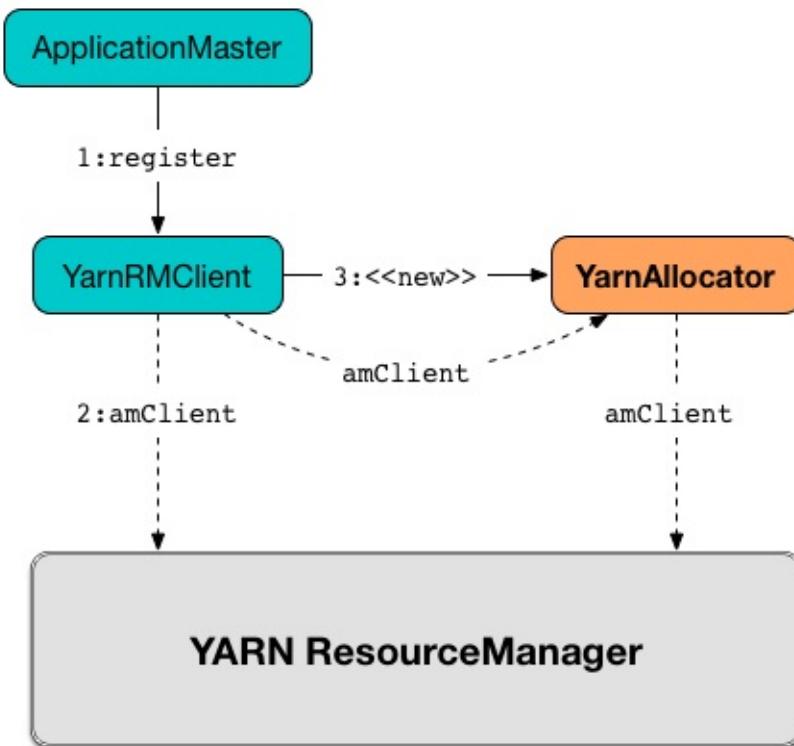


Figure 2. Creating YarnAllocator

All the input parameters for `YarnAllocator` (but `appAttemptId` and `amClient`) are passed directly from the input parameters of `YarnRMClient`.

```

YarnAllocator(
  driverUrl: String,
  driverRef: RpcEndpointRef,
  conf: Configuration,
  sparkConf: SparkConf,
  amClient: AMRMClient[ContainerRequest],
  appAttemptId: ApplicationAttemptId,
  securityMgr: SecurityManager,
  localResources: Map[String, LocalResource])
  
```

The input `amClient` parameter is created in and owned by `YarnRMClient`.

When `YarnAllocator` is created, it sets the `org.apache.hadoop.util.RackResolver` logger to `WARN` (unless set to some log level already).

It creates the following empty registries:

- `releasedContainers`
- `allocatedHostToContainersMap`
- `allocatedContainerToHostMap`
- `pendingLossReasonRequests`

- `releasedExecutorLossReasons`
- `executorIdToContainer`
- `containerIdToExecutorId`
- `hostToLocalTaskCounts`

It sets the following internal counters:

- `numExecutorsRunning` to 0
- `executorIdCounter` to the last allocated executor id (it seems quite an extensive operation that uses a RPC system)
- `numUnexpectedContainerRelease` to 0L
- `numLocalityAwareTasks` to 0
- `targetNumExecutors` to the initial number of executors

It creates an empty queue of failed executors.

It sets the internal `executorFailuresValidityInterval` to `spark.yarn.executor.failuresValidityInterval`.

It sets the internal `executorMemory` to `spark.executor.memory`.

It sets the internal `memoryOverhead` to `spark.yarn.executor.memoryOverhead`. If unavailable, it is set to the maximum of 10% of `executorMemory` and 384 .

It sets the internal `executorCores` to `spark.executor.cores`.

It creates the internal `resource` to Hadoop YARN's Resource with both `executorMemory + memoryOverhead` memory and `executorCores` CPU cores.

It creates the internal `launcherPool` called **ContainerLauncher** with maximum `spark.yarn.containerLauncherMaxThreads` threads.

It sets the internal `launchContainers` to `spark.yarn.launchContainers`.

It sets the internal `labelExpression` to `spark.yarn.executor.nodeLabelExpression`.

It sets the internal `nodeLabelConstructor` to...FIXME

Caution	FIXME nodeLabelConstructor?
---------	-----------------------------

It sets the internal `containerPlacementStrategy` to...FIXME

Caution	FIXME LocalityPreferredContainerPlacementStrategy?
---------	--

## Requesting Executors with Locality Preferences (`requestTotalExecutorsWithPreferredLocalities` method)

```
requestTotalExecutorsWithPreferredLocalities(  
    requestedTotal: Int,  
    localityAwareTasks: Int,  
    hostToLocalTaskCount: Map[String, Int]): Boolean
```

`requestTotalExecutorsWithPreferredLocalities` returns `true` if the current desired total number of executors is different than the input `requestedTotal`.

Note	<code>requestTotalExecutorsWithPreferredLocalities</code> should instead have been called <code>shouldRequestTotalExecutorsWithPreferredLocalities</code> since it answers the question whether to request total executors or not.
------	--

`requestTotalExecutorsWithPreferredLocalities` sets the internal `numLocalityAwareTasks` and `hostToLocalTaskCounts` attributes to the input `localityAwareTasks` and `hostToLocalTaskCount` arguments, respectively.

If the input `requestedTotal` is different than the internal `targetNumExecutors` attribute you should see the following INFO message in the logs:

```
INFO YarnAllocator: Driver requested a total number of [requestedTotal] executor(s).
```

It sets the internal `targetNumExecutors` attribute to the input `requestedTotal` and returns `true`. Otherwise, it returns `false`.

Note	<code>requestTotalExecutorsWithPreferredLocalities</code> is executed in response to <a href="#">RequestExecutors message to ApplicationMaster</a> .
------	--

## numLocalityAwareTasks Internal Counter

```
numLocalityAwareTasks: Int = 0
```

It tracks the number of locality-aware tasks to be used as container placement hint when [YarnAllocator](#) is requested for executors given locality preferences.

It is used as an input to `containerPlacementStrategy.localityOfRequestedContainers` when [YarnAllocator](#) updates YARN container allocation requests.

## Adding or Removing Executor Container Requests (`updateResourceRequests` method)

```
updateResourceRequests(): Unit
```

`updateResourceRequests` requests new or cancels outstanding executor containers from the [YARN ResourceManager](#).

**Note**

In YARN, you have to request containers for resources first (using [AMRMClient.addContainerRequest](#)) before calling [AMRMClient.allocate](#).

It gets the list of outstanding YARN's `ContainerRequests` (using the constructor's [AMRMClient\[ContainerRequest\]](#)) and aligns their number to current workload.

`updateResourceRequests` consists of two main branches:

1. [missing executors](#), i.e. when the number of executors allocated already or pending does not match the needs and so there are missing executors.
2. [executors to cancel](#), i.e. when the number of pending executor allocations is positive, but the number of all the executors is more than Spark needs.

## Case 1. Missing Executors

You should see the following INFO message in the logs:

```
INFO YarnAllocator: Will request [count] executor containers, each with [vCores] cores  
and [memory] MB memory including [memoryOverhead] MB overhead
```

It then splits pending container allocation requests per locality preference of pending tasks (in the internal [hostToLocalTaskCounts](#) registry).

**Caution**

[FIXME Review](#) `splitPendingAllocationsByLocality`

It removes stale container allocation requests (using YARN's [AMRMClient.removeContainerRequest](#)).

**Caution**

[FIXME Stale?](#)

You should see the following INFO message in the logs:

```
INFO YarnAllocator: Canceled [cancelledContainers] container requests (locality no lon  
ger needed)
```

It computes locality of requested containers (based on the internal [numLocalityAwareTasks](#), [hostToLocalTaskCounts](#) and [allocatedHostToContainersMap](#) lookup table).

**Caution**

**FIXME** Review `containerPlacementStrategy.localityOfRequestedContainers` + the code that follows.

For any new container needed `updateResourceRequests` adds a container request (using YARN's [AMRMClient.addContainerRequest](#)).

You should see the following INFO message in the logs:

```
INFO YarnAllocator: Submitted container request (host: [host], capability: [resource])
```

## Case 2. Cancelling Pending Executor Allocations

When there are executors to cancel (case 2.), you should see the following INFO message in the logs:

```
INFO Canceling requests for [numToCancel] executor container(s) to have a new desired total [targetNumExecutors] executors.
```

It checks whether there are pending allocation requests and removes the excess (using YARN's [AMRMClient.removeContainerRequest](#)). If there are no pending allocation requests, you should see the WARN message in the logs:

```
WARN Expected to find pending requests, but found none.
```

## killExecutor

**Caution****FIXME**

## Handling Allocated Containers for Executors (`handleAllocatedContainers` internal method)

When the [YARN ResourceManager](#) has allocated new containers for executors in `allocateResources`, the call is then passed on to `handleAllocatedContainers` procedure.

```
handleAllocatedContainers(allocatedContainers: Seq[Container]): Unit
```

`handleAllocatedContainers` handles allocated YARN containers.

Internally, `handleAllocatedContainers` matches requests to host, rack, and any host (a container allocation).

If there are any allocated containers left (without having been matched), you should see the following DEBUG message in the logs:

```
DEBUG Releasing [size] unneeded containers that were allocated to us
```

It then [releases the containers](#).

It [runs the allocated and matched containers](#).

At the end of the method, you should see the following INFO message in the logs:

```
INFO Received [allocatedContainersSize] containers from YARN, launching executors on [containersToUseSize] of them.
```

## Launching Spark Executors in Allocated YARN Containers (`runAllocatedContainers` internal method)

```
runAllocatedContainers(containersToUse: ArrayBuffer[Container]): Unit
```

For each YARN's [Container](#) in the input `containersToUse` collection, `runAllocatedContainers` attempts to run a [ExecutorRunnable](#) (on [ContainerLauncher](#) thread pool).

Internally, `runAllocatedContainers` increases the internal `executorIdCounter` counter and asserts that the amount of memory of (the resource allocated to) the container is greater than the requested memory for executors.

You should see the following INFO message in the logs:

```
INFO YarnAllocator: Launching container [containerId] for on host [executorHostname]
```

Unless `runAllocatedContainers` runs in [spark.yarn.launchContainers](#) testing mode (when it merely [updates internal state](#)), you should see the following INFO message in the logs:

```
INFO YarnAllocator: Launching ExecutorRunnable. driverUrl: [driverUrl], executorHostname: [executorHostname]
```

Note	<code>driverUrl</code> is of the form <code>spark://CoarseGrainedScheduler@[host]:[port]</code> .
------	---

It executes [ExecutorRunnable](#) on [ContainerLauncher](#) thread pool and [updates internal state](#).

Any non-fatal exception while running `ExecutorRunnable` is caught and you should see the following ERROR message in the logs:

```
ERROR Failed to launch executor [executorId] on container [containerId]
```

It then immediately releases the failed container (using the internal [AMRMClient](#)).

## updateInternalState

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Releasing YARN Container ([internalReleaseContainer](#) internal procedure)

All unnecessary YARN containers (that were allocated but are either [of no use](#) or [no longer needed](#)) are released using the internal `internalReleaseContainer` procedure.

```
internalReleaseContainer(container: Container): Unit
```

`internalReleaseContainer` records `container` in the internal [releasedContainers](#) registry and releases it to the [YARN ResourceManager](#) (calling [AMRMClient\[ContainerRequest\].releaseAssignedContainer](#) using the internal `amclient` ).

## Deciding on Use of YARN Container ([matchContainerToRequest](#) internal method)

When `handleAllocatedContainers` handles allocated containers for executors, it uses `matchContainerToRequest` to match the containers to `ContainerRequests` (and hence to workload and location preferences).

```
matchContainerToRequest(
    allocatedContainer: Container,
    location: String,
    containersToUse: ArrayBuffer[Container],
    remaining: ArrayBuffer[Container]): Unit
```

`matchContainerToRequest` puts `allocatedContainer` in `containersToUse` or `remaining` collections per available outstanding `ContainerRequests` that match the priority of the input `allocatedContainer`, the input `location`, and the memory and vcore capabilities for Spark executors.

Note	The input <code>location</code> can be host, rack, or <code>*</code> (star), i.e. any host.
------	---

It gets the outstanding `ContainerRequests` (from the [YARN ResourceManager](#)).

If there are any outstanding `ContainerRequests` that meet the requirements, it simply takes the first one and puts it in the input `containersToUse` collection. It also removes the `ContainerRequest` so it is not submitted again (it uses the internal `AMRMClient[ContainerRequest]` ).

Otherwise, it puts the input `allocatedContainer` in the input `remaining` collection.

## ContainerLauncher Thread Pool

Caution	<a href="#">FIXME</a>
---------	-----------------------

## processCompletedContainers

```
processCompletedContainers(completedContainers: Seq[ContainerStatus]): Unit
```

`processCompletedContainers` accepts a collection of YARN's [ContainerStatus](#)'es.

Note	<code>ContainerStatus</code> represents the current status of a YARN <code>Container</code> and provides details such as:
	<ul style="list-style-type: none"> <li>• Id</li> <li>• State</li> <li>• Exit status of a completed container.</li> <li>• Diagnostic message for a failed container.</li> </ul>

For each completed container in the collection, `processCompletedContainers` removes it from the internal [releasedContainers](#) registry.

It looks the host of the container up (in the internal [allocatedContainerToHostMap](#) lookup table). The host may or may not exist in the lookup table.

Caution	<a href="#">FIXME</a> The host may or may not exist in the lookup table?
---------	--

The `ExecutorExited` exit reason is computed.

When the host of the completed container has been found, the internal [numExecutorsRunning](#) counter is decremented.

You should see the following INFO message in the logs:

```
INFO Completed container [containerId] [host] (state: [containerState], exit status: [containerExitStatus])
```

For `ContainerExitStatus.SUCCESS` and `ContainerExitStatus.PREEMPTED` exit statuses of the container (which are not considered application failures), you should see one of the two possible INFO messages in the logs:

```
INFO Executor for container [id] exited because of a YARN event (e.g., pre-emption) and not because of an error in the running job.
```

```
INFO Container [id] [host] was preempted.
```

Other exit statuses of the container are considered application failures and reported as a WARN message in the logs:

```
WARN Container killed by YARN for exceeding memory limits. [diagnostics] Consider boosting spark.yarn.executor.memoryOverhead.
```

or

```
WARN Container marked as failed: [id] [host]. Exit status: [containerExitStatus]. Diagnostics: [containerDiagnostics]
```

The host is looked up in the internal `allocatedHostToContainersMap` lookup table. If found, the container is removed from the containers registered for the host or the host itself is removed from the lookup table when this container was the last on the host.

The container is removed from the internal `allocatedContainerToHostMap` lookup table.

The container is removed from the internal `containerIdToExecutorId` translation table. If an executor is found, it is removed from the internal `executorIdToContainer` translation table.

If the executor was recorded in the internal `pendingLossReasonRequests` lookup table, the exit reason (as calculated earlier as `ExecutorExited`) is sent back for every pending RPC message recorded.

If no executor was found, the executor and the exit reason are recorded in the internal `releasedExecutorLossReasons` lookup table.

In case the container was not in the internal `releasedContainers` registry, the internal `numUnexpectedContainerRelease` counter is increased and a `RemoveExecutor` RPC message is sent to the driver (as specified when `YarnAllocator` was created) to notify about

the failure of the executor.

## **numUnexpectedContainerRelease Internal Counter**

## **releasedExecutorLossReasons Internal Lookup Table**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **pendingLossReasonRequests Internal Lookup Table**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **executorIdToContainer Internal Translation Table**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **containerIdToExecutorId Internal Translation Table**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **allocatedHostToContainersMap Internal Lookup Table**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **numExecutorsRunning Internal Counter**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **allocatedContainerToHostMap Internal Lookup Table**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **Allocating YARN Containers for Executors and Cancelling Outstanding Containers (allocateResources method)**

After `ApplicationMaster` is registered to the YARN ResourceManager Spark calls `allocateResources`.

```
allocateResources(): Unit
```

`allocateResources` claims new resource containers from [YARN ResourceManager](#) and cancels any outstanding resource container requests.

**Note**

In YARN, you have to submit requests for resource containers to [YARN ResourceManager](#) first (using [AMRMClient.addContainerRequest](#)) before claiming them by calling [AMRMClient.allocate](#).

Internally, `allocateResources` starts by [submitting requests for new containers and cancelling previous container requests](#).

`allocateResources` then [claims the containers](#) (using the internal reference to YARN's [AMRMClient](#)) with progress indicator of `0.1f`.

You can see the exact moment in the YARN console for the Spark application with the progress bar at 10%.

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	Blacklisted Nodes	ApplicationMaster
application_1469955900130_0001	jacek	Spark shell	SPARK	default	Sun Jul 31 11:05:33 +0200 2016	N/A	RUNNING	UNDEFINED	<div style="width: 10%; height: 10px; background-color: #ccc;"></div>		0	

Figure 3. YARN Console after Allocating YARN Containers (Progress at 10%)

`allocateResources` [gets the list of allocated containers](#) from the [YARN ResourceManager](#).

If the number of allocated containers is greater than `0`, you should see the following DEBUG message in the logs (in stderr on YARN):

```
DEBUG YarnAllocator: Allocated containers: [allocatedContainersSize]. Current executor count: [numExecutorsRunning]. Cluster resources: [availableResources].
```

`allocateResources` [launches executors on the allocated YARN containers](#).

`allocateResources` [gets the list of completed containers' statuses](#) from YARN.

If the number of completed containers is greater than `0`, you should see the following DEBUG message in the logs (in stderr on YARN):

```
DEBUG YarnAllocator: Completed [completedContainersSize] containers
```

`allocateResources` [processes completed containers](#).

You should see the following DEBUG message in the logs (in stderr on YARN):

```
DEBUG YarnAllocator: Finished processing [completedContainersSize] completed containers. Current running executor count: [numExecutorsRunning].
```

## Internal Registries

### hostToLocalTaskCounts

```
hostToLocalTaskCounts: Map[String, Int] = Map.empty
```

Caution	<a href="#">FIXME</a>
---------	-----------------------

### containerIdToExecutorId

Caution	<a href="#">FIXME</a>
---------	-----------------------

### executorIdToContainer

Caution	<a href="#">FIXME</a>
---------	-----------------------

### releasedExecutorLossReasons

Caution	<a href="#">FIXME</a>
---------	-----------------------

### pendingLossReasonRequests

Caution	<a href="#">FIXME</a>
---------	-----------------------

### failedExecutorsTimeStamps

Caution	<a href="#">FIXME</a>
---------	-----------------------

### releasedContainers Internal Registry

`releasedContainers` contains containers of no use anymore by their globally unique identifier `ContainerId` (for a `Container` in the cluster).

**Note**

Hadoop YARN's `Container` represents an allocated resource in the cluster. The YARN ResourceManager is the sole authority to allocate any `Container` to applications. The allocated `Container` is always on a single node and has a unique `containerId`. It has a specific amount of `Resource` allocated.

## Desired Total Number of Executors (targetNumExecutors Internal Attribute)

Initially, when `YarnAllocator` is created, `targetNumExecutors` corresponds to the [initial number of executors](#).

`targetNumExecutors` is eventually reached after `YarnAllocator` updates YARN container allocation requests.

It may later be changed when `YarnAllocator` is requested for executors given locality preferences.

# Introduction to Hadoop YARN

Apache Hadoop 2.0 introduced a framework for job scheduling and cluster resource management and negotiation called **Hadoop YARN (Yet Another Resource Negotiator)**.

YARN is a general-purpose application scheduling framework for distributed applications that was initially aimed at improving MapReduce job management but quickly turned itself into supporting non-MapReduce applications equally, like Spark on YARN.

YARN comes with two components — ResourceManager and NodeManager — running on their own machines.

- [ResourceManager](#) is the master daemon that communicates with YARN clients, tracks resources on the cluster (on NodeManagers), and orchestrates work by assigning tasks to NodeManagers. It coordinates work of ApplicationMasters and NodeManagers.
- [NodeManager](#) is a worker process that offers resources (memory and CPUs) as resource containers. It launches and tracks processes spawned on them.
- **Containers** run tasks, including ApplicationMasters. YARN offers container allocation.

YARN currently defines two **resources**: vcores and memory. **vcore** is a usage share of a CPU core.

YARN ResourceManager keeps track of the cluster's resources while NodeManagers tracks the local host's resources.

It can optionally work with two other components:

- **History Server** for job history
- **Proxy Server** for viewing application status and logs from outside the cluster.

YARN ResourceManager accepts application submissions, schedules them, and tracks their status (through ApplicationMasters). A YARN NodeManager registers with the ResourceManager and provides its local CPUs and memory for resource negotiation.

In a real YARN cluster, there are one ResourceManager (two for High Availability) and multiple NodeManagers.

## YARN ResourceManager

**YARN ResourceManager** manages the global assignment of compute resources to [applications](#), e.g. memory, cpu, disk, network, etc.

## YARN NodeManager

- 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.

## YARN ApplicationMaster

**YARN ResourceManager** manages the global assignment of compute resources to [applications](#), e.g. memory, cpu, disk, network, etc.

- 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.

## YARN's Model of Computation (aka YARN components)

**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 Spark tasks.

An ApplicationMaster is like a *queen bee* that starts creating *worker bees* (in their own containers) in the YARN cluster.

## Others

- 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.
- 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`.
- 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**.
- Distributed Cache for application jar files.
- Preemption (for high-priority applications)

- Queues and nested queues
- User authentication via Kerberos

## Hadoop YARN

- 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.

## ContainerExecutors

- [LinuxContainerExecutor and Docker](#)
- [WindowsContainerExecutor](#)

## LinuxContainerExecutor and Docker

[YARN-3611 Support Docker Containers In LinuxContainerExecutor](#) is an umbrella JIRA issue for Hadoop YARN to support Docker natively.

## Further reading or watching

- [Introduction to YARN](#)
- [Untangling Apache Hadoop YARN, Part 1](#)
- [Quick Hadoop Startup in a Virtual Environment](#)
- (video) [HUG Meetup Apr 2016: The latest of Apache Hadoop YARN and running your docker apps on YARN](#)

# Setting up YARN Cluster

YARN uses the following environment variables:

- `YARN_CONF_DIR`
- `HADOOP_CONF_DIR`
- `HADOOP_HOME`

# Kerberos

- Microsoft incorporated Kerberos authentication into Windows 2000
- Two open source Kerberos implementations exist: the MIT reference implementation and the Heimdal Kerberos implementation.

YARN supports user authentication via Kerberos (so do the other services: HDFS, HBase, Hive).

## Service Delegation Tokens

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Further reading or watching

- (video training) [Introduction to Hadoop Security](#)
- [Hadoop Security](#)
- [Kerberos: The Definitive Guide](#)

# ConfigurableCredentialManager

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Obtaining Security Tokens from Credential Providers (`obtainCredentials` method)

# ClientDistributedCacheManager

`ClientDistributedCacheManager` is a mere *wrapper* to hold the collection of cache-related resource entries `CacheEntry` (as `distCacheEntries`) to [add resources to](#) and later [update Spark configuration with files to distribute](#).

Caution

[FIXME](#) What is a resource? Is this a file only?

## Adding Cache-Related Resource (addResource method)

```
addResource(
  fs: FileSystem,
  conf: Configuration,
  destPath: Path,
  localResources: HashMap[String, LocalResource],
  resourceType: LocalResourceType,
  link: String,
  statCache: Map[URI, FileStatus],
  appMasterOnly: Boolean = false): Unit
```

## Updating Spark Configuration with Resources to Distribute (updateConfiguration method)

```
updateConfiguration(conf: SparkConf): Unit
```

`updateConfiguration` sets the following internal Spark configuration settings in the input `conf` [Spark configuration](#):

- [spark.yarn.cache.filenames](#)
- [spark.yarn.cache.sizes](#)
- [spark.yarn.cache.timestamps](#)
- [spark.yarn.cache.visibilities](#)
- [spark.yarn.cache.types](#)

It uses the internal `distCacheEntries` with [resources to distribute](#).

Note

[It is later used in ApplicationMaster when it prepares local resources.](#)



# YarnSparkHadoopUtil

`YarnSparkHadoopUtil` is...[FIXME](#)

It can only be created when `SPARK_YARN_MODE` flag is enabled.

Note	It belongs to <code>org.apache.spark.deploy.yarn</code> package.
Tip	<p>Enable <code>DEBUG</code> logging level for <code>org.apache.spark.deploy.yarn.YarnSparkHadoopUtil</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.deploy.yarn.YarnSparkHadoopUtil=DEBUG</pre> <p>Refer to <a href="#">Logging</a>.</p>

## MEMORY\_OVERHEAD\_FACTOR

`MEMORY_OVERHEAD_FACTOR` is a constant that equals to `10%` for memory overhead.

## MEMORY\_OVERHEAD\_MIN

`MEMORY_OVERHEAD_MIN` is a constant that equals to `384L` for memory overhead.

## getApplicationAclsForYarn

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Resolving Environment Variable (expandEnvironment method)

```
expandEnvironment(environment: Environment): String
```

`expandEnvironment` resolves `environment` variable using YARN's `Environment.$` or `Environment.$$` methods (depending on the version of Hadoop used).

## Getting YarnSparkHadoopUtil Instance (get method)

Caution

FIXME

## Computing YARN's ContainerId (getContainerId method)

```
getContainerId: ContainerId
```

`getContainerId` is a `private[spark]` method that gets YARN's `ContainerId` from the YARN environment variable `ApplicationConstants.Environment.CONTAINER_ID` and converts it to the return object using YARN's `ConverterUtils.toContainerId`.

## startExecutorDelegationTokenRenewer

Caution

FIXME

## stopExecutorDelegationTokenRenewer

Caution

FIXME

## Calculating Initial Number of Executors (getInitialTargetExecutorNumber method)

```
getInitialTargetExecutorNumber(  
    conf: SparkConf,  
    numExecutors: Int = DEFAULT_NUMBER_EXECUTORS): Int
```

`getInitialTargetExecutorNumber` calculates the initial number of executors for Spark on YARN. It varies by whether [dynamic allocation is enabled or not](#).

Note

The default number of executors (aka `DEFAULT_NUMBER_EXECUTORS`) is `2`.

If [dynamic allocation is enabled](#), `getInitialTargetExecutorNumber` returns the value of `spark.dynamicAllocation.initialExecutors` or `spark.dynamicAllocation.minExecutors` or `0`.

If however [dynamic allocation is disabled](#), `getInitialTargetExecutorNumber` returns the value of `spark.executor.instances` setting or `SPARK_EXECUTOR_INSTANCES` environment variable, or the default value (of the input parameter `numExecutors`) `2`.

Note

It is used to calculate `totalExpectedExecutors` to [start Spark on YARN in client mode](#) or [cluster mode](#).

## addPathToEnvironment

```
addPathToEnvironment(env: HashMap[String, String], key: String, value: String): Unit
```

Caution	<a href="#">FIXME</a>
---------	-----------------------

# Settings

The following settings (aka system properties) are specific to Spark on YARN.

## **spark.yarn.credentials.renewalTime**

`spark.yarn.credentials.renewalTime` (default: `Long.MaxValue ms`) is an internal setting for the time of the next credentials renewal.

See [prepareLocalResources](#).

## **spark.yarn.credentials.updateTime**

`spark.yarn.credentials.updateTime` (default: `Long.MaxValue ms`) is an internal setting for the time of the next credentials update.

## **spark.yarn.rolledLog.includePattern**

`spark.yarn.rolledLog.includePattern`

## **spark.yarn.rolledLog.excludePattern**

`spark.yarn.rolledLog.excludePattern`

## **spark.yarn.am.nodeLabelExpression**

`spark.yarn.am.nodeLabelExpression`

## **spark.yarn.am.attemptFailuresValidityInterval**

`spark.yarn.am.attemptFailuresValidityInterval`

## **spark.yarn.tags**

`spark.yarn.tags`

## **spark.yarn.am.extraLibraryPath**

`spark.yarn.am.extraLibraryPath`

## **spark.yarn.am.extraJavaOptions**

`spark.yarn.am.extraJavaOptions`

## **spark.yarn.scheduler.initial-allocation.interval**

`spark.yarn.scheduler.initial-allocation.interval` (default: `200ms`) controls the initial allocation interval.

It is used when `ApplicationMaster` is instantiated.

## **spark.yarn.scheduler.heartbeat.interval-ms**

`spark.yarn.scheduler.heartbeat.interval-ms` (default: `3s`) is the heartbeat interval to YARN ResourceManager.

It is used when `ApplicationMaster` is instantiated.

## **spark.yarn.max.executor.failures**

`spark.yarn.max.executor.failures` is an optional setting that sets the maximum number of executor failures before...TK

It is used when `ApplicationMaster` is instantiated.

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **spark.yarn.maxAppAttempts**

`spark.yarn.maxAppAttempts` is the maximum number of attempts to register `ApplicationMaster` before deploying a Spark application to YARN is deemed failed.

It is used when `YarnRMClient` computes `getMaxRegAttempts`.

## **spark.yarn.app.id**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **spark.yarn.am.port**

Caution	<a href="#">FIXME</a>
---------	-----------------------

## **spark.yarn.user.classpath.first**

Caution

FIXME

## spark.yarn.archive

`spark.yarn.archive` is the location of the archive containing jars files with Spark classes. It cannot be a `local:` URI.

It is used to populate `CLASSPATH` for `ApplicationMaster` and `executors`.

## spark.yarn.queue

`spark.yarn.queue` (default: `default`) is the name of the YARN resource queue that `client` uses to submit a Spark application to.

You can specify the value using `spark-submit`'s `--queue` command-line argument.

The value is used to set YARN's `ApplicationSubmissionContext.setQueue`.

## spark.yarn.jars

`spark.yarn.jars` is the location of the Spark jars.

```
--conf spark.yarn.jar=hdfs://master:8020/spark/spark-assembly-2.0.0-hadoop2.7.2.jar
```

It is used to populate the `CLASSPATH` for `ApplicationMaster` and `ExecutorRunnables` (when `spark.yarn.archive` is not defined).

Note

`spark.yarn.jar` setting is deprecated as of Spark 2.0.

## spark.yarn.report.interval

`spark.yarn.report.interval` (default: `1s`) is the interval (in milliseconds) between reports of the current application status.

It is used in `Client.monitorApplication`.

## spark.yarn.dist.jars

`spark.yarn.dist.jars` (default: empty) is a collection of additional jars to distribute.

It is used when `Client` distributes additional resources as specified using `--jars` command-line option for `spark-submit`.

## spark.yarn.dist.files

`spark.yarn.dist.files` (default: empty) is a collection of additional files to distribute.

It is used when [Client distributes additional resources as specified using `--files` command-line option for spark-submit](#).

## spark.yarn.dist.archives

`spark.yarn.dist.archives` (default: empty) is a collection of additional archives to distribute.

It is used when [Client distributes additional resources as specified using `--archives` command-line option for spark-submit](#).

## spark.yarn.principal

`spark.yarn.principal` — See the corresponding `--principal` command-line option for spark-submit.

## spark.yarn.keytab

`spark.yarn.keytab` — See the corresponding `--keytab` command-line option for spark-submit.

## spark.yarn.submit.file.replication

`spark.yarn.submit.file.replication` is the replication factor (number) for files uploaded by Spark to HDFS.

## spark.yarn.config.gatewayPath

`spark.yarn.config.gatewayPath` (default: `null`) is the root of configuration paths that is present on gateway nodes, and will be replaced with the corresponding path in cluster machines.

It is used when [client resolves a path to be YARN NodeManager-aware](#).

## spark.yarn.config.replacementPath

`spark.yarn.config.replacementPath` (default: `null`) is the path to use as a replacement for `spark.yarn.config.gatewayPath` when launching processes in the YARN cluster.

It is used when [client resolves a path to be YARN NodeManager-aware](#).

## **spark.yarn.historyServer.address**

`spark.yarn.historyServer.address` is the optional address of the History Server.

## **spark.yarn.access.namenodes**

`spark.yarn.access.namenodes` (default: empty) is a list of extra NameNode URLs for which to request delegation tokens. The NameNode that hosts `fs.defaultFS` does not need to be listed here.

## **spark.yarn.cache.types**

`spark.yarn.cache.types` is an internal setting...

## **spark.yarn.cache.visibilities**

`spark.yarn.cache.visibilities` is an internal setting...

## **spark.yarn.cache.timestamps**

`spark.yarn.cache.timestamps` is an internal setting...

## **spark.yarn.cache.filenames**

`spark.yarn.cache.filenames` is an internal setting...

## **spark.yarn.cache.sizes**

`spark.yarn.cache.sizes` is an internal setting...

## **spark.yarn.cache.confArchive**

`spark.yarn.cache.confArchive` is an internal setting...

## **spark.yarn.secondary.jars**

`spark.yarn.secondary.jars` is...

## **spark.yarn.executor.nodeLabelExpression**

`spark.yarn.executor.nodeLabelExpression` is a node label expression for executors.

## spark.yarn.containerLauncherMaxThreads

`spark.yarn.containerLauncherMaxThreads` (default: `25`)...[FIXME](#)

## spark.yarn.executor.failuresValidityInterval

`spark.yarn.executor.failuresValidityInterval` (default: `-1L`) is an interval (in milliseconds) after which Executor failures will be considered independent and not accumulate towards the attempt count.

## spark.yarn.submit.waitAppCompletion

`spark.yarn.submit.waitAppCompletion` (default: `true`) is a flag to control whether to wait for the application to finish before exiting the launcher process in cluster mode.

## spark.yarn.executor.memoryOverhead

`spark.yarn.executor.memoryoverhead` (in MiBs) is an optional setting for the executor memory overhead (in addition to [spark.executor.memory](#)) when requesting YARN containers from a YARN cluster.

If not set, [Client](#) uses [10%](#) of the [executor memory](#) or [384](#) whatever is larger.

Note

[10%](#) and [384](#) are constants and cannot be changed.

## spark.yarn.am.cores

`spark.yarn.am.cores` (default: `1`) sets the number of CPU cores for ApplicationMaster's JVM.

## spark.yarn.driver.memoryOverhead

`spark.yarn.driver.memoryoverhead` (in MiBs)

## spark.yarn.am.memoryOverhead

`spark.yarn.am.memoryoverhead` (in MiBs)

## spark.yarn.am.memory

`spark.yarn.am.memory` (default: `512m`) sets the memory size of ApplicationMaster's JVM (in MiBs)

## spark.yarn.stagingDir

`spark.yarn.stagingDir` is a staging directory used while submitting applications.

## spark.yarn.preserve.staging.files

`spark.yarn.preserve.staging.files` (default: `false`) controls whether to preserve temporary files in a staging directory (as pointed by [spark.yarn.stagingDir](#)).

## spark.yarn.credentials.file

`spark.yarn.credentials.file` ...

## spark.yarn.launchContainers

`spark.yarn.launchContainers` (default: `true`) is a flag used for testing only so `YarnAllocator` does not run launch `ExecutorRunnables` on allocated YARN containers.

# Spark Standalone cluster

**Spark Standalone cluster** (aka *Spark deploy cluster* or *standalone cluster*) is Spark's own built-in clustered environment. Since Spark Standalone is available in the default distribution of Apache Spark it is the easiest way to run your Spark applications in a clustered environment in many cases.

**Standalone Master** (often written *standalone Master*) is the resource manager for the Spark Standalone cluster (read [Standalone Master](#) for in-depth coverage).

**Standalone Worker** (aka *standalone slave*) is the worker in the Spark Standalone cluster (read [Standalone Worker](#) for in-depth coverage).

**Note**

Spark Standalone cluster is one of the three available clustering options in Spark (refer to [Running Spark on cluster](#)).

**Caution**

**FIXME** A figure with SparkDeploySchedulerBackend sending messages to AppClient and AppClient RPC Endpoint and later to Master.

SparkDeploySchedulerBackend → AppClient → AppClient RPC Endpoint → Master

Add SparkDeploySchedulerBackend as AppClientListener in the picture

In Standalone cluster mode Spark allocates resources based on cores. By default, an application will grab all the cores in the cluster (read [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 (read [Master URLs](#)).

**Caution**

**FIXME** That might be **very** confusing!

You can deploy, i.e. `spark-submit`, your applications to Spark Standalone in `client` or `cluster` deploy mode (read [Deployment modes](#)).

## Deployment modes

**Caution**

**FIXME**

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	<a href="#">FIXME</a>
---------	-----------------------

```
→ 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

If enabled (using [spark.deploy.spreadOut](#)), standalone Master attempts to spread out an application's executors on as many workers as possible (instead of trying to consolidate it onto a small number of nodes).

Note	It is enabled by default.
------	---------------------------

## scheduleExecutorsOnWorkers

Caution	<a href="#">FIXME</a>
---------	-----------------------

```
scheduleExecutorsOnWorkers(
  app: ApplicationInfo,
  usableWorkers: Array[WorkerInfo],
  spreadOutApps: Boolean): Array[Int]
```

`scheduleExecutorsOnWorkers` schedules executors on workers.

## 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 a 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](#).

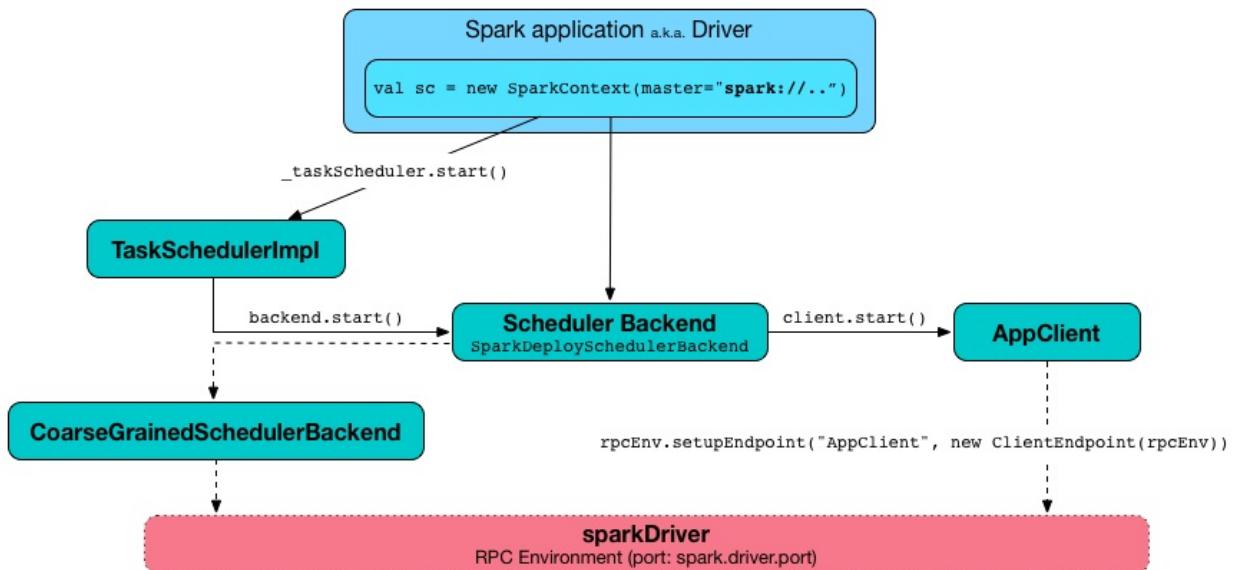


Figure 1. `SparkDeploySchedulerBackend.start()` (while `SparkContext` starts)

Caution	<a href="#">FIXME</a> <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](#)).

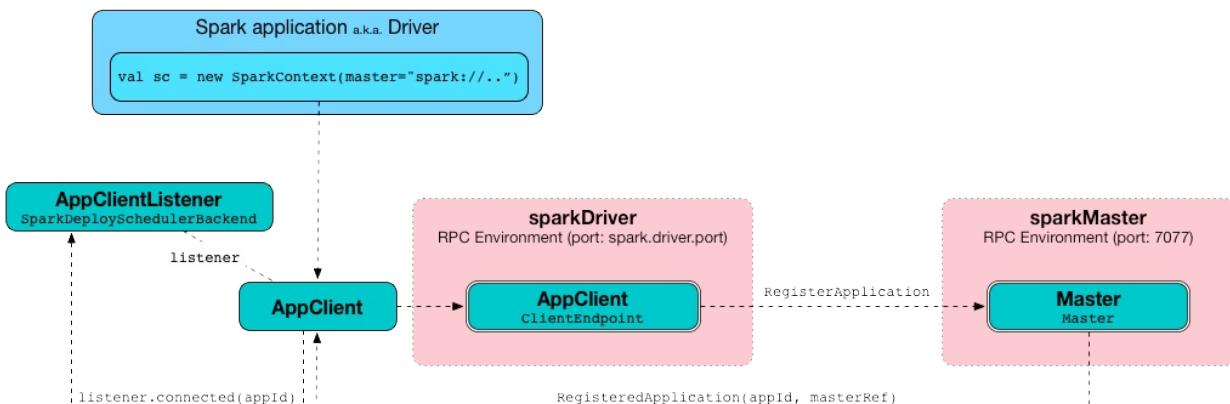


Figure 2. AppClient registers application to standalone Master

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	<a href="#">FIXME</a> 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	<a href="#">FIXME</a> the message
---------	-----------------------------------

```
INFO Executor updated: %s is now %s%
```

## MasterChanged RPC message

`MasterChanged` is received from the primary master to inform about...[FIXME](#)

Caution	<b>FIXME</b> 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.

## Settings

### spark.deploy.spreadOut

`spark.deploy.spreadout` (default: `true`) controls whether standalone Master should perform [round-robin scheduling across the nodes](#).

# Standalone 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.

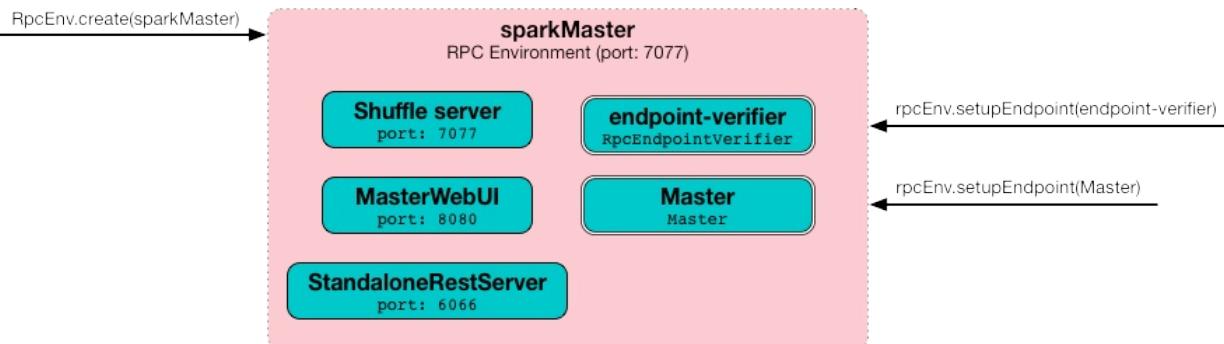


Figure 1. `sparkMaster` - the RPC Environment for Spark Standalone's master  
Master endpoint is a [ThreadSafeRpcEndpoint](#) and [LeaderElectable](#) (see [Leader Election](#)).

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 ni
d=0x7f03 waiting on condition [0x000000011cad9000]
```

## 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**

Caution	<b>FIXME</b> <ul style="list-style-type: none"> <li>• Review <code>org.apache.spark.metrics.MetricsConfig</code></li> <li>• How to access the metrics for master? See <code>Master#onStart</code></li> <li>• Review <code>masterMetricsSystem</code> and <code>applicationMetricsSystem</code></li> </ul>
---------	---

## 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 <a href="#">Spark Standalone - Using ZooKeeper for High-Availability of Master</a> .
-----	---

## Leader Election

Master endpoint is `LeaderElectable`, i.e. [FIXME](#)

Caution	<a href="#">FIXME</a>
---------	-----------------------

## 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.

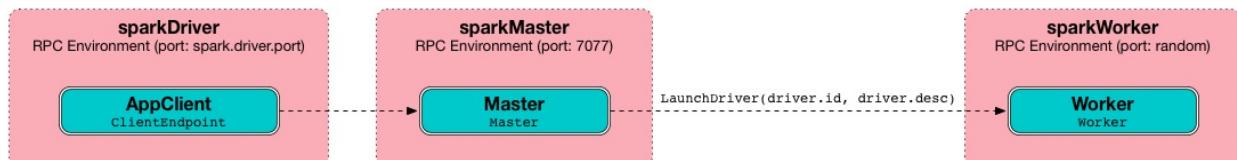


Figure 2. Master finds a place for a driver (posts LaunchDriver)

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	<a href="#">FIXME</a> 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 <a href="#">Spark Tips and Tricks</a> document.
-----	---

When in troubles, consult [Spark Tips and Tricks](#) document.

It starts [RPC Environment](#) with necessary endpoints and lives until the RPC environment terminates.

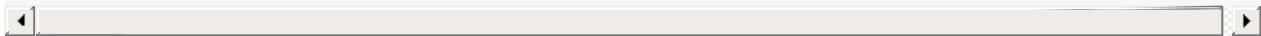
## Worker Management

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	<b>FIXME</b>
	<ul style="list-style-type: none"> <li>• Where are `RETAINED_`'s properties used?</li> </ul>

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` to perform [round-robin scheduling across the 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](#)

# Standalone Worker

**Standalone Worker** (aka *standalone slave*) is the worker in Spark Standalone cluster.

You can have one or many standalone workers in a standalone cluster. They can be started and stopped using [custom management scripts for standalone workers](#).

# 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 address bar shows "localhost:8080/app/?appId=app-20160218212811-0000". The main content area has the "Spark 2.0.0-SNAPSHOT" logo and the title "Application: Spark shell". Below it, application details are listed:

- 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
- State: RUNNING

A blue link "Application Detail UI" is visible. Below this is a section titled "Executor Summary" with a table:

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.

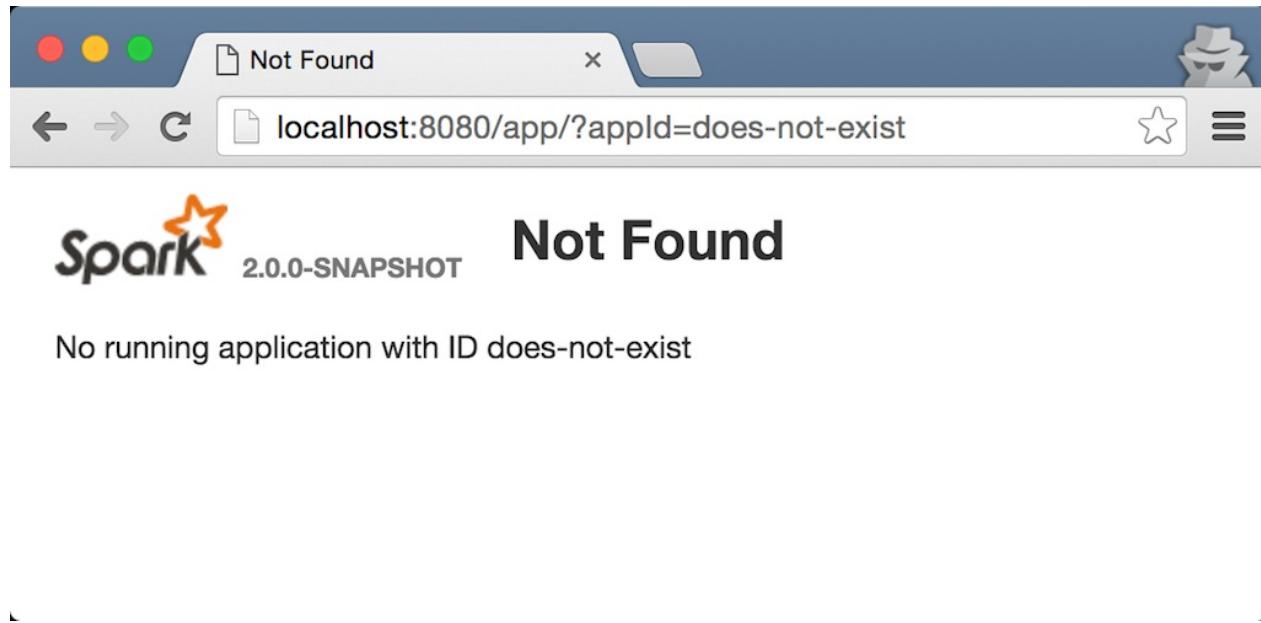


Figure 2. Application Not Found Page

# Submission Gateways

Caution	<a href="#">FIXME</a>
---------	-----------------------

From `SparkSubmit.submit` :

In standalone cluster mode, there are two submission gateways:

1. The traditional legacy RPC gateway using `o.a.s.deploy.Client` as a wrapper
2. The new REST-based gateway introduced in Spark 1.3

The latter is the default behaviour as of Spark 1.3, but `Spark submit` will fail over to use the legacy gateway if the master endpoint turns out to be not a REST server.

# 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 <code>SPARK_PRINT_LAUNCH_COMMAND</code> environment variable to print out the launch command to standard error output. Refer to <a href="#">Print Launch Command of Spark Scripts</a> .
------	--

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:7077
```

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 8
080
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 8
080
```

## 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 8
080

$ 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: <ul style="list-style-type: none"><li>• <a href="#">Operating Spark master</a></li><li>• <a href="#">Starting Spark workers on node using sbin/start-slave.sh</a></li></ul>
Important	You can use the Spark Standalone cluster in the following ways: <ul style="list-style-type: none"><li>• Use <code>spark-shell</code> with <code>--master MASTER_URL</code></li><li>• Use <code>SparkConf.setMaster(MASTER_URL)</code> in your Spark application</li></ul> For our learning purposes, <code>MASTER_URL</code> is <code>spark://localhost:7077</code> .

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.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

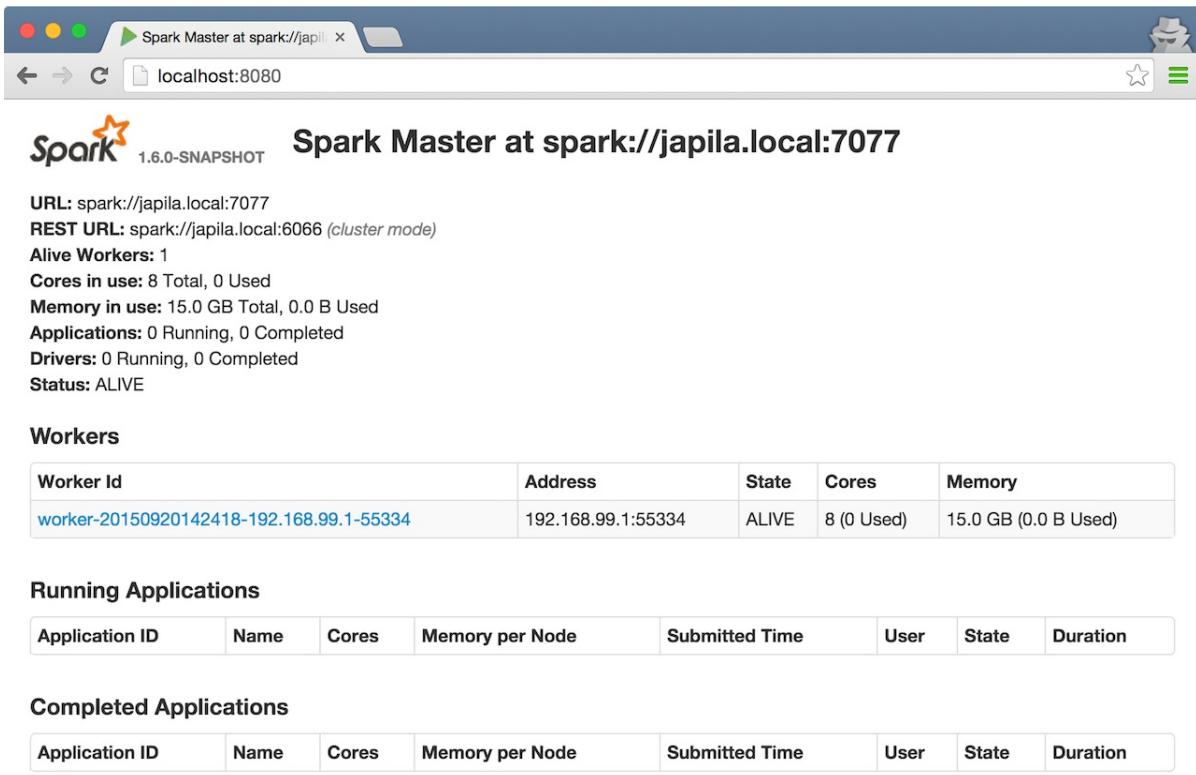
3. Start the first worker.

```
./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's web interface at <http://localhost:8080>. The title bar says "Spark Master at spark://japila.local:7077". The page 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: 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**

Worker Id	Address	State	Cores	Memory
<code>worker-20150920142418-192.168.99.1-55334</code>	<code>192.168.99.1:55334</code>	ALIVE	8 (0 Used)	15.0 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 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.

**Spark Master at spark://japila.local:7077**

URL: spark://japila.local:7077  
 REST URL: spark://japila.local:6066 (cluster mode)  
 Alive Workers: 0  
 Cores in use: 0 Total, 0 Used  
 Memory in use: 0.0 B 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-20150920142418-192.168.99.1-55334	192.168.99.1:55334	DEAD	8 (0 Used)	15.0 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 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 --memory 4g
```

Note	The command translates to <code>org.apache.spark.deploy.worker.Worker --webui-port 8081 spark://japila.local:7077 --cores 2 --memory 4g</code>
------	--

- 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`. The UI includes a summary of cluster resources, a table of workers, and sections for running and completed applications.

**Summary:**

- 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**

Worker Id	Address	State	Cores	Memory
<a href="#">worker-20150920142418-192.168.99.1-55334</a>	192.168.99.1:55334	DEAD	8 (0 Used)	15.0 GB (0.0 B Used)
<a href="#">worker-20150920144742-192.168.99.1-55538</a>	192.168.99.1:55538	ALIVE	2 (0 Used)	4.0 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 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)

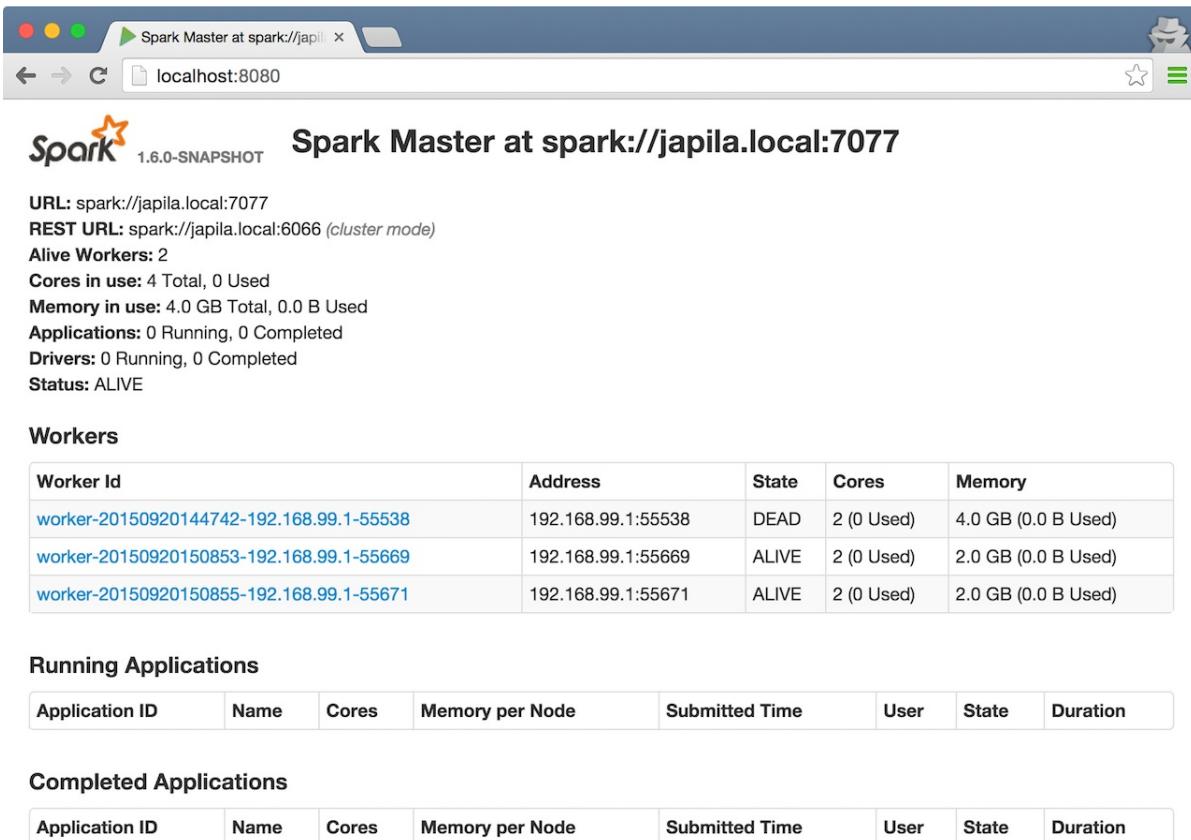
#### 10. Start the workers.

```
./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
./logs/spark-jacek-org.apache.spark.deploy.worker.Worker-1-
japila.local.out
starting org.apache.spark.deploy.worker.Worker, logging to
./logs/spark-jacek-org.apache.spark.deploy.worker.Worker-2-
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 interface at <http://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

<b>Note</b>	Use <code>jps</code> on master to see the instances given they all run on the same machine, e.g. <code>localhost</code> ).
	<pre>\$ jps 6580 Worker 4872 Master 6874 Jps 6539 Worker</pre>

12. Stop all instances - the driver and the workers.

```
./sbin/stop-all.sh
```

# StandaloneSchedulerBackend

Caution

[FIXME](#)

## Starting StandaloneSchedulerBackend (start method)

```
start(): Unit
```

# Spark on Mesos

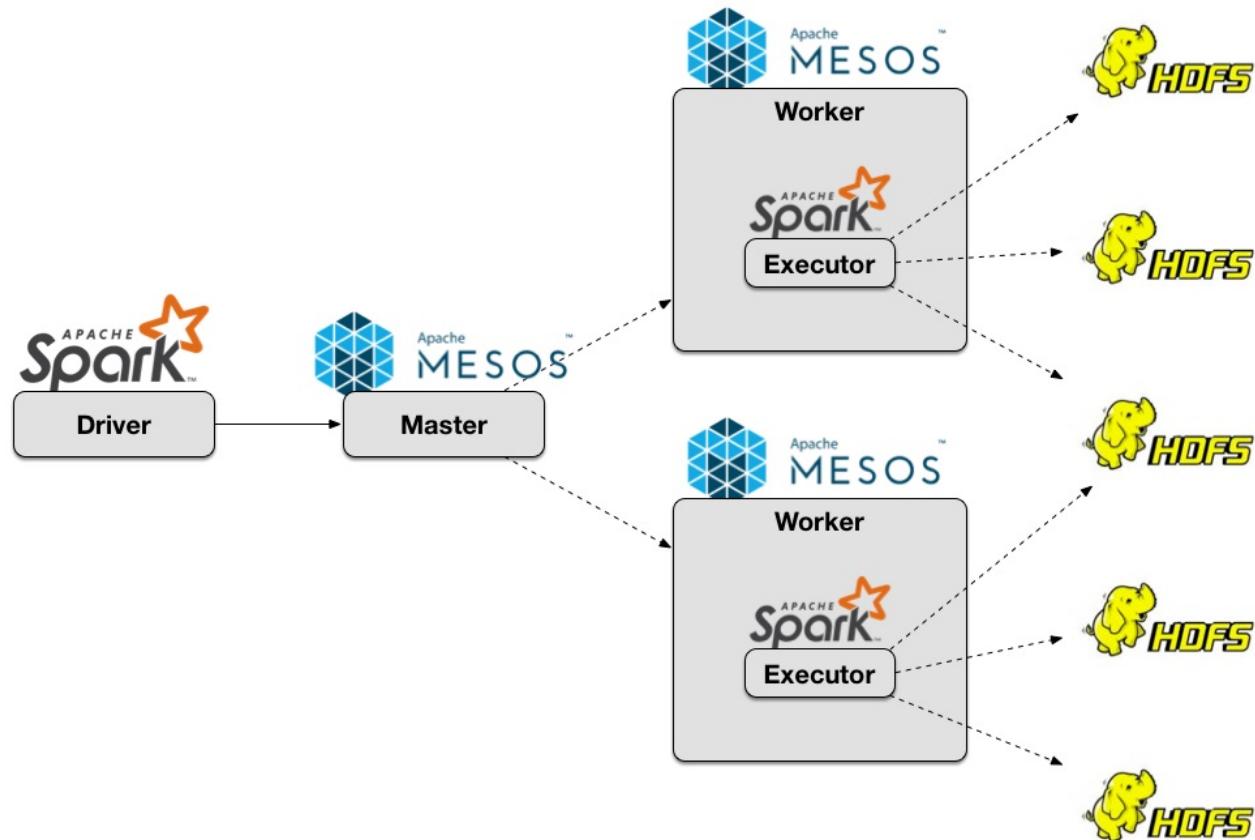


Figure 1. Spark on Mesos Architecture

## 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
I0401 00:12:01.955883 1916461824 main.cpp:237] Build: 2016-03-17 14:20:58 by brew
I0401 00:12:01.956457 1916461824 main.cpp:239] Version: 0.28.0
I0401 00:12:01.956538 1916461824 main.cpp:260] Using 'HierarchicalDRF' allocator
I0401 00:12:01.957381 1916461824 main.cpp:471] Starting Mesos master
I0401 00:12:01.964118 1916461824 master.cpp:375] Master 9867c491-5370-48cc-8e25-e1aff1
d86542 (localhost) started on 127.0.0.1:5050
...
```

Visit the management console at <http://localhost:5050>.

The screenshot shows the Mesos Management Console interface. At the top, there are tabs for Mesos, Frameworks, Slaves, and Offers. The Slaves tab is selected. In the center, there is a section titled "Active Tasks" which displays a table with columns for ID, Name, State, Started, and Host. A note says "No active tasks." Below it is a section titled "Completed Tasks" with a similar table showing "No completed tasks." To the left, there are several summary sections: "Cluster: (Unnamed)", "Server: 127.0.0.1:5050", "Version: 0.28.0", "Built: 2 weeks ago by brew", "Started: 2 minutes ago", and "Elected: 2 minutes ago". Other sections include "LOG", "Slaves" (with Activated and Deactivated counts), "Tasks" (with Staging, Starting, Running, Killing, Finished, Killed, Failed, and Lost counts), and "Resources" (with CPU, Mem, and Disk usage). A small icon of a person is visible in the bottom right corner.

Figure 2. Mesos Management Console

Run Mesos Slave onto which Master will dispatch jobs.

```
$ mesos-slave --master=127.0.0.1:5050
I0401 00:15:05.850455 1916461824 main.cpp:223] Build: 2016-03-17 14:20:58 by brew
I0401 00:15:05.850772 1916461824 main.cpp:225] Version: 0.28.0
I0401 00:15:05.852812 1916461824 containerizer.cpp:149] Using isolation: posix/cpu,posix/mem,filesystem posix
I0401 00:15:05.866186 1916461824 main.cpp:328] Starting Mesos slave
I0401 00:15:05.869470 218980352 slave.cpp:193] Slave started on 1)@10.1.47.199:5051
...
I0401 00:15:05.906355 218980352 slave.cpp:832] Detecting new master
I0401 00:15:06.762917 220590080 slave.cpp:971] Registered with master master@127.0.0.1:5050; given slave ID 9867c491-5370-48cc-8e25-e1aff1d86542-S0
...
```

Switch to the management console at <http://localhost:5050/#/slaves> to see the slaves available.

The screenshot shows the Mesos Management Console with the Slaves tab selected. At the top, there are tabs for Mesos, Frameworks, Slaves, and Offers. The Slaves tab is selected. Below the tabs, there is a breadcrumb navigation showing "Master / Slaves". The main area is titled "Slaves" and contains a table with columns for ID, Host, CPUs, Mem, Disk, Registered, and Re-Registered. One row is visible: "...aaea-5dffbfd44e5d-S0" with host 192.168.1.4, 8 CPUs, 15.0 GB Mem, 459.8 GB Disk, registered a minute ago, and re-registered a minute ago. There is also a search bar with a dropdown arrow and the placeholder "Find...".

Figure 3. Mesos Management Console (Slaves tab) with one slave running

<p><b>Important</b></p>	<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>
-------------------------	--

<p><b>Note</b></p>	<p>The preferred approach to launch Spark on Mesos and to give the location of Spar 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` ...
I0401 00:17:41.806743 581939200 sched.cpp:222] Version: 0.28.0
I0401 00:17:41.808825 579805184 sched.cpp:326] New master detected at master@127.0.0.1:5050
I0401 00:17:41.808976 579805184 sched.cpp:336] No credentials provided. Attempting to register without authentication
I0401 00:17:41.809605 579268608 sched.cpp:703] Framework registered with 9867c491-5370-48cc-8e25-e1aff1d86542-0001
Spark context available as sc (master = mesos://127.0.0.1:5050, app id = 9867c491-5370-48cc-8e25-e1aff1d86542-0001).
...
```

In [Frameworks tab](#) you should see a single active framework for `spark-shell`.

ID	Host	User	Name	Active Tasks	CPU	Mem	Disk	Max Share	Registered	Re-Registered
...8e25-e1aff1d86542-0001	japila.local	jacek	Spark shell	1	8	1.4 GB	0 B	100%	a minute ago	-

Figure 4. Mesos Management Console (Frameworks tab) with Spark shell active

<p><b>Tip</b></p>	<p>Consult slave logs under <code>/tmp/mesos/slaves</code> when facing troubles.</p>
-------------------	--

<p><b>Important</b></p>	<p>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.</p>
-------------------------	--

```
scala> sc.parallelize(0 to 10, 8).count
res0: Long = 11
```

The screenshot shows the Mesos Management Console interface. On the left, there's a sidebar with information about the executor: Name, Source, Cluster (Unnamed), Master (localhost), Active Tasks (0), and Resources (CPU: 0.011 Used, 1 Allocated; Mem: 402 MB Used, 1.4 GB Allocated; Disk: 0 B Used, 0 B Allocated). The main area is titled 'Completed Tasks' and contains a table with the following data:

ID	Name	State	CPU (allocated)	Mem (allocated)	
7	task 7.0 in stage 0.0	TASK_FINISHED	1	0 B	Sandbox
6	task 6.0 in stage 0.0	TASK_FINISHED	1	0 B	Sandbox
5	task 5.0 in stage 0.0	TASK_FINISHED	1	0 B	Sandbox
4	task 4.0 in stage 0.0	TASK_FINISHED	1	0 B	Sandbox
3	task 3.0 in stage 0.0	TASK_FINISHED	1	0 B	Sandbox
2	task 2.0 in stage 0.0	TASK_FINISHED	1	0 B	Sandbox
1	task 1.0 in stage 0.0	TASK_FINISHED	1	0 B	Sandbox
0	task 0.0 in stage 0.0	TASK_FINISHED	1	0 B	Sandbox

Figure 5. 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
-aaea-5dffbfd44e5d-0002'
```

## 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	<a href="#">FIXME</a>
---------	-----------------------

## Settings

- `spark.mesos.coarse` (default: `true`) controls whether the scheduler backend for Mesos works in coarse- (`CoarseMesosSchedulerBackend`) or fine-grained mode (`MesosSchedulerBackend`).

Caution	<a href="#">FIXME</a> Review <ul style="list-style-type: none"> <li>• <a href="#">MesosClusterScheduler.scala</a></li> <li>• <a href="#">MesosExternalShuffleService</a></li> </ul>
---------	---

## 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 --executor-memory 3G --conf spark.mesos.role=dev ./examples/src/main/python/pi.py 100
```

## 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.

# MesosCoarseGrainedSchedulerBackend — Coarse-Grained Scheduler Backend for Mesos

Caution	<a href="#">FIXME</a>
---------	-----------------------

## (executorLimitOption attribute)

`executorLimitOption` is an internal attribute to...[FIXME](#)

# About Mesos

[Apache Mesos](#) is an Apache Software Foundation open source cluster management and scheduling framework. It abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual).

Mesos provides API for resource management and scheduling across multiple nodes (in datacenter and cloud environments).

Tip

Visit [Apache Mesos](#) to learn more about the project.

Mesos is a *distributed system kernel* with a pool of resources.

"If a service fails, kill and replace it".

An Apache Mesos cluster consists of three major components: masters, agents, and frameworks.

## Concepts

A Mesos *master* manages agents. It is responsible for tracking, pooling and distributing agents' resources, managing active applications, and task delegation.

A Mesos *agent* is the worker with resources to execute tasks.

A Mesos *framework* is an application running on a Apache Mesos cluster. It runs on agents as tasks.

The Mesos master *offers resources* to frameworks that can *accept* or *reject* them based on specific *constraints*.

A *resource offer* is an offer with CPU cores, memory, ports, disk.

Frameworks: Chronos, Marathon, Spark, HDFS, YARN (Myriad), Jenkins, Cassandra.

- Mesos API
- Mesos is a *scheduler of schedulers*
- 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

# Execution Model

Caution

**FIXME** This is the **single** place for explaining jobs, stages, tasks. Move relevant parts from the other places.

# Optimising Spark

- Caching and Persistence
- Broadcast variables
- Accumulators

# RDD Caching and Persistence

**Caching or persistence** are optimisation techniques for (iterative and interactive) Spark computations. They help saving interim partial results so they can be reused in subsequent stages. These interim results as RDDs are thus kept in memory (default) or more solid storages like disk and/or replicated.

RDDs can be **cached** using `cache` operation. They can also be **persisted** using `persist` operation.

The difference between `cache` and `persist` operations is purely syntactic. `cache` is a synonym of `persist` or `persist(MEMORY_ONLY)`, i.e. `cache` is merely `persist` with the default storage level `MEMORY_ONLY`.

Note

Due to the very small and purely syntactic difference between caching and persistence of RDDs the two terms are often used interchangeably and I will follow the "pattern" here.

RDDs can also be **unpersisted** to remove RDD from a permanent storage like memory and/or disk.

## Caching RDD (cache method)

```
cache(): this.type = persist()
```

`cache` is a synonym of `persist` with `StorageLevel.MEMORY_ONLY` storage level.

## Persisting RDD (persist method)

```
persist(): this.type
persist(newLevel: StorageLevel): this.type
```

`persist` marks a RDD for persistence using `newLevel` storage level.

You can only change the storage level once or a `UnsupportedOperationException` is thrown:

Cannot change storage level of an RDD after it was already assigned a level

Note

You can *pretend* to change the storage level of an RDD with already-assigned storage level only if the storage level is the same as it is currently assigned.

If the RDD is marked as persistent the first time, the RDD is registered to `ContextCleaner` (if available) and `SparkContext`.

The internal `storageLevel` attribute is set to the input `newLevel` storage level.

## 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.

```
val lines = sc.textFile("README.md")  
  
scala> lines.getStorageLevel  
res0: org.apache.spark.storage.StorageLevel = StorageLevel(disk=false, memory=false, offheap=false, deserialized=false, replication=1)
```

## Unpersisting RDDs (Clearing Blocks) (unpersist method)

```
unpersist(blocking: Boolean = true): this.type
```

When called, `unpersist` prints the following INFO message to the logs:

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

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

# 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.

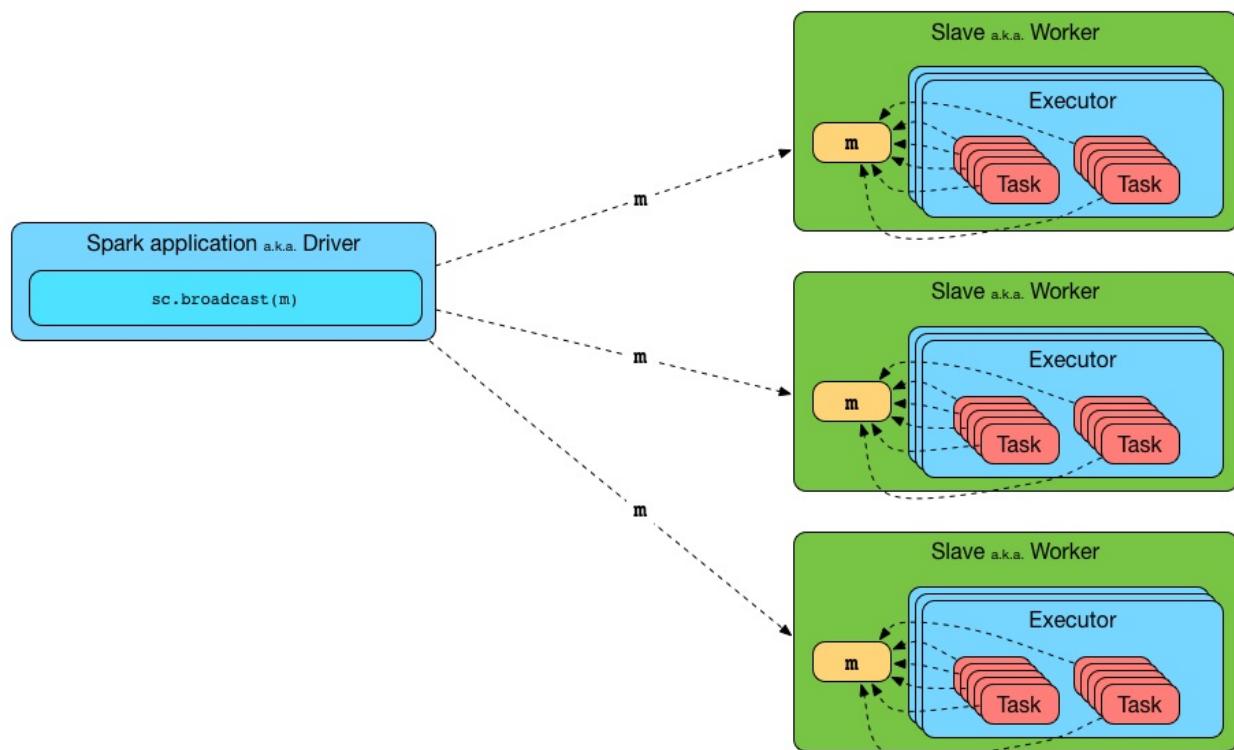


Figure 1. Broadcasting a value to executors

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](#) section.

The Broadcast feature in Spark uses `SparkContext` to create broadcast values and `BroadcastManager` and `ContextCleaner` to manage their lifecycle.

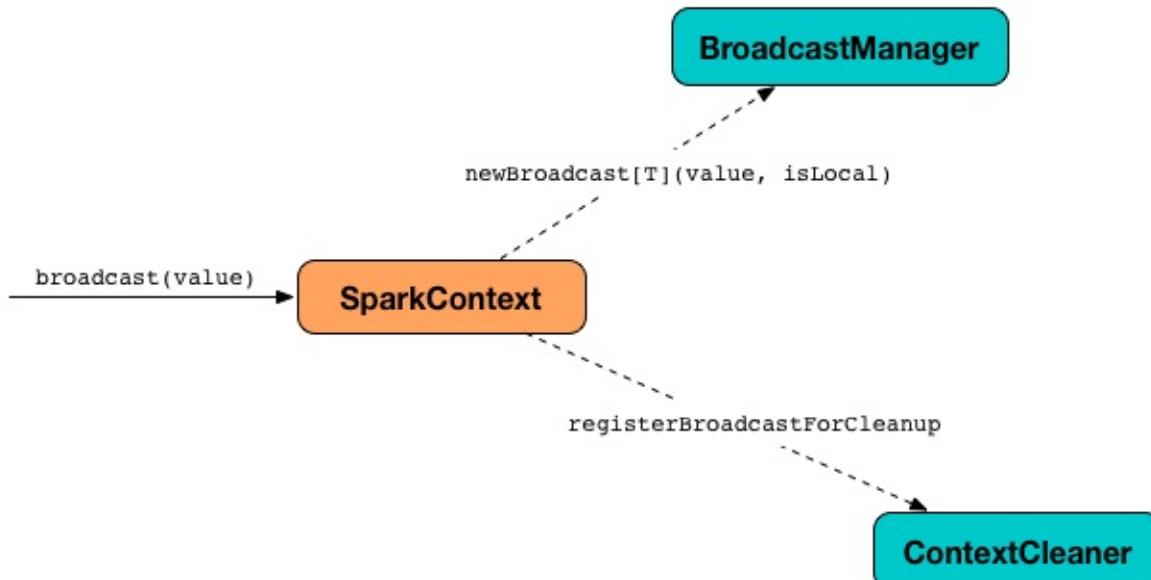


Figure 2. SparkContext to broadcast using BroadcastManager and ContextCleaner

## 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.scala-lang.org/")
pws: scala.collection.immutable.Map[String,String] = Map(Apache Spark -> http://spark.apache.org/, Scala -> http://www.scala-lang.org/)

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.

```
val pwsB = sc.broadcast(pws)
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.

You use broadcast variable to implement **map-side join**, i.e. a join using a `map`. For this, lookup tables are distributed across nodes in a cluster using `broadcast` and then looked up inside `map` (to do the join implicitly).

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.

It appears that a Spark idiom emerges that uses `broadcast` with `collectAsMap` to create a `Map` for broadcast. When an RDD is `map` over to a smaller dataset (column-wise not record-wise), `collectAsMap`, and `broadcast`, using the very big RDD to map its elements to the broadcast RDDs is computationally faster.

```
val acMap = sc.broadcast(myRDD.map { case (a,b,c,b) => (a, c) }.collectAsMap)
val otherMap = sc.broadcast(myOtherRDD.collectAsMap)

myBigRDD.map { case (a, b, c, d) =>
  (acMap.value.get(a).get, otherMap.value.get(c).get)
}.collect
```

Use large broadcasted HashMaps over RDDs whenever possible and leave RDDs with a key to lookup necessary data as demonstrated above.

Spark comes with a BitTorrent implementation.

It is not enabled by default.

## SparkContext.broadcast

Read about `SparkContext.broadcast` method in [Creating broadcast variables](#).

## Further Reading

- [Map-Side Join in Spark](#)

# Accumulators

**Accumulators** are variables that are "added" to through an associative and commutative "add" operation. They act as a container for accumulating partial values across multiple tasks running on executors. They are designed to be used safely and efficiently in parallel and distributed Spark computations and are meant for distributed counters and sums.

You can create built-in accumulators for [longs](#), [doubles](#), or [collections](#) or register custom accumulators using the [SparkContext.register](#) methods. You can create accumulators with or without a name, but only [named accumulators](#) are displayed in [web UI](#) (under Stages tab for a given stage).

Accumulators									
Accumulable									
Tasks									
Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		

Figure 1. Accumulators in the Spark UI

Accumulator are write-only variables for executors. They can be added to by executors and read by the driver only.

```

executor1: accumulator.add(incByExecutor1)
executor2: accumulator.add(incByExecutor2)

driver: println(accumulator.value)

```

Accumulators are not thread-safe. They do not really have to since the [DAGScheduler.updateAccumulators](#) method that the driver uses to update the values of accumulators after a task completes (successfully or with a failure) is only executed on a [single thread that runs scheduling loop](#). Beside that, they are write-only data structures for workers that have their own local accumulator reference whereas accessing the value of an accumulator is only allowed by the driver.

Accumulators are serializable so they can safely be referenced in the code executed in executors and then safely send over the wire for execution.

```
val counter = sc.longAccumulator("counter")
sc.parallelize(1 to 9).foreach(x => counter.add(x))
```

Internally, [longAccumulator](#), [doubleAccumulator](#), and [collectionAccumulator](#) methods create the built-in typed accumulators and call [SparkContext.register](#).

Tip

Read the official documentation about [Accumulators](#).

## AccumulatorV2

```
abstract class AccumulatorV2[IN, OUT]
```

`AccumulatorV2` parameterized class represents an accumulator that accumulates `IN` values to produce `OUT` result.

## Registering Accumulator (register method)

```
register(
  sc: SparkContext,
  name: Option[String] = None,
  countFailedValues: Boolean = false): Unit
```

`register` is a `private[spark]` method of the `AccumulatorV2` abstract class.

It creates a `AccumulatorMetadata` metadata object for the accumulator (with a new unique identifier) and registers the accumulator with [AccumulatorContext](#). The accumulator is then registered with [ContextCleaner](#) for cleanup.

## AccumulatorContext

`AccumulatorContext` is a `private[spark]` internal object used to track accumulators by Spark itself using an internal `originals` lookup table. Spark uses the `AccumulatorContext` object to register and unregister accumulators.

The `originals` lookup table maps accumulator identifier to the accumulator itself.

Every accumulator has its own unique accumulator id that is assigned using the internal `nextId` counter.

## AccumulatorContext.SQL\_ACCUM\_IDENTIFIER

`AccumulatorContext.SQL_ACCUM_IDENTIFIER` is an internal identifier for Spark SQL's internal accumulators. The value is `sql` and Spark uses it to distinguish [Spark SQL metrics](#) from others.

## Named Accumulators

An accumulator can have an optional name that you can specify when [creating an accumulator](#).

```
val counter = sc.longAccumulator("counter")
```

## AccumulableInfo

`AccumulableInfo` contains information about a task's local updates to an [Accumulable](#).

- `id` of the accumulator
- optional `name` of the accumulator
- optional partial `update` to the accumulator from a task
- `value`
- whether or not it is `internal`
- whether or not to `countFailedValues` to the final value of the accumulator for failed tasks
- optional `metadata`

`AccumulableInfo` is used to transfer accumulator updates from executors to the driver every executor heartbeat or when a task finishes.

Create an representation of this with the provided values.

## When are Accumulators Updated?

### Examples

#### Example: Distributed Counter

Imagine you are requested to write a distributed counter. What do you think about the following solutions? What are the pros and cons of using it?

```
val ints = sc.parallelize(0 to 9, 3)

var counter = 0
ints.foreach { n =>
  println(s"int: $n")
  counter = counter + 1
}
println(s"The number of elements is $counter")
```

How would you go about doing the calculation using accumulators?

## Example: Using Accumulators in Transformations and Guarantee Exactly-Once Update

Caution	<b>FIXME</b> Code with failing transformations (tasks) that update accumulator ( <code>Map</code> ) with <code>TaskContext</code> info.
---------	---

## Example: Custom Accumulator

Caution	<b>FIXME</b> Improve the earlier example
---------	--

## Example: Distributed Stopwatch

Note	This is <i>almost</i> a raw copy of org.apache.spark.ml.util.DistributedStopwatch.
------	--

```
class DistributedStopwatch(sc: SparkContext, val name: String) {

  val elapsedTime: Accumulator[Long] = sc.accumulator(0L, s"DistributedStopwatch($name)")

  override def elapsed(): Long = elapsedTime.value

  override protected def add(duration: Long): Unit = {
    elapsedTime += duration
  }
}
```

## Further reading or watching

- 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.

## SecurityManager

Caution	<a href="#">FIXME</a> Likely move to a separate page with references here.
---------	--

# Securing Web UI

Tip	Read the official document <a href="#">Web UI</a> .
-----	---

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[LongWritable], classOf[Text])
```

## 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).

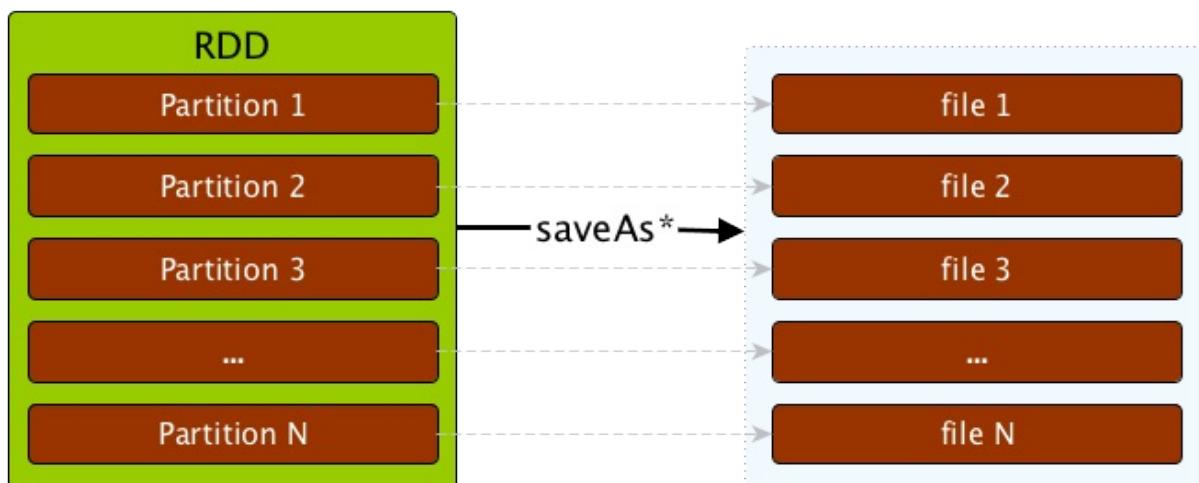


Figure 1. `saveAs` on RDD

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 file:/Users/jacek/dev/oss/spark/numbers/_temporary/0/task_201511050904_0000_m_00000
1
INFO FileOutputCommitter: Saved output of task 'attempt_201511050904_0000_m_000002_2'
to file:/Users/jacek/dev/oss/spark/numbers/_temporary/0/task_201511050904_0000_m_00000
2
INFO FileOutputCommitter: Saved output of task 'attempt_201511050904_0000_m_000000_0'
to file:/Users/jacek/dev/oss/spark/numbers/_temporary/0/task_201511050904_0000_m_00000
0
INFO FileOutputCommitter: Saved output of task 'attempt_201511050904_0000_m_000003_3'
to file:/Users/jacek/dev/oss/spark/numbers/_temporary/0/task_201511050904_0000_m_00000
3
...
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 file:/Users/jacek/dev/oss/spark/numbers1/_temporary/0/task_201511050907_0002_m_0000
00
```

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.g
z:0+38
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 spark.
Welcome to

    ___
   / _ \
  _\ \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:258)
    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)
...
...
```

# 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** and **schema evolution**.
- faster than json/gzip
- [Used in Spark SQL](#).

# Serialization

Serialization systems:

- Java serialization
- Kryo
- Avro
- Thrift
- Protobuf

# Spark and Apache Cassandra

[DataStax Spark Cassandra Connector](#)

## Rules for Effective Spark-Cassandra Setup

1. Use Cassandra nodes to host Spark executors for **data locality**. In this setup a Spark executor will talk to a local Cassandra node and will only query for local data. It is supposed to make queries faster by reducing the usage of network to send data between Spark executors (to process data) and Cassandra nodes (where data lives).
2. Set up a dedicated Cassandra cluster for a Spark analytics batch workload - **Analytics Data Center**. Since it is more about batch processing with lots of table scans they would interfere with caches for real-time data reads and writes.
3. Spark jobs write results back to Cassandra.

## Core Concepts of Cassandra

- A **keyspace** is a space for tables and resembles a schema in a relational database.
- A **table** stores data (and is a table in a relational database).
- A table uses **partitions** to group data.
- Partitions are groups of **rows**.
- **Partition keys** to determine the location of partitions. They are used for grouping.
- **Clustering keys** to determine ordering of rows in partitions. They are used for sorting.
- **CQL** (aka *Cassandra Query Language*) to create tables and query data.

## Further reading or watching

- [Excellent write-up about how to run Cassandra inside Docker](#) from DataStax. Read it as early as possible!
- (video) [Getting Started with Spark & Cassandra](#)

# Spark and Apache Kafka

[Apache Kafka](#) is a distributed partitioned commit log.

Caution	<p><a href="#">FIXME</a>:</p> <ol style="list-style-type: none"><li>1. Kafka Direct API in Spark Streaming</li><li>2. Getting information on the current topic being consumed by each executor</li></ol>
---------	--

# Couchbase Spark Connector

The Couchbase Spark Connector provides an open source integration between Apache Spark and Couchbase Server.

Tip	Read the official documentation in <a href="#">Spark Connector 1.2</a> .
Caution	<b>FIXME</b> Describe the features listed in the document and how Spark features contributed
Caution	<b>FIXME</b> How do predicate pushdown and data locality / topology awareness work?

## Further reading or watching

- [Announcing the New Couchbase Spark Connector](#).
- [Why Spark and NoSQL?](#)

# 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](#)

## Table of Contents

- [Spark SQL](#)
  - [DataFrame](#)
  - [Creating DataFrames](#)
  - [Handling data in Avro format](#)
  - [Group and aggregate](#)
  - [More examples](#)
  - [Further reading or watching](#)

# Spark SQL

**Spark SQL** is a distributed SQL framework for loading, querying and persisting structured and semi-structured data in Apache Spark. It is de facto the primary and feature-rich interface to Spark to start with (hiding Spark Core's RDDs behind higher-level abstractions).

Spark SQL offers a standard ANSI SQL support with subqueries and the higher-level [Dataset API](#) on top of Spark Core's distributed computation model (based on RDD).

It offers performance optimizations using [Catalyst query optimizer](#) and [Tungsten execution engine](#).

Tip	Visit <a href="#">Spark SQL</a> home page to learn more.
-----	--

The following snippet shows a batch ETL pipeline to process JSON files and saving their subset as CSVs.

```
spark.read  
  .format("json")  
  .load("input-json")  
  .select("field1", "field2")  
  .where(field2 > 15)  
  .write  
  .format("csv")  
  .save("output-csv")
```

With [Structured Streaming](#) feature however, the above static batch query becomes dynamic and continuous paving the way for **continuous applications**.

```
spark.readStream
  .format("json")
  .load("input-json")
  .select("field1", "field2")
  .where(field2 > 15)
  .writeStream
  .format("console")
  .start
```

As of Spark 2.0, the main data abstraction of Spark SQL is [Dataset](#). It represents a **structured data** which is records with a known schema. This structured data representation [Dataset](#) enables [compact binary representation](#) using compressed columnar format that is stored in managed objects outside JVM's heap. It is supposed to speed computations up by reducing memory usage and GCs.

Spark SQL supports [predicate pushdown](#) to optimize performance of Dataset queries and can also [generate optimized code at runtime](#).

Spark SQL comes with the different APIs to work with:

1. [Dataset API](#) (formerly [DataFrame API](#)) with a strongly-typed LINQ-like Query DSL that Scala programmers will likely find very appealing to use.
2. [Structured Streaming API \(aka Streaming Datasets\)](#) for continuous incremental execution of structured queries.
3. Non-programmers will likely use SQL as their query language through direct integration with Hive
4. JDBC/ODBC fans can use JDBC interface (through Thrift JDBC/ODBC server) and connect their tools to Spark's distributed query engine.

Spark SQL comes with a uniform interface for data access in distributed storage systems like Cassandra or HDFS (Hive, Parquet, JSON) using specialized [DataFrameReader](#) and [DataFrameWriter](#) objects.

Spark SQL allows you to execute SQL-like queries on large volume of data that can live in Hadoop HDFS or Hadoop-compatible file systems like S3. It can access data from different data sources - files or tables.

Spark SQL defines three types of functions:

- [Built-in functions](#) or [User-Defined Functions \(UDFs\)](#) that take values from a single row as input to generate a single return value for every input row.
- [Aggregate functions](#) that operate on a group of rows and calculate a single return value per group.

- [Windowed Aggregates \(Windows\)](#) that operate on a group of rows and calculate a single return value for each row in a group.

There are two supported **catalog** implementations — `in-memory` (default) and `hive` — that you can set using [spark.sql.catalogImplementation](#) setting.

## 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.

Note

Found the following note about Apache Drill, but appears to apply to Spark SQL perfectly:

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.

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.

You can parse data from external data sources and let the *schema inferencer* to deduct the schema.

## 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")

spark.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?

## 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://dl.bintray.com/databricks/maven"

```

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 = spark.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")
```

## Further reading or watching

1. (video) [Spark's Role in the Big Data Ecosystem - Matei Zaharia](#)
2. [Introducing Apache Spark 2.0](#)

# Logical Query Plan Analyzer

`Analyzer` is a [logical query plan](#) analyzer. It is available through [analyzer attribute](#) of the current `SparkSession`.

```
sparkSession.sessionState.analyzer
```

Note	<code>sessionState</code> attribute in <code>SparkSession</code> is a <code>private[sql]</code> value so to access it your code has to be in <code>org.apache.spark.sql</code> package.
------	---

`Analyzer` is a [RuleExecutor](#) with `CheckAnalysis` and defines `Substitution`, `Resolution`, `Nondeterministic`, `UDF`, `FixNullability`, and `Cleanup batches`.

`Analyzer` uses a [SessionCatalog](#), a `CatalystConf`, and a configurable number of iterations (as `maxIterations`).

`Analyzer` defines `extendedResolutionRules` attribute being a collection of rules (that process a `LogicalPlan`) as an extension point that a custom `Analyzer` can use to extend the `Resolution` batch. The collection of rules is added at the end of the `Resolution` batch.

You can access the result of executing `Analyzer` against the [logical plan](#) of a [Dataset](#) using [explain](#) method or [QueryExecution](#):

```

val dataset = spark.range(5).withColumn("new_column", 'id + 5 as "plus5")

scala> dataset.explain(extended = true)
== Parsed Logical Plan ==
'Project [* , ('id + 5) AS plus5#148 AS new_column#149]
+- Range (0, 5, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint, new_column: bigint
Project [id#145L, (id#145L + cast(5 as bigint)) AS new_column#149L]
+- Range (0, 5, step=1, splits=Some(8))

== Optimized Logical Plan ==
Project [id#145L, (id#145L + 5) AS new_column#149L]
+- Range (0, 5, step=1, splits=Some(8))

== Physical Plan ==
*Project [id#145L, (id#145L + 5) AS new_column#149L]
+- *Range (0, 5, step=1, splits=Some(8))

scala> dataset.queryExecution.analyzed
res14: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Project [id#145L, (id#145L + cast(5 as bigint)) AS new_column#149L]
+- Range (0, 5, step=1, splits=Some(8))

```

**Tip**

Enable `TRACE` or `DEBUG` logging level for `org.apache.spark.sql.hive.HiveSessionState$$anon$1` (when [Hive support is enabled](#)) or `org.apache.spark.sql.internal.SessionState$$anon$1` logger to see what happens inside `Analyzer`.

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

```

# when Hive support is enabled
log4j.logger.org.apache.spark.sql.hive.HiveSessionState$$anon$1=TRACE

# with no Hive support
log4j.logger.org.apache.spark.sql.internal.SessionState$$anon$1=TRACE

```

Refer to [Logging](#).

The reason for such a weird-looking logger name is that `analyzer` attribute is created as an anonymous subclass of `Analyzer` class.

## RuleExecutor—Abstract Rule Executor

`RuleExecutor` can [apply](#) collection of rules (as `batches`) to a tree.

Scala-wise, `RuleExecutor` is an abstract type generator parameterized by the `TreeType` type parameter.

```
abstract class RuleExecutor[TreeType <: TreeNode[_]]
```

`RuleExecutor` defines the protected `batches` method that implementations are supposed to define with the collection of `Batch` instances to `execute`.

```
protected def batches: Seq[Batch]
```

## Applying Rules to Tree (execute method)

```
execute(plan: TreeType): TreeType
```

`execute` iterates over `batches` and applies rules sequentially to the input `plan`.

It tracks the number of iterations and the time of executing each rule (with a plan).

When a rule changes a plan, you should see the following TRACE message in the logs:

```
TRACE HiveSessionState$$anon$1:  
==== Applying Rule [ruleName] ====  
[currentAndModifiedPlansSideBySide]
```

After the number of iterations has reached the number of iterations for the batch's `strategy` it stops execution and prints out the following WARN message to the logs:

```
WARN HiveSessionState$$anon$1: Max iterations ([iteration]) reached for batch [batchName]
```

When the plan has not changed (after applying rules), you should see the following TRACE message in the logs and `execute` moves on to applying the rules in the next batch. The moment is called **fixed point** (i.e. when the execution **converges**).

```
TRACE HiveSessionState$$anon$1: Fixed point reached for batch [batchName] after [iteration] iterations.
```

After the batch finishes, if the plan has been changed by the rules, you should see the following DEBUG message in the logs:

```
DEBUG HiveSessionState$$anon$1:  
==== Result of Batch [batchName] ===  
[currentAndModifiedPlansSideBySide]
```

Otherwise, when the rules had no changes to a plan, you should see the following TRACE message in the logs:

```
TRACE HiveSessionState$$anon$1: Batch [batchName] has no effect.
```

## Batch

A **batch** is a named collection of rules with a strategy, e.g.

```
Batch("Substitution", fixedPoint,  
      CTESubstitution,  
      WindowsSubstitution,  
      EliminateUnions,  
      new SubstituteUnresolvedOrdinals(conf)),
```

A `strategy` can be `Once` or `FixedPoint` (with a number of iterations).

Note

`Once` `strategy` is a `FixedPoint` `strategy` with one iteration.

# SparkSession — The Entry Point

`SparkSession` is the entry point to developing Spark SQL applications using the fully-typed `Dataset` (and untyped `DataFrame`) data abstractions.

Note

`SparkSession` has replaced `SQLContext` as of Spark **2.0.0**.

You should use the `builder` method to create an instance of `sparkSession`.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = SparkSession.builder
  .master("local[*]")
  .appName("My Spark Application")
  .getOrCreate
```

You can have multiple `SparkSession`s in a single Spark application.

Internally, `sparkSession` requires a `SparkContext` and an optional `SharedState` (that represents the shared state across `SparkSession` instances).

## Accessing Builder for SparkSession (builder method)

```
builder(): Builder
```

`builder` method creates a new `Builder` that you can use to build a fully-configured `SparkSession` using a *fluent API*.

```
import org.apache.spark.sql.SparkSession
val builder = SparkSession.builder
```

Tip

Read about [Fluent interface](#) design pattern in Wikipedia, the free encyclopedia.

## Implicits — `SparkSession.implicits`

The `implicits` object is a helper class with the methods to convert Scala objects to `Datasets` and `DataFrames`. It defines `Encoders` for Scala's "primitive" types and their products and collections.

Import the implicits by `import spark.implicits._`.

**Note**

```
val spark = SparkSession.builder.getOrCreate()
import spark.implicits._
```

It holds [Encoders](#) for Scala's "primitive" types like `Int`, `Double`, `String`, and their products and collections.

It offers support for creating `Dataset` from `RDD` of any type (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 "primitive" types but `Int`, `Long`, and `String`.

## readStream

```
readStream: DataStreamReader
```

`readStream` returns a new [DataStreamReader](#).

## Creating Empty Dataset (`emptyDataset` method)

```
emptyDataset[T: Encoder]: Dataset[T]
```

`emptyDataset` creates an empty [Dataset](#) (assuming that future records being of type `T`).

```
scala> val strings = spark.emptyDataset[String]
strings: org.apache.spark.sql.Dataset[String] = [value: string]

scala> strings.printSchema
root
 |-- value: string (nullable = true)
```

The [LogicalPlan](#) is `LocalRelation`.

## Creating Dataset from Collections and RDDs (createDataset methods)

```
createDataset[T : Encoder](data: Seq[T]): Dataset[T]
createDataset[T : Encoder](data: RDD[T]): Dataset[T]
```

`createDataset` creates a [Dataset](#) from a local collection or distributed [RDD](#).

```
scala> val ns = 1 to 5
ns: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> val nums = spark.createDataset(ns)
nums: org.apache.spark.sql.Dataset[Int] = [value: int]

scala> nums.show
+---+
|value|
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
```

The [LogicalPlan](#) is [LocalRelation](#) (for the input `data` collection) or [LogicalRDD](#) (for the input `RDD[T]`).

## Creating Dataset With Single Long Column (range methods)

```
range(end: Long): Dataset[java.lang.Long]
range(start: Long, end: Long): Dataset[java.lang.Long]
range(start: Long, end: Long, step: Long): Dataset[java.lang.Long]
range(start: Long, end: Long, step: Long, numPartitions: Int): Dataset[java.lang.Long]
```

`range` family of methods create a [Dataset](#) of `Long` numbers.

```
scala> spark.range(start = 0, end = 4, step = 2, numPartitions = 5).show
+---+
| id|
+---+
| 0 |
| 2 |
+---+
```

**Note**

The three first variants (that do not specify `numPartitions` explicitly) use `SparkContext.defaultParallelism` for the number of partitions `numPartitions`.

Internally, `range` creates a new `Dataset[Long]` with `Range` logical plan and `Encoders.LONG encoder`.

## emptyDataFrame

```
emptyDataFrame: DataFrame
```

`emptyDataFrame` creates an empty `DataFrame` (with no rows and columns).

It calls `createDataFrame` with an empty `RDD[Row]` and an empty schema `StructType(Nil)`.

## createDataFrame method

```
createDataFrame(rowRDD: RDD[Row], schema: StructType): DataFrame
```

`createDataFrame` creates a `DataFrame` using `RDD[Row]` and the input `schema`. It is assumed that the rows in `rowRDD` all match the `schema`.

## streams Attribute

```
streams: StreamingQueryManager
```

`streams` attribute gives access to `StreamingQueryManager` (through `SessionState`).

```
val spark: SparkSession = ...
spark.streams.active.foreach(println)
```

## udf Attribute

```
udf: UDFRegistration
```

`udf` attribute gives access to `UDFRegistration` that allows registering `user-defined functions` for SQL queries.

```

val spark: SparkSession = ...
spark.udf.register("myUpper", (s: String) => s.toUpperCase)

val strs = ('a' to 'c').map(_.toString).toDS
strs.registerTempTable("strs")

scala> sql("select myUpper(value) from strs").show
+-----+
|UDF(value)|
+-----+
|      A|
|      B|
|      C|
+-----+

```

Internally, it uses `SessionState.udf`.

## catalog Attribute

`catalog` attribute is an interface to the current `catalog` (of databases, tables, functions, table columns, and temporary views).

```

scala> spark.catalog.listTables.show
+-----+-----+-----+-----+
|       name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|my_permanent_table| default|      null|  MANAGED|    false|
|      strs|    null|      null|TEMPORARY|     true|
+-----+-----+-----+-----+

```

## table method

```
table(tableName: String): DataFrame
```

`table` creates a `DataFrame` from records in the `tableName` table (if exists).

```
val df = spark.table("mytable")
```

## streamingQueryManager Attribute

```
streamingQueryManager is...
```

## listenerManager Attribute

```
listenerManager is...
```

## ExecutionListenerManager

```
ExecutionListenerManager is...
```

## functionRegistry Attribute

```
functionRegistry is...
```

## experimentalMethods Attribute

```
experimental: ExperimentalMethods
```

`experimentalMethods` is an extension point with `ExperimentalMethods` that is a per-session collection of extra strategies and `Rule[LogicalPlan]`s.

Note

`experimental` is used in [SparkPlanner](#) and [SparkOptimizer](#). Hive and Structured Streaming use it for extra strategies and optimizations.

## newSession method

```
newSession(): SparkSession
```

`newSession` creates (starts) a new `sparkSession` (with the current [SparkContext](#) and [SharedState](#)).

```
scala> println(sc.version)
2.0.0-SNAPSHOT

scala> val newSession = spark.newSession
newSession: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@122f
58a
```

## sharedState Attribute

`sharedState` is the current [SharedState](#). It is created lazily when first accessed.

## SharedState

`SharedState` is an internal class that holds the shared state across active SQL sessions (as `SparkSession` instances) by sharing `CacheManager`, `SQLListener`, and `ExternalCatalog`.

`SharedState` requires a `SparkContext` when being created. It also adds `hive-site.xml` to Hadoop's configuration in the current `SparkContext` if found on CLASSPATH.

The fully-qualified class name is `org.apache.spark.sql.internal.SharedState`.

`SharedState` is created lazily, i.e. when first accessed after `sparkSession` is created. It can happen when a new session is created or when the shared services are accessed. It is created with a `SparkContext`.

Caution	<a href="#">FIXME</a> Describe <code>hive.metastore.warehouse.dir</code>
---------	--

## Creating SparkSession Instance

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Creating Datasets (createDataset methods)

```
createDataset[T: Encoder](data: Seq[T]): Dataset[T]
createDataset[T: Encoder](data: RDD[T]): Dataset[T]

// For Java
createDataset[T: Encoder](data: java.util.List[T]): Dataset[T]
```

`createDataset` is an experimental API to create a `Dataset` from a local Scala collection, i.e. `Seq[T]` or Java's `List[T]`, or an `RDD[T]`.

```
val ints = spark.createDataset(0 to 9)
```

Note	You'd rather not be using <code>createDataset</code> since you have the <a href="#">Scala implicits and toDS method</a> .
------	---

## Accessing DataFrameReader (read method)

```
read: DataFrameReader
```

`read` method returns a `DataFrameReader` that is used to read data from external storage systems and load it into a `DataFrame`.

```
val spark: SparkSession = // create instance
val dfReader: DataFrameReader = spark.read
```

## Runtime Configuration (conf attribute)

```
conf: RuntimeConfig
```

`conf` returns the current runtime configuration (as `RuntimeConfig`) that wraps `SQLConf`.

Caution

FIXME

## sessionState

`sessionState` is a transient lazy value that represents the current `SessionState`.

Note

`sessionState` is a `private[sql]` value so you can only access it in a code inside `org.apache.spark.sql` package.

`sessionState` is a lazily-created value based on the internal `spark.sql.catalogImplementation` setting that can be:

- `org.apache.spark.sql.hive.HiveSessionState` for `hive`
- `org.apache.spark.sql.internal.SessionState` for `in-memory`

## Executing SQL (sql method)

```
sql(sqlText: String): DataFrame
```

`sql` executes the `sqlText` SQL statement.

```
scala> sql("SHOW TABLES")
res0: org.apache.spark.sql.DataFrame = [tableName: string, isTemporary: boolean]

scala> sql("DROP TABLE IF EXISTS testData")
res1: org.apache.spark.sql.DataFrame = []

// Let's create a table to SHOW it
spark.range(10).write.option("path", "/tmp/test").saveAsTable("testData")

scala> sql("SHOW TABLES").show
+-----+-----+
|tableName|isTemporary|
+-----+-----+
| testdata|      false|
+-----+-----+
```

Internally, it creates a [Dataset](#) using the current `SparkSession` and a [logical plan](#). The plan is created by parsing the input `sqlText` using `sessionState.sqlParser`.

Caution	<a href="#">FIXME See Executing SQL Queries.</a>
---------	--

# Builder — Building SparkSession with Fluent API

`Builder` is the fluent API to build a fully-configured `SparkSession`.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = SparkSession.builder
  .master("local[*]")
  .appName("My Spark Application")
  .getOrCreate
```

You can use the fluent design pattern to set the various properties of a `SparkSession` that opens a session to Spark SQL.

Note

You can have multiple `sparkSession`s in a single Spark application.

## Enabling Hive Support (`enableHiveSupport` method)

When creating a `SparkSession`, you can enable Hive support using `enableHiveSupport` method.

```
enableHiveSupport(): Builder
```

Caution

**FIXME** What exactly do you get from it?

# CacheManager

## Caching Logical Plan (cacheQuery method)

When you `cache` or `persist` a `Dataset`, they pass the call to `cacheQuery` method.

```
cacheQuery(  
    query: Dataset[_],  
    tableName: Option[String] = None,  
    storageLevel: StorageLevel = MEMORY_AND_DISK): Unit
```

`cacheQuery` obtains `analyzed` logical plan and saves it as a `InMemoryRelation` in the internal `cachedData` cached queries collection.

If however the query has already been cached, you should instead see the following WARN message in the logs:

```
WARN CacheManager: Asked to cache already cached data.
```

# Caching

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

You can use `CACHE TABLE [tableName]` to cache `tableName` table in memory. It is an eager operation which is executed as soon as the statement is executed.

```
sql("CACHE TABLE [tableName]")
```

You could use `LAZY` keyword to make caching lazy.

# Debugging Query Execution

`debug` package object contains tools for **debugging query execution** that you can use to do the full analysis of your [structured queries](#) (i.e. `Datasets` ).

Note

Let's make it clear — they are methods, *my dear*.

The methods are in `org.apache.spark.sql.execution.debug` package and work on your `Datasets` and [SparkSession](#).

Caution

[FIXME](#) Expand on the `sparkSession` part.

```
debug()  
debugCodegen()
```

Import the package and do the full analysis using `debug` method.

```
import org.apache.spark.sql.execution.debug._  
  
scala> spark.range(10).where('id === 4).debug  
Results returned: 1  
== WholeStageCodegen ==  
Tuples output: 1  
  id LongType: {java.lang.Long}  
== Filter (id#25L = 4) ==  
Tuples output: 0  
  id LongType: {}  
== Range (0, 10, splits=8) ==  
Tuples output: 0  
  id LongType: {}
```

You can also perform `debugCodegen` .

```

import org.apache.spark.sql.execution.debug._

scala> spark.range(10).where('id === 4).debugCodegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Filter (id#29L = 4)
+- *Range (0, 10, splits=8)

Generated code:
/* 001 */ public Object generate(Object[] references) {
/* 002 */     return new GeneratedIterator(references);
/* 003 */ }
/* 004 */
/* 005 */ final class GeneratedIterator extends org.apache.spark.sql.execution.BufferedRowIterator {
/* 006 */     private Object[] references;
...

```

```

scala> spark.range(1, 1000).select('id+1+2+3, 'id+4+5+6).queryExecution.debug.codegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Project [(id#33L + 6) AS (((id + 1) + 2) + 3)#36L, (id#33L + 15) AS (((id + 4) + 5) +
6)#37L]
+- *Range (1, 1000, splits=8)

Generated code:
/* 001 */ public Object generate(Object[] references) {
/* 002 */     return new GeneratedIterator(references);
/* 003 */ }
/* 004 */
/* 005 */ final class GeneratedIterator extends org.apache.spark.sql.execution.BufferedRowIterator {
/* 006 */     private Object[] references;
...

```

# Dataset

**Dataset** is the Spark SQL API for working with structured data, i.e. records with a known schema.

Datasets are "lazy" and structured query expressions are only triggered when an action is invoked. Internally, a `Dataset` represents a [logical plan](#) that describes the computation query required to produce the data (for a given [Spark SQL session](#)).

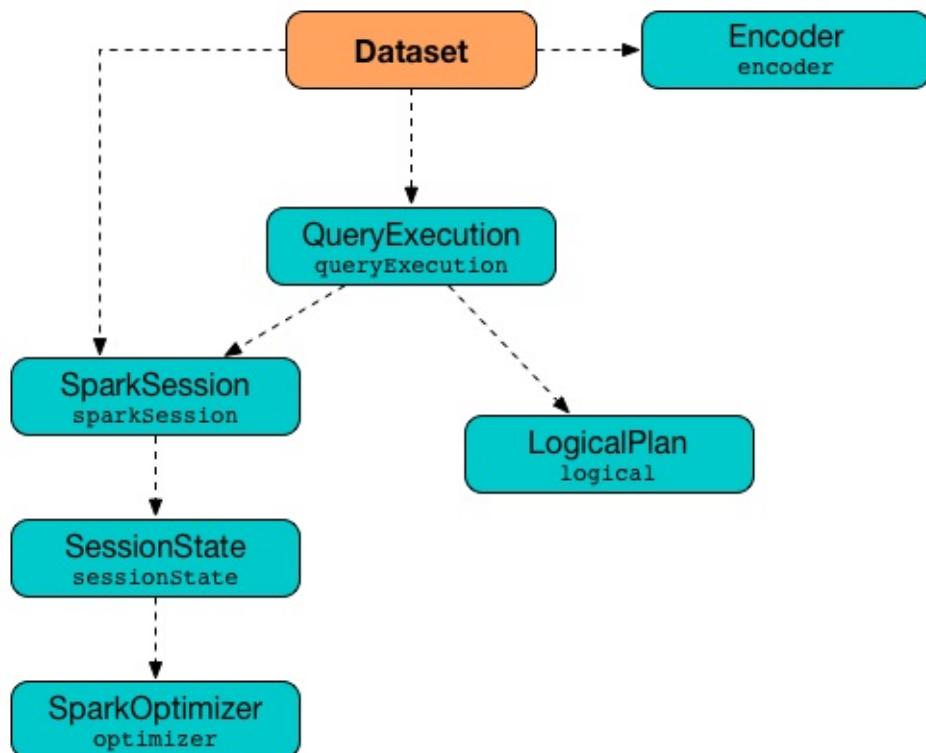


Figure 1. Dataset's Internals

A Dataset is a result of executing a query expression against data storage like files or databases. The structured query expression can be described by a SQL query, a Column-based SQL expression or a Scala/Java lambda function. And that is why Dataset operations are available in three variants.

```
scala> val dataset = (0 to 4).toDS
dataset: org.apache.spark.sql.Dataset[Int] = [value: int]

// Variant 1: filter operator accepts a Scala function
dataset.filter(n => n % 2 == 0).count

// Variant 2: filter operator accepts a Column-based SQL expression
dataset.filter('value % 2 === 0).count

// Variant 3: filter operator accepts a SQL query
dataset.filter("value % 2 = 0").count
```

The Dataset API offers declarative and type-safe operators that makes for an improved experience for data processing (comparing to [DataFrames](#) that were a set of index- or column name-based [Rows](#)).

Note

`Dataset` was first introduced in Apache Spark **1.6.0** as an experimental feature, and has since turned itself into a fully supported API.

As of Spark **2.0.0**, [DataFrame](#) - the flagship data abstraction of previous versions of Spark SQL - is currently a *mere* type alias for `Dataset[Row]` :

```
type DataFrame = Dataset[Row]
```

See [package object sql](#).

`Dataset` offers convenience of RDDs with the performance optimizations of DataFrames and the strong static type-safety of Scala. The last feature of bringing the strong type-safety to [DataFrame](#) makes Dataset so appealing. All the features together give you a more functional programming interface to work with structured data.

```

scala> spark.range(1).filter('id === 0).explain(true)
== Parsed Logical Plan ==
'Filter ('id = 0)
+- Range (0, 1, splits=8)

== Analyzed Logical Plan ==
id: bigint
Filter (id#51L = cast(0 as bigint))
+- Range (0, 1, splits=8)

== Optimized Logical Plan ==
Filter (id#51L = 0)
+- Range (0, 1, splits=8)

== Physical Plan ==
*Filter (id#51L = 0)
+- *Range (0, 1, splits=8)

scala> spark.range(1).filter(_ == 0).explain(true)
== Parsed Logical Plan ==
'TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], un
resolveddeserializer(newInstance(class java.lang.Long))
+- Range (0, 1, splits=8)

== Analyzed Logical Plan ==
id: bigint
TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], new
Instance(class java.lang.Long)
+- Range (0, 1, splits=8)

== Optimized Logical Plan ==
TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], new
Instance(class java.lang.Long)
+- Range (0, 1, splits=8)

== Physical Plan ==
*Filter <function1>.apply
+- *Range (0, 1, splits=8)

```

It is only with Datasets to have syntax and analysis checks at compile time (that was not possible using [DataFrame](#), regular SQL queries or even RDDs).

Using `Dataset` objects turns `DataFrames` of `Row` instances into a `DataFrames` of case classes with proper names and types (following their equivalents in the case classes). Instead of using indices to access respective fields in a DataFrame and cast it to a type, all this is automatically handled by Datasets and checked by the Scala compiler.

Datasets use [Catalyst Query Optimizer](#) and [Tungsten](#) to optimize query performance.

A `Dataset` object requires a [SparkSession](#), a [QueryExecution](#) plan, and an [Encoder](#). If however a [LogicalPlan](#) is used to [create a `Dataset`](#), the logical plan is first [executed](#) (using the current [SessionState](#) in the `SparkSession`) that yields the [QueryExecution](#) plan.

A `Dataset` is [Queryable](#) and [Serializable](#), i.e. can be saved to a persistent storage.

**Note** `SparkSession` and `QueryExecution` are transient attributes of a `Dataset` and therefore do not participate in `Dataset` serialization. The only *firmly-tied* feature of a `Dataset` is the [Encoder](#).

It also has a [schema](#).

You can convert a type-safe `Dataset` to a "untyped" `DataFrame` (see [Type Conversions to `Dataset\[T\]`](#)) or access the `RDD` that sits underneath (see [Converting Datasets into RDDs \(using rdd method\)](#)). It is supposed to give you a more pleasant experience while transitioning from the legacy `RDD`-based or `DataFrame`-based APIs you may have used in the earlier versions of Spark SQL or encourage migrating from Spark Core's `RDD` API to Spark SQL's `Dataset` API.

The default storage level for `Datasets` is [MEMORY\\_AND\\_DISK](#) because recomputing the in-memory columnar representation of the underlying table is expensive. See [Persisting Dataset \(persist method\)](#) in this document.

Spark 2.0 has introduced a new query model called [Structured Streaming](#) for continuous incremental execution of structured queries. That made possible to consider `Datasets` a static and bounded as well as streaming and unbounded data sets with a single unified API for different execution models.

## queryExecution Attribute

`queryExecution` is a required parameter of a `Dataset`.

```
val dataset: Dataset[Int] = ...
dataset.queryExecution
```

It is a part of the Developer API of the `Dataset` class.

## Creating Datasets

If [LogicalPlan](#) is used to [create a `Dataset`](#), it is [executed](#) (using the current [SessionState](#)) to create a corresponding [QueryExecution](#).

## Accessing DataFrameWriter (write method)

```
write: DataFrameWriter[T]
```

`write` method returns [DataFrameWriter](#) for records of type `T`.

```
import org.apache.spark.sql.{DataFrameWriter, Dataset}
val ints: Dataset[Int] = (0 to 5).toDS

val writer: DataFrameWriter[Int] = ints.write
```

## Accessing DataStreamWriter (writeStream method)

```
writeStream: DataStreamWriter[T]
```

`writeStream` method returns [DataStreamWriter](#) for records of type `T`.

```
val papers = spark.readStream.text("papers").as[String]

import org.apache.spark.sql.streaming.DataStreamWriter
val writer: DataStreamWriter[String] = papers.writeStream
```

## Display Records (show methods)

Caution	<a href="#">FIXME</a>
---------	-----------------------

Internally, `show` relays to a private `showString` to do the formatting. It turns the `Dataset` into a `DataFrame` (by calling `toDF()`) and [takes first n records](#).

## Taking First n Records (take method)

```
take(n: Int): Array[T]
```

`take` is an action on a `Dataset` that returns a collection of `n` records.

Warning	<code>take</code> loads all the data into the memory of the Spark application's driver process and for a large <code>n</code> could result in <code>OutOfMemoryError</code> .
---------	---

Internally, `take` creates a new `Dataset` with `Limit` logical plan for `Literal` expression and the current `LogicalPlan`. It then runs the [SparkPlan](#) that produces a `Array[InternalRow]` that is in turn decoded to `Array[T]` using a bounded [encoder](#).

## join

Caution

FIXME

## where

Caution

FIXME

## groupBy

Caution

FIXME

## foreachPartition method

```
foreachPartition(f: Iterator[T] => Unit): Unit
```

`foreachPartition` applies the `f` function to each partition of the `Dataset`.

```
case class Record(id: Int, city: String)
val ds = Seq(Record(0, "Warsaw"), Record(1, "London")).toDS

ds.foreachPartition { iter: Iterator[Record] => iter.foreach(println) }
```

Note

`foreachPartition` is used to [save a `DataFrame` to a JDBC table](#) (indirectly through `JdbcUtils.saveTable`) and [ForeachSink](#).

## mapPartitions method

```
mapPartitions[U: Encoder](func: Iterator[T] => Iterator[U]): Dataset[U]
```

`mapPartitions` returns a new `Dataset` (of type `U`) with the function `func` applied to each partition.

Caution

FIXME Example

## Creating Zero or More Records (flatMap method)

```
flatMap[U: Encoder](func: T => TraversableOnce[U]): Dataset[U]
```

`flatMap` returns a new `Dataset` (of type `U`) with all records (of type `T`) mapped over using the function `func` and then flattening the results.

**Note**

`flatMap` can create new records. It deprecated `explode`.

```
final case class Sentence(id: Long, text: String)
val sentences = Seq(Sentence(0, "hello world"), Sentence(1, "witaj swiecie")).toDS

scala> sentences.flatMap(s => s.text.split("\\s+")).show
+-----+
| value|
+-----+
| hello|
| world|
| witaj|
| swiecie|
+-----+
```

Internally, `flatMap` calls `mapPartitions` with the partitions `flatMap(ped)`.

## Caching Dataset (cache method)

```
cache(): this.type
```

`cache` merely passes the calls to no-argument `persist` method.

## Persisting Dataset (persist method)

```
persist(): this.type
persist(newLevel: StorageLevel): this.type
```

`persist` caches the `Dataset` using the default storage level `MEMORY_AND_DISK` or `newLevel`.

Internally, it requests the `CacheManager` to cache the query (that is accessible through `SharedState` of the current `SparkSession`).

## Unpersisting Dataset (unpersist method)

```
unpersist(blocking: Boolean): this.type
```

`unpersist` uncache the `Dataset` possibly by `blocking` the call.

Internally, it requests the `CacheManager` to uncache the query.

## Repartitioning Dataset (repartition method)

```
repartition(numPartitions: Int): Dataset[T]
```

`repartition` repartition the `Dataset` to exactly `numPartitions` partitions.

## Features of Dataset API

The features of the Dataset API in Spark SQL:

- **Type-safety** as Datasets are Scala domain objects and operations operate on their attributes. All is checked by the Scala compiler at compile time.

## Type Conversions to Dataset[T] (and DataFrame) (toDS and toDF methods)

`DatasetHolder` case class offers three methods that do the conversions from `Seq[T]` or `RDD[T]` type to `Dataset[T]`:

- `toDS(): Dataset[T]`
- `toDF(): DataFrame`
- `toDF(colNames: String*): DataFrame`

Note	<code>DataFrame</code> is a <i>mere</i> type alias for <code>Dataset[Row]</code> since Spark 2.0.0.
------	---

`DatasetHolder` is used by `SQLImplicits` that is available to use after [importing implicits object of SQLContext](#).

```
scala> val ds = Seq("I am a shiny Dataset!").toDS
ds: org.apache.spark.sql.Dataset[String] = [value: string]

scala> val df = Seq("I am an old grumpy DataFrame!").toDF
df: org.apache.spark.sql.DataFrame = [value: string]

scala> val df = Seq("I am an old grumpy DataFrame!").toDF("text")
df: org.apache.spark.sql.DataFrame = [text: string]

scala> val ds = sc.parallelize(Seq("hello")).toDS
ds: org.apache.spark.sql.Dataset[String] = [value: string]
```

**Note**

This import is automatically executed in [Spark Shell](#).

```
scala> sc.version
res11: String = 2.0.0-SNAPSHOT

scala> :imports
1) import spark.implicits._  (59 terms, 38 are implicit)
2) import spark.sql          (1 terms)
```

```
import spark.implicits._

case class Token(name: String, productId: Int, score: Double)
val data = Seq(
  Token("aaa", 100, 0.12),
  Token("aaa", 200, 0.29),
  Token("bbb", 200, 0.53),
  Token("bbb", 300, 0.42))

// Transform data to a Dataset[Token]
// It doesn't work with type annotation yet
// https://issues.apache.org/jira/browse/SPARK-13456
val ds: Dataset[Token] = data.toDS

// Transform data into a DataFrame with no explicit schema
val df = data.toDF

// Transform DataFrame into a Dataset
val ds = df.as[Token]

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)

// In DataFrames we work with Row instances
scala> df.map(_.getClass.getName).show(false)
+-----+
|value|
+-----+
|org.apache.spark.sql.catalyst.expressions.GenericRowWithSchema|
```

```
|org.apache.spark.sql.catalyst.expressions.GenericRowWithSchema|
|org.apache.spark.sql.catalyst.expressions.GenericRowWithSchema|
|org.apache.spark.sql.catalyst.expressions.GenericRowWithSchema|
+-----+  
  
// In Datasets we work with case class instances  
scala> ds.map(_.getClass.getName).show(false)
+-----+
|value          |  
+-----+
|$line40.$read$$iw$$iw$Token|
|$line40.$read$$iw$$iw$Token|
|$line40.$read$$iw$$iw$Token|
|$line40.$read$$iw$$iw$Token|
+-----+  
  
scala> ds.map(_.name).show
+---+
|value|
+---+
| aaa|
| aaa|
| bbb|
| bbb|
+---+
```

## Converting Datasets into RDDs (using rdd method)

Whenever you are in need to convert a `Dataset` into a `RDD`, executing `rdd` method gives you a RDD of the proper input object type (not [Row as in DataFrames](#)).

```
scala> val rdd = tokens.rdd
rdd: org.apache.spark.rdd.RDD[Token] = MapPartitionsRDD[11] at rdd at <console>:30
```

## Schema

A `dataset` has a **schema**.

```
schema: StructType
```

Tip	<p>You may also use the following methods to learn about the schema:</p> <ul style="list-style-type: none"> <li>• <code>printSchema(): Unit</code></li> <li>• <a href="#">explain</a></li> </ul>
-----	--

## Supported Types

Caution

[FIXME](#) What types are supported by Encoders

## toJSON

`toJSON` maps the content of `Dataset` to a `Dataset` of JSON strings.

Note

A new feature in Spark 2.0.0.

```
scala> val ds = Seq("hello", "world", "foo bar").toDS
ds: org.apache.spark.sql.Dataset[String] = [value: string]

scala> ds.toJSON.show
+-----+
|      value|
+-----+
| {"value":"hello"}|
| {"value":"world"}|
| {"value":"foo bar"}|
+-----+
```

## explain

```
explain(): Unit
explain(extended: Boolean): Unit
```

`explain` prints the [logical](#) and physical plans to the console. You can use it for debugging.

Tip

If you are serious about query debugging you could also use the [Debugging Query Execution facility](#).

Internally, `explain` executes a `ExplainCommand` logical plan on [the logical plan of the QueryExecution of the Dataset](#).

```
scala> spark.range(10).explain(extended = true)
== Parsed Logical Plan ==
Range (0, 10, splits=8)

== Analyzed Logical Plan ==
id: bigint
Range (0, 10, splits=8)

== Optimized Logical Plan ==
Range (0, 10, splits=8)

== Physical Plan ==
*Range (0, 10, splits=8)
```

## select

```
select[U1: Encoder](c1: TypedColumn[T, U1]): Dataset[U1]
select[U1, U2](c1: TypedColumn[T, U1], c2: TypedColumn[T, U2]): Dataset[(U1, U2)]
select[U1, U2, U3](
  c1: TypedColumn[T, U1],
  c2: TypedColumn[T, U2],
  c3: TypedColumn[T, U3]): Dataset[(U1, U2, U3)]
select[U1, U2, U3, U4](
  c1: TypedColumn[T, U1],
  c2: TypedColumn[T, U2],
  c3: TypedColumn[T, U3],
  c4: TypedColumn[T, U4]): Dataset[(U1, U2, U3, U4)]
select[U1, U2, U3, U4, U5](
  c1: TypedColumn[T, U1],
  c2: TypedColumn[T, U2],
  c3: TypedColumn[T, U3],
  c4: TypedColumn[T, U4],
  c5: TypedColumn[T, U5]): Dataset[(U1, U2, U3, U4, U5)]
```

Caution	<a href="#">FIXME</a>
---------	-----------------------

## filter

Caution	<a href="#">FIXME</a>
---------	-----------------------

## selectExpr

```
selectExpr(exprs: String*): DataFrame
```

`selectExpr` is like `select`, but accepts SQL expressions `exprs`.

```
val ds = spark.range(5)

scala> ds.selectExpr("rand() as random").show
16/04/14 23:16:06 INFO HiveSqlParser: Parsing command: rand() as random
+-----+
|      random|
+-----+
| 0.887675894185651|
| 0.36766085091074086|
| 0.2700020856675186|
| 0.1489033635529543|
| 0.5862990791950973|
+-----+
```

Internally, it executes `select` with every expression in `exprs` mapped to [Column](#) (using [SparkSqlParser.parseExpression](#)).

```
scala> ds.select(expr("rand() as random")).show
+-----+
|      random|
+-----+
| 0.5514319279894851|
| 0.2876221510433741|
| 0.4599999092045741|
| 0.5708558868374893|
| 0.6223314406247136|
+-----+
```

Note	A new feature in Spark 2.0.0.
------	-------------------------------

## isStreaming

`isStreaming` returns `true` when `Dataset` contains [StreamingRelation](#) or [StreamingExecutionRelation](#) **streaming sources**.

Note	Streaming datasets are created using <a href="#">DataFrameReader.stream</a> method (for <a href="#">StreamingRelation</a> ) and contain <a href="#">StreamingExecutionRelation</a> after <a href="#">DataFrameWriter.startStream</a> .
------	--

```
val reader = spark.read
val helloStream = reader.stream("hello")

scala> helloStream.isStreaming
res9: Boolean = true
```

Note	A new feature in Spark <b>2.0.0</b> .
------	---------------------------------------

## randomSplit

```
randomSplit(weights: Array[Double]): Array[Dataset[T]]
randomSplit(weights: Array[Double], seed: Long): Array[Dataset[T]]
```

`randomSplit` randomly splits the `Dataset` per `weights`.

`weights` doubles should sum up to `1` and will be normalized if they do not.

You can define `seed` and if you don't, a random `seed` will be used.

Note	It is used in <a href="#">TrainValidationSplit</a> to split dataset into training and validation datasets.
------	--

```
val ds = spark.range(10)
scala> ds.randomSplit(Array[Double](2, 3)).foreach(_.show)
+---+
| id|
+---+
| 0|
| 1|
| 2|
+---+
+---+
| id|
+---+
| 3|
| 4|
| 5|
| 6|
| 7|
| 8|
| 9|
+---+
```

Note	A new feature in Spark <b>2.0.0</b> .
------	---------------------------------------

## Queryable

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Tracking Multi-Job SQL Query Executions (withNewExecutionId method)

```
withNewExecutionId[U](body: => U): U
```

`withNewExecutionId` is a `private[sql]` method that executes the input `body` action using `SQLExecution.withNewExecutionId` that sets the **execution id** local property set.

Note

It is used in `foreach`, `foreachPartition`, and (private) `collect`.

## Further reading or watching

- (video) [Structuring Spark: DataFrames, Datasets, and Streaming](#)

# Logical Query Plan

Caution

[FIXME](#)

**Logical Query Plan** is the base representation of a structured query expression (that makes for a [Dataset](#)).

It is modelled as the `LogicalPlan` abstract class which is a custom [QueryPlan](#).

```
// an example dataset to work with
val dataset = spark.range(1)

scala> val plan = dataset.queryExecution.logical
plan: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Range (0, 1, splits=8)
```

A logical plan can be **analyzed** which is to say that the plan (including children) has gone through analysis and verification. It can also be **resolved** to a specific schema.

```
scala> plan.analyzed
res1: Boolean = true

scala> plan.resolved
res2: Boolean = true
```

A logical plan knows the size of objects that are results of query operators, like `join`, through `statistics` object.

```
scala> val stats = plan.statistics
stats: org.apache.spark.sql.catalyst.plans.logical.Statistics = Statistics(8, false)
```

A logical plan knows the maximum number of records it can compute.

```
scala> val maxRows = plan.maxRows
maxRows: Option[Long] = None
```

## Specialized LogicalPlans

- `LeafNode`
- `UnaryNode`

- `BinaryNode`

## Join Logical Plan

`Join` is a `LogicalPlan` that acts on two `LogicalPlan` objects. It has a join type and an optional expression for the join.

The following is a list of join types:

- `INNER`
- `LEFT OUTER`
- `RIGHT OUTER`
- `FULL OUTER`
- `LEFT SEMI`
- `NATURAL`

## ExplainCommand Logical Plan

`ExplainCommand` logical plan allows users to see how a structured query will be executed.

`ExplainCommand` is used to implement [explain operator](#) and `EXPLAIN` SQL query (with `EXTENDED` and `CODEGEN` options).

```

scala> spark.catalog.listTables.explain
== Physical Plan ==
LocalTableScan [name#42, database#43, description#44, tableType#45, isTemporary#46]

scala> spark.catalog.listTables.explain(true)
== Parsed Logical Plan ==
LocalRelation [name#54, database#55, description#56, tableType#57, isTemporary#58]

== Analyzed Logical Plan ==
name: string, database: string, description: string, tableType: string, isTemporary: boolean
LocalRelation [name#54, database#55, description#56, tableType#57, isTemporary#58]

== Optimized Logical Plan ==
LocalRelation [name#54, database#55, description#56, tableType#57, isTemporary#58]

== Physical Plan ==
LocalTableScan [name#54, database#55, description#56, tableType#57, isTemporary#58]

// Explain in SQL

scala> sql("EXPLAIN EXTENDED show tables").show(false)
+-----+
| plan
+-----+
|== Parsed Logical Plan ==
|ShowTablesCommand
+-----+
|== Analyzed Logical Plan ==
|tableName: string, isTemporary: boolean
|ShowTablesCommand
+-----+
|== Optimized Logical Plan ==
|ShowTablesCommand
+-----+
|== Physical Plan ==
|ExecutedCommand
|  +- ShowTablesCommand|
+-----+

```

The following EXPLAIN variants in SQL queries are not supported:

- EXPLAIN FORMATTED

- EXPLAIN LOGICAL

```
scala> sql("EXPLAIN LOGICAL show tables")
org.apache.spark.sql.catalyst.parser.ParseException:
Operation not allowed: EXPLAIN LOGICAL(line 1, pos 0)

== SQL ==
EXPLAIN LOGICAL show tables
^^^

    at org.apache.spark.sql.catalyst.parser.ParserUtils$.operationNotAllowed(ParserUtils
.scala:39)
    at org.apache.spark.sql.execution.SparkSqlAstBuilder$$anonfun$visitExplain$1.apply(S
parkSqlParser.scala:258)
    at org.apache.spark.sql.execution.SparkSqlAstBuilder$$anonfun$visitExplain$1.apply(S
parkSqlParser.scala:253)
    at org.apache.spark.sql.catalyst.parser.ParserUtils$.withOrigin(ParserUtils.scala:93
)
    at org.apache.spark.sql.execution.SparkSqlAstBuilder.visitExplain(SparkSqlParser.sca
la:253)
    at org.apache.spark.sql.execution.SparkSqlAstBuilder.visitExplain(SparkSqlParser.sca
la:52)
    at org.apache.spark.sql.catalyst.parser.SqlBaseParser$ExplainContext.accept(SqlBaseP
arser.java:448)
    at org.antlr.v4.runtime.tree.AbstractParseTreeVisitor.visit(AbstractParseTreeVisitor
.java:42)
    at org.apache.spark.sql.catalyst.parser.AstBuilder$$anonfun$visitSingleStatement$1.a
pply(AstBuilder.scala:64)
    at org.apache.spark.sql.catalyst.parser.AstBuilder$$anonfun$visitSingleStatement$1.a
pply(AstBuilder.scala:64)
    at org.apache.spark.sql.catalyst.parser.ParserUtils$.withOrigin(ParserUtils.scala:93
)
    at org.apache.spark.sql.catalyst.parser.AstBuilder.visitSingleStatement(AstBuilder.s
cala:63)
    at org.apache.spark.sql.catalyst.parser.AbstractSqlParser$$anonfun$parsePlan$1.appl
y(ParseDriver.scala:54)
    at org.apache.spark.sql.catalyst.parser.AbstractSqlParser$$anonfun$parsePlan$1.appl
y(ParseDriver.scala:53)
    at org.apache.spark.sql.catalyst.parser.AbstractSqlParser.parse(ParseDriver.scala:82
)
    at org.apache.spark.sql.execution.SparkSqlParser.parse(SparkSqlParser.scala:45)
    at org.apache.spark.sql.catalyst.parser.AbstractSqlParser.parsePlan(ParseDriver.sca
la:53)
    at org.apache.spark.sql.SparkSession.sql(SparkSession.scala:572)
    ... 48 elided
```

# Schema — Shape of Records

A **schema** is the description of the structure of your records (which together create a [Dataset](#) in Spark SQL). It can be **implicit** (and [inferred at runtime](#)) or **explicit** (and known at compile time).

A schema is described using [StructType](#) which is a collection of [StructField](#) objects.

`StructType` and `StructField` belong to the `org.apache.spark.sql.types` package.

```
import org.apache.spark.sql.types.StructType
val schemaUntyped = new StructType()
  .add("a", "int")
  .add("b", "string")
```

You can use the canonical string representation of SQL types to describe the types in a schema (that is inherently untyped at compile type) or use type-safe types from the `org.apache.spark.sql.types` package.

```
// it is equivalent to the above expression
import org.apache.spark.sql.types.{IntegerType, StringType}
val schemaTyped = new StructType()
  .add("a", IntegerType)
  .add("b", StringType)
```

Tip

Read up on [SQL Parser Framework](#) in Spark SQL to learn about `CatalystSqlParser` that is responsible for parsing data types.

`StructType` offers [printTreeString](#) that make presenting the schema more user-friendly.

```

scala> schemaTyped.printTreeString
root
|-- a: integer (nullable = true)
|-- b: string (nullable = true)

scala> println(schema1.prettyJson)
{
  "type" : "struct",
  "fields" : [ {
    "name" : "a",
    "type" : "integer",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "b",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  } ]
}

```

## StructType Data Type

`StructType` is a built-in [data type](#) in Spark SQL to represent a collection of [StructFields](#).

	<p><code>StructType</code> is a <code>Seq[StructField]</code> and therefore all things <code>Seq</code> apply equally here.</p>
Note	<pre> scala&gt; schemaTyped.foreach(println) StructField(a,IntegerType,true) StructField(b,StringType,true) </pre> <p>Read the official documentation of <a href="#">scala.collection.Seq</a>.</p>

You can compare two `StructType` instances to see whether they are equal.

```

import org.apache.spark.sql.types.StructType

val schemaUntyped = new StructType()
  .add("a", "int")
  .add("b", "string")

import org.apache.spark.sql.types.{IntegerType, StringType}
val schemaTyped = new StructType()
  .add("a", IntegerType)
  .add("b", StringType)

scala> schemaUntyped == schemaTyped
res0: Boolean = true

```

`StructType` presents itself as `<struct>` or `STRUCT` in query plans or SQL.

## Adding Fields to Schema (add methods)

You can add a new `StructField` to your `StructType`. There are different variants of `add` method that all make for a new `StructType` with the field added.

```

add(field: StructField): StructType
add(name: String, dataType: DataType): StructType
add(name: String, dataType: DataType, nullable: Boolean): StructType
add(
  name: String,
  dataType: DataType,
  nullable: Boolean,
  metadata: Metadata): StructType
add(
  name: String,
  dataType: DataType,
  nullable: Boolean,
  comment: String): StructType
add(name: String, dataType: String): StructType
add(name: String, dataType: String, nullable: Boolean): StructType
add(
  name: String,
  dataType: String,
  nullable: Boolean,
  metadata: Metadata): StructType
add(
  name: String,
  dataType: String,
  nullable: Boolean,
  comment: String): StructType

```

## DataType Name Conversions

```
simpleString: String
catalogString: String
sql: String
```

`StructType` as a custom `DataType` is used in query plans or SQL. It can present itself using `simpleString`, `catalogString` or `sql` (see [DataType Contract](#)).

```
scala> schemaTyped.simpleString
res0: String = struct<a:int,b:string>

scala> schemaTyped.catalogString
res1: String = struct<a:int,b:string>

scala> schemaTyped.sql
res2: String = STRUCT<`a`: INT, `b`: STRING>
```

## Accessing StructField (apply method)

```
apply(name: String): StructField
```

`StructType` defines its own `apply` method that gives you an easy access to a `StructField` by name.

```
scala> schemaTyped.printTreeString
root
|-- a: integer (nullable = true)
|-- b: string (nullable = true)

scala> schemaTyped("a")
res4: org.apache.spark.sql.types.StructField = StructField(a, IntegerType, true)
```

## Creating StructType from Existing StructType (apply method)

```
apply(names: Set[String]): StructType
```

This variant of `apply` lets you create a `StructType` out of an existing `StructType` with the names only.

```
scala> schemaTyped(names = Set("a"))
res0: org.apache.spark.sql.types.StructType = StructType(StructField(a, IntegerType, true
))
```

It will throw an `IllegalArgumentException` exception when a field could not be found.

```
scala> schemaTyped(names = Set("a", "c"))
java.lang.IllegalArgumentException: Field c does not exist.
  at org.apache.spark.sql.types.StructType.apply(StructType.scala:275)
  ... 48 elided
```

## Printing Out Schema (printTreeString method)

```
printTreeString(): Unit
```

`printTreeString` prints out the schema to standard output.

```
scala> schemaTyped.printTreeString
root
|-- a: integer (nullable = true)
|-- b: string (nullable = true)
```

Internally, it uses `treeString` method to build the tree and then `println` it.

## StructField

A `structField` describes a single field in a `structType`. It has a name, the type and whether or not it be empty, and an optional metadata and a comment.

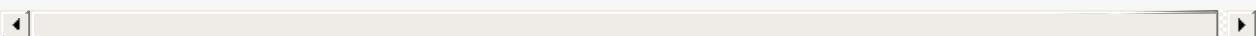
A comment is a part of metadata under `comment` key and is used to build a Hive column or when describing a table.

```
scala> schemaTyped("a").getComment
res0: Option[String] = None

scala> schemaTyped("a").withComment("this is a comment").getComment
res1: Option[String] = Some(this is a comment)
```

## Implicit Schema

```
val df = Seq((0, s"""hello\tworld"""), (1, "two  spaces inside")).toDF("label", "sentence")  
  
scala> df.printSchema  
root  
| -- label: integer (nullable = false)  
| -- sentence: string (nullable = true)  
  
scala> df.schema  
res0: org.apache.spark.sql.types.StructType = StructType(StructField(label,IntegerType,  
false), StructField(sentence,StringType,true))  
  
scala> df.schema("label").dataType  
res1: org.apache.spark.sql.types.DataType = IntegerType
```



# Data Types

`DataType` abstract class is the base type of all [built-in data types](#) in Spark SQL, e.g. strings, longs. It is possible to extend the type system by creating [user-defined types](#).

The [DataType Contract](#) defines methods to build SQL, JSON and string representations.

`DataType` (and the concrete Spark SQL types) live in `org.apache.spark.sql.types` package.

```
import org.apache.spark.sql.types.StringType

scala> StringType.json
res0: String = "string"

scala> StringType.sql
res1: String = STRING

scala> StringType.catalogString
res2: String = string
```

You should use [DataTypes](#) object in your code to create complex Spark SQL types, i.e. arrays or maps.

```
import org.apache.spark.sql.types.DataTypes

scala> val arrayType = DataTypes.createArrayType(BooleanType)
arrayType: org.apache.spark.sql.types.ArrayType = ArrayType(BooleanType, true)

scala> val mapType = DataTypes.createMapType(StringType, LongType)
mapType: org.apache.spark.sql.types.MapType = MapType(StringType, LongType, true)
```

`DataType` has support for Scala's pattern matching using `unapply` method.

```
???
```

## DataType Contract

Any type in Spark SQL follows the `DataType` contract which means that the types define the following methods:

- `json` and `prettyJson` to build JSON representations of a data type
- `defaultSize` to know the default size of values of a type

- `simpleString` and `catalogString` to build user-friendly string representations (with the latter for external catalogs)
- `sql` to build SQL representation

## Built-In Data Types

Spark SQL comes with the following built-in [data types](#):

- `CalendarIntervalType`
- `StructType`
- `MapType`
- `ArrayType`
- `NullType`
- Atomic Types
  - `TimestampType`
  - `StringType`
  - `BooleanType`
  - `DateType`
  - `BinaryType`
- Fractional Types
  - `DoubleType`
  - `FloatType`
- Integral Types
  - `ByteType`
  - `IntegerType`
  - `LongType`
  - `ShortType`

You can also create your own [user-defined types \(UDTs\)](#).

## DataTypes — Factory Methods for Complex DataTypes

---

`DataTypes` is an Scala object with methods to create complex Spark SQL types, i.e. arrays and maps. It lives in `org.apache.spark.sql.types` package.

```
import org.apache.spark.sql.types.DataTypes

scala> val arrayType = DataTypes.createArrayType(BooleanType)
arrayType: org.apache.spark.sql.types.ArrayType = ArrayType(BooleanType, true)

scala> val mapType = DataTypes.createMapType(StringType, LongType)
mapType: org.apache.spark.sql.types.MapType = MapType(StringType, LongType, true)
```

Tip

`DataTypes` offers singleton objects for Spark SQL data types but they seem to be only for Java programmers. Since the data types are also defined as `case object` alongside their definitions, you are lucky to code in Scala and have no reason to use `DataTypes` objects to access the types. Just import the package and you have access to the objects.

## UDTs — User-Defined Types

Caution

FIXME

# Encoders — Internal Row Converters

**Encoders** are the fundamental concept in the serialization and deserialization (SerDe) framework in [Catalyst Optimizer](#). An encoder of type `T` is used to convert (`encode` and `decode`) a JVM object of type `T` (that could be your domain object) and primitives to and from the Spark SQL internal binary row format representation using Catalyst expressions and code generation.

Note

`Encoder` is also called "*a container of serde expressions in Dataset*".

Encoders know the [schema](#) of the records. They allow for significantly faster serialization and deserialization (comparing to the default Java or Kryo serializers).

The Encoder concept is represented by `trait Encoder[T]`.

```
trait Encoder[T] extends Serializable {
  def schema: StructType
  def clsTag: ClassTag[T]
}
```

The one and only implementation of the `Encoder` trait in Spark 2.0 is [ExpressionEncoder](#).

```

case class Person(id: Long, name: String)

import org.apache.spark.sql.Encoders

scala> val personEncoder = Encoders.product[Person]
personEncoder: org.apache.spark.sql.Encoder[Person] = class[id[0]: bigint, name[0]: string]

scala> personEncoder.schema
res0: org.apache.spark.sql.types.StructType = StructType(StructField(id,LongType, false), StructField(name,StringType, true))

scala> personEncoder.clsTag
res1: scala.reflect.ClassTag[Person] = Person

import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder

scala> val personExprEncoder = personEncoder.asInstanceOf[ExpressionEncoder[Person]]
personExprEncoder: org.apache.spark.sql.catalyst.encoders.ExpressionEncoder[Person] = class[id[0]: bigint, name[0]: string]

scala> personExprEncoder.namedExpressions
res2: Seq[org.apache.spark.sql.catalyst.expressions.NamedExpression] = List(assertnotnull(input[0, Person, true], top level non-flat input object).id AS id#2L, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(input[0, Person, true], top level non-flat input object).name, true) AS name#3)

scala> val row = personExprEncoder.toRow(Person(0, "Jacek"))
row: org.apache.spark.sql.catalyst.InternalRow = [0,0,1800000005,6b6563614a]

// WARNING: ERROR A bug?
scala> val jacek = personExprEncoder.fromRow(row)
java.lang.RuntimeException: Error while decoding: java.lang.UnsupportedOperationException: Cannot evaluate expression: upcast('id, LongType, - field (class: "scala.Long", name: "id"), - root class: "Person")
...

```

Dataset owns a `Encoder` that serializes and deserializes the type of the `Dataset`.

You can [create custom encoders using `Encoders` object](#). Encoders for many Scala types are however available through `SparkSession.implicits` object so in most cases you don't need to worry about them whatsoever and simply import the `implicits` object.

```

val spark = SparkSession.builder.getOrCreate()
import spark.implicits._

```

Encoders map columns (of your dataset) to fields (of your JVM object) by name. It is by Encoders that you can bridge JVM objects to data sources (CSV, JDBC, Parquet, Avro, JSON, Cassandra, Elasticsearch, memsql) and vice versa.

```
import org.apache.spark.sql.Encoders

case class Person(id: Int, name: String, speaksPolish: Boolean)

scala> val personEncoder = Encoders.product[Person]
personEncoder: org.apache.spark.sql.Encoder[Person] = class[id[0]: int, name[0]: string, speaksPolish[0]: boolean]

scala> personEncoder.schema
res11: org.apache.spark.sql.types.StructType = StructType(StructField(id, IntegerType, false), StructField(name, StringType, true), StructField(speaksPolish, BooleanType, false))

scala> personEncoder.clsTag
res12: scala.reflect.ClassTag[Person] = Person
```

## ExpressionEncoder

```
case class ExpressionEncoder[T](
    schema: StructType,
    flat: Boolean,
    serializer: Seq[Expression],
    deserializer: Expression,
    clsTag: ClassTag[T])
extends Encoder[T]
```

`ExpressionEncoder` is the one and only implementation of the `Encoder` trait in Spark 2.0 with additional properties, i.e. `flat`, one or many serializers and a deserializer expressions.

A `ExpressionEncoder` can be **flat** is which case there is only one Catalyst expression for the serializer.

**Serializer expressions** are used to encode an object of type `T` to a `InternalRow`. It is assumed that all serializer expressions contain at least one and the same `BoundReference`.

Caution	<a href="#">FIXME</a> What's <code>BoundReference</code> ?
---------	--

**Deserializer expression** is used to decode an `InternalRow` to an object of type `T`.

Internally, a `ExpressionEncoder` creates a `UnsafeProjection` (for the input serializer), a `InternalRow` (of size `1`), and a safe `Projection` (for the input deserializer). They are all internal lazy attributes of the encoder.

## Creating Custom Encoders (Encoders object)

Encoders factory object defines methods to create Encoder instances.

Import org.apache.spark.sql package to have access to the Encoders factory object.

```
import org.apache.spark.sql.Encoders

scala> Encoders.LONG
res1: org.apache.spark.sql.Encoder[Long] = class[value[0]: bigint]
```

You can find methods to create encoders for Java's object types, e.g. Boolean , Integer , Long , Double , String , java.sql.Timestamp or Byte array, that could be composed to create more advanced encoders for Java bean classes (using bean method).

```
import org.apache.spark.sql.Encoders

scala> Encoders.STRING
res2: org.apache.spark.sql.Encoder[String] = class[value[0]: string]
```

You can also create encoders based on Kryo or Java serializers.

```
import org.apache.spark.sql.Encoders

case class Person(id: Int, name: String, speaksPolish: Boolean)

scala> Encoders.kryo[Person]
res3: org.apache.spark.sql.Encoder[Person] = class[value[0]: binary]

scala> Encoders.javaSerialization[Person]
res5: org.apache.spark.sql.Encoder[Person] = class[value[0]: binary]
```

You can create encoders for Scala's tuples and case classes, Int , Long , Double , etc.

```
import org.apache.spark.sql.Encoders

scala> Encoders.tuple(Encoders.scalaLong, Encoders.STRING, Encoders.scalaBoolean)
res9: org.apache.spark.sql.Encoder[(Long, String, Boolean)] = class[_1[0]: bigint, _2[0]: string, _3[0]: boolean]
```

# InternalRow — Internal Binary Row Format

```
// The type of your business objects
case class Person(id: Long, name: String)

// The encoder for Person objects
import org.apache.spark.sql.Encoders
val personEncoder = Encoders.product[Person]

// The expression encoder for Person objects
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
val personExprEncoder = personEncoder.asInstanceOf[ExpressionEncoder[Person]]

// Convert Person objects to InternalRow
scala> val row = personExprEncoder.toRow(Person(0, "Jacek"))
row: org.apache.spark.sql.catalyst.InternalRow = [0,0,1800000005,6b6563614a]

// How many fields are available in Person's InternalRow?
scala> row.numFields
res0: Int = 2

// Are there any NULLs in this InternalRow?
scala> row.isNullAt(0)
res1: Boolean = false

// You can create your own InternalRow objects
import org.apache.spark.sql.catalyst.InternalRow

scala> val ir = InternalRow(5, "hello", (0, "nice"))
ir: org.apache.spark.sql.catalyst.InternalRow = [5,hello,(0,nice)]
```

There are methods to create `InternalRow` objects using the factory methods in the `InternalRow` object.

```
import org.apache.spark.sql.catalyst.InternalRow

scala> InternalRow.empty
res0: org.apache.spark.sql.catalyst.InternalRow = [empty row]

scala> InternalRow(0, "string", (0, "pair"))
res1: org.apache.spark.sql.catalyst.InternalRow = [0,string,(0,pair)]

scala> InternalRow.fromSeq(Seq(0, "string", (0, "pair")))
res2: org.apache.spark.sql.catalyst.InternalRow = [0,string,(0,pair)]
```

# UnsafeRow

`UnsafeRow` is a mutable internal row that is `Externalizable` and `KryoSerializable`.

Caution

**FIXME** What does being `Externalizable` and `KryoSerializable` mean?  
What's the protocol to follow?

# Columns

Caution	<a href="#">FIXME</a>
---------	-----------------------

`Column` type represents...[FIXME](#)

## like

Caution	<a href="#">FIXME</a>
---------	-----------------------

```
scala> df("id") like "0"
res0: org.apache.spark.sql.Column = id LIKE 0

scala> df.filter('id like "0").show
+---+-----+
| id| text|
+---+-----+
|  0|hello|
+---+-----+
```

## Symbols As Column Names

```
scala> val df = Seq((0, "hello"), (1, "world")).toDF("id", "text")
df: org.apache.spark.sql.DataFrame = [id: int, text: string]

scala> df.select('id)
res0: org.apache.spark.sql.DataFrame = [id: int]

scala> df.select('id).show
+---+
| id|
+---+
|  0|
|  1|
+---+
```

## over function

```
over(window: expressions.WindowSpec): Column
```

`over` function defines a **windowing column** that allows for window computations to be applied to a window. Window functions are defined using [WindowSpec](#).

Tip	Read about Windows in <a href="#">Windows</a> .
-----	---

## cast

`cast` method casts a column to a data type. It makes for type-safe maps with `Row` objects of the proper type (not `Any`).

```
cast(to: String): Column
cast(to: DataType): Column
```

It uses `CatalystSqlParser` to parse the data type from its canonical string representation.

## cast Example

```
scala> val df = Seq((0f, "hello")).toDF("label", "text")
df: org.apache.spark.sql.DataFrame = [label: float, text: string]

scala> df.printSchema
root
|-- label: float (nullable = false)
|-- text: string (nullable = true)

// without cast
import org.apache.spark.sql.Row
scala> df.select("label").map { case Row(label) => label.getClass.getName }.show(false)
)
+-----+
|value      |
+-----+
|java.lang.Float|
+-----+

// with cast
import org.apache.spark.sql.types.DoubleType
scala> df.select(col("label").cast(DoubleType)).map { case Row(label) => label.getClass.getName }.show(false)
)
+-----+
|value      |
+-----+
|java.lang.Double|
+-----+
```

# DataFrame (aka Dataset[Row])

A **DataFrame** is a data abstraction or a domain-specific language (DSL) for working with **structured** and **semi-structured data**, i.e. datasets with a schema. A DataFrame is thus a collection of [rows](#) with a schema.

It uses the immutable, in-memory, resilient, distributed and parallel capabilities of [RDD](#), and applies a [schema](#) to the data.

	In Spark <b>2.0.0</b> <code>DataFrame</code> is a <i>mere</i> type alias for <code>Dataset[Row]</code> .
Note	<pre>type DataFrame = Dataset[Row]</pre> <p>See <a href="#">org.apache.spark.package.scala</a>.</p>

`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 with 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).

	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 structured 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](#)).

```

val df = Seq(("one", 1), ("one", 1), ("two", 1))
  .toDF("word", "count")

scala> df.show
+---+---+
|word|count|
+---+---+
| one|    1|
| one|    1|
| two|    1|
+---+---+

val counted = df.groupBy('word).count

scala> counted.show
+---+---+
|word|count|
+---+---+
| two|    1|
| one|    2|
+---+---+

```

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 (as in the above example). See [DataFrame API](#). You can read any format given you have appropriate Spark SQL extension of [DataFrameReader](#) to format the dataset appropriately.

**Caution**

[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" world 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](#)

DataFrames use the [Catalyst query optimizer](#) to produce efficient queries (and so they are supposed to be faster than corresponding RDD-based queries).

**Note**

Your DataFrames can also be type-safe and moreover further improve their performance through [specialized encoders](#) that can significantly cut serialization and deserialization times.

You can enforce types on [generic rows](#) and hence bring type safety (at compile time) by [encoding rows into type-safe `dataset` object](#). As of Spark 2.0 it is a preferred way of developing Spark applications.

## Features of DataFrame

A `DataFrame` is a collection of "generic" `Row` instances (as `RDD[Row]`) and a [schema](#).

**Note**

Regardless of how you create a `DataFrame`, it will always be a pair of `RDD[Row]` and [StructType](#).

## Enforcing Types (as method)

`DataFrame` is a type alias for `Dataset[Row]`. You can enforce types of the fields using `as` method.

`as` gives you a conversion from `Dataset[Row]` to `Dataset[T]`.

```
// Create DataFrame of pairs
val df = Seq("hello", "world!").zipWithIndex.map(_.swap).toDF("id", "token")

scala> df.printSchema
root
|-- id: integer (nullable = false)
|-- token: string (nullable = true)

scala> val ds = df.as[(Int, String)]
ds: org.apache.spark.sql.Dataset[(Int, String)] = [id: int, token: string]

// It's more helpful to have a case class for the conversion
final case class MyRecord(id: Int, token: String)

scala> val myRecords = df.as[MyRecord]
myRecords: org.apache.spark.sql.Dataset[MyRecord] = [id: int, token: string]
```

## 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|
+-----+-----+

```

## Writing DataFrames to External Storage (write method)

Caution

[FIXME](#)

## SQLContext, spark, 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 `spark` object.

```

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

```

As you may have noticed, `spark` 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](#).

## Using toDF

After you `import spark.implicits._` (which is done for you by Spark shell) you may apply `toDF` method to convert objects to DataFrames.

```
scala> val df = Seq("I am a DataFrame!").toDF("text")
df: org.apache.spark.sql.DataFrame = [text: string]
```

## 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 = spark.createDataFrame(people)
df: org.apache.spark.sql.DataFrame = [name: string, age: int]

scala> df.show
+---+---+
| name|age|
+---+---+
| Jacek| 42|
| Patryk| 19|
| Maksym| 5|
+---+---+
```

## 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,String
Type,true), StructField(bid,StringType,true), StructField(bidtime,StringType,true), St
ructField(bidder,StringType,true), StructField(bidderrate,StringType,true), StructFiel
d(openbid,StringType,true), StructField(price,StringType,true))

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,Strin
gType,true), StructField(bid,StringType,true), StructField(bidtime,StringType,true), S
tructField(bidder,StringType,true), StructField(bidderrate,StringType,true), StructFie
ld(openbid,StringType,true), StructField(price,StringType,true))

scala> val noheaders = lines.filter(_ != header)
noheaders: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[10] at filter at <conso
le>:33

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 <console>:35

scala> val auctions = spark.createDataFrame(rows, schema)
auctions: org.apache.spark.sql.DataFrame = [auctionid: string, bid: string, bidtime: s
tring, bidder: string, bidderrate: string, openbid: string, price: 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), (bidt
ime,StringType), (bidder,StringType), (bidderrate,StringType), (openbid,StringType), (p
rice,StringType))

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, bidderrate: Int, openbid: Float, price: Float)
defined class Auction

scala> val noheader = lines.filter(_ != header)
noheader: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[53] at filter at <console>:31

scala> val auctions = noheader.map(_.split(",")).map(r => Auction(r(0), r(1).toFloat,
r(2).toFloat, r(3), r(4).toInt, r(5).toFloat, r(6).toFloat))
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, bidder: string, bidderrate: int, openbid: float, price: float]

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-
-2.11/spark-assembly-1.5.0-SNAPSHOT-hadoop2.7.1.jar!/org/apache/ivy/core/settings/ivys
ettings.xml
com.databricks#spark-csv_2.11 added as a dependency

scala> val df = spark.read.format("com.databricks.spark.csv").option("header", "true")
.load("Cartier+for+WinnersCurse.csv")
df: org.apache.spark.sql.DataFrame = [auctionid: string, bid: string, bidtime: string,
bidder: string, bidderrate: string, openbid: string, price: string]

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| adamurry|        5|    300| 740|
|1638844464|  500|6.736469907| adamurry|        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

```

## Reading Data from External Data Sources (read method)

You can create DataFrames by loading data from structured files (JSON, Parquet, CSV), RDDs, tables in Hive, or external databases (JDBC) using `SQLContext.read` method.

```
read: DataFrameReader
```

`read` returns a `DataFrameReader` instance.

Among the supported structured data (file) formats are (consult [Specifying Data Format \(format method\)](#) for `DataFrameReader`):

- JSON
- parquet
- JDBC
- ORC
- Tables in Hive and any JDBC-compliant database
- libsvm

```
val reader = spark.read
r: org.apache.spark.sql.DataFrameReader = org.apache.spark.sql.DataFrameReader@59e67a1
8

reader.parquet("file.parquet")
reader.json("file.json")
reader.format("libsvm").load("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 = spark.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=[count#148L])
  TungstenExchange SinglePartition
    TungstenAggregate(key=[], functions=[(count(1),mode=Partial,isDistinct=false)], output=[currentCount#156L])
      TungstenProject
        Scan PhysicalRDD[auctionid#49,bid#50,bidtime#51,bidder#52,bidderrate#53,openbid#54,price#55]

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

# Row

`Row` is a data abstraction of an ordered collection of fields that can be accessed by an ordinal / an index (aka *generic access by ordinal*), a name (aka *native primitive access*) or using Scala's pattern matching. A `Row` instance may or may not have a [schema](#).

The traits of `Row` :

- `length` or `size` - `Row` knows the number of elements (columns).
- `schema` - `Row` knows the schema

`Row` belongs to `org.apache.spark.sql.Row` package.

```
import org.apache.spark.sql.Row
```

## Field Access

Fields of a `Row` instance can be accessed by index (starting from `0`) using `apply` or `get`.

```
scala> val row = Row(1, "hello")
row: org.apache.spark.sql.Row = [1,hello]

scala> row(1)
res0: Any = hello

scala> row.get(1)
res1: Any = hello
```

Note	Generic access by ordinal (using <code>apply</code> or <code>get</code> ) returns a value of type <code>Any</code> .
------	--

You can query for fields with their proper types using `getAs` with an index

```
val row = Row(1, "hello")

scala> row.getAs[Int](0)
res1: Int = 1

scala> row.getAs[String](1)
res2: String = hello
```

**FIXME****Note**`row.getAs[String](null)`

## Schema

A `Row` instance can have a schema defined.

**Note**

Unless you are instantiating `Row` yourself (using [Row Object](#)), a `Row` has always a schema.

**Note**

It is `RowEncoder` to take care of assigning a schema to a `Row` when `toDF` on a [Dataset](#) or when instantiating [DataFrame](#) through [DataFrameReader](#).

## Row Object

`Row` companion object offers factory methods to create `Row` instances from a collection of elements (`apply`), a sequence of elements (`fromSeq`) and tuples (`fromTuple`).

```
scala> Row(1, "hello")
res0: org.apache.spark.sql.Row = [1,hello]

scala> Row.fromSeq(Seq(1, "hello"))
res1: org.apache.spark.sql.Row = [1,hello]

scala> Row.fromTuple((0, "hello"))
res2: org.apache.spark.sql.Row = [0,hello]
```

`Row` object can merge `Row` instances.

```
scala> Row.merge(Row(1), Row("hello"))
res3: org.apache.spark.sql.Row = [1,hello]
```

It can also return an empty `Row` instance.

```
scala> Row.empty == Row()
res4: Boolean = true
```

## Pattern Matching on Row

`Row` can be used in pattern matching (since [Row Object](#) comes with `unapplySeq` ).

```
scala> Row.unapplySeq(Row(1, "hello"))
res5: Some[Seq[Any]] = Some(wrappedArray(1, hello))

Row(1, "hello") match { case Row(key: Int, value: String) =>
  key -> value
}
```

# DataSource API — Loading and Saving Datasets

## Reading Datasets

Spark SQL can read data from external storage systems like files, Hive tables and JDBC databases through [DataFrameReader](#) interface.

You use [SparkSession](#) to access [DataFrameReader](#) using [read](#) operation.

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder.getOrCreate

val reader = spark.read
```

[DataFrameReader](#) is an interface to create [DataFrames](#) (aka [Dataset\[Row\]](#)) from [files](#), [Hive tables](#) or [JDBC](#).

```
val people = reader.csv("people.csv")
val cities = reader.format("json").load("cities.json")
```

As of Spark 2.0, [DataFrameReader](#) can read text files using [textFile](#) methods that return [Dataset\[String\]](#) (not [DataFrames](#)).

```
spark.read.textFile("README.md")
```

You can also [define your own custom file formats](#).

```
val countries = reader.format("customFormat").load("countries.cf")
```

There are two operation modes in Spark SQL, i.e. batch and [streamed](#) (called **structured streaming**).

You can access [DataStreamReader](#) for reading streaming datasets through [SparkSession.readStream](#) method.

```
import org.apache.spark.sql.streaming.DataStreamReader
val stream: DataStreamReader = spark.readStream
```

The available methods in [DataStreamReader](#) are similar to [DataFrameReader](#).

## Saving Datasets

Spark SQL can save data to external storage systems like files, Hive tables and JDBC databases through [DataFrameWriter](#) interface.

You use `write` method on a `Dataset` to access `DataFrameWriter`.

```
import org.apache.spark.sql.{DataFrameWriter, Dataset}
val ints: Dataset[Int] = (0 to 5).toDS

val writer: DataFrameWriter[Int] = ints.write
```

`DataFrameWriter` is an interface to persist a [Datasets](#) to an external storage system in a batch fashion.

You can access [DataStreamWriter](#) for writing streaming datasets through `Dataset.writeStream` method.

```
val papers = spark.readStream.text("papers").as[String]

import org.apache.spark.sql.streaming.DataStreamWriter
val writer: DataStreamWriter[String] = papers.writeStream
```

The available methods in `DataStreamWriter` are similar to `DataFrameWriter`.

# DataFrameReader

`DataFrameReader` is an interface to create `DataFrames` from [files](#), [Hive tables](#) or [JDBC](#).

Note	You can <a href="#">define your own custom file formats</a> .
------	---

You use [SparkSession.read](#) to access an instance of `DataFrameReader`.

```
val spark: SparkSession = SparkSession.builder.getOrCreate

import org.apache.spark.sql.DataFrameReader
val reader: DataFrameReader = spark.read
```

`DataFrameReader` supports many [file formats](#) and [interface for new ones](#). It assumes `parquet` as the default data source format that you can change using `spark.sql.sources.default` setting.

As of Spark 2.0, `DataFrameReader` can read text files using [textFile](#) methods that return `Dataset[String]` (not `DataFrames` which are `Dataset[Row]` and therefore untyped).

## Specifying Data Format (`format` method)

```
format(source: String): DataFrameReader
```

You use `format` to configure `DataFrameReader` to use appropriate `source` format.

Supported data formats:

- `json`
- `csv` (since **2.0.0**)
- `parquet` (see [Parquet](#))
- `orc`
- `text`
- `jdbc`
- `libsvm` — only when used in `format("libsvm")`

Note	You can improve your understanding of <code>format("jdbc")</code> with the exercise <a href="#">Creating DataFrames from Tables using JDBC and PostgreSQL</a> .
------	---

## Specifying Input Schema (schema method)

```
schema(schema: StructType): DataFrameReader
```

You can specify a `schema` of the input data source.

Tip

Refer to [Schema](#).

## Option Support (option and options methods)

```
option(key: String, value: String): DataFrameReader
option(key: String, value: Boolean): DataFrameReader (1)
option(key: String, value: Long): DataFrameReader (1)
option(key: String, value: Double): DataFrameReader (1)
```

1. Available since Spark 2.0.0

You can also use `options` method to describe different options in a single `Map`.

```
options(options: scala.collection.Map[String, String]): DataFrameReader
```

## load methods

```
load(): DataFrame
load(path: String): DataFrame
```

`load` loads input data as a `DataFrame`.

```
val csv = spark.read
  .format("csv")
  .option("header", "true")
  .load("*.csv")
```

## Creating DataFrames from Files

`DataFrameReader` supports the following file formats:

- [JSON](#)
- [CSV](#)
- [parquet](#)

- ORC
- text

## json method

```
json(path: String): DataFrame
json(paths: String*): DataFrame
json(jsonRDD: RDD[String]): DataFrame
```

New in **2.0.0**: prefersDecimal

## csv method

```
csv(path: String): DataFrame
csv(paths: String*): DataFrame
```

## parquet method

```
parquet(path: String): DataFrame
parquet(paths: String*): DataFrame
```

The supported options:

- compression (default: snappy )

New in **2.0.0**: snappy is the default Parquet codec. See [\[SPARK-14482\]\[SQL\] Change default Parquet codec from gzip to snappy](#).

The compressions supported:

- none Or uncompressed
- snappy - the default codec in Spark **2.0.0**.
- gzip - the default codec in Spark before **2.0.0**
- lzo

```
val tokens = Seq("hello", "henry", "and", "harry")
  .zipWithIndex
  .map(_.swap)
  .toDF("id", "token")

val parquetWriter = tokens.write
```

```

parquetWriter.option("compression", "none").save("hello-none")

// The exception is mostly for my learning purposes
// so I know where and how to find the trace to the compressions
// Sorry...
scala> parquetWriter.option("compression", "unsupported").save("hello-unsupported")
java.lang.IllegalArgumentException: Codec [unsupported] is not available. Available co
decs are uncompressed, gzip, lzo, snappy, none.
    at org.apache.spark.sql.execution.datasources.parquet.ParquetOptions.<init>(ParquetO
ptions.scala:43)
    at org.apache.spark.sql.execution.datasources.parquet.DefaultSource.prepareWrite(Par
quetRelation.scala:77)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$ru
n$1$$anonfun$4.apply(InsertIntoHadoopFsRelation.scala:122)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$ru
n$1$$anonfun$4.apply(InsertIntoHadoopFsRelation.scala:122)
    at org.apache.spark.sql.execution.datasources.BaseWriterContainer.driverSideSetup(Wr
iterContainer.scala:103)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$ru
n$1.apply$mcV$sp(InsertIntoHadoopFsRelation.scala:141)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$ru
n$1.apply(InsertIntoHadoopFsRelation.scala:116)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$ru
n$1.apply(InsertIntoHadoopFsRelation.scala:116)
    at org.apache.spark.sql.execution.SQLExecution$.withNewExecutionId(SQLExecution.sca
la:53)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation.run(InsertI
ntoHadoopFsRelation.scala:116)
    at org.apache.spark.sql.execution.command.ExecutedCommand.sideEffectResult$lzycomput
e(commands.scala:61)
    at org.apache.spark.sql.execution.command.ExecutedCommand.sideEffectResult(commands.s
cala:59)
    at org.apache.spark.sql.execution.command.ExecutedCommand.doExecute(commands.scala:73
)
    at org.apache.spark.sql.execution.SparkPlan$$anonfun$execute$1.apply(SparkPlan.scala:
118)
    at org.apache.spark.sql.execution.SparkPlan$$anonfun$execute$1.apply(SparkPlan.scala:
118)
    at org.apache.spark.sql.execution.SparkPlan$$anonfun$executeQuery$1.apply(SparkPlan.s
cala:137)
    at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
    at org.apache.spark.sql.execution.SparkPlan.executeQuery(SparkPlan.scala:134)
    at org.apache.spark.sql.execution.SparkPlan.execute(SparkPlan.scala:117)
    at org.apache.spark.sql.execution.QueryExecution.toRdd$lzycompute(QueryExecution.sca
la:65)
    at org.apache.spark.sql.execution.QueryExecution.toRdd(QueryExecution.scala:65)
    at org.apache.spark.sql.execution.datasources.DataSource.write(DataSource.scala:390)
    at org.apache.spark.sql.DataFrameWriter.save(DataFrameWriter.scala:247)
    at org.apache.spark.sql.DataFrameWriter.save(DataFrameWriter.scala:230)
    ... 48 elided

```

## orc method

```
orc(path: String): DataFrame  
orc(paths: String*): DataFrame
```

**Optimized Row Columnar (ORC)** file format is a highly efficient columnar format to store Hive data with more than 1,000 columns and improve performance. ORC format was introduced in Hive version 0.11 to use and retain the type information from the table definition.

Tip

Read [ORC Files](#) document to learn about the ORC file format.

## text method

`text` method loads a text file.

```
text(path: String): DataFrame  
text(paths: String*): DataFrame
```

### Example

```
val lines: Dataset[String] = spark.read.text("README.md").as[String]

scala> lines.show
+-----+
|          value|
+-----+
|      # Apache Spark|
|      |
|Spark is a fast a...|
|high-level APIs i...|
|supports general ...|
|rich set of highe...|
|MLlib for machine...|
|and Spark Streami...|
|      |
|<http://spark.apa...|
|      |
|      |
|## Online Documen...|
|      |
|You can find the ...|
|guide, on the [pr...|
|and [project wiki...|
|This README file ...|
|      |
|  ## Building Spark|
+-----+
only showing top 20 rows
```

## Creating DataFrames from Tables

### table method

```
table(tableName: String): DataFrame
```

`table` method returns the `tableName` table as a `DataFrame`.

```

scala> spark.sql("SHOW TABLES").show(false)
+-----+-----+
|tableName|isTemporary|
+-----+-----+
|dafa     |false      |
+-----+-----+

scala> spark.read.table("dafa").show(false)
+---+---+
|id |text   |
+---+---+
|1  |swiecie|
|0  |hello   |
+---+---+

```

**Caution**

**FIXME** The method uses `spark.sessionState.sqlParser.parseTableIdentifier(tableName)` and `spark.sessionState.catalog.lookupRelation`. Would be nice to learn a bit more on their internals, huh?

## jdbc method

**Note**

`jdbc` method uses `java.util.Properties` (and appears so Java-centric). Use `format("jdbc")` instead.

```

jdbc(url: String, table: String, properties: Properties): DataFrame
jdbc(url: String, table: String,
  parts: Array[Partition],
  connectionProperties: Properties): DataFrame
jdbc(url: String, table: String,
  predicates: Array[String],
  connectionProperties: Properties): DataFrame
jdbc(url: String, table: String,
  columnName: String,
  lowerBound: Long,
  upperBound: Long,
  numPartitions: Int,
  connectionProperties: Properties): DataFrame

```

`jdbc` allows you to create `DataFrame` that represents `table` in the database available as `url`.

## Reading Text Files (textFile methods)

```
textFile(path: String): Dataset[String]
textFile(paths: String*): Dataset[String]
```

`textFile` methods query text files as a `Dataset[String]`.

```
spark.read.textFile("README.md")
```

Note	<code>textFile</code> are similar to <code>text</code> family of methods in that they both read text files but <code>text</code> methods return untyped <code>DataFrame</code> while <code>textFile</code> return typed <code>Dataset[String]</code> .
------	--

Internally, `textFile` passes calls on to `text` method and **selects** the only `value` column before it applies `Encoders.STRING encoder`.

# DataFrameWriter

`DataFrameWriter` is an interface to persist a `Datasets` to an external storage system in a batch fashion.

Note	Read <a href="#">DataStreamWriter</a> for streaming write API.
------	--

Use `write` method on a `DataFrame` to access `DataFrameWriter`.

```
import org.apache.spark.sql.{DataFrame, DataFrameWriter, Row}

val ints: DataFrame = (0 to 5).toDF
val writer: DataFrameWriter[Row] = ints.write
```

`DataFrameWriter` has a direct support for many [file formats](#), [JDBC databases](#) and [an interface to plug in new formats](#). It assumes [parquet](#) as the default data source that you can change using [spark.sql.sources.default](#) setting or [format](#) method.

```
// see above for writer definition

// Save dataset in Parquet format
writer.save(path = "ints")

// Save dataset in JSON format
writer.format("json").save(path = "ints-json")
```

Interestingly, a `DataFrameWriter` is really a type constructor in Scala that keeps a reference to a source `DataFrame` during its lifecycle (starting right from the moment it was created).

## Internal State

`DataFrameWriter` uses the following mutable attributes to build a properly-defined write specification for [insertInto](#), [saveAsTable](#), and [save](#):

Table 1. Attributes and Corresponding Setters

Attribute	Setters
source	format
mode	mode
extraOptions	option, options, save
partitioningColumns	partitionBy
bucketColumnNames	bucketBy
numBuckets	bucketBy
sortColumnNames	sortBy

## saveAsTable method

```
saveAsTable(tableName: String): Unit
```

`saveAsTable` saves the content of a `DataFrame` as the `tableName` table.

First, `tableName` is parsed to an internal table identifier. `saveAsTable` then checks whether the table exists or not and uses [save mode](#) to decide what to do.

`saveAsTable` uses the [SessionCatalog](#) for the current session.

Table 2. `saveAsTable`'s Behaviour per Save Mode

Does table exist?	Save Mode	Behaviour
yes	Ignore	Do nothing
yes	ErrorIfExists	Throws a <code>AnalysisException</code> exception with <code>Table [tableIdent] already exists.</code> error message.
anything	anything	It creates a <code>catalogTable</code> and executes the <code>createTable</code> plan.

```
val ints = 0 to 9 toDF
val options = Map("path" -> "/tmp/int")
ints.write.options(options).saveAsTable("ints")
sql("show tables").show
```

## Persisting DataFrame (save method)

```
save(): Unit
```

When you `save` a `DataFrame`, the method will first check whether the `DataFrame` is not bucketed.

Caution	<a href="#">FIXME</a> What does <code>bucketing</code> mean?
---------	--

It then creates a [DataSource](#) and calls `write` on it.

Note

It uses `source`, `partitioningColumns`, `extraOptions`, and `mode` internal attributes directly. They are specified through the API.

## jdbc

```
jdbc(url: String, table: String, connectionProperties: Properties): Unit
```

`jdbc` method saves the content of the `DataFrame` to an external database table via JDBC.

You can use `mode` to control **save mode**, i.e. what happens when an external table exists when `save` is executed.

It is assumed that the `jdbc` save pipeline is not `partitioned` and `bucketed`.

All `options` are overridden by the input `connectionProperties`.

The required options are:

- `driver` which is the class name of the JDBC driver (that is passed to Spark's own `DriverRegistry.register` and later used to `connect(url, properties)`).

When `table` exists and the `override save mode` is in use, `DROP TABLE table` is executed.

It creates the input `table` (using `CREATE TABLE table (schema)` where `schema` is the schema of the `DataFrame`).

## bucketBy method

Caution

[FIXME](#)

## partitionBy method

```
partitionBy(colNames: String*): DataFrameWriter[T]
```

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Specifying Save Mode (mode method)

```
mode(saveMode: String): DataFrameWriter[T]
mode(saveMode: SaveMode): DataFrameWriter[T]
```

You can control the behaviour of write using `mode` method, i.e. what happens when an external file or table exist when `save` is executed.

- `SaveMode.Ignore` or
- `SaveMode.ErrorIfExists` or
- `SaveMode.Overwrite` or

## Writer Configuration (option and options methods)

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Writing DataFrames to Files

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Specifying Data Format (format method)

Caution	<a href="#">FIXME</a> Compare to DataFrameReader.
---------	---

## Parquet

Caution	<a href="#">FIXME</a>
---------	-----------------------

Note	Parquet is the default data source format.
------	--

## insertInto method

Caution	<a href="#">FIXME</a>
---------	-----------------------



# DataSource

`DataSource` case class belongs to the Data Source API (along with [DataFrameReader](#) for loading datasets and [DataFrameWriter](#) for saving datasets).

`DataSource` acts as the canonical set of parameters that describe a data source to load from or write data to. It uses a [SparkSession](#), a class name, a collection of `paths`, optional user-specified [schema](#), a collection of partition columns, a bucket specification, and configuration options.

## Creating DataSource Instance

When being created, `DataSource` first [looks up the providing class](#) given the input `className` (considering it an alias or a fully-qualified class name) and computes the [name](#) and [schema](#) of the data source.

### lookupDataSource Internal Method

```
lookupDataSource(provider0: String): Class[_]
```

It first searches the classpath for available [DataSourceRegister](#) classes (using Java's [ServiceLoader.load](#) method) and finds the requested data source by short name.

If the `DataSource` could not be found by its alias, it tries to load the class by the input `provider0` fully-qualified class name or its variant `provider0.DefaultSource` (with `.DefaultSource` suffix).

There has to be one data source registered only or you will see the following

```
RuntimeException :
```

```
Multiple sources found for [provider] ([comma-separated class names]), please specify the fully qualified class name.
```

### sourceSchema Internal Method

```
sourceSchema(): SourceInfo
```

`sourceSchema` returns the name and [schema](#) of the data source for streamed reading.

**Caution**

**FIXME** Why is the method called? Why does this bother with streamed reading and data sources?!

It supports two class hierarchies, i.e. `StreamSourceProvider` and `FileFormat` data sources.

Internally, `sourceSchema` first creates an instance of the data source and...

**Caution**

**FIXME** Finish...

For `StreamSourceProvider` data sources, `sourceSchema` relays calls to `StreamSourceProvider.sourceSchema`.

For `FileFormat` data sources, `sourceSchema` makes sure that `path` option was specified.

**Tip**

`path` is looked up in a case-insensitive way so `paTh` and `PATH` and `pATH` are all acceptable. Use the lower-case version of `path`, though.

**Note**

`path` can use [glob pattern](#) (not regex syntax), i.e. contain any of `{[]}*?\\` characters.

It checks whether the path exists if a glob pattern is not used. In case it did not exist you will see the following `AnalysisException` exception in the logs:

```
scala> spark.read.load("the.file.does.not.exist.parquet")
org.apache.spark.sql.AnalysisException: Path does not exist: file:/Users/jacek/dev/oss
/spark/the.file.does.not.exist.parquet;
    at org.apache.spark.sql.execution.datasources.DataSource$$anonfun$12.apply(DataSourc
e.scala:375)
    at org.apache.spark.sql.execution.datasources.DataSource$$anonfun$12.apply(DataSourc
e.scala:364)
    at scala.collection.TraversableLike$$anonfun$flatMap$1.apply(TraversableLike.scala:2
41)
    at scala.collection.TraversableLike$$anonfun$flatMap$1.apply(TraversableLike.scala:2
41)
    at scala.collection.immutable.List.foreach(List.scala:381)
    at scala.collection.TraversableLike$class.flatMap(TraversableLike.scala:241)
    at scala.collection.immutable.List.flatMap(List.scala:344)
    at org.apache.spark.sql.execution.datasources.DataSource.resolveRelation(DataSource.
scala:364)
    at org.apache.spark.sql.DataFrameReader.load(DataFrameReader.scala:149)
    at org.apache.spark.sql.DataFrameReader.load(DataFrameReader.scala:132)
    ... 48 elided
```

If `spark.sql.streaming.schemaInference` is disabled and the data source is different than `TextFileFormat`, and the input `userSpecifiedSchema` is not specified, the following `IllegalArgumentException` exception is thrown:

Schema must be specified when creating a streaming source DataFrame. If some files already exist in the directory, then depending on the file format you may be able to create a static DataFrame on that directory with 'spark.read.load(directory)' and infer schema from it.

**Caution**

**FIXME** I don't think the exception will ever happen for non-streaming sources since the schema is going to be defined earlier. When?

Eventually, it returns a `SourceInfo` with `FileSource[path]` and the schema (as calculated using the `inferFileFormatSchema` internal method).

For any other data source, it throws `UnsupportedOperationException` exception:

```
Data source [className] does not support streamed reading
```

## inferFileFormatSchema Internal Method

```
inferFileFormatSchema(format: FileFormat): StructType
```

`inferFileFormatSchema` private method computes (aka *infers*) schema (as `StructType`). It returns `userSpecifiedSchema` if specified or uses `FileFormat.inferSchema`. It throws a `AnalysisException` when is unable to infer schema.

It uses `path` option for the list of directory paths.

**Note**

It is used by `DataSource.sourceSchema` and `DataSource.createSource` when `FileFormat` is processed.

## write

**Caution**

**FIXME**

## createSource

```
createSource(metadataPath: String): Source
```

**Caution**

**FIXME**

## resolveRelation

```
resolveRelation(checkPathExist: Boolean = true): BaseRelation
```

`resolveRelation` creates a `BaseRelation` for a given `DataSource`.

Caution

[FIXME](#) What's `BaseRelation`? Why is the name?

# DataSourceRegister

`DataSourceRegister` is an interface to register [DataSources](#) under their `shortName` aliases (and [look it up](#) later).

```
package org.apache.spark.sql.sources

trait DataSourceRegister {
  def shortName(): String
}
```

It allows users to use the data source alias as the format type over the fully qualified class name.

## Custom Formats

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

See [spark-mf-format](#) project at GitHub for a complete solution.

# Standard Functions (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. Some functions are transformations of `Column` objects (or column names) into other `Column` objects or transform `DataFrame` into `DataFrame`.

The functions are grouped by functional areas:

- [Defining UDFs](#)
- [String functions](#)
  - [split](#)
  - [upper](#) (chained with `reverse`)
- [Aggregate functions](#)
- [Non-aggregate functions \(aka \*normal functions\*\)](#)
  - [struct](#)
  - [broadcast](#) (for `DataFrame`)
  - [expr](#)
- [Date time functions](#)
- [...and others](#)

**Tip** You should read the [official documentation of the functions object](#).

## window

Caution

FIXME

## 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|
+---+---+
```

## String functions

### split function

```
split(str: Column, pattern: String): Column
```

`split` function splits `str` column using `pattern`. It returns a new `column`.

Note	<code>split</code> UDF uses <a href="#">java.lang.String.split(String regex, int limit)</a> method.
------	---

```
val df = Seq((0, "hello|world"), (1, "witaj|swiecie")).toDF("num", "input")
val withSplit = df.withColumn("split", split($"input", "[|]"))

scala> withSplit.show
+---+-----+-----+
|num|      input|      split|
+---+-----+-----+
| 0| hello|world|[hello, world]|
| 1|witaj|swiecie|[witaj, swiecie]|
+---+-----+-----+
```

Note	<code>.\$ ()[{^?*+\`</code> are RegEx's meta characters and are considered special.
------	---

### upper function

```
upper(e: Column): Column
```

`upper` function converts a string column into one with all letter upper. It returns a new `Column`.

**Note**

The following example uses two functions that accept a `Column` and return another to showcase how to chain them.

```
val df = Seq((0, "hello"), (2, 3, "world"), (2, 4, "ala")).toDF("id", "val", "name")
val withUpperReversed = df.withColumn("upper", reverse(upper($"name")))

scala> withUpperReversed.show
+---+---+-----+
| id|val| name|upper|
+---+---+-----+
|  0|  1|hello|OLLEH|
|  2|  3|world|DLROW|
|  2|  4|  ala|   ALA|
+---+---+-----+
```

## Non-aggregate functions

They are also called **normal functions**.

### struct functions

```
struct(cols: Column*): Column
struct(colName: String, colNames: String*): Column
```

`struct` family of functions allows you to create a new struct column based on a collection of `Column` or their names.

**Note**

The difference between `struct` and another similar `array` function is that the types of the columns can be different (in `struct` ).

```
scala> df.withColumn("struct", struct($"name", $"val")).show
+---+---+-----+
| id|val| name|  struct|
+---+---+-----+
|  0|  1|hello|[hello,1]|
|  2|  3|world|[world,3]|
|  2|  4|  ala| [ala,4]|
+---+---+-----+
```

## broadcast function

```
broadcast[T](df: Dataset[T]): Dataset[T]
```

`broadcast` function marks the input `Dataset` small enough to be used in broadcast `join`.

Tip	Consult <a href="#">Broadcast Join</a> document.
-----	--

```
val left = Seq((0, "aa"), (0, "bb")).toDF("id", "token").as[(Int, String)]
val right = Seq(("aa", 0.99), ("bb", 0.57)).toDF("token", "prob").as[(String, Double)]

scala> left.join(broadcast(right), "token").explain(extended = true)
== Parsed Logical Plan ==
'Join UsingJoin(Inner, List('token))
:- Project [_1#42 AS id#45, _2#43 AS token#46]
: +- LocalRelation [_1#42, _2#43]
+- BroadcastHint
  +- Project [_1#55 AS token#58, _2#56 AS prob#59]
    +- LocalRelation [_1#55, _2#56]

== Analyzed Logical Plan ==
token: string, id: int, prob: double
Project [token#46, id#45, prob#59]
+- Join Inner, (token#46 = token#58)
  :- Project [_1#42 AS id#45, _2#43 AS token#46]
  : +- LocalRelation [_1#42, _2#43]
  +- BroadcastHint
    +- Project [_1#55 AS token#58, _2#56 AS prob#59]
      +- LocalRelation [_1#55, _2#56]

== Optimized Logical Plan ==
Project [token#46, id#45, prob#59]
+- Join Inner, (token#46 = token#58)
  :- Project [_1#42 AS id#45, _2#43 AS token#46]
  : +- Filter isnotnull(_2#43)
  :   +- LocalRelation [_1#42, _2#43]
  +- BroadcastHint
    +- Project [_1#55 AS token#58, _2#56 AS prob#59]
      +- Filter isnotnull(_1#55)
        +- LocalRelation [_1#55, _2#56]

== Physical Plan ==
*Project [token#46, id#45, prob#59]
+- *BroadcastHashJoin [token#46], [token#58], Inner, BuildRight
  :- *Project [_1#42 AS id#45, _2#43 AS token#46]
  : +- *Filter isnotnull(_2#43)
  :   +- LocalTableScan [_1#42, _2#43]
  +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
    +- *Project [_1#55 AS token#58, _2#56 AS prob#59]
      +- *Filter isnotnull(_1#55)
        +- LocalTableScan [_1#55, _2#56]
```

## expr function

```
expr(expr: String): Column
```

`expr` function parses the input `expr` SQL string to a `Column` it represents.

```
val ds = Seq((0, "hello"), (1, "world"))
  .toDF("id", "token")
  .as[(Long, String)]  
  
scala> ds.show
+---+-----+
| id|token|
+---+-----+
|  0|hello|
|  1|world|
+---+-----+  
  
val filterExpr = expr("token = 'hello'")  
  
scala> ds.filter(filterExpr).show
+---+-----+
| id|token|
+---+-----+
|  0|hello|
+---+-----+
```

Internally, `expr` uses the active session's `sqlParser` or creates a new `SparkSqlParser` to call `parseExpression` method.

# 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 aggregate 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 spark.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.

# UDFs — User-Defined Functions

**User-Defined Functions** (aka **UDF**) is a feature of Spark SQL to define new [Column](#)-based functions that extend the vocabulary of Spark SQL's DSL to transform [Datasets](#).

**Note** UDFs play a vital role in Spark MLlib to define new [Transformers](#).

**Tip** Use [higher-level standard column functions](#) and [Dataset operators](#) available in Spark already before reverting to using your own custom functions since Spark may or may not be able to optimize the query (e.g. for efficient [joins](#) when comparing two columns).

You define a new UDF by defining a Scala function as an input parameter of `udf` function. You can use Scala functions of up to 10 input parameters. See the section [udf Functions \(in functions object\)](#).

```
val df = Seq((0, "hello"), (1, "world")).toDF("id", "text")

// Define a "regular" Scala function
val upper: String => String = _.toUpperCase

import org.apache.spark.sql.functions.udf
val upperUDF = udf(upper)

scala> df.withColumn("upper", upperUDF('text)).show
+---+-----+
| id| text|upper|
+---+-----+
|  0|hello|HELLO|
|  1|world|WORLD|
+---+-----+
```

## udf Functions (in functions object)

```
udf[RT: TypeTag](f: Function0[RT]): UserDefinedFunction
...
udf[RT: TypeTag, A1: TypeTag, A2: TypeTag, A3: TypeTag, A4: TypeTag, A5: TypeTag, A6: TypeTag, A7: TypeTag, A8: TypeTag, A9: TypeTag, A10: TypeTag](f: Function10[A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, RT]): UserDefinedFunction
```

`org.apache.spark.sql.functions` object comes with `udf` function to let you define a UDF for a Scala function `f`.

```

val df = Seq(
  (0, "hello"),
  (1, "world")).toDF("id", "text")

// Define a "regular" Scala function
// It's a clone of upper UDF
val toUpper: String => String = _.toUpperCase

import org.apache.spark.sql.functions.udf
val upper = udf(toUpper)

scala> df.withColumn("upper", upper('text)).show
+---+-----+
| id| text|upper|
+---+-----+
|  0|hello|HELLO|
|  1|world|WORLD|
+---+-----+

// You could have also defined the UDF this way
val upperUDF = udf { s: String => s.toUpperCase }

// or even this way
val upperUDF = udf[String, String](_.toUpperCase)

scala> df.withColumn("upper", upperUDF('text)).show
+---+-----+
| id| text|upper|
+---+-----+
|  0|hello|HELLO|
|  1|world|WORLD|
+---+-----+

```

**Tip**

Define custom UDFs based on "standalone" Scala functions (e.g. `toUpperUDF`) so you can test the Scala functions using Scala way (without Spark SQL's "noise") and once they are defined reuse the UDFs in [UnaryTransformers](#).

# Window Aggregates (Windows)

**Window Aggregates** (aka **Windows**) operate on a group of rows (a row set) called a **window** to apply aggregation on. They calculate a value for every input row for its window.

**Note**

Window-based framework is available as an experimental feature since Spark **1.4.0**.

Spark SQL supports three kinds of [window aggregate function](#): **ranking** functions, **analytic** functions, and **aggregate** functions.

A [window specification](#) defines the **partitioning**, **ordering**, and **frame boundaries**.

## Window Aggregate Functions

A **window aggregate function** calculates a return value over a set of rows called **window** that are *somewhat* related to the current row.

**Note**

Window functions are also called **over functions** due to how they are applied using [Column's over function](#).

Although similar to [aggregate functions](#), a window function does not group rows into a single output row and retains their separate identities. A window function can access rows that are linked to the current row.

**Tip**

See [Examples](#) section in this document.

Spark SQL supports three kinds of window functions:

- **ranking** functions
- **analytic** functions
- **aggregate** functions

Table 1. Window functions in Spark SQL (see [Introducing Window Functions in Spark SQL](#))

	SQL	DataFrame API
<b>Ranking functions</b>	RANK	rank
	DENSE_RANK	dense_rank
	PERCENT_RANK	percent_rank
	NTILE	ntile
	ROW_NUMBER	row_number
<b>Analytic functions</b>	CUME_DIST	cume_dist
	LAG	lag
	LEAD	lead

For aggregate functions, you can use the existing [aggregate functions](#) as window functions, e.g. `sum`, `avg`, `min`, `max` and `count`.

You can mark a function `window` by `OVER` clause after a function in SQL, e.g. `avg(revenue) OVER (...)` or [over method](#) on a function in the Dataset API, e.g. `rank().over(...)`.

When executed, a window function computes a value for each row in a window.

Note	Window functions belong to <a href="#">Window functions group</a> in Spark's Scala API.
------	---

## WindowSpec - Window Specification

A window function needs a window specification which is an instance of `WindowSpec` class.

Note	<code>WindowSpec</code> class is marked as experimental since <b>1.4.0</b> .
------	--

Tip	Consult <a href="#">org.apache.spark.sql.expressions.WindowSpec</a> API.
-----	--

A **window specification** defines which rows are included in a **window** (aka a *frame*), i.e. set of rows, that is associated with a given input row. It does so by **partitioning** an entire data set and specifying **frame boundary** with **ordering**.

Note	Use static methods in <a href="#">Window object</a> to create a <code>WindowSpec</code> .
------	---

```
import org.apache.spark.sql.expressions.Window

scala> val byHTokens = Window.partitionBy('token startsWith "h")
byHTokens: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@574985d8
```

A window specification includes three parts:

1. **Partitioning Specification** defines which rows will be in the same partition for a given row.
2. **Ordering Specification** defines how 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"?](#)) defines the rows to be 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.

Once `WindowSpec` instance has been created using [Window object](#), you can further expand on window specification using the following methods to define [frames](#):

- `rowsBetween(start: Long, end: Long): WindowSpec`
- `rangeBetween(start: Long, end: Long): WindowSpec`

Besides the two above, you can also use the following methods (that correspond to the methods in [Window object](#)):

- `partitionBy`
- `orderBy`

## Window object

`Window` object provides functions to define windows (as [WindowSpec instances](#)).

`Window` object lives in `org.apache.spark.sql.expressions` package. Import it to use `Window` functions.

```
import org.apache.spark.sql.expressions.Window
```

There are two families of the functions available in `Window` object that create `WindowSpec` instance for one or many `Column` instances:

- `partitionBy`
- `orderBy`

## partitionBy

```
partitionBy(colName: String, colNames: String*): WindowSpec
partitionBy(cols: Column*): WindowSpec
```

`partitionBy` creates an instance of `WindowSpec` with partition expression(s) defined for one or more columns.

```
// partition records into two groups
// * tokens starting with "h"
// * others
val byHTokens = Window.partitionBy('token startsWith "h")  
  

// count the sum of ids in each group
val result = tokens.select('*', sum('id) over byHTokens as "sum over h tokens").orderBy(
'id)  
  

scala> .show
+---+-----+
| id|token|sum over h tokens|
+---+-----+
|  0|hello|        4|
|  1|henry|        4|
|  2|  and|        2|
|  3|harry|        4|
+---+-----+
```

## orderBy

```
orderBy(colName: String, colNames: String*): WindowSpec
orderBy(cols: Column*): WindowSpec
```

## Window 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)
```

## Frame

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

In the current implementation of [WindowSpec](#) you can use two methods to define a frame:

- rowsBetween
- rangeBetween

See [WindowSpec](#) for their coverage.

## Examples

### Top N per Group

Top N per Group is useful when you need to compute 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\_REVENU

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?

```

val dataset = 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))
  .toDF("product", "category", "revenue")

scala> dataset.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> data.where('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.

```

import org.apache.spark.sql.expressions.Window
val overCategory = Window.partitionBy('category).orderBy('revenue.desc)
val rank = dense_rank.over(overCategory)

val ranked = data.withColumn("rank", dense_rank.over(overCategory))

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('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|
+-----+-----+-----+----+

```

## Revenue Difference per Category

**Note**

This example is the 2nd example from an excellent article [Introducing Window Functions in Spark SQL](#).

```
import org.apache.spark.sql.expressions.Window
val reveDesc = Window.partitionBy('category).orderBy('revenue.desc)
val reveDiff = max('revenue).over(reveDesc) - 'revenue

scala> data.select('*', reveDiff as 'revenue_diff).show
+-----+-----+-----+
| product| category|revenue|revenue_diff|
+-----+-----+-----+
|    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|
+-----+-----+-----+
```

## Difference on Column

Compute a difference between values in rows in a column.

```

val pairs = for {
  x <- 1 to 5
  y <- 1 to 2
} yield (x, 10 * x * y)
val ds = pairs.toDF("ns", "tens")

scala> ds.show
+---+---+
| ns|tens|
+---+---+
| 1| 10|
| 1| 20|
| 2| 20|
| 2| 40|
| 3| 30|
| 3| 60|
| 4| 40|
| 4| 80|
| 5| 50|
| 5| 100|
+---+---+

import org.apache.spark.sql.expressions.Window
val overNs = Window.partitionBy('ns).orderBy('tens)
val diff = lead('tens, 1).over(overNs)

scala> ds.withColumn("diff", diff - 'tens).show
+---+---+---+
| ns|tens|diff|
+---+---+---+
| 1| 10| 10|
| 1| 20|null|
| 3| 30| 30|
| 3| 60|null|
| 5| 50| 50|
| 5| 100|null|
| 4| 40| 40|
| 4| 80|null|
| 2| 20| 20|
| 2| 40|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).

## Running Total

The **running total** is the sum of all previous lines including the current one.

```
val sales = Seq(
  (0, 0, 0, 5),
  (1, 0, 1, 3),
  (2, 0, 2, 1),
  (3, 1, 0, 2),
  (4, 2, 0, 8),
  (5, 2, 2, 8))
  .toDF("id", "orderID", "prodID", "orderQty")

scala> sales.show
+---+-----+-----+-----+
| id|orderID|prodID|orderQty|
+---+-----+-----+-----+
| 0|      0|      0|      5|
| 1|      0|      1|      3|
| 2|      0|      2|      1|
| 3|      1|      0|      2|
| 4|      2|      0|      8|
| 5|      2|      2|      8|
+---+-----+-----+-----+

val orderedByID = Window.orderBy('id)

val totalQty = sum('orderQty).over(orderedByID).as('running_total)
val salesTotalQty = sales.select('*', totalQty).orderBy('id)

scala> salesTotalQty.show
16/04/10 23:01:52 WARN Window: No Partition Defined for Window operation! Moving all data to a single partition, this can cause serious performance degradation.
+---+-----+-----+-----+
| id|orderID|prodID|orderQty|running_total|
+---+-----+-----+-----+
| 0|      0|      0|      5|      5|
| 1|      0|      1|      3|      8|
| 2|      0|      2|      1|      9|
| 3|      1|      0|      2|     11|
| 4|      2|      0|      8|     19|
| 5|      2|      2|      8|     27|
+---+-----+-----+-----+

val byOrderId = orderedByID.partitionBy('orderID)
val totalQtyPerOrder = sum('orderQty).over(byOrderId).as('running_total_per_order)
val salesTotalQtyPerOrder = sales.select('*', totalQtyPerOrder).orderBy('id)

scala> salesTotalQtyPerOrder.show
+---+-----+-----+-----+
| id|orderID|prodID|orderQty|running_total_per_order|
+---+-----+-----+-----+
```

	0	0	0	5	5
	1	0	1	3	8
	2	0	2	1	9
	3	1	0	2	2
	4	2	0	8	8
	5	2	2	8	16
+	-----+	-----+	-----+	-----+	

## 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.

## Accessing values of earlier rows

**FIXME** What's the value of rows before current one?

## Calculate rank of row

## Moving Average

## Cumulative Aggregates

Eg. cumulative sum

## 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.

## Further reading or watching

- [3.5. Window Functions](#) in the official documentation of PostgreSQL
- [Window Functions in SQL](#)
- [Working with Window Functions in SQL Server](#)
- [OVER Clause \(Transact-SQL\)](#)

- An introduction to windowed functions
- Probably the Coolest SQL Feature: Window Functions
- Window Functions

# Structured Streaming (aka Streaming Datasets)

**Structured Streaming** is a new computation model introduced in Spark 2.0.0 for building end-to-end streaming applications termed as **continuous applications**. Structured streaming offers a high-level declarative streaming API built on top of [Datasets](#) (inside Spark SQL engine) for continuous incremental execution of [structured queries](#).

The semantics of the Structured Streaming model is as follows (see the article [Structured Streaming In Apache Spark](#)):

At any time, the output of a continuous application is equivalent to executing a batch job on a prefix of the data.

Structured streaming is an attempt to unify streaming, interactive, and batch queries that paves the way for continuous applications like continuous aggregations using [groupBy](#) operator or continuous windowed aggregations using [groupBy](#) operator with [window](#) function.

Spark 2.0 aims at simplifying **streaming analytics** without having to reason about streaming at all.

The new model introduces the **streaming datasets** that are *infinite datasets* with primitives like input [streaming sources](#) and output [streaming sinks](#), **event time**, **windowing**, and **sessions**. You can specify [output mode](#) of a streaming dataset which is what gets written to a streaming sink when there is new data available.

It lives in `org.apache.spark.sql.streaming` package with the following main data abstractions:

- [StreamingQueryManager](#)
- [StreamingQuery](#)
- [Input Source](#)
- [Output Sink](#)

With Datasets being Spark SQL's view of structured data, structured streaming checks input sources for new data every [trigger](#) (time) and executes the (continuous) queries.

**Tip** Watch [SPARK-8360 Streaming DataFrames](#) to track progress of the feature.

**Tip** Read the official programming guide of Spark about [Structured Streaming](#).

## Note

The feature has also been called **Streaming Spark SQL Query**, **Streaming DataFrames**, **Continuous DataFrame** or **Continuous Query**. There have been lots of names before the Spark project settled on Structured Streaming.

## Example — Streaming Query for Running Counts (over Words from Socket with Output to Console)

## Note

The example is "borrowed" from [the official documentation of Spark](#). Changes and errors are only mine.

## Tip

You need to run `nc -l 9999` first before running the example.

```
val lines = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()

import org.apache.spark.sql.Dataset
val wordsDS: Dataset[String] = lines.as[String]

val words = wordsDS.flatMap(_.split("\\\\W+"))

scala> words.printSchema
root
 |-- value: string (nullable = true)

val wordCounts = words.groupBy("value").count

val query = wordCounts.writeStream
  .outputMode("complete")
  .format("console")
  .start // nc -l 9999 is supposed to be up at this point
```

## Example — Streaming Query over CSV Files with Output to Console Every 5 Seconds

Below you can find a complete example of a streaming query in a form of `DataFrame` of data from `csv-logs` files in `csv` format of a given schema into a `ConsoleSink` every 5 seconds.

## Tip

Copy and paste it to Spark Shell in `:paste` mode to run it.

```
// Explicit schema with nullables false
import org.apache.spark.sql.types._
val schemaExp = StructType(
  StructField("name", StringType, false) ::
```

```

StructField("city", StringType, true) ::  

StructField("country", StringType, true) ::  

StructField("age", IntegerType, true) ::  

StructField("alive", BooleanType, false) :: Nil  

)  
  

// Implicit inferred schema  

val schemaImp = spark.read  

  .format("csv")  

  .option("header", true)  

  .option("inferSchema", true)  

  .load("csv-logs")  

  .schema  
  

val in = spark.readStream  

  .schema(schemaImp)  

  .format("csv")  

  .option("header", true)  

  .option("maxFilesPerTrigger", 1)  

  .load("csv-logs")  
  

scala> in.printSchema  

root  

|-- name: string (nullable = true)  

|-- city: string (nullable = true)  

|-- country: string (nullable = true)  

|-- age: integer (nullable = true)  

|-- alive: boolean (nullable = true)  
  

println("Is the query streaming" + in.isStreaming)  
  

println("Are there any streaming queries?" + spark.streams.active.isEmpty)  
  

import scala.concurrent.duration._  

import org.apache.spark.sql.streaming.ProcessingTime  

import org.apache.spark.sql.streaming.OutputMode.Append  

val out = in.writeStream  

  .format("console")  

  .trigger(ProcessingTime(5.seconds))  

  .queryName("consoleStream")  

  .outputMode(Append)  

  .start()  
  

16/07/13 12:32:11 TRACE FileStreamSource: Listed 3 file(s) in 4.274022 ms  

16/07/13 12:32:11 TRACE FileStreamSource: Files are:  

  file:///Users/jacek/dev/oss/spark/csv-logs/people-1.csv  

  file:///Users/jacek/dev/oss/spark/csv-logs/people-2.csv  

  file:///Users/jacek/dev/oss/spark/csv-logs/people-3.csv  

16/07/13 12:32:11 DEBUG FileStreamSource: New file: file:///Users/jacek/dev/oss/spark/  

csv-logs/people-1.csv  

16/07/13 12:32:11 TRACE FileStreamSource: Number of new files = 3  

16/07/13 12:32:11 TRACE FileStreamSource: Number of files selected for batch = 1  

16/07/13 12:32:11 TRACE FileStreamSource: Number of seen files = 1

```

```

16/07/13 12:32:11 INFO FileStreamSource: Max batch id increased to 0 with 1 new files
16/07/13 12:32:11 INFO FileStreamSource: Processing 1 files from 0:0
16/07/13 12:32:11 TRACE FileStreamSource: Files are:
    file:///Users/jacek/dev/oss/spark/csv-logs/people-1.csv
-----
Batch: 0
-----
+---+---+---+---+
| name| city|country|age|alive|
+---+---+---+---+
| Jacek| Warszawa| Polska| 42| true|
+---+---+---+---+
spark.streams
  .active
  .foreach(println)
// Streaming Query - consoleStream [state = ACTIVE]

scala> spark.streams.active(0).explain
== Physical Plan ==
*Scan csv [name#130,city#131,country#132,age#133,alive#134] Format: CSV, InputPaths: f
ile:/Users/jacek/dev/oss/spark/csv-logs/people-3.csv, PushedFilters: [], ReadSchema: s
truct<name:string,city:string,country:string,age:int,alive:boolean>

```

## Further reading or watching

- [Structured Streaming In Apache Spark](#)
- (video) [The Future of Real Time in Spark](#) from Spark Summit East 2016 in which Reynold Xin presents the concept of **Streaming DataFrames** to the public.
- (video) [Structuring Spark: DataFrames, Datasets, and Streaming](#)
- [What Spark's Structured Streaming really means](#)
- (video) [A Deep Dive Into Structured Streaming](#) by Tathagata "TD" Das from Spark Summit 2016

# DataStreamReader

`DataStreamReader` is an interface for [reading](#) streaming data in [DataFrame](#) from data sources with specified [format](#), [schema](#) and [options](#).

`DataStreamReader` offers support for the built-in formats: [json](#), [csv](#), [parquet](#), [text](#). [parquet](#) format is the default data source as configured using [spark.sql.sources.default](#) setting.

`DataStreamReader` is available using [SparkSession.readStream](#) method.

```
val spark: SparkSession = ...  
  
val schema = spark.read  
  .format("csv")  
  .option("header", true)  
  .option("inferSchema", true)  
  .load("csv-logs/*.csv")  
  .schema  
  
val df = spark.readStream  
  .format("csv")  
  .schema(schema)  
  .load("csv-logs/*.csv")
```

## format

```
format(source: String): DataStreamReader
```

`format` specifies the `source` format of the streaming data source.

## schema

```
schema(schema: StructType): DataStreamReader
```

`schema` specifies the `schema` of the streaming data source.

## option Methods

```
option(key: String, value: String): DataStreamReader
option(key: String, value: Boolean): DataStreamReader
option(key: String, value: Long): DataStreamReader
option(key: String, value: Double): DataStreamReader
```

`option` family of methods specifies additional options to a streaming data source.

There is support for values of `String`, `Boolean`, `Long`, and `Double` types for user convenience, and internally are converted to `String` type.

Note

You can also set options in bulk using [options](#) method. You have to do the type conversion yourself, though.

## options

```
options(options: scala.collection.Map[String, String]): DataStreamReader
```

`options` method allows specifying one or many options of the streaming input data source.

Note

You can also set options one by one using [option](#) method.

## load Methods

```
load(): DataFrame
load(path: String): DataFrame (1)
```

1. Specifies `path` option before passing calls to `load()`

`load` loads streaming input data as [DataFrame](#).

Internally, `load` creates a `DataFrame` from the current [SparkSession](#) and a [StreamingRelation](#) (of a [DataSource](#) based on `schema`, `format`, and `options`).

## Built-in Formats

```
json(path: String): DataFrame
csv(path: String): DataFrame
parquet(path: String): DataFrame
text(path: String): DataFrame
```

`DataStreamReader` can load streaming data from data sources of the following [formats](#):

- `json`
- `csv`
- `parquet`
- `text`

The methods simply pass calls to `format` followed by `load(path)`.

# DataStreamWriter

Caution

FIXME

`DataFrameWriter` is a part of Structured Streaming API.

```
val df: DataFrame = ...

import org.apache.spark.sql.streaming.ProcessingTime
import scala.concurrent.duration._

df.writeStream
  .queryName("textStream")
  .trigger(ProcessingTime(10.seconds))
  .format("console")
  .start
```

- `trigger` to set the Trigger for a stream query.
- `queryName`
- `startStream` to start a continuous write.

## Data Streams (startStream methods)

`DataFrameWriter` comes with two `startStream` methods to return a `StreamingQuery` object to continually write data.

```
startStream(): StreamingQuery
startStream(path: String): StreamingQuery (1)
```

1. Sets `path` option to `path` and calls `startStream()`

Note

`startStream` uses `StreamingQueryManager.startQuery` to create `StreamingQuery`.

Note

Whether or not you have to specify `path` option depends on the `DataSource` in use.

Recognized options:

- `queryName` is the name of active streaming query.
- `checkpointLocation` is the directory for checkpointing.

Note	Define options using <a href="#">option</a> or <a href="#">options</a> methods.
------	---

Note	It is a new feature of Spark <b>2.0.0</b> .
------	---

## Specifying Output Mode (`outputMode` method)

```
outputMode(outputMode: OutputMode): DataStreamWriter[T]
```

`outputMode` specifies **output mode** of a streaming [Dataset](#) which is what gets written to a [streaming sink](#) when there is a new data available.

Currently, the following output modes are supported:

- `OutputMode.Append` — only the new rows in the streaming dataset will be written to a sink.
- `OutputMode.Complete` — entire streaming dataset (with all the rows) will be written to a sink every time there are updates. It is supported only for streaming queries with aggregations.

## queryName

```
queryName(queryName: String): DataStreamWriter[T]
```

`queryName` sets the name of a [streaming query](#).

Internally, it is just an additional [option](#) with the key `queryName`.

## trigger

```
trigger(trigger: Trigger): DataStreamWriter[T]
```

`trigger` method sets the time interval of the **trigger** (batch) for a streaming query.

Note	<code>Trigger</code> specifies how often results should be produced by a <a href="#">StreamingQuery</a> . See <a href="#">Trigger</a> .
------	--

The default trigger is [ProcessingTime\(0L\)](#) that runs a streaming query as often as possible.

Tip	Consult <a href="#">Trigger</a> to learn about <code>Trigger</code> and <code>ProcessingTime</code> types.
-----	--

## start methods

```
start(path: String): StreamingQuery  
start(): StreamingQuery
```

## foreach

# Streaming Sources

A **Streaming Source** represents a continuous stream of data for a streaming query. It generates batches of [DataFrame](#) for given start and end offsets. For fault tolerance, a source must be able to replay data given a start offset.

A streaming source should be able to replay an arbitrary sequence of past data in the stream using a range of offsets. This means that only streaming sources like Kafka and Kinesis (which have the concept of per-record offset) fit into this model. This is the assumption so structured streaming can achieve end-to-end exactly-once guarantees.

`Source` trait has the following features:

- **schema** of the data (as [StructType](#) using `schema` method)
- the maximum **offset** (of type `Offset` using `getOffset` method)
- Returns a **batch** for start and end offsets (of type [DataFrame](#)).

It lives in `org.apache.spark.sql.execution.streaming` package.

```
import org.apache.spark.sql.execution.streaming.Source
```

There are two available `Source` implementations:

- [FileStreamSource](#)
- [MemoryStream](#)

## MemoryStream

# FileStreamSource

`FileStreamSource` is a [Source](#) that reads text files from `path` directory as they appear. It uses `LongOffset` offsets.

**Note**

It is used by [DataSource.createSource](#) for `FileFormat`.

You can provide the `schema` of the data and `dataFrameBuilder` - the function to build a `DataFrame` in [getBatch](#) at instantiation time.

```
// NOTE The source directory must exist
// mkdir text-logs

val df = spark.readStream
  .format("text")
  .option("maxFilesPerTrigger", 1)
  .load("text-logs")

scala> df.printSchema
root
 |-- value: string (nullable = true)
```

Batches are indexed.

It lives in `org.apache.spark.sql.execution.streaming` package.

It tracks already-processed files in `seenFiles` hash map.

**Tip**

Enable `DEBUG` or `TRACE` logging level for `org.apache.spark.sql.execution.streaming.FileStreamSource` to see what happens inside.

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

```
log4j.logger.org.apache.spark.sql.execution.streaming.FileStreamSource=TRACE
```

Refer to [Logging](#).

## Options

### maxFilesPerTrigger

`maxFilesPerTrigger` option specifies the maximum number of files per trigger (batch). It limits the file stream source to read the `maxFilesPerTrigger` number of files specified at a time and hence enables rate limiting.

It allows for a static set of files be used like a stream for testing as the file set is processed `maxFilesPerTrigger` number of files at a time.

## schema

If the schema is specified at instantiation time (using optional `dataSchema` constructor parameter) it is returned.

Otherwise, `fetchAllFiles` internal method is called to list all the files in a directory.

When there is at least one file the schema is calculated using `dataFrameBuilder` constructor parameter function. Else, an `IllegalArgumentException("No schema specified")` is thrown unless it is for **text** provider (as `providerName` constructor parameter) where the default schema with a single `value` column of type `StringType` is assumed.

Note

**text** as the value of `providerName` constructor parameter denotes **text file stream provider**.

## getOffset

The maximum offset (`getOffset`) is calculated by fetching all the files in `path` excluding files that start with `_` (underscore).

When computing the maximum offset using `getOffset`, you should see the following DEBUG message in the logs:

```
DEBUG Listed ${files.size} in ${(endTime.toDouble - startTime) / 1000000}ms
```

When computing the maximum offset using `getOffset`, it also filters out the files that were already seen (tracked in `seenFiles` internal registry).

You should see the following DEBUG message in the logs (depending on the status of a file):

```
new file: $file
// or
old file: $file
```

## getBatch

`FileStreamSource.getBatch` asks `metadataLog` for the batch.

You should see the following INFO and DEBUG messages in the logs:

```
INFO Processing ${files.length} files from ${startId + 1}:$endId  
DEBUG Streaming ${files.mkString(", ")}
```

The method to create a result batch is given at instantiation time (as `dataFrameBuilder` constructor parameter).

## metadataLog

`metadataLog` is a metadata storage using `metadataPath` path (which is a constructor parameter).

Note	It extends <code>HDFSMetadataLog[Seq[String]]</code> .
Caution	<a href="#">FIXME</a> Review <code>HDFSMetadataLog</code>

# Streaming Sinks

A **Streaming Sink** represents an external storage to write streaming datasets to. It is modeled as `Sink` trait that can [process batches](#) of data given as [DataFrames](#).

The following sinks are currently available in Spark:

- [ConsoleSink](#) for `console` format.
- [FileStreamSink](#) for `parquet` format.
- [ForeachSink](#) used in `foreach` operator.
- [MemorySink](#) for `memory` format.

You can create your own streaming format implementing [StreamSinkProvider](#).

## Sink Contract

Sink Contract is described by `Sink` trait. It defines the one and only `addBatch` method to add `data as batchId`.

```
package org.apache.spark.sql.execution.streaming

trait Sink {
  def addBatch(batchId: Long, data: DataFrame): Unit
}
```

## FileStreamSink

`FileStreamSink` is the streaming sink for the `parquet` format.

Caution	<a href="#">FIXME</a>
---------	-----------------------

```
import scala.concurrent.duration._
import org.apache.spark.sql.streaming.{OutputMode, ProcessingTime}
val out = in.writeStream
  .format("parquet")
  .option("path", "parquet-output-dir")
  .option("checkpointLocation", "checkpoint-dir")
  .trigger(ProcessingTime(5.seconds))
  .outputMode(OutputMode.Append)
  .start()
```

`FileStreamSink` supports `Append` output mode only.

It uses `spark.sql.streaming.fileSink.log.deletion` (as `isDeletingExpiredLog`)

## MemorySink

`MemorySink` is an memory-based `Sink` particularly useful for testing. It stores the results in memory.

It is available as `memory` format that requires a query name (by `queryName` method or `queryName` option).

...FIXME

Note	It was introduced in the <a href="#">pull request for [SPARK-14288][SQL] Memory Sink for streaming</a> .
------	--

Use `toDebugString` to see the batches.

Its aim is to allow users to test streaming applications in the Spark shell or other local tests.

You can set `checkpointLocation` using `option` method or it will be set to `spark.sql.streaming.checkpointLocation` setting.

If `spark.sql.streaming.checkpointLocation` is set, the code uses `$location/$queryName` directory.

Finally, when no `spark.sql.streaming.checkpointLocation` is set, a temporary directory `memory.stream` under `java.io.tmpdir` is used with `offsets` subdirectory inside.

Note	The directory is cleaned up at shutdown using <code>ShutdownHookManager.registerShutdownDeleteDir</code> .
------	--

```
val nums = (0 to 10).toDF("num")

scala> val outStream = nums.write
       .format("memory")
       .queryName("memStream")
       .startStream()
16/04/11 19:37:05 INFO HiveSqlParser: Parsing command: memStream
outStream: org.apache.spark.sql.StreamingQuery = Continuous Query - memStream [state = ACTIVE]
```

It creates `MemorySink` instance based on the schema of the DataFrame it operates on.

It creates a new DataFrame using `MemoryPlan` with `MemorySink` instance created earlier and registers it as a temporary table (using `DataFrame.registerTempTable` method).

Note

At this point you can query the table as if it were a regular non-streaming table using `sql` method.

A new `StreamingQuery` is started (using `StreamingQueryManager.startQuery`) and returned.

Caution

**FIXME** Describe `else` part.

# ConsoleSink

`ConsoleSink` is a streaming sink that is registered as the `console` format.

```
val spark: SparkSession = ...
spark.readStream
  .format("text")
  .load("server-logs/*.out")
  .as[String]
  .writeStream
  .queryName("server-logs processor")
  .format("console") // <-- uses ConsoleSink
  .start

scala> spark.streams.active.foreach(println)
Streaming Query - server-logs processor [state = ACTIVE]

// in another terminal
$ echo hello > server-logs/hello.out

// in the terminal with Spark
-----
Batch: 0
-----
+---+
| value|
+---+
|hello|
+---+
```

# ConsoleSinkProvider

`ConsoleSinkProvider` is a `StreamSinkProvider` for `ConsoleSink`. As a `DataSourceRegister`, it registers the `ConsoleSink` streaming sink as `console` format.

# ForeachSink

`ForeachSink` is a typed [Sink](#) that passes records (of the type `T`) to [ForeachWriter](#) (one record at a time per partition).

It is used exclusively in [foreach](#) operator.

```
val records = spark.readStream
  .format("text")
  .load("server-logs/*.out")
  .as[String]

import org.apache.spark.sql.ForeachWriter
val writer = new ForeachWriter[String] {
  override def open(partitionId: Long, version: Long) = true
  override def process(value: String) = println(value)
  override def close(errorOrNull: Throwable) = {}
}

records.writeStream
  .queryName("server-logs processor")
  .foreach(writer)
  .start
```

Internally, `addBatch` (the only method from the [Sink Contract](#)) takes records from the input [DataFrame](#) (as `data`), transforms them to expected type `T` (of this `ForeachSink`) and (now as a [Dataset](#)) processes each partition.

```
addBatch(batchId: Long, data: DataFrame): Unit
```

It then opens the constructor's [ForeachWriter](#) (for the [current partition](#) and the input batch) and passes the records to process (one at a time per partition).

Caution	<a href="#">FIXME</a> Why does Spark track whether the writer failed or not? Why couldn't it <code>finally</code> and do <code>close</code> ?
---------	---

Caution	<a href="#">FIXME</a> Can we have a constant for <code>"foreach"</code> for <code>source</code> in <code>DataStreamWriter</code> ?
---------	--

# ForeachWriter

Caution	<a href="#">FIXME</a>
---------	-----------------------



# StreamSourceProvider

Caution	<a href="#">FIXME</a>
---------	-----------------------

# StreamSinkProvider

`StreamSinkProvider` is an interface for objects that can create [streaming sinks](#) for a specific format or system, e.g. [ConsoleSinkProvider](#) for `console` format.

It defines the one and only method `createSink` that creates a [Sink](#).

```
package org.apache.spark.sql.sources

trait StreamSinkProvider {
  def createSink(
    sqlContext: SQLContext,
    parameters: Map[String, String],
    partitionColumns: Seq[String],
    outputMode: OutputMode): Sink
}
```

# StreamingQueryManager — Streaming Query Management

**Note**

`StreamingQueryManager` is an experimental feature of Spark 2.0.0.

A `streamingQueryManager` is the Management API for [continuous queries](#) per `SQLContext`.

**Note**

There is a single `StreamingQueryManager` instance per `SQLContext` session.

You can access `StreamingQueryManager` for the current `SQLContext` using `SQLContext.streams` method. It is lazily created when a `SQLContext` instance starts.

```
val queries = spark.streams
```

## Initialization

`StreamingQueryManager` manages the following instances:

- `StateStoreCoordinatorRef` (as `stateStoreCoordinator`)
- [StreamingQueryListenerBus](#) (as `listenerBus`)
- `activeQueries` which is a mutable mapping between query names and `StreamingQuery` objects.

## StreamingQueryListenerBus

**Caution**[FIXME](#)

### startQuery

```
startQuery(name: String,
checkpointLocation: String,
df: DataFrame,
sink: Sink,
trigger: Trigger = ProcessingTime(0)): StreamingQuery
```

`startQuery` is a `private[sql]` method to start a [StreamingQuery](#).

**Note**

It is called exclusively by [DataFrameWriter.startStream](#).

Note	By default, <code>trigger</code> is <code>ProcessingTime(0)</code> .
------	--

`startQuery` makes sure that `activeQueries` internal registry does not contain the query under `name`. It throws an `IllegalArgumentException` if it does.

It transforms the `LogicalPlan` of the input DataFrame `df` so all `StreamingRelation` "nodes" become `StreamingExecutionRelation`. It uses `DataSource.createSource(metadataPath)` where `metadataPath` is `$checkpointLocation/sources/$nextSourceId`. Otherwise, it returns the `LogicalPlan` untouched.

It finally creates `StreamExecution` and starts it. It also registers the `StreamExecution` instance in `activeQueries` internal registry.

## Return All Active Continuous Queries per SQLContext

```
active: Array[StreamingQuery]
```

`active` method returns a collection of `StreamingQuery` instances for the current `SQLContext`.

## Getting Active Continuous Query By Name

```
get(name: String): StreamingQuery
```

`get` method returns a `StreamingQuery` by `name`.

It may throw an `IllegalArgumentException` when no `StreamingQuery` exists for the `name`.

```
java.lang.IllegalArgumentException: There is no active query with name hello
  at org.apache.spark.sql.StreamingQueryManager$$anonfun$get$1.apply(StreamingQueryManager.scala:59)
  at org.apache.spark.sql.StreamingQueryManager$$anonfun$get$1.apply(StreamingQueryManager.scala:59)
  at scala.collection.MapLike$class.getOrElse(MapLike.scala:128)
  at scala.collection.AbstractMap.getOrElse(Map.scala:59)
  at org.apache.spark.sql.StreamingQueryManager.get(StreamingQueryManager.scala:58)
  ... 49 elided
```

## StreamingQueryListener Management - Adding or Removing Listeners

- `addListener(listener: StreamingQueryListener): Unit` adds `listener` to the internal `listenerBus`.
- `removeListener(listener: StreamingQueryListener): Unit` removes `listener` from the internal `listenerBus`.

## postListenerEvent

```
postListenerEvent(event: StreamingQueryListener.Event): Unit
```

`postListenerEvent` posts a `StreamingQueryListener.Event` to `listenerBus`.

## StreamingQueryListener

Caution	<a href="#">FIXME</a>
---------	-----------------------

`StreamingQueryListener` is an interface for listening to query life cycle events, i.e. a query start, progress and termination events.

## lastTerminatedQuery - internal barrier

Caution	<a href="#">FIXME</a> Why is <code>lastTerminatedQuery</code> needed?
---------	---

Used in:

- `awaitAnyTermination`
- `awaitAnyTermination(timeoutMs: Long)`

They all wait `10` millis before doing the check of `lastTerminatedQuery` being non-null.

It is set in:

- `resetTerminated()` resets `lastTerminatedQuery`, i.e. sets it to `null`.
- `notifyQueryTermination(terminatedQuery: StreamingQuery)` sets `lastTerminatedQuery` to be `terminatedQuery` and notifies all the threads that wait on `awaitTerminationLock`.

It is called from [StreamExecution.runBatches](#).

# StreamingQuery

`StreamingQuery` provides an interface for interacting with a query that executes continually in background.

Note	<code>StreamingQuery</code> is called <b>continuous query</b> or <b>stream query</b> .
------	--

A `StreamingQuery` has a name. It belongs to a single `SQLContext`.

Note	<code>StreamingQuery</code> is a Scala trait with the only implementation being <a href="#">StreamExecution</a>
------	---

It can be in two states: active (started) or inactive (stopped). If inactive, it may have transitioned into the state due to an `StreamingQueryException` (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 `StreamingQuery` with many `Source's.

`StreamingQuery` can be stopped by `stop` or an exception.

# Trigger

`Trigger` is used to define how often a [streaming query](#) should be executed to produce results.

## Note

`Trigger` is a `sealed trait` so all available implementations are in the same file [Trigger.scala](#).

## Note

A trigger can also be considered a batch (as in Spark Streaming).

Import `org.apache.spark.sql` to work with `Trigger` and the only implementation [ProcessingTime](#).

```
import org.apache.spark.sql._
```

## Note

It was introduced in [the commit for \[SPARK-14176\]\[SQL\] Add DataFrameWriter.trigger to set the stream batch period](#).

# ProcessingTime

`ProcessingTime` is the only available implementation of `Trigger` sealed trait. It assumes that milliseconds is the minimum time unit.

You can create an instance of `ProcessingTime` using the following constructors:

- `ProcessingTime(Long)` that accepts non-negative values that represent milliseconds.

```
ProcessingTime(10)
```

- `ProcessingTime(interval: String)` or `ProcessingTime.create(interval: String)` that accept `CalendarInterval` instances with or without leading `interval` string.

```
ProcessingTime("10 milliseconds")
ProcessingTime("interval 10 milliseconds")
```

- `ProcessingTime(Duration)` that accepts `scala.concurrent.duration.Duration` instances.

```
ProcessingTime(10.seconds)
```

- `ProcessingTime.create(interval: Long, unit: TimeUnit)` for `Long` and `java.util.concurrent.TimeUnit` instances.

```
ProcessingTime.create(10, TimeUnit.SECONDS)
```

# 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.

`StreamExecution` is a `StreamingQuery` with additional attributes:

- `checkpointRoot`
- `LogicalPlan`
- `Sink`
- `Trigger`

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.

Tip	<p>Enable <code>DEBUG</code> logging level for <code>org.apache.spark.sql.execution.streaming.StreamExecution</code> to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.execution.streaming.StreamExecution=DEBUG</pre> <p>Refer to <a href="#">Logging</a>.</p>
-----	--

## runBatches

```

scala> val out = in.write
     .format("memory")
     .queryName("memStream")
     .startStream()
out: org.apache.spark.sql.StreamingQuery = Continuous Query - memStream [state = ACTIVE]
16/04/16 00:48:47 INFO StreamExecution: Starting new continuous query.

scala> 16/04/16 00:48:47 INFO StreamExecution: Committed offsets for batch 1.
16/04/16 00:48:47 DEBUG StreamExecution: Stream running from {} to {FileSource[hello]: #0}
16/04/16 00:48:47 DEBUG StreamExecution: Retrieving data from FileSource[hello]: None
-> #0
16/04/16 00:48:47 DEBUG StreamExecution: Optimized batch in 163.940239ms
16/04/16 00:48:47 INFO StreamExecution: Completed up to {FileSource[hello]: #0} in 703
.573981ms

```

## toDebugString

You can call `toDebugString` on `StreamExecution` to learn about the internals.

```

scala> out.asInstanceOf[StreamExecution].toDebugString
res3: String =
"
== Continuous Query ==
Name: memStream
Current Offsets: {FileSource[hello]: #0}

Current State: ACTIVE
Thread State: RUNNABLE

Logical Plan:
FileSource[hello]

"

```

# StreamingRelation

`StreamingRelation` is the [LogicalPlan](#) of the `DataFrame` being the result of executing `DataFrameReader.stream` method.

```
val reader = spark.read
val helloDF = reader.stream("hello")

scala> helloDF.explain(true)
== Parsed Logical Plan ==
FileSource[hello]
== Analyzed Logical Plan ==
id: bigint
FileSource[hello]
== Optimized Logical Plan ==
FileSource[hello]
== Physical Plan ==
java.lang.AssertionError: assertion failed: No plan for FileSource[hello]
```

# StreamingExecutionRelation

# StreamingQueryListenerBus

Caution	<a href="#">FIXME</a>
---------	-----------------------

# Joins

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

## Broadcast Join (aka Map-Side Join)

Caution	FIXME: Review <a href="#">BroadcastNestedLoop</a> .
---------	---

You can use `broadcast` function to mark a [Dataset](#) to be broadcast when used in a `join` operator.

Note	According to the article <a href="#">Map-Side Join in Spark</a> , <b>broadcast join</b> is also called a <b>replicated join</b> (in the distributed system community) or a <b>map-side join</b> (in the Hadoop community).
------	--

Note	At long last! I have always been wondering what a map-side join is and it appears I am close to uncover the truth!
------	--

And later in the article [Map-Side Join in Spark](#), you can find that with the broadcast join, you can very effectively join a large table (fact) with relatively small tables (dimensions), i.e. to perform a **star-schema join** you can avoid sending all data of the large table over the network.

`CanBroadcast` object matches a [LogicalPlan](#) with output small enough for broadcast join.

Note	Currently statistics are only supported for Hive Metastore tables where the command <code>ANALYZE TABLE [tableName] COMPUTE STATISTICS noscan</code> has been run.
------	--

It uses `spark.sql.autoBroadcastJoinThreshold` setting to control the size of a table that will be broadcast to all worker nodes when performing a join.

# SQLConf

`SQLConf` is a key-value configuration store for [parameters](#) and [hints](#) used in Spark SQL. It offers methods to [get](#), [set](#), [unset](#) or [clear](#) their values.

You can access the current `SQLConf` using [sparkSession.conf](#).

Note	<code>SQLConf</code> is a <code>private[sql]</code> serializable class in <code>org.apache.spark.sql.internal</code> package.
------	---

## Getting Parameters and Hints

You can get the current parameters and hints using the following family of `get` methods.

```
getConfString(key: String): String  
getConf[T](entry: ConfigEntry[T], defaultValue: T): T  
getConf[T](entry: ConfigEntry[T]): T  
getConf[T](entry: OptionalConfigEntry[T]): Option[T]  
getConfString(key: String, defaultValue: String): String  
getAllConfs: immutable.Map[String, String]  
getAllDefinedConfs: Seq[(String, String, String)]
```

## Setting Parameters and Hints

You can set parameters and hints using the following family of `set` methods.

```
setConf(props: Properties): Unit  
setConfString(key: String, value: String): Unit  
setConf[T](entry: ConfigEntry[T], value: T): Unit
```

## Unsetting Parameters and Hints

You can unset parameters and hints using the following family of `unset` methods.

```
unsetConf(key: String): Unit  
unsetConf(entry: ConfigEntry[_]): Unit
```

## Clearing All Parameters and Hints

```
clear(): Unit
```

You can use `clear` to remove all the parameters and hints in `SQLConf`.

## Parameters and Hints

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

### **spark.sql.streaming.fileSink.log.deletion**

`spark.sql.streaming.fileSink.log.deletion` (default: `true`) is an internal flag to control whether to delete the expired log files in [file stream sink](#).

### **spark.sql.streaming.fileSink.log.compactInterval**

`spark.sql.streaming.fileSink.log.compactInterval`

### **spark.sql.streaming.fileSink.log.cleanupDelay**

`spark.sql.streaming.fileSink.log.cleanupDelay`

### **spark.sql.streaming.schemaInference**

`spark.sql.streaming.schemaInference`

# Catalog

`Catalog` is the interface to work with database(s), local and external tables, functions, table columns, and temporary views in Spark.

It is available as `SparkSession.catalog` attribute.

```
scala> spark
res1: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@4daee083

scala> spark.catalog
res2: org.apache.spark.sql.catalog.Catalog = org.apache.spark.sql.internal.CatalogImpl@1b42eb0f

scala> spark.catalog.listTables.show
+-----+-----+-----+-----+
|       name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|my_permanent_table| default|      null|  MANAGED|    false|
|          strs|     null|      null|TEMPORARY|     true|
+-----+-----+-----+-----+
```

The one and only implementation of the `Catalog` contract is `CatalogImpl`.

## Catalog Contract

```
package org.apache.spark.sql.catalog

abstract class Catalog {
    def currentDatabase: String
    def setCurrentDatabase(dbName: String): Unit
    def listDatabases(): Dataset[Database]
    def listTables(): Dataset[Table]
    def listTables(dbName: String): Dataset[Table]
    def listFunctions(): Dataset[Function]
    def listFunctions(dbName: String): Dataset[Function]
    def listColumns(tableName: String): Dataset[Column]
    def listColumns(dbName: String, tableName: String): Dataset[Column]
    def createExternalTable(tableName: String, path: String): DataFrame
    def createExternalTable(tableName: String, path: String, source: String): DataFrame
    def createExternalTable(
        tableName: String,
        source: String,
        options: Map[String, String]): DataFrame
    def createExternalTable(
        tableName: String,
        source: String,
        schema: StructType,
        options: Map[String, String]): DataFrame
    def dropTempView(viewName: String): Unit
    def isCached(tableName: String): Boolean
    def cacheTable(tableName: String): Unit
    def uncacheTable(tableName: String): Unit
    def clearCache(): Unit
    def refreshTable(tableName: String): Unit
    def refreshByPath(path: String): Unit
}
```

## CatalogImpl

`CatalogImpl` is the one and only `Catalog` that relies on a per-session `SessionCatalog` (through `SparkSession`) to obey the `Catalog contract`.

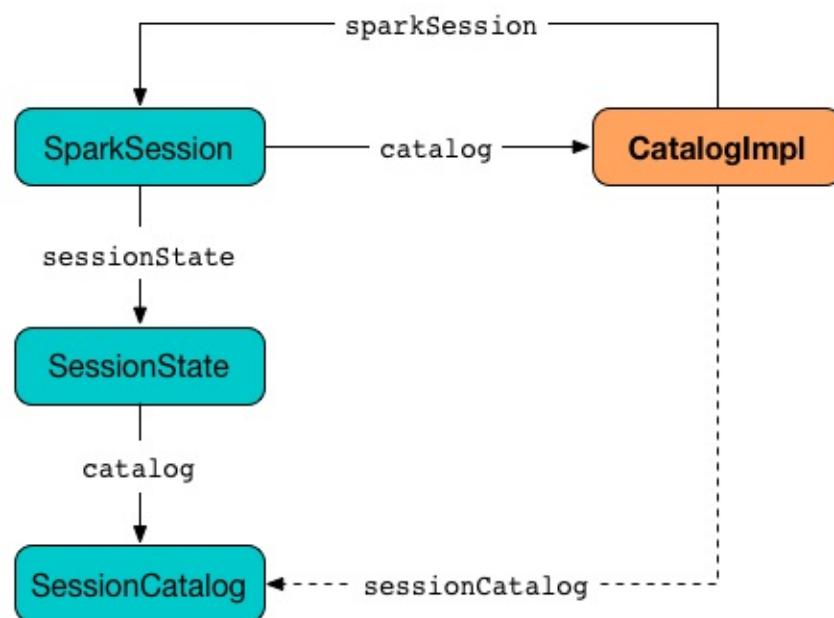


Figure 1. CatalogImpl uses SessionCatalog (through SparkSession)

It lives in `org.apache.spark.sql.internal` package.

# 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.

# SessionState

`SessionState` is the state separation layer between sessions, including SQL configuration, tables, functions, UDFs, the SQL parser, and everything else that depends on a `SQLConf`.

It uses a `SparkSession` and manages its own `SQLConf`.

Note	Given the package <code>org.apache.spark.sql.internal</code> that <code>SessionState</code> belongs to, this one is truly <i>internal</i> . You've been warned.
------	---

Note	<code>SessionState</code> is a <code>private[sql]</code> class.
------	---

`SessionState` offers the following services:

- optimizer
- analyzer
- catalog
- streamingQueryManager
- udf
- newHadoopConf to create a new Hadoop's `Configuration`.
- sessionState
- sqlParser
- executePlan

## catalog Attribute

<code>catalog: SessionCatalog</code>
--------------------------------------

`catalog` attribute points at shared internal `SessionCatalog` for managing tables and databases.

It is used to create the shared `analyzer`, `optimizer`

## SessionCatalog

`SessionCatalog` is a proxy between [SparkSession](#) and the underlying metastore, e.g. `HiveSessionCatalog`.

## analyzer Attribute

```
analyzer: Analyzer
```

`analyzer` is the [Analyzer](#) for the current Spark SQL session.

## optimizer Attribute

```
optimizer: Optimizer
```

`optimizer` is a [Catalyst optimizer](#) for logical query plans.

It is (lazily) set to [SparkOptimizer](#). It is created for the session-owned [SessionCatalog](#), [SQLConf](#), and `ExperimentalMethods` (as defined in [experimentalMethods](#) attribute).

## experimentalMethods

`experimentalMethods` is...

## sqlParser Attribute

`sqlParser` is...

## planner method

`planner` is the [SparkPlanner](#) for the current session.

Whenever called, `planner` returns a new `SparkPlanner` instance with the [SparkContext](#) of the current [SparkSession](#), the [SQLConf](#), and a collection of extra [SparkStrategies](#) (via [experimentalMethods](#) attribute).

## executePlan method

```
executePlan(plan: LogicalPlan): QueryExecution
```

`executePlan` executes the input [LogicalPlan](#) to produce a [QueryExecution](#) in the current [SparkSession](#).

## refreshTable method

`refreshTable` is...

## addJar method

`addJar` is...

## analyze method

`analyze` is...

## streamingQueryManager Attribute

`streamingQueryManager: StreamingQueryManager`

`streamingQueryManager` attribute points at shared [StreamingQueryManager](#) (e.g. to [start streaming queries in DataStreamWriter](#) ).

## udf Attribute

`udf: UDFRegistration`

`udf` attribute points at shared `UDFRegistration` for a given Spark session.

## Creating New Hadoop Configuration (newHadoopConf method)

`newHadoopConf(): Configuration`

`newHadoopConf` returns Hadoop's `Configuration` that it builds using [SparkContext.hadoopConfiguration](#) (through [SparkSession](#)) with all configuration settings added.

**Note**

`newHadoopConf` is used by [HiveSessionState](#) (for `HiveSessionCatalog` ), `ScriptTransformation` , `ParquetRelation` , `StateStoreRDD` , and `SessionState` itself, and few other places.

**Caution**

**FIXME** What is `ScriptTransformation` ? `StateStoreRDD` ?



# SQL Parser Framework

**SQL Parser Framework** in Spark SQL uses ANTLR to parse a SQL text and then creates [data types](#), Catalyst's `Expression`, `TableIdentifier`, and [LogicalPlan](#).

The contract of the SQL Parser Framework is described by [ParserInterface](#) interface. The contract is then abstracted in [AbstractSqlParser](#) class so subclasses have only to provide custom [AstBuilder](#).

There are two concrete implementations of `AbstractSqlParser`:

1. [SparkSqlParser](#) that is the default parser of the SQL expressions into Spark's types.
2. [CatalystSqlParser](#) that is used to parse data types from their canonical string representation.

## ParserInterface — SQL Parser Contract

`ParserInterface` is the parser contract for extracting [LogicalPlan](#), Catalyst `Expressions` (to create [Columns](#) from), and `TableIdentifiers` from a given SQL string.

```
package org.apache.spark.sql.catalyst.parser

trait ParserInterface {
    def parsePlan(sqlText: String): LogicalPlan

    def parseExpression(sqlText: String): Expression

    def parseTableIdentifier(sqlText: String): TableIdentifier
}
```

It has the only single abstract subclass [AbstractSqlParser](#).

## AbstractSqlParser

`AbstractSqlParser` abstract class is a [ParserInterface](#) that provides the foundation for the SQL parsing infrastructure in Spark SQL with two concrete implementations available at the moment:

1. [SparkSqlParser](#)
2. [CatalystSqlParser](#)

`AbstractSqlParser` creates an layer of indirection and expects that subclasses provide custom `AstBuilder` that in turn converts a ANTLR `ParseTree` into a `data type`, `Expression`, `TableIdentifier`, or `LogicalPlan`.

```
protected def astBuilder: AstBuilder
```

`AbstractSqlParser` simply routes all the final parsing calls to translate sql string into a respective Spark SQL object to that `AstBuilder`.

When parsing a SQL string, it first uses its own `parse` protected method that sets up a proper ANTLR parsing infrastructure.

## parse method

```
parse[T](command: String)(toResult: SqlBaseParser => T): T
```

`parse` is a protected method that sets up a proper ANTLR parsing infrastructure with `SqlBaseLexer` and `SqlBaseParser` with are the ANTLR-specific classes of Spark SQL that are auto-generated at build time.

Tip

Review the definition of ANTLR grammar for Spark SQL in [sql/catalyst/src/main/antlr4/org/apache/spark/sql/catalyst/parser/SqlBase.g4](https://github.com/apache/spark/blob/master/sql/catalyst/src/main/antlr4/org/apache/spark/sql/catalyst/parser/SqlBase.g4).

When called, `parse` prints out the following INFO message to the logs:

```
INFO SparkSqlParser: Parsing command: [command]
```

Tip

Enable `INFO` logging level for `SparkSqlParser` or `CatalystSqlParser` to see the INFO message.

## AstBuilder

`AstBuilder` is a ANTLR `SqlBaseBaseVisitor` to convert a ANTLR `ParseTree` (that represents a SQL string) into Spark SQL's corresponding entity using the following methods:

1. `visitSingleDataType` to produce a `DataType`
2. `visitSingleExpression` to produce a `Expression`
3. `visitSingleTableIdentifier` to produce a `TableIdentifier`
4. `visitSingleStatement` for a `LogicalPlan`

`AstBuilder` belongs to `org.apache.spark.sql.catalyst.parser` package.

Note

`SqlBaseBaseVisitor` is a ANTLR-specific base class for parser visitors that is auto-generated at build time.

## SparkSqlParser

`SparkSqlParser` is the default parser of the SQL statements supported in Spark SQL. It is available as `sqlParser` (as the current `ParserInterface` object) through `SessionState`.

The common idiom in Spark SQL is as follows:

```
sparkSession.sessionState.sqlParser
```

`SparkSqlParser` is `AbstractSqlParser` with the `astBuilder` being `SparkSqlAstBuilder`. It supports `variable substitution`.

`SparkSqlParser` is used to parse expression strings into their corresponding `Columns` objects in the following:

1. `expr` function
2. `selectExpr` method (of `dataset` )
3. `filter` method (of `Dataset` )
4. `where` method (of `Dataset` )

```
scala> expr("token = 'hello'")
16/07/07 18:32:53 INFO SparkSqlParser: Parsing command: token = 'hello'
res0: org.apache.spark.sql.Column = (token = hello)
```

`SparkSqlParser` is used to parse table strings into their corresponding table identifiers in the following:

1. `table` methods in `DataFrameReader` and `SparkSession`
2. `insertInto` and `saveAsTable` methods of `DataFrameWriter`
3. `createExternalTable` and `refreshTable` methods of `Catalog` (and `SessionState`)

`SparkSqlParser` is used to parse sql strings into their corresponding `logical query plans` in the following:

1. `sql` method in `sparkSession`

Enable `INFO` logging level for `org.apache.spark.sql.execution.SparkSqlParser` logger to see what happens inside.

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

Tip

```
log4j.logger.org.apache.spark.sql.execution.SparkSqlParser=INFO
```

Refer to [Logging](#).

## Variable Substitution

Caution

[FIXME](#) See `SparkSqlParser` and `substitutor`.

## CatalystSqlParser

`CatalystSqlParser` is an [AbstractSqlParser](#) object with the `astBuilder` being [AstBuilder](#).

`CatalystSqlParser` is used to parse data types (using their canonical string representation), e.g. when [adding fields to a schema](#) or [casting column to different data types](#).

```
import org.apache.spark.sql.types.StructType
scala> val struct = new StructType().add("a", "int")
struct: org.apache.spark.sql.types.StructType = StructType(StructField(a,IntegerType,true))

scala> val asInt = expr("token = 'hello'").cast("int")
asInt: org.apache.spark.sql.Column = CAST((token = hello) AS INT)
```

When parsing, you should see INFO messages in the logs:

```
INFO CatalystSqlParser: Parsing command: int
```

It is also used in `HiveClientImpl` (when converting columns from Hive to Spark) and in `OrcFileOperator` (when inferring the schema for ORC files).

**Tip**

Enable `INFO` logging level for  
`org.apache.spark.sql.catalyst.parser.CatalystSqlParser` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.sql.catalyst.parser.CatalystSqlParser=INFO
```

Refer to [Logging](#).

# SQLExecution Helper Object

`SQLExecution` defines `spark.sql.execution.id` key that is used to track multiple jobs that constitute a single SQL query execution. Whenever a SQL query is to be executed, `withNewExecutionId` static method is used that sets the key.

Note

Jobs without `spark.sql.execution.id` key are not considered to belong to SQL query executions.

## spark.sql.execution.id EXECUTION\_ID\_KEY Key

```
val EXECUTION_ID_KEY = "spark.sql.execution.id"
```

## Tracking Multi-Job SQL Query Executions (`withNewExecutionId` methods)

```
withExecutionId[T](
  sc: SparkContext,
  executionId: String)(body: => T): T  (1)

withNewExecutionId[T](
  sparkSession: SparkSession,
  queryExecution: QueryExecution)(body: => T): T  (2)
```

1. With explicit execution identifier
2. `QueryExecution` variant with an auto-generated execution identifier

`SQLExecution.withNewExecutionId` allow executing the input `body` query action with the **execution id** local property set (as `executionId` or auto-generated). The execution identifier is set as `spark.sql.execution.id` local property (using `SparkContext.setLocalProperty`).

The use case is to track Spark jobs (e.g. when running in separate threads) that belong to a single SQL query execution.

Note

It is used in `Dataset.withNewExecutionId`.

Caution

**FIXME** Where is the proxy-like method used? How important is it?

If there is another execution local property set (as `spark.sql.execution.id`), it is replaced for the course of the current action.

In addition, the `QueryExecution` variant posts `SparkListenerSQLExecutionStart` and `SparkListenerSQLExecutionEnd` events (to `LiveListenerBus` event bus) before and after executing the `body` action, respectively. It is used to inform `SQLListener` when a SQL query execution starts and ends.

Note	Nested execution ids are not supported in the <code>QueryExecution</code> variant.
------	--

# SQLContext

**Caution**

As of Spark **2.0.0** `SQLContext` is only for backward compatibility and is a *mere wrapper of [SparkSession](#)*.

In the older Spark 1.x, **SQLContext** was 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 Dataset\[Long\] \(range method\)](#)
- [Creating DataFrames](#)
- [Creating DataFrames for Table](#)
- [Accessing DataFrameReader](#)
- [Accessing StreamingQueryManager](#)
- [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](#)
- [Executing SQL Queries](#)

## 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 <code>spark.sql</code> 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 you access to `UDFRegistration` to manipulate user-defined functions. Functions registered using `udf` are available for Hive queries only.

Tip

Read up on UDFs in [UDFs — User-Defined Functions](#) document.

```
// Create a DataFrame
val df = Seq("hello", "world!").zip(0 to 1).toDF("text", "id")

// Register the DataFrame as a temporary table in Hive
df.registerTempTable("texts")

scala> sql("SHOW TABLES").show
+-----+
|tableName|isTemporary|
+-----+-----+
| texts | true |
+-----+-----+

scala> sql("SELECT * FROM texts").show
+---+---+
| text | id |
+---+---+
| hello | 0 |
| world! | 1 |
+---+---+

// Just a Scala function
val my_upper: String => String = _.toUpperCase

// Register the function as UDF
spark.udf.register("my_upper", my_upper)

scala> sql("SELECT *, my_upper(text) AS MY_UPPER FROM texts").show
+---+---+-----+
| text | id | MY_UPPER |
+---+---+-----+
| hello | 0 | HELLO |
| world! | 1 | WORLD! |
+---+---+-----+
```

## 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.implicitlys

The `implicitlys` 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.

Import the implicits by `import spark.implicitlys._` as follows:

Note  
`val spark = new SQLContext(sc)`  
`import spark.implicitlys._`

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 <code>SQLContext.implicits</code> brings many <code>encoders</code> available in scope.
------	---

## Accessing DataFrameReader (read method)

```
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[String, String]): DataFrame
```

The experimental `createExternalTable` family of methods is used to create an external table `tableName` and return a corresponding `DataFrame`.

Caution	<a href="#">FIXME</a> What is an external table?
---------	--

It assumes **parquet** as the default data source format that you can change using `spark.sql.sources.default` setting.

## Dropping Temporary Tables

```
dropTempTable(tableName: String): Unit
```

`dropTempTable` method drops a temporary table `tableName`.

Caution	<a href="#">FIXME</a> What is a temporary table?
---------	--

## Creating Dataset[Long] (range method)

```
range(end: Long): Dataset[Long]
range(start: Long, end: Long): Dataset[Long]
range(start: Long, end: Long, step: Long): Dataset[Long]
range(start: Long, end: Long, step: Long, numPartitions: Int): Dataset[Long]
```

The `range` family of methods creates a `Dataset[Long]` 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`.

```
scala> spark.range(5)
res0: org.apache.spark.sql.Dataset[Long] = [id: bigint]

scala> .show
+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
|  4|
+---+
```

## 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.

```
scala> spark.tables.show
+-----+-----+
|tableName|isTemporary|
+-----+-----+
|      t1|     true|
|      t2|     true|
+-----+-----+
```

The schema consists of two columns - `tableName` of `StringType` and `isTemporary` of `BooleanType`.

Note	<code>tables</code> is a result of <code>SHOW TABLES [IN databaseName]</code> .
------	---

```
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 StreamingQueryManager

```
streams: StreamingQueryManager
```

The `streams` method returns a [StreamingQueryManager](#) that is used to...TK

Caution	<a href="#">FIXME</a>
---------	-----------------------

## 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(spark: SQLContext): Unit
clearActive(): Unit
```

## Executing SQL Queries

```
sql(sqlText: String): DataFrame
```

`sql` executes the `sqlText` SQL query.

Note	It supports Hive statements through <a href="#">HiveContext</a> .
------	---

```

scala> sql("set spark.sql.hive.version").show(false)
16/04/10 15:19:36 INFO HiveSqlParser: Parsing command: set spark.sql.hive.version
+-----+-----+
|key          |value|
+-----+-----+
|spark.sql.hive.version|1.2.1|
+-----+-----+


scala> sql("describe database extended default").show(false)
16/04/10 15:21:14 INFO HiveSqlParser: Parsing command: describe database extended default
+-----+-----+
|database_description_item|database_description_value|
+-----+-----+
|Database Name           |default           |
|Description              |Default Hive database   |
|Location                 |file:/user/hive/warehouse |
|Properties               |                   |
+-----+-----+


// Create temporary table
scala> spark.range(10).registerTempTable("t")
16/04/14 23:34:31 INFO HiveSqlParser: Parsing command: t

scala> sql("CREATE temporary table t2 USING PARQUET OPTIONS (PATH 'hello') AS SELECT * FROM t")
16/04/14 23:34:38 INFO HiveSqlParser: Parsing command: CREATE temporary table t2 USING PARQUET OPTIONS (PATH 'hello') AS SELECT * FROM t

scala> spark.tables.show
+-----+-----+
|tableName|isTemporary|
+-----+-----+
|      t|     true|
|     t2|     true|
+-----+-----+

```

`sql` parses `sqlText` using a dialect that can be set up using `spark.sql.dialect` setting.

	<p><code>sql</code> is imported in spark-shell so you can execute Hive statements without <code>spark</code> prefix.</p>
Note	<pre> scala&gt; println(s"This is Spark \${sc.version}") This is Spark 2.0.0-SNAPSHOT  scala&gt; :imports 1) import spark.implicits._  (52 terms, 31 are implicit) 2) import spark.sql        (1 terms) </pre>

Tip	You may also use <code>spark-sql shell script</code> to interact with Hive.
-----	---

Internally, it uses `SessionState.sqlParser.parsePlan(sql)` method to create a [LogicalPlan](#).

Caution	<a href="#">FIXME</a> Review
---------	------------------------------

```
scala> sql("show tables").show(false)
16/04/09 13:05:32 INFO HiveSqlParser: Parsing command: show tables
+-----+-----+
|tableName|isTemporary|
+-----+-----+
|dafa    |false     |
+-----+-----+
```

Enable `INFO` logging level for the loggers that correspond to the [implementations of AbstractSqlParser](#) to see what happens inside `sql`.

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

Tip

```
log4j.logger.org.apache.spark.sql.hive.execution.HiveSqlParser=INFO
```

Refer to [Logging](#).

## Creating New Session

```
newSession(): SQLContext
```

You can use `newSession` method to create a new session without a cost of instantiating a new `SQLContext` from scratch.

`newSession` returns a new `SQLContext` that shares `sparkContext`, `CacheManager`, [SQLListener](#), and `ExternalCatalog`.

Caution	<a href="#">FIXME</a> Why would I need that?
---------	--

# Catalyst Query Optimizer

Caution	<a href="#">FIXME Review</a> <a href="https://github.com/apache/spark/blob/main/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/Optimizer.scala">sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/Optimizer.scala</a>
---------	---

Catalyst is a framework for manipulating trees of relational operators and expressions. It contains logical query plans that it can translate to executable code (as Spark RDDs).

Catalyst is an extensible **query plan optimizer** with [constant folding](#), [predicate pushdown](#), [nullability \(NULL value\) propagation](#), [vectorized Parquet decoder](#) and [whole-stage code generation](#) optimization techniques.

Catalyst supports both rule-based and cost-based optimization.

Among the rule-based optimizations is [nullability \(NULL value\) propagation](#).

## SparkOptimizer

Caution	<a href="#">FIXME</a>
---------	-----------------------

## RowEncoder

`RowEncoder` is a factory object that maps `StructType` to `ExpressionEncoder[Row]`

It belongs to `org.apache.spark.sql.catalyst.encoders` package.

## Further reading or watching

- [Deep Dive into Spark SQL's Catalyst Optimizer](#)

# Predicate Pushdown

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

When you execute `where` operator right after loading a data (into a `Dataset`), Spark SQL will push the "where" predicate down to the source using a corresponding SQL query with `WHERE` clause (or whatever is the proper language for the source).

This optimization is called **predicate pushdown** that pushes down the filtering to a data source engine (rather than dealing with it after the entire dataset has been loaded to Spark's memory and filtering out records afterwards).

Given the following code:

```
val df = spark.read  
  .format("jdbc")  
  .option("url", "jdbc:...")  
  .option("dbtable", "people")  
  .load()  
  .as[Person]  
  .where(_.name === "Jacek")
```

Spark translates it to the following SQL query:

```
SELECT * FROM people WHERE name = 'Jacek'
```

Caution	FIXME Show the database logs with the query.
---------	--

# Constant Folding

`ConstantFolding` object is a operator optimization in [Catalyst](#) that replaces expressions that can be statically evaluated with equivalent literal values.

`ConstantFolding` object is a `Rule[LogicalPlan]`.

# Nullability (NULL Value) Propagation

NullPropagation object is a operator optimization in Catalyst.

NullPropagation object is a Rule[LogicalPlan] .

```
scala> (0 to 9).toDF("num").as[String].where('num === null).explain(extended = true)
== Parsed Logical Plan ==
'Filter ('num = null)
+- Project [value#137 AS num#139]
  +- LocalRelation [value#137]

== Analyzed Logical Plan ==
num: int
Filter (num#139 = cast(null as int))
+- Project [value#137 AS num#139]
  +- LocalRelation [value#137]

== Optimized Logical Plan ==
LocalRelation <empty>, [num#139]

== Physical Plan ==
LocalTableScan <empty>, [num#139]
```

# Vectorized Parquet Decoder

Caution

[FIXME](#)

# Query Plan

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

`QueryPlan` abstract class has a [output](#) (that is a sequence of [Attribute](#) instances). You can also ask for [schema](#).

Note	<code>QueryPlan</code> is a super type of <a href="#">SparkPlan</a> and <a href="#">LogicalPlan</a> abstract classes.
------	---

## schema

You can find out about the schema of a `QueryPlan` using `schema` that builds `StructType` from the [output attributes](#).

## Output Attributes

### Attribute

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

# SparkPlan — Spark Query Planner

`SparkPlan` is a base [QueryPlan](#) for physical operators. It can be executed to produce a `RDD[InternalRow]`.

Note	The naming convention for physical operators in Spark's source code is to have their names end with the <code>Exec</code> prefix, e.g. <code>DebugExec</code> .
------	---

Tip	Read <a href="#">InternalRow</a> about the internal binary row format.
-----	--

`SparkPlan` has the following attributes:

- `metadata`
- `metrics`
- `outputPartitioning`
- `outputOrdering`

`SparkPlan` has the following `final` methods that prepare environment and pass calls on to corresponding methods that constitute [SparkPlan Contract](#):

- `execute` calls `doExecute`
- `prepare` calls `doPrepare`
- `executeBroadcast` calls `doExecuteBroadcast`

## SparkPlan Contract

The contract of `SparkPlan` requires that concrete implementations define the following method:

- `doExecute(): RDD[InternalRow]`

They may also define their own custom overrides:

- `doPrepare`
- `doExecuteBroadcast`

Caution	<a href="#">FIXME</a> Why are there two executes?
---------	---

## SQLMetric

`SQLMetric` is an [accumulator](#) that accumulate and produce long values.

There are three known `SQLMetrics` :

- `sum`
- `size`
- `timing`

## metrics Lookup Table

```
metrics: Map[String, SQLMetric] = Map.empty
```

`metrics` is a `private[sql]` lookup table of supported [SQLMetrics](#) by their names.

# QueryPlanner

`QueryPlanner` transforms a [LogicalPlan](#) through a chain of `GenericStrategy` objects to produce a physical execution plan, e.g. [SparkPlan](#) for [SparkPlanner](#) or the [Hive-Specific SparkPlanner for HiveSessionState](#).

## QueryPlanner Contract

`QueryPlanner` contract defines the following operations:

- `strategies`
- `plan`
- `collectPlaceholders`
- `prunePlans`

### strategies

The abstract `strategies` method that returns a collection of `GenericStrategy` objects (that are used in the other `plan` method).

### plan

`plan(plan: LogicalPlan)` that returns an `Iterator[PhysicalPlan]` with elements being the result of applying each `GenericStrategy` object from `strategies` collection to `plan` input parameter.

### collectPlaceholders

`collectPlaceholders` that returns a collection of pairs of a physical plan and a corresponding [LogicalPlan](#).

### prunePlans

`prunePlans` that prunes bad physical plans.

## SparkStrategies

`SparkStrategies` is an abstract base `QueryPlanner` that produces a [SparkPlan](#).

It merely serves as a "container" with concrete `SparkStrategy` objects, e.g. `SpecialLimits`, `JoinSelection`, `StatefulAggregationStrategy`, `Aggregation`, `InMemoryScans`, `StreamingRelationStrategy`, etc.

**Note**

`Strategy` is a type alias of `SparkStrategy` that is defined in [org.apache.spark.sql package object](#).

[SparkPlanner](#) is the one and only concrete implementation of `SparkStrategies` available.

**Caution**

[`FIXME`](#) What is `singleRowRdd` for?

## SparkPlanner

`SparkPlanner` is a concrete `QueryPlanner` (indirectly through extending [SparkStrategies](#)) that allows for `extraStrategies`, i.e. a collection of additional `SparkStrategy` transformations (beside the ones defined in `SparkPlanner` itself).

It requires a `SparkContext`, a `SQLConf`, and a collection of `SparkStrategy` objects (as `extraStrategies`) when created.

It defines `numPartitions` method that returns the value of [spark.sql.shuffle.partitions](#) for the number of partitions to use for joins and aggregations.

`strategies` collection uses predefined `Strategy` objects as well as the constructor's `extraStrategies`.

Among the `SparkStrategy` objects are `FileSourceStrategy` and `JoinSelection`.

## Hive-Specific SparkPlanner for HiveSessionState

`HiveSessionState` class uses an custom anonymous `SparkPlanner` for `planner` method (part of the [SessionState](#) contract).

The custom anonymous `SparkPlanner` uses `Strategy` objects defined in `HiveStrategies`.

# Query Execution

`QueryExecution` is a part of the public `Dataset` API and represents the query execution that will eventually produce the data in a `Dataset`.

You can access the `QueryExecution` of a `Dataset` using `queryExecution` attribute.

Note

```
val spark: SparkSession = ...
spark.emptyDataset[Long].queryExecution
```

`QueryExecution` is the result of executing a `LogicalPlan` in a `SparkSession` (and so you could create a `Dataset` from a `LogicalPlan` or use the `QueryExecution` after executing the `LogicalPlan`).

`QueryExecution` uses the input `SparkSession` to access the current `SparkPlanner` (through `SessionState` that could also return a `HiveSessionState`) when it is created. It then computes a `SparkPlan` (a `PhysicalPlan` exactly) using the planner. It is available as the `sparkPlan` (lazy) attribute.

A streaming variant of `QueryExecution` is `IncrementalExecution`.

`debug package object` contains methods for **debugging query execution** that you can use to do the full analysis of your queries (as `Dataset` objects).

Caution

`FIXME What's planner ? analyzed ? Why do we need assertAnalyzed and assertSupported ?`

It belongs to `org.apache.spark.sql.execution` package.

Note

`QueryExecution` is a transient feature of a `Dataset`, i.e. it is not preserved across serializations.

```

val ds = spark.range(5)
scala> ds.queryExecution
res17: org.apache.spark.sql.execution.QueryExecution =
== Parsed Logical Plan ==
Range 0, 5, 1, 8, [id#39L]

== Analyzed Logical Plan ==
id: bigint
Range 0, 5, 1, 8, [id#39L]

== Optimized Logical Plan ==
Range 0, 5, 1, 8, [id#39L]

== Physical Plan ==
WholeStageCodegen
: +- Range 0, 1, 8, 5, [id#39L]

```

## Creating QueryExecution Instance

```

class QueryExecution(
  val sparkSession: SparkSession,
  val logical: LogicalPlan)

```

`QueryExecution` requires a [SparkSession](#) and a [LogicalPlan](#).

## Accessing SparkPlanner (planner method)

```
planner: SparkPlanner
```

`planner` returns [SparkPlanner](#).

`planner` is merely to expose internal `planner` (in the current [SessionState](#)).

## Lazy Attributes

`QueryExecution` holds lazy attributes that are initialized at the first read.

## analyzed Attribute

`analyzed` lazy value is a [SparkPlan](#) that is a result of executing the input `logical` logical plan by the current `Analyzer` (of the current [SparkSession](#)).

**Note**

`Analyzer` resolves unresolved attributes and relations to typed objects using information in a `sessionCatalog` and `FunctionRegistry`.

## withCachedData Attribute

`withCachedData` being a `LogicalPlan` that is the `analyzed` plan after being analyzed, checked (for unsupported operations) and replaced with cached segments.

## Optimized Logical Query Plan (optimizedPlan Attribute)

`optimizedPlan` is the `LogicalPlan` (of a query) that is a result of executing the session-owned [Optimizer](#) to `withCachedData`.

## sparkPlan Attribute

`sparkPlan` is the [SparkPlan](#) that is the result of requesting [SparkPlanner](#) to plan a [optimized logical query plan](#).

**Note**

In fact, the result `SparkPlan` is the first Spark query plan from the collection of possible query plans from [SparkPlanner](#).

## executedPlan Attribute

`executedPlan` attribute is computed lazily and represents a [SparkPlan](#) ready for execution. It is the `sparkPlan` plan with all the [preparation rules](#) applied.

```
val dataset: Dataset[Long] = ...
dataset.queryExecution.executedPlan
```

## toRdd Attribute

`toRdd` is the `RDD[InternalRow]` that is the result of executing `executedPlan` plan, i.e.

```
executedPlan.execute()
```

**Tip**

[InternalRow](#) is the internal optimized binary row format.

## preparations — SparkPlan Optimization Rules (to apply before Query Execution)

`preparations` is a sequence of [SparkPlan](#) optimization rules.

Tip	A <code>SparkPlan</code> optimization rule transforms a <code>SparkPlan</code> to another <code>SparkPlan</code> .
-----	--

This collection is an intermediate phase of query execution that developers can use to introduce further optimizations.

The current list of `SparkPlan` transformations in `preparations` is as follows:

1. `ExtractPythonUDFs`
2. `PlanSubqueries`
3. `EnsureRequirements`
4. `CollapseCodegenStages`
5. `ReuseExchange`
6. `ReuseSubquery`

Note	The transformation rules applied in order to the physical plan before execution, i.e. they generate a <code>SparkPlan</code> when <code>executedPlan</code> lazy value is accessed.
------	---

## IncrementalExecution

`IncrementalExecution` is a custom `QueryExecution` with `OutputMode`, `checkpointLocation`, and `currentBatchId`.

It lives in `org.apache.spark.sql.execution.streaming` package.

Caution	<a href="#">FIXME</a> What is <code>stateStrategy</code> ?
---------	--

Stateful operators in the query plan are numbered using `operatorId` that starts with `0`.

`IncrementalExecution` adds one `Rule[SparkPlan]` called `state` to `preparations` sequence of rules as the first element.

Caution	<a href="#">FIXME</a> What does <code>IncrementalExecution</code> do? Where is it used?
---------	---

# Whole-Stage Code Generation (aka Whole-Stage CodeGen)

## Note

Review [SPARK-12795 Whole stage codegen](#) to learn about the work to support it.

**Whole-Stage Code Generation** (aka `WholeStageCodegen` or `WholeStageCodegenExec`) fuses multiple operators (as a subtree of [plans that support codegen](#)) together into a single Java function that is aimed at improving execution performance. It collapses a query into a single optimized function that eliminates virtual function calls and leverages CPU registers for intermediate data.

`WholeStageCodegenExec` case class works with a [SparkPlan](#) to produce a **codegened pipeline**. It is a unary node in [SparkPlan](#) with [support for codegen](#).

## Tip

Use [Dataset.explain](#) method to know the physical plan of a query and find out whether or not `WholeStageCodegen` is in use.

## Tip

Consider using [Debugging Query Execution facility](#) to deep dive into whole stage codegen.

```
scala> spark.range(10).select('id as 'asId).where('id === 4).explain
== Physical Plan ==
WholeStageCodegen
: +- Project [id#0L AS asId#3L]
:   +- Filter (id#0L = 4)
:     +- Range 0, 1, 8, 10, [id#0L]
```

`SparkPlan` plans with support for codegen extend [CodegenSupport](#).

## Note

Whole stage codegen is used by some modern massively parallel processing (MPP) databases to archive great performance. See [Efficiently Compiling Efficient Query Plans for Modern Hardware \(PDF\)](#).

Whole stage codegen uses `spark.sql.codegen.wholeStage` setting to control...FIXME

## Note

Janino is used to compile a Java source code into a Java class.

Before a query is executed, `CollapseCodegenStages` case class is used to find the plans that support codegen and collapse them together as `WholeStageCodegen`. It is part of the sequence of rules `QueryExecution.preparations` that will be applied in order to the physical plan before execution.

## CodegenSupport Contract

`CodegenSupport` is a custom [SparkPlan](#) for operators that support codegen.

It however allows custom implementations to optionally disable codegen using `supportCodegen` predicate (that defaults to `true`).

It assumes that custom implementations define:

- `doProduce(ctx: CodegenContext): String`

## Codegen Operators

`SparkPlan` plans that support codegen extend [CodegenSupport](#).

- `ProjectExec` for `as`
- `FilterExec` for `where` or `filter`
- `Range`
- [SampleExec](#) for `sample`
- [RangeExec](#) for `SQLContext.range`
- `RowDataSourceScanExec`

Caution	<a href="#">FIXME</a> Where is <code>RowDataSourceScanExec</code> used?
---------	---

- `BatchedDataSourceScanExec`
- `ExpandExec`
- `BaseLimitExec`
- `SortExec`
- `WholeStageCodegenExec` and `InputAdapter`
- `TungstenAggregate`
- [BroadcastHashJoinExec](#)
- `SortMergeJoinExec`

## BroadcastHashJoinExec

`BroadcastHashJoinExec` variables are prefixed with `bhj` (see `CodegenSupport.variablePrefix`).

```

val ds = Seq((0,"playing"), (1, "with"), (2, "broadcast")).toDS

scala> spark.conf.get("spark.sql.autoBroadcastJoinThreshold")
res18: String = 10485760

scala> ds.join(ds).explain(extended=true)
== Parsed Logical Plan ==
'Join Inner
:- LocalRelation [_1#21, _2#22]
+- LocalRelation [_1#21, _2#22]

== Analyzed Logical Plan ==
_1: int, _2: string, _1: int, _2: string
Join Inner
:- LocalRelation [_1#21, _2#22]
+- LocalRelation [_1#32, _2#33]

== Optimized Logical Plan ==
Join Inner
:- LocalRelation [_1#21, _2#22]
+- LocalRelation [_1#32, _2#33]

== Physical Plan ==
BroadcastNestedLoopJoin BuildRight, Inner, true
:- LocalTableScan [_1#21, _2#22]
+- BroadcastExchange IdentityBroadcastMode
  +- LocalTableScan [_1#32, _2#33]

// Use broadcast function to mark the right-side Dataset
// eligible for broadcasting explicitly

scala> ds.join(broadcast(ds)).explain(extended=true)
== Parsed Logical Plan ==
'Join Inner
:- LocalRelation [_1#21, _2#22]
+- BroadcastHint
  +- LocalRelation [_1#21, _2#22]

== Analyzed Logical Plan ==
_1: int, _2: string, _1: int, _2: string
Join Inner
:- LocalRelation [_1#21, _2#22]
+- BroadcastHint
  +- LocalRelation [_1#43, _2#44]

== Optimized Logical Plan ==
Join Inner
:- LocalRelation [_1#21, _2#22]
+- BroadcastHint
  +- LocalRelation [_1#43, _2#44]

== Physical Plan ==

```

```
BroadcastNestedLoopJoin BuildRight, Inner, true
:- LocalTableScan [_1#21, _2#22]
+- BroadcastExchange IdentityBroadcastMode
  +- LocalTableScan [_1#43, _2#44]
```

## SampleExec

```
scala> spark.range(10).sample(false, 0.4).explain
== Physical Plan ==
WholeStageCodegen
: +- Sample 0.0, 0.4, false, -7634498724724501829
:     +- Range 0, 1, 8, 10, [id#15L]
```

## RangeExec

```
scala> spark.range(10).explain
== Physical Plan ==
WholeStageCodegen
: +- Range 0, 1, 8, 10, [id#20L]
```

## CollapseCodegenStages

`CollapseCodegenStages` is a `Rule[SparkPlan]`, i.e. a transformation of `SparkPlan` into another `SparkPlan`.

Note	<code>CollapseCodegenStages</code> is used in the sequence of rules to apply to a <code>SparkPlan</code> before query execution.
------	--

It searches for sub-plans (aka *stages*) that support codegen and collapse them together as a `wholeStageCodegen`.

Note	Only <code>CodegenSupport</code> <code>SparkPlans</code> support codegen for which <code>supportCodegen</code> is enabled ( <code>true</code> ).
------	--

It is assumed that all `Expression` instances except `codegenFallback` support codegen.

`CollapseCodegenStages` uses the internal setting `spark.sql.codegen.maxFields` (default: 200) to control the number of fields in input and output schemas before deactivating whole-stage codegen. It counts the fields included in complex types, i.e. `StructType`, `MapType`, `ArrayType`, `UserDefinedType`, and their combinations, recursively. See [SPARK-14554](#).

It inserts `InputAdapter` leaf nodes in a `SparkPlan` recursively that is then used to generate code that consumes an `RDD` iterator of `InternalRow`.

## BenchmarkWholeStageCodegen - Performance Benchmark

`BenchmarkWholeStageCodegen` class provides a benchmark to measure whole stage codegen performance.

You can execute it using the command:

```
build/sbt 'sql/testOnly *BenchmarkWholeStageCodegen'
```

**Note**

You need to un-ignore tests in `BenchmarkWholeStageCodegen` by replacing `ignore` with `test`.

```
$ build/sbt 'sql/testOnly *BenchmarkWholeStageCodegen'
...
Running benchmark: range/limit/sum
  Running case: range/limit/sum codegen=false
22:55:23.028 WARN org.apache.hadoop.util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
  Running case: range/limit/sum codegen=true

Java HotSpot(TM) 64-Bit Server VM 1.8.0_77-b03 on Mac OS X 10.10.5
Intel(R) Core(TM) i7-4870HQ CPU @ 2.50GHz

range/limit/sum:                                Best/Avg Time(ms)      Rate(M/s)    Per Row(ns)   Rel
ative
-----
-----
range/limit/sum codegen=false                  376 / 433          1394.5        0.7
1.0X
range/limit/sum codegen=true                  332 / 388          1581.3        0.6
1.1X

[info] - range/limit/sum (10 seconds, 74 milliseconds)
```

# Project Tungsten

One of the main motivations of **Project Tungsten** is to greatly reduce the usage of Java objects to minimum by introducing its own memory management. Tungsten uses a compact storage format for data representation that also reduces memory footprint. With a known schema for datasets, the proper data layout is possible immediately with the data being already serialized (that further reduces or completely avoids serialization between JVM object representation and Spark's internal one).

Project Tungsten uses `sun.misc.unsafe` API for direct memory access to bypass the JVM in order to avoid garbage collection.

The optimizations provided by the project Tungsten:

1. Memory Management using Binary In-Memory Data Representation aka **Tungsten row format**.
2. Cache-Aware Computations with Cache-Aware Layout for high cache hit rates
3. [Whole-Stage Code Generation](#)

Tungsten does code generation, i.e. generates JVM bytecode on the fly, to access Tungsten-managed memory structures that gives a very fast access.

Tungsten also introduces **cache-aware data structures** that are aware of the physical machine caches at different levels - L1, L2, L3.

## Further reading or watching

- [Project Tungsten: Bringing Spark Closer to Bare Metal](#)

# Hive Integration

Spark SQL supports [Apache Hive](#) using `HiveContext`. It uses the Spark SQL execution engine to work with data stored in Hive.

From [Wikipedia, the free encyclopedia](#):

Apache Hive supports analysis of large datasets stored in Hadoop's HDFS and compatible file systems such as Amazon S3 filesystem.

**Note**

It provides an SQL-like language called HiveQL with schema on read and transparently converts queries to Hadoop MapReduce, Apache Tez and Apache Spark jobs.

All three execution engines can run in Hadoop YARN.

`HiveContext` is a specialized `SQLContext` to work with Hive.

There is also a dedicated tool [spark-sql](#) that...[FIXME](#)

**Tip**

Import `org.apache.spark.sql.hive` package to use `HiveContext`.

Enable `DEBUG` logging level for `HiveContext` to see what happens inside.

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

**Tip**

```
log4j.logger.org.apache.spark.sql.hive.HiveContext=DEBUG
```

Refer to [Logging](#).

## Hive Functions

[SQLContext.sql](#) (or simply `sql`) allows you to interact with Hive.

You can use `show functions` to learn about the Hive functions supported through the Hive integration.

```

scala> sql("show functions").show(false)
16/04/10 15:22:08 INFO HiveSqlParser: Parsing command: show functions
+-----+
|function      |
+-----+
| !
|%
|&
|*
|+
|-|
|/
|<
|<=
|<=>
|=|
|==|
|>
|>=
|^
|abs
|acos
|add_months
|and
|approx_count_distinct|
+-----+
only showing top 20 rows

```

## Hive Configuration - `hive-site.xml`

The configuration for Hive is in `hive-site.xml` on the classpath.

The default configuration uses Hive 1.2.1 with the default warehouse in `/user/hive/warehouse`.

```

16/04/09 13:37:54 INFO HiveContext: Initializing execution hive, version 1.2.1
16/04/09 13:37:58 WARN ObjectStore: Version information not found in metastore. hive.metastore.schema.verification is not enabled so recording the schema version 1.2.0
16/04/09 13:37:58 WARN ObjectStore: Failed to get database default, returning NoSuchObjectException
16/04/09 13:37:58 INFO HiveContext: default warehouse location is /user/hive/warehouse
16/04/09 13:37:58 INFO HiveContext: Initializing HiveMetastoreConnection version 1.2.1 using Spark classes.
16/04/09 13:38:01 DEBUG HiveContext: create HiveContext

```

## current\_database function

`current_database` function returns the current database of Hive metadata.

```
scala> sql("select current_database()").show(false)
16/04/09 13:52:13 INFO HiveSqlParser: Parsing command: select current_database()
+-----+
|currentdatabase()|
+-----+
|default      |
+-----+
```

`current_database` function is registered when `HiveContext` is initialized.

Internally, it uses private `CurrentDatabase` class that uses `HiveContext.sessionState.catalog.getCurrentDatabase`.

## Analyzing Tables

```
analyze(tableName: String)
```

`analyze` analyzes `tableName` table for query optimizations. It currently supports only Hive tables.

```
scala> sql("show tables").show(false)
16/04/09 14:04:10 INFO HiveSqlParser: Parsing command: show tables
+-----+
|tableName|isTemporary|
+-----+
|dafa     |false      |
+-----+
```

```
scala> spark.asInstanceOf[HiveContext].analyze("dafa")
16/04/09 14:02:56 INFO HiveSqlParser: Parsing command: dafa
java.lang.UnsupportedOperationException: Analyze only works for Hive tables, but dafa
is a LogicalRelation
    at org.apache.spark.sql.hive.HiveContext.analyze(HiveContext.scala:304)
    ... 50 elided
```

## Hive Thrift server

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

## Experimental: Metastore Tables with non-Hive SerDe

Caution	FIXME Review the uses of <code>convertMetastoreParquet</code> , <code>convertMetastoreParquetWithSchemaMerging</code> , <code>convertMetastoreOrc</code> , <code>convertCTAS</code> .
---------	---

## Settings

- `spark.sql.hive.metastore.version` (default: `1.2.1`) - the version of the Hive metastore. Supported versions from `0.12.0` up to and including `1.2.1`.
- `spark.sql.hive.version` (default: `1.2.1`) - the version of Hive used by Spark SQL.

Caution	<a href="#">FIXME</a> Review <code>HiveContext</code> object.
---------	---

# Spark SQL CLI - spark-sql

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

Tip	Read about Spark SQL CLI in Spark's official documentation in <a href="#">Running the Spark SQL CLI</a> .
-----	---

Tip	<pre>spark-sql&gt; describe function `&lt;&gt;`; Function: &lt;&gt; Usage: a &lt;&gt; b - Returns TRUE if a is not equal to b</pre>
-----	---

Tip	Functions are registered in <a href="#">FunctionRegistry</a> .
-----	--

	<pre>spark-sql&gt; show functions;</pre>
--	--

	<pre>spark-sql&gt; explain extended show tables;</pre>
--	--

# Settings

The following list are the settings used to configure Spark SQL applications.

You can apply them to `SQLContext` using `setConf` method:

```
spark.setConf("spark.sqlcodegen.wholeStage", "false")
```

## spark.sql.catalogImplementation

`spark.sql.catalogImplementation` (default: `in-memory`) is an internal setting to select the active catalog implementation.

There are two acceptable values:

- `in-memory` (default)
- `hive`

Caution

[FIXME](#) What is a catalog?

## spark.sql.shuffle.partitions

`spark.sql.shuffle.partitions` (default: `200`) — the default number of partitions to use when shuffling data for joins or aggregations.

## spark.sql.allowMultipleContexts

`spark.sql.allowMultipleContexts` (default: `true`) controls whether creating multiple `SQLContexts/HiveContexts` is allowed.

## spark.sql.autoBroadcastJoinThreshold

`spark.sql.autoBroadcastJoinThreshold` (default: `10 * 1024 * 1024`) configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. If the size of the statistics of the logical plan of a `DataFrame` is at most the setting, the `DataFrame` is broadcast for join.

Negative values or `0` disable broadcasting.

Consult [Broadcast Join](#) for more information about the topic.

## spark.sql.columnNameOfCorruptRecord

`spark.sql.columnNameOfCorruptRecord` ...[FIXME](#)

## spark.sql.dialect

`spark.sql.dialect` - [FIXME](#)

## spark.sql.sources.default

`spark.sql.sources.default` (default: `parquet`) sets the default data source to use in input/output.

It is used when reading or writing data in [DataFrameWriter](#), [DataFrameReader](#), [createExternalTable](#) as well as the streaming [DataStreamReader](#) and [DataStreamWriter](#).

## spark.sql.streaming.checkpointLocation

`spark.sql.streaming.checkpointLocation` is the default location for storing checkpoint data for [continuously executing queries](#).

## spark.sqlcodegen.wholeStage

`spark.sqlcodegen.wholeStage` (default: `true`) controls whether the whole stage (of multiple operators) will be compiled into single java method (`true`) or not (`false`).

# Spark Streaming

**Spark Streaming** is the incremental stream processing framework for Spark.

Spark Streaming offers the data abstraction called [DStream](#) that hides the complexity of dealing with a continuous data stream and makes it as easy for programmers as using one single RDD at a time.

That is why Spark Streaming is also called a **micro-batching streaming framework** as a batch is one RDD at a time.

Note	I think Spark Streaming shines on performing the <b>T</b> stage well, i.e. the transformation stage, while leaving the <b>E</b> and <b>L</b> stages for more specialized tools like <a href="#">Apache Kafka</a> or frameworks like Akka.
------	---

For a software developer, a `DStream` is similar to work with as a `RDD` with the DStream API to match RDD API. Interestingly, you can reuse your RDD-based code and apply it to `DStream` - a stream of RDDs - with no changes at all (through [foreachRDD](#)).

It runs [streaming jobs](#) every [batch duration](#) to pull and process data (often called *records*) from one or many [input streams](#).

Each batch [computes \(generates\)](#) a RDD for data in input streams for a given batch and [submits a Spark job to compute the result](#). 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.

Spark Streaming allows for integration with real-time data sources ranging from such basic ones like a HDFS-compatible file system or socket connection to more advanced ones like Apache Kafka or Apache Flume.

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 <a href="#">SparkHadoopUtil.get.conf</a> .
------	---

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!)

Tip	<p><code>StreamingContext</code> will warn you when you use <code>local</code> or <code>local[1]</code> <a href="#">master URLs</a>:</p> <pre>WARN StreamingContext: spark.master should be set as local[n], n &gt; 1 in local mode if you have receivers to get data, otherwise Spark jobs will not get resources to process the received data.</pre>
-----	--

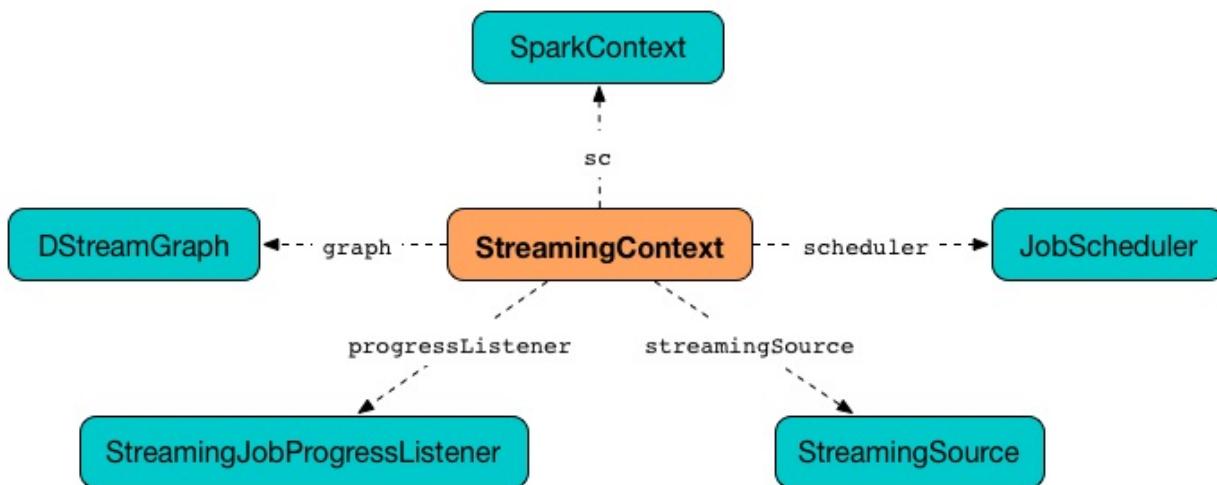


Figure 1. StreamingContext and Dependencies

A `DStreamGraph` is created.

A `JobScheduler` is created.

A `StreamingJobProgressListener` is created.

`Streaming tab` in web UI is created (when `spark.ui.enabled` is enabled).

A `StreamingSource` is instantiated.

At this point, `StreamingContext` enters `INITIALIZED` state.

## Creating ReceiverInputDStreams

`StreamingContext` offers the following methods to create `ReceiverInputDStreams`:

- `receiverStream(receiver: Receiver[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]`

## receiverStream method

```
receiverStream[T: ClassTag](receiver: Receiver[T]): ReceiverInputDStream[T]
```

You can register a custom input dstream using `receiverStream` method. It accepts a `Receiver`.

Note	You can find an example of a custom <code>Receiver</code> in <a href="#">Custom Receiver</a> .
------	--

## transform method

```
transform[T](dstreams: Seq[DStream[_]], transformFunc: (Seq[RDD[_]], Time) => RDD[T]): DStream[T]
```

## 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	<a href="#">FIXME</a> 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 <b>graph checkpointing interval</b> .
------	---

[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	"A <i>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 <a href="#">Checkpointing</a> 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 <a href="#">StreamingContext is created</a> and <a href="#">spark.streaming.checkpoint.directory</a> 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 <a href="#">States</a> section in this document to learn about the states of <code>StreamingContext</code> .
------	--

## 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 started in this JVM. Currently running StreamingContext was
started at [startSite]
```

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](#), 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 `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.

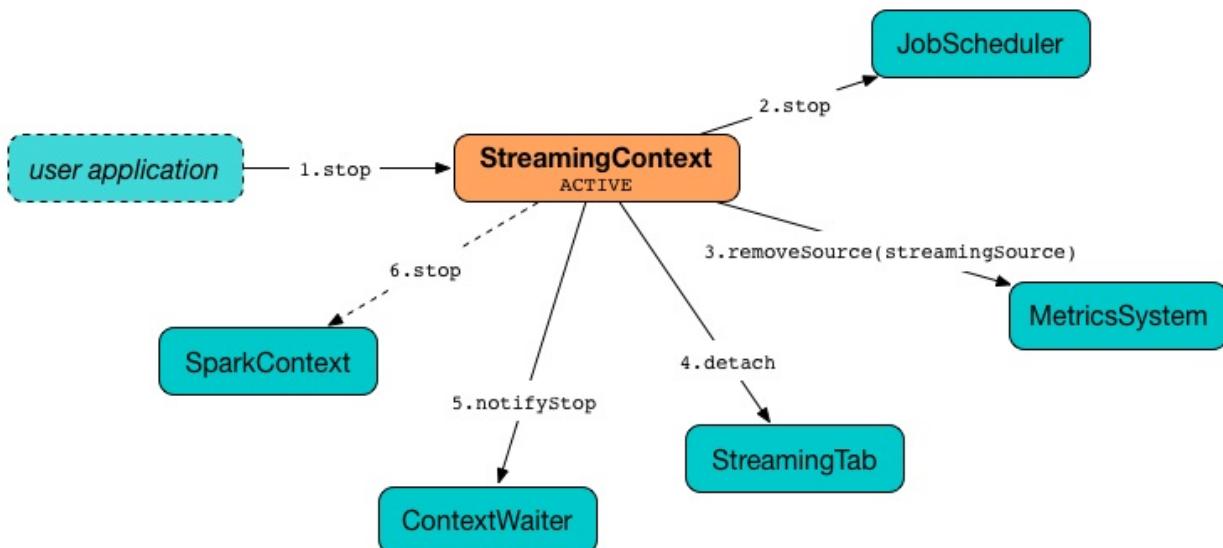


Figure 3. StreamingContext Stop Procedure

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 `ShutdownHookManager` when [StreamingContext starts](#).

**Note** `ShutdownHookManager` uses `org.apache.hadoop.util.ShutdownHookManager` 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 <a href="#">started</a> .
------	--

It first asserts that `DStreamGraph` has been assigned (i.e. `graph` field is not `null`) and triggers [validation of DStreamGraph](#).

Caution	It appears that <code>graph</code> could never be <code>null</code> , 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 application. Enabling Dynamic allocation for Spark Streaming applications can cause data loss if Write Ahead Log is not enabled for non-replayable sources like Flume. See the programming guide for details on how to enable the Write Ahead Log
```

## Registering Streaming Listeners

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Streaming Metrics Source

Caution	<a href="#">FIXME</a>
---------	-----------------------

## 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 <a href="#">TransformedDStream</a> .

## 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 spark.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 <a href="#">TransformedDStream</a> .
------	--

## 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, time) =>
    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 <a href="#">Window Operations</a> to read the official documentation.
Note	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): DStream[T]
reduceByWindow(reduceFunc: (T, T) => T, invReduceFunc: (T, T) => T, windowDuration: Duration, slideDuration: Duration): DStream[T]
```

`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 <a href="#">window</a> and <a href="#">reduce</a> 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 = Second
s(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.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).

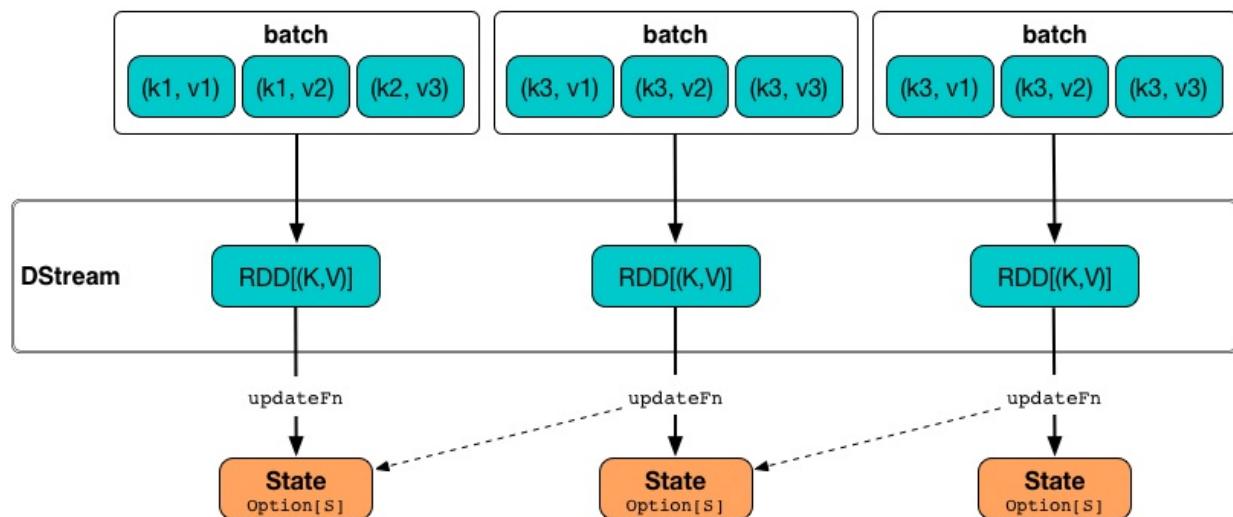


Figure 1. `updateStateByKey` in motion

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 yet")
    println(s">>> >>> Building the state being the number of calls to update state function, i.e. the number of batches")
    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*).

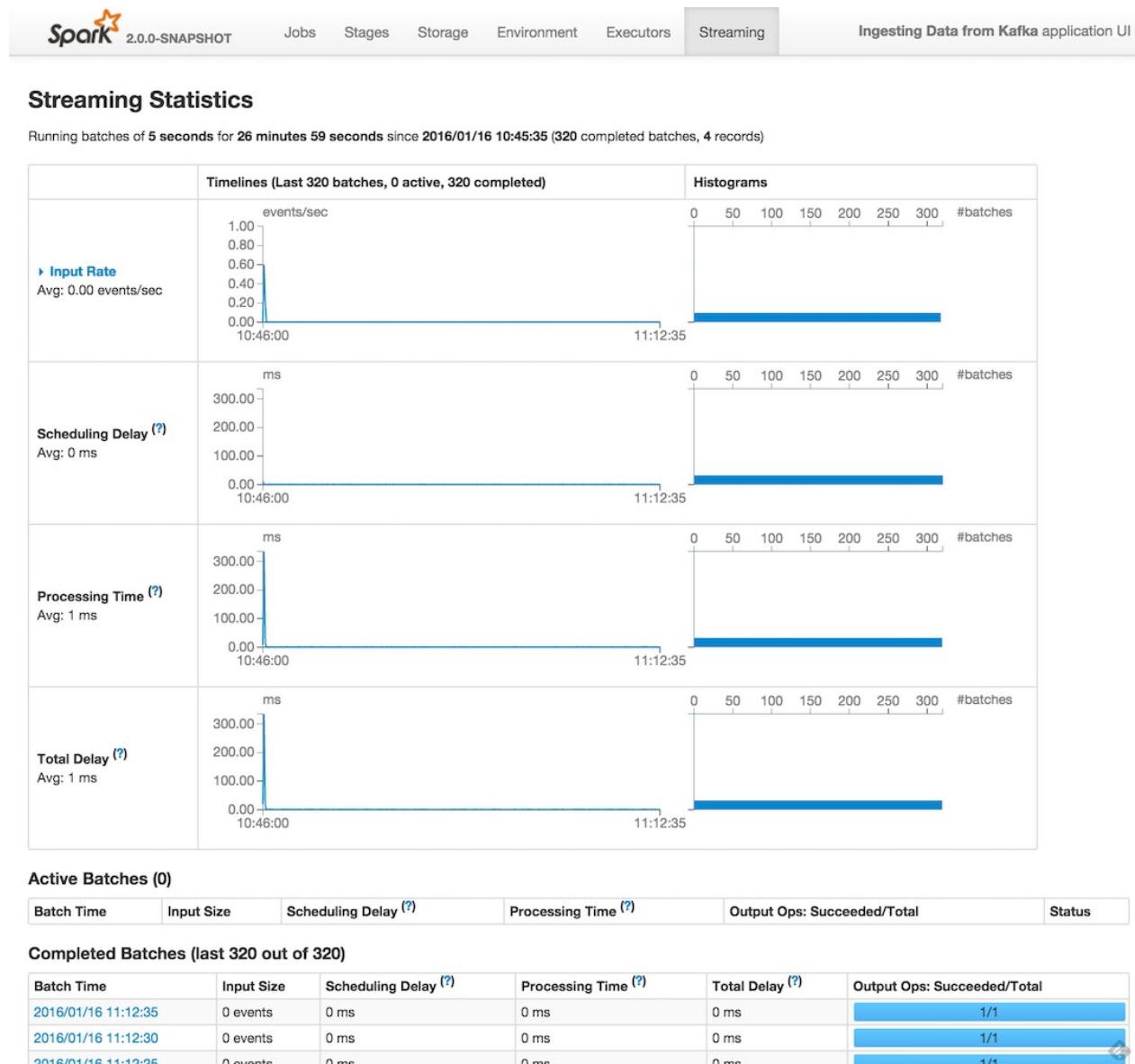


Figure 1. Streaming Tab in web UI

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)

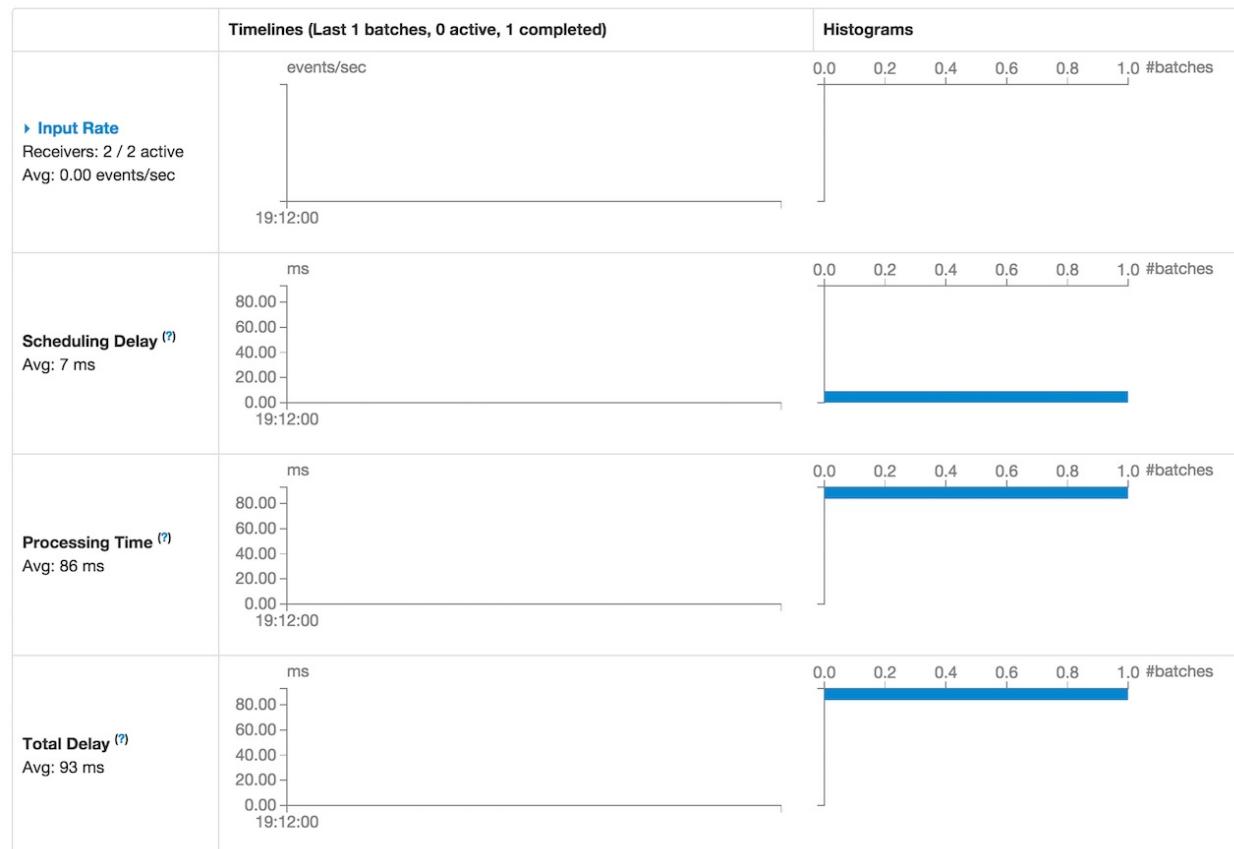


Figure 2. Basic Information section in Streaming Page (with Receivers)

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](#).

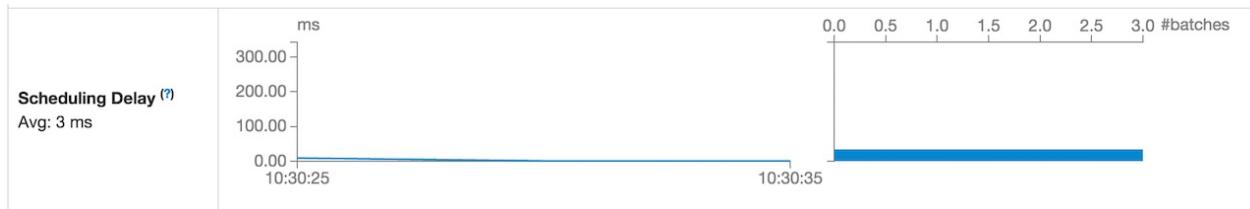


Figure 3. Scheduling Delay in Streaming Page

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)
}
```

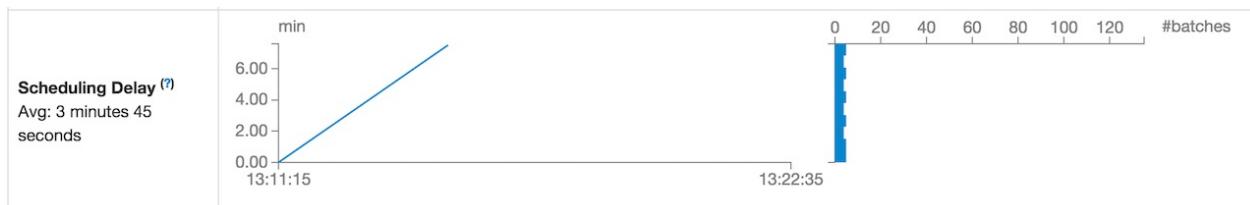


Figure 4. Scheduling Delay Increased in Streaming Page

## Processing Time

**Processing Time** is the time spent to complete all the streaming jobs of a batch.

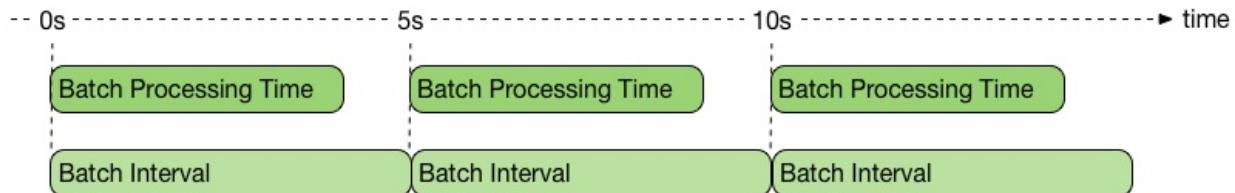


Figure 5. Batch Processing Time and Batch Intervals

## 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 <a href="#">spark.streaming.ui.retainedBatches</a> .
------	--

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 <a href="#">StreamingContext is created</a> and later registered as a <code>StreamingListener</code> and <code>SparkListener</code> when <a href="#">Streaming tab</a> 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 <a href="#">Logging</a> .
-----	------------------------------------

## 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 <a href="#">DStream.updateCheckpointData(currentTime: Time)</a> .
------	---

## 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	Initial checkpoint 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 queue
```

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 [time] ms.
```

`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 <a href="#">CheckpointWriter</a> to queue a <a href="#">checkpoint write for a batch time</a> .
------	--

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 '[checkpointFile]'
```

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. <b>10</b> , 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 '[checkpointFile]', took [bytes] bytes and [time] ms
```

[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 [checkpointFile]
```

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 [checkpointFile]
```

## 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 `read` 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 `read`.

`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.

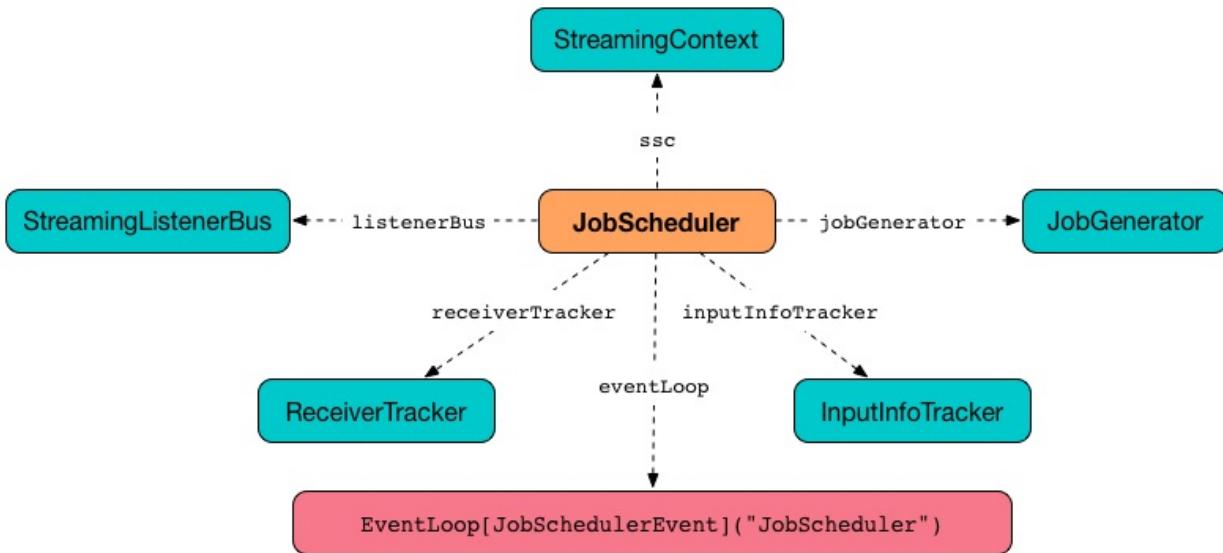


Figure 1. JobScheduler and Dependencies

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.

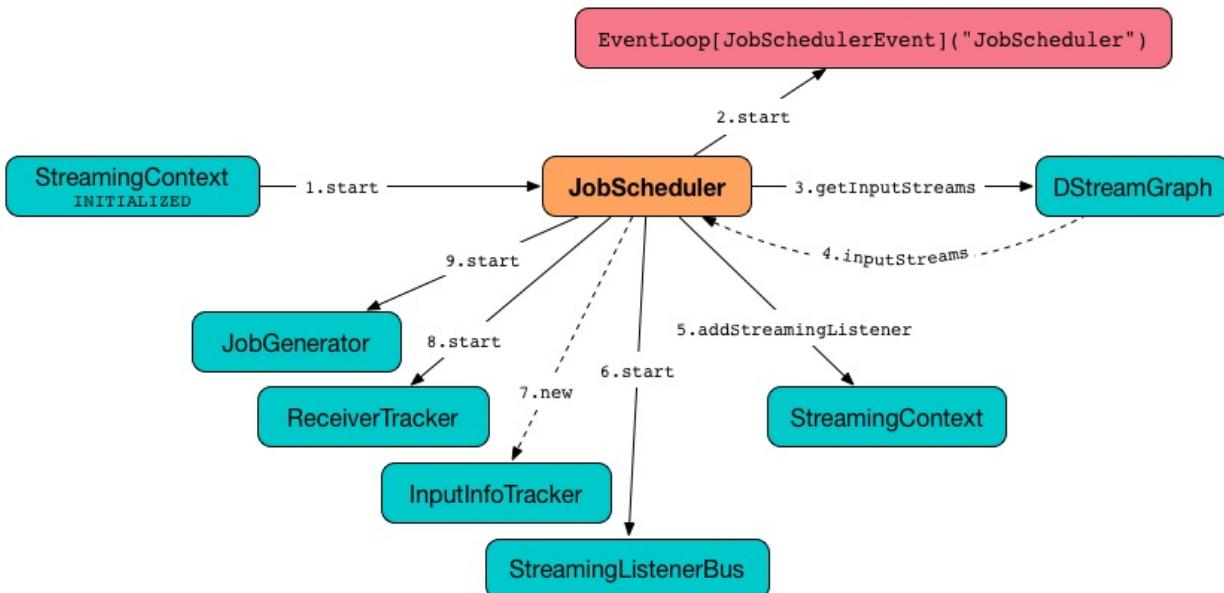


Figure 2. JobScheduler Start procedure

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	<a href="#">FIXME</a>
---------	-----------------------

## Stopping JobScheduler (stop method)

```
stop(processAllReceivedData: Boolean): Unit
```

`stop` stops `JobScheduler`.

Note	It is called when <a href="#">StreamingContext</a> is being stopped.
------	--

You should see the following DEBUG message in the logs:

```
DEBUG JobScheduler: Stopping JobScheduler
```

[ReceiverTracker](#) is stopped.

Note	<a href="#">ReceiverTracker</a> 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 creat
eDirectStream at <console>:36)
...
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 <a href="#">JobHandler</a> to post <code>JobStarted</code> .
------	--

`handleJobStart(job: Job, startTime: Long)` takes a `JobSet` (from `jobs`) 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: [processingDelay] s)
```

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).

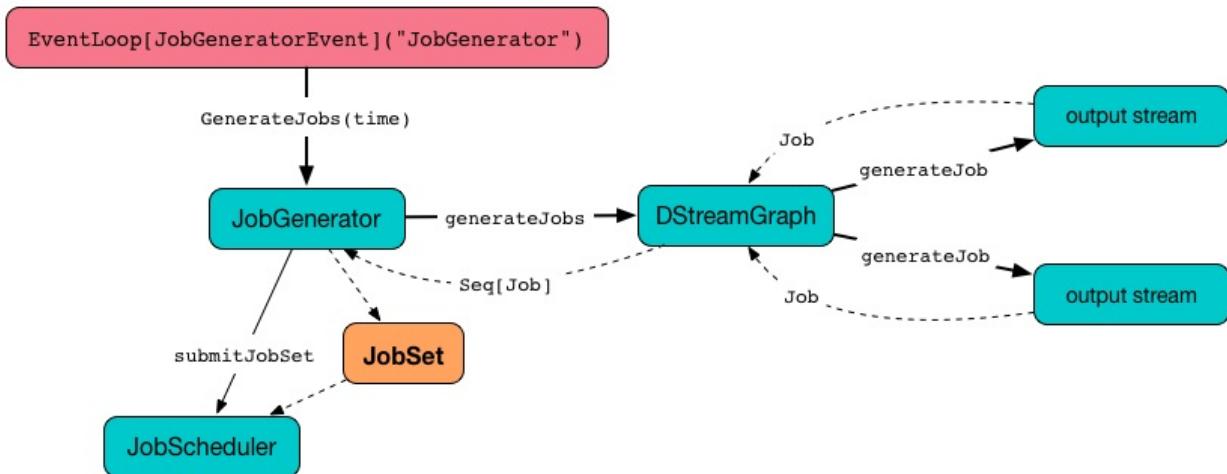


Figure 3. JobSet Created and Submitted to JobScheduler

`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	<b>FIXME</b> 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).

Note	<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"><li>• <a href="#">DirectKafkaInputDStream</a></li><li>• <a href="#">ReceiverInputDStreams - Input Streams with Receivers</a></li><li>• <a href="#">FileInputDStream</a></li></ul>
------	---

## 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	<a href="#">FIXME</a> 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.

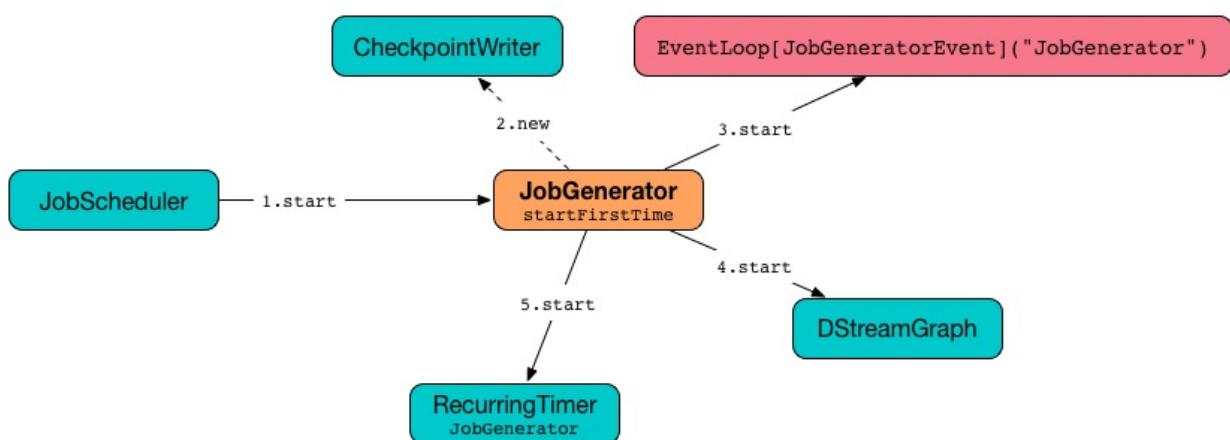


Figure 1. JobGenerator Start (First Time) procedure (tip: follow the numbers)

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.

	<code>JobGenerator</code> is stopped as <a href="#">JobScheduler stops</a> .
Note	<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	<a href="#">FIXME</a> 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 <a href="#">generateJobs</a> 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 <b>eligible for checkpointing</b> when the time interval between current batch <code>time</code> and zero time is a multiple of <code>checkpoint interval</code> .
------	---

Caution	<b>FIXME</b> 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	<b>FIXME</b> 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	clearMetadata events 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	<a href="#">FIXME</a> Describe cleanups of <a href="#">ReceiverTracker</a> and <a href="#">InputInfoTracker</a> .
---------	---

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 <a href="#">checkpoint is saved</a> and <a href="#">checkpoint cleanup is requested</a> .
------	--

`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](#).

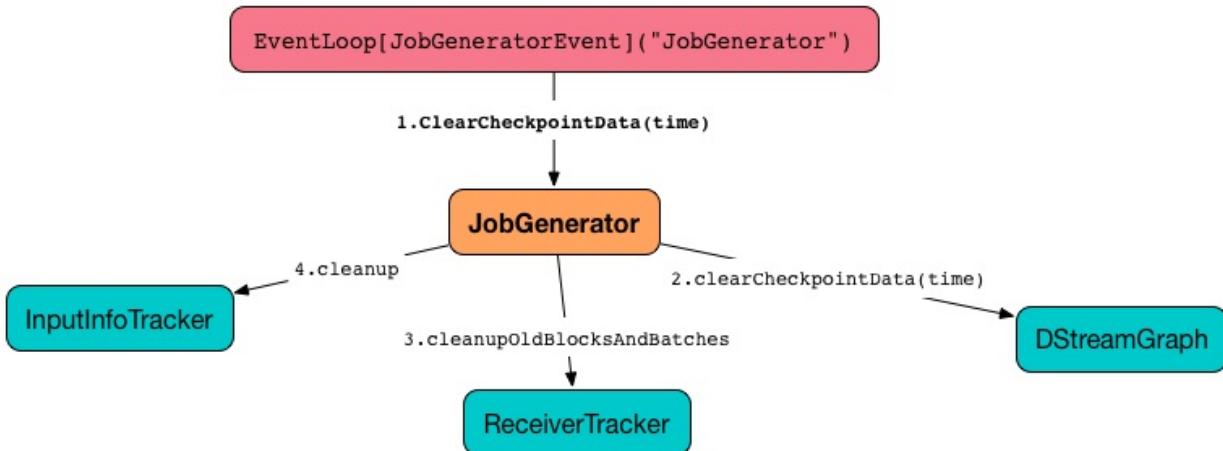


Figure 2. JobGenerator and ClearCheckpointData event

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>
Caution	<p><a href="#">FIXME Report an issue</a></p> <p>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?</p> <p>Answer: See <a href="#">Setup Validation</a>.</p>
Caution	Potential bug: Can <code>streamingContext</code> have no checkpoint duration set? At least, the batch interval <b>must</b> 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.

<b>Tip</b>	<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 <a href="#">Logging</a>.</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 <b>zero time</b> .
------	---

## 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	<a href="#">FIXME</a>
---------	-----------------------

## Restarting DStreamGraph

```
restart(time: Time): Unit
```

`restart` sets [start time](#) to be `time` input parameter.

Note	This is the only moment when <a href="#">zero time</a> can be different than <a href="#">start time</a> .
------	---

## 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 <a href="#">JobGenerator</a> 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 <a href="#">StreamingContext starts</a> .
------	---

## Metadata Cleanup

Note	It is called when <a href="#">JobGenerator clears metadata</a> .
------	--

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 duration already set as [rememberDuration] ms. Cannot set it again.
```

```
at scala.Predef$.require(Predef.scala:219)
at
org.apache.spark.streaming.DStreamGraph.remember(DStreamGraph.scala:79)
at
org.apache.spark.streaming.StreamingContext.remember(StreamingContext.scala:222)
... 43 elided
```

Note	It only makes sense to call <code>remember</code> method before <code>DStreamGraph</code> is started, i.e. before <code>StreamingContext</code> is started, since the output dstreams are only given the remember interval when <code>DStreamGraph</code> 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 <a href="#">DStreamGraph is started</a> .
------	--

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 <a href="#">StateDStreams</a> .
------	---

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 after streaming context has started
  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 been set. Please set it by StreamingContext.checkpoint().
```

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	<a href="#">FIXME</a> 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 RDD.
------	--

Caution	<a href="#">FIXME</a> What happens when <code>SparkContext.runJob(rdd, emptyFunc)</code> is called with the empty function, i.e. <code>(iterator: Iterator[T]) =&gt; {}</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	<a href="#">FIXME</a>
---------	-----------------------

## Checkpoint Cleanup

Caution	<a href="#">FIXME</a>
---------	-----------------------

## `restoreCheckpointData`

```
restoreCheckpointData(): Unit
```

`restoreCheckpointData` does its work only when the internal `transient restoredFromCheckpointData` flag is disabled (i.e. `false`) and is so initially.

Note	<a href="#">restoreCheckpointData</a> method is called when <a href="#">DStreamGraph</a> is requested to <a href="#">restore state of output dstreams</a> .
------	---

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	<a href="#">FIXME</a>
---------	-----------------------

[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 <a href="#">window operators</a> .
------	--

`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	<a href="#">FIXME</a>
---------	-----------------------

## 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	<b>FIXME</b> When could <code>getOrCompute</code> <b>not</b> return an RDD? How does this apply to the StateDStream? 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 RDD (per partition).
------	---

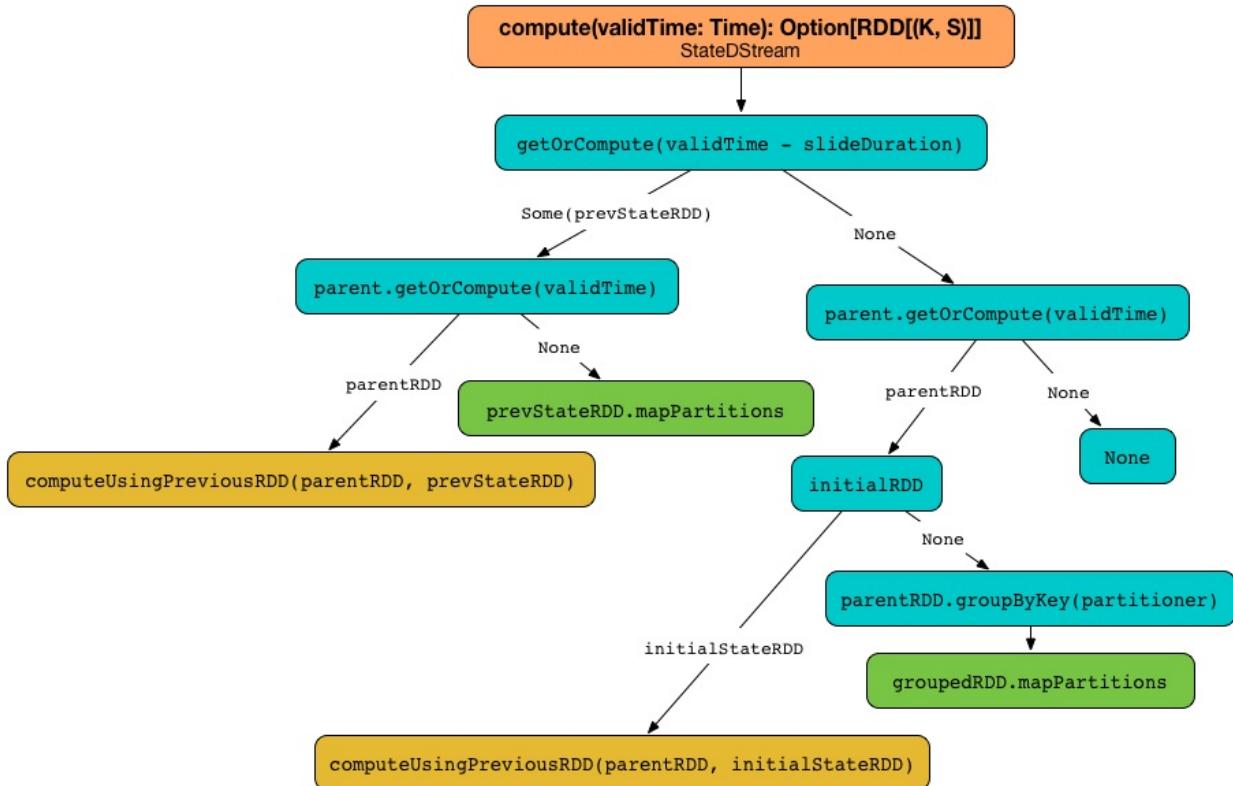


Figure 1. Computing stateful RDDs (StateDStream.compute)

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 `RDD.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, S)]]
```

The `computeUsingPreviousRDD` method uses `cogroup` and `mapPartitions` to build the final state RDD.

Note	Regardless of the return type <code>Option[RDD[(K, S)]]</code> that really allows no state, it will always return <i>some</i> 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 <code>cogroup</code> is as follows and applies to key-value pair RDDs, i.e. <a href="#">RI</a>
	<code>cogroup[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Iterable[V],</code>

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	<a href="#">FIXME</a> Why is the transformation from an Iterable into a Seq so important? Why could not the constructor's <code>updateFunc</code> 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	<a href="#">FIXME</a> Why is <code>preservePartitioning</code> important? What happens when <code>mapPartitions</code> does not preserve partitioning (which by default it does <b>not!</b> )
---------	---



# 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. Return SparkContext.emptyRDD() instead to represent
no element as the result of transformation.
    at
org.apache.spark.streaming.dstream.TransformedDStream.compute(Tr
ansformedDStream.scala:48)
```

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 the type of the elements it processes as well as [StorageLevel](#).

**Note**

You use [StreamingContext.receiverStream](#) method to register a custom `Receiver` to a streaming context.

The abstract `Receiver` class requires the following methods to be implemented (see [Custom Receiver](#)):

- `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`.

## Custom Receiver

```
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.receiver.Receiver

final class MyStringReceiver extends Receiver[String](StorageLevel.NONE) {

    def onStart() = {
        println("onStart called")
    }

    def onStop() = {
        println("onStop called")
    }
}

val ssc = new StreamingContext(sc, Seconds(5))
val strings = ssc.receiverStream(new MyStringReceiver)
strings.print

ssc.start

// MyStringReceiver will print "onStart called"

ssc.stop()

// MyStringReceiver will print "onStop called"
```

# ReceiverTracker

## Introduction

`ReceiverTracker` manages execution of all [Receivers](#).

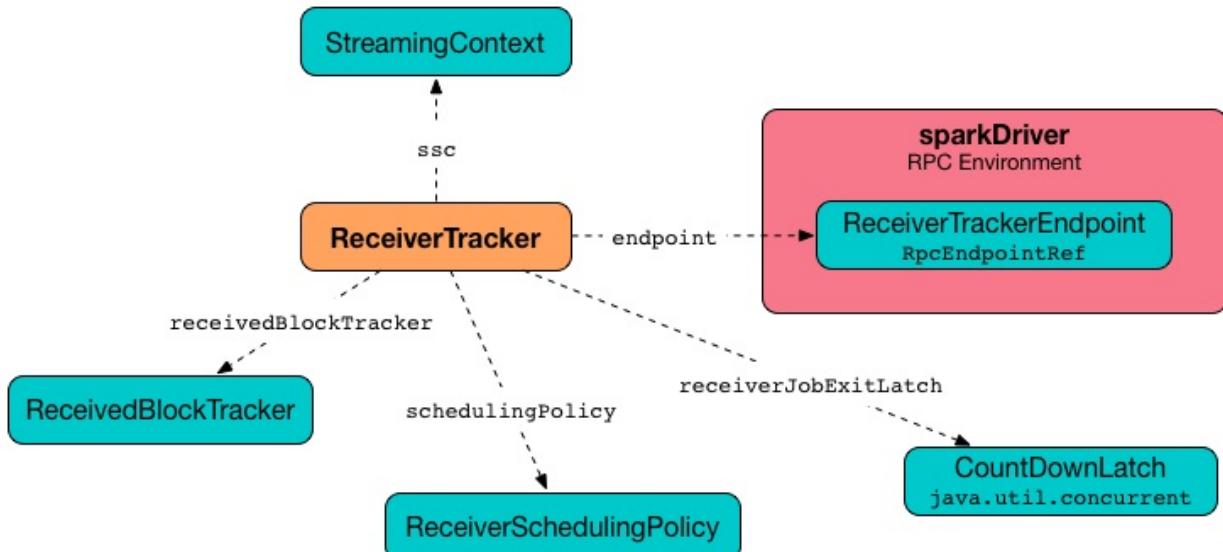


Figure 1. ReceiverTracker and Dependencies

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	<a href="#">FIXME</a>
---------	-----------------------

## hasUnallocatedBlocks

Caution	<a href="#">FIXME</a>
---------	-----------------------

## ReceiverTracker RPC endpoint

Caution	<a href="#">FIXME</a>
---------	-----------------------

## 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	<a href="#">FIXME</a> What does <code>ReceiverSchedulingPolicy.scheduleReceivers(receivers, getExecutors)</code> do?
---------	--

It does *some* bookkeeping.

Caution	<a href="#">FIXME</a> What is <i>the</i> bookkeeping?
---------	---

It finally starts every receiver (using the helper method [ReceiverTrackerEndpoint.startReceiver](#)).

### **ReceiverTrackerEndpoint.startReceiver**

Caution	<a href="#">FIXME</a> When is the method called?
---------	--

`ReceiverTrackerEndpoint.startReceiver(receiver: Receiver[_], scheduledLocations: Seq[TaskLocation])` starts a receiver `Receiver` at the given `Seq[TaskLocation]` locations.

Caution	<a href="#">FIXME</a> 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`.

Tip	Read about <code>TaskContext</code> in <a href="#">TaskContext</a> .
-----	--

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	<a href="#">FIXME</a> 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	<a href="#">FIXME</a> 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	<a href="#">FIXME</a>
---------	-----------------------

## AllReceiverIds

Caution	<a href="#">FIXME</a>
---------	-----------------------

## 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 <a href="#">JobScheduler</a> 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 <a href="#">receiver input streams</a> in use, the method does nothing.
------	---

## ReceivedBlockTracker

Caution	<a href="#">FIXME</a>
---------	-----------------------

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	<a href="#">FIXME</a> 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`.

	<p><b>FIXME Prepare exercises</b></p> <ul style="list-style-type: none"> <li>• for a receiver to call <code>stop(message: String)</code> when a custom "TERMINATE" message arrives</li> <li>• send <code>StopReceiver</code> to a <code>ReceiverTracker</code></li> </ul>
--	---

## ReceiverSupervisorImpl

`ReceiverSupervisorImpl` is the implementation of [ReceiverSupervisor contract](#).

	<p><b>Note</b> A dedicated <code>ReceiverSupervisorImpl</code> is started for every receiver when <a href="#">ReceiverTracker starts</a>. See <a href="#">ReceiverTrackerEndpoint.startReceiver</a>.</p>
--	--

It communicates with [ReceiverTracker](#) that runs on the driver (by posting messages using the [ReceiverTracker RPC endpoint](#)).

	<p><b>Enable DEBUG logging level for</b> <code>org.apache.spark.streaming.receiver.ReceiverSupervisorImpl</code> <b>logger</b> to see what happens in <code>ReceiverSupervisorImpl</code> .</p> <p><b>Tip</b> Add the following line to <code>conf/log4j.properties</code> :</p> <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 <a href="#">ReceiverSupervisorImpl</a> (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.
DEBUG)
Logger.getLogger("org.apache.spark.streaming.DStreamGraph").setLevel(Level.DEBUG)
Logger.getLogger("org.apache.spark.streaming.scheduler.JobGenerator").setLevel(Level.D
BUG)

// 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](ssc, kafkaParams, kafkaTopics)

// 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.

The screenshot shows the Spark Web UI interface. At the top, there's a navigation bar with tabs for Jobs, Stages, Storage, Environment, Executors, Streaming, and "Ingesting Data from Kafka application UI". Below the navigation bar, the title "Spark Jobs (3)" is displayed. A summary section shows "Total Uptime: 15 min", "Scheduling Mode: FIFO", and "Completed Jobs: 3". There's a link to "Event Timeline". The main area shows a table of completed jobs:

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.

2016/01/09 22:17:15	1 events	0 ms	24 ms	24 ms	1/1
---------------------	----------	------	-------	-------	-----

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.

The screenshot shows the Spark Web UI interface. At the top, there's a navigation bar with tabs for Jobs, Stages, Storage, Environment, Executors, Streaming, and "Ingesting Data from Kafka application UI". The "Streaming" tab is selected. Below the navigation bar, the title "Details of batch at 2016/01/09 22:17:15" is displayed. The page displays batch details and a table of output operations:

**Batch Duration:** 15 s  
**Input data size:** 1 records  
**Scheduling delay:** 0 ms  
**Processing time:** 24 ms  
**Total delay:** 24 ms  
**Input Metadata:**

Input	Metadata
Kafka direct stream [0]	topic: spark-topic partition: 0 offsets: 5 to 6

Output Op Id	Description	Duration	Status	Job Id	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total	Error
0	print at <console>:32	+details	Succeeded	2	18 ms	1/1	1/1	

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 <a href="#">DStreamGraph</a> 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] -> [toOffset]
```

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**

`getRestartTime` 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**

`start()` method uses the internal [getStartTime](#) method to calculate `startTime` and calls `start(startTime: Long)`.

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 `callback` one more/last time, i.e. `callback` is executed for a period `after` `RecurringTimer` 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/kb0d3rqn4zb9fcc91pxhgn8w0000gn/T/spark-71dbd43d-2db3-4527-adb8-f1174d799b0d/repl-a6b9bf12-fec2-4004-9236-3b0ab772cc94
INFO ShutdownHookManager: Deleting directory /private/var/folders/0w/kb0d3rqn4zb9fcc91pxhgn8w0000gn/T/spark-71dbd43d-2db3-4527-adb8-f1174d799b0d
```

# 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], integral = [integral], derivative = [derivative], min rate = [minRate]
```

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 delay = [schedulingDelay]
```

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`).

# ExecutorAllocationManager

Caution	<a href="#">FIXME</a>
---------	-----------------------

**requestExecutors**

**killExecutor**

# Settings

The following list are the settings used to configure Spark Streaming applications.

Caution	<a href="#">FIXME</a> 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 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 (a library / an extension) of Apache Spark to provide distributed machine learning algorithms on top of Spark's RDD abstraction. 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](#)

Note	There are two libraries for Machine Learning in Spark MLlib: <code>org.apache.spark.mllib</code> for RDD-based Machine Learning and a higher-level API under <code>org.apache.spark.ml</code> for DataFrame-based Machine Learning with Pipelines.
------	---

**Machine Learning** uses large datasets to identify (infer) patterns and make decisions (aka *predictions*). Automated decision making is what makes Machine Learning so appealing. You can teach a system from a dataset and let the system act by itself to predict future.

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):

- Security monitoring and fraud detection
- Operational optimizations
- Product recommendations or (more broadly) Marketing optimization
- Ad serving and optimization

## Concepts

This section introduces the concepts of Machine Learning and how they are modeled in Spark MLlib.

## Observation

An **observation** is used to learn about or evaluate (i.e. draw conclusions about) the observed item's target value.

Spark models observations as rows in a `DataFrame`.

## Feature

A **feature** (aka *dimension* or *variable*) is an attribute of an observation. It is an **independent variable**.

Spark models features as columns in a `DataFrame` (one per feature or a set of features).

Note	Ultimately, it is up to an algorithm to expect one or many features per column.
------	---

There are two classes of features:

- **Categorical** with *discrete* values, i.e. the set of possible values is limited, and can range from one to many thousands. There is no ordering implied, and so the values are incomparable.
- **Numerical** with *quantitative* values, i.e. any numerical values that you can compare to each other. You can further classify them into **discrete** and **continuous** features.

## Label

A **label** is a variable that a machine learning system learns to predict that are assigned to observations.

There are **categorical** and **numerical** labels.

A label is a **dependent variable** that depends on other dependent or independent variables like features.

## 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

## Further reading or watching

- [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](#)
- (video) [GOTO 2015 • A Taste of Random Decision Forests on Apache Spark • Sean Owen](#)

# ML Pipelines - High-Level API for MLlib

Note	Both <a href="#">scikit-learn</a> and <a href="#">GraphLab</a> have the concept of <b>pipelines</b> built into their system.
------	--

Use of a machine learning algorithm is only one component of a **predictive analytic workflow**. There can also be additional **pre-processing steps** for the machine learning algorithm to work.

Note	It appears that what is called a <i>RDD computation</i> (Spark Core) or a <i>DataFrame manipulation</i> (Spark SQL) or a <i>continuous DStream computation</i> (Spark Streaming) is called a <b>Machine Learning algorithm</b> in Spark MLlib.
------	--

A typical standard machine learning workflow is as follows:

1. Loading data (aka *data ingestion*)
2. Extracting features (aka *feature extraction*)
3. Training model (aka *model training*)
4. Evaluate (or *predictionize*)

You may also think of two additional steps before the final model becomes production ready and hence of any use:

1. Testing model (aka *model testing*)
2. Selecting the best model (aka *model selection* or *model tuning*)
3. Deploying model (aka *model deployment and integration*)

The goal of the **Pipeline API** (aka **Spark ML** or **spark.ml** given the package the API lives in) is to let users quickly and easily assemble and configure practical distributed machine learning pipelines (aka workflows) by standardizing the APIs for different Machine Learning concepts.

The ML Pipeline API is a new DataFrame-based API developed under the `spark.ml` package.

Note	The old RDD-based API has been developed in parallel under the <code>spark.mllib</code> package. It has been proposed to switch RDD-based MLlib APIs to maintenance mode in Spark 2.0.
------	--

Note	The Pipeline API lives under <a href="#">org.apache.spark.ml</a> package.
------	---

The key concepts of Pipeline API (aka **spark.ml Components**):

- [Pipelines](#) and [PipelineStages](#)
- [Transformers](#)
  - [Models](#)
- [Estimators](#)
- [Evaluators](#)
- [Params \(and ParamMaps\)](#)

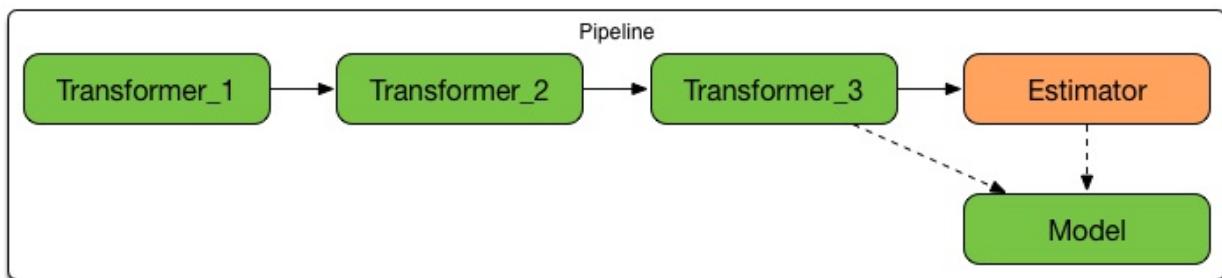


Figure 1. Pipeline with Transformers and Estimator (and corresponding Model)

The beauty of using Spark ML is that the **ML dataset** is simply a [DataFrame](#) (and all calculations are simply [UDF applications](#) on columns).

Given the Pipeline Components, a typical machine learning pipeline is as follows:

- You use a collection of `Transformer` instances to prepare input `DataFrame` - the dataset with proper input data (in columns) for a chosen ML algorithm.
- You then fit (aka *build*) a `Model`.
- With a `Model` you can calculate predictions (in `prediction` column) on `features` input column through DataFrame transformation.

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 fit a [PipelineModel](#) to an input dataset.

```
pipeline: DataFrame =[fit]=> DataFrame (using transformers and estimators)
```

A pipeline is represented by [Pipeline class](#).

```
import org.apache.spark.ml.Pipeline
```

`Pipeline` is also 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.

```
val pipeline = new Pipeline()

scala> println(pipeline.uid)
pipeline_94be47c3b709

val pipeline = new Pipeline("my_pipeline")

scala> println(pipeline.uid)
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 = (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 `Transformer` 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):

- [Estimators](#)
- [Models](#)
- [Pipeline](#)
- [Predictor](#)
- [Transformer](#)

Each `PipelineStage` transforms schema using `transformSchema` family of methods:

```
transformSchema(schema: StructType): StructType
transformSchema(schema: StructType, logging: Boolean): StructType
```

#### Note

[StructType](#) describes a schema of a `DataFrame`.

**Tip**

Enable `DEBUG` logging level for the respective `PipelineStage` implementations to see what happens beneath.

## 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 \(Databricks\)](#)
- (video) [Spark MLlib: Making Practical Machine Learning Easy and Scalable](#)

# Transformers

A **transformer** is a function object that maps (aka *transforms*) a `DataFrame` into another `DataFrame` (both called *datasets*).

```
transformer: DataFrame =[transform]=> DataFrame
```

Transformers prepare a dataset for an machine learning algorithm to work with. They are also very helpful to transform DataFrames in general (even outside the machine learning space).

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[_]*): DataFrame
```

A `Transformer` is a [PipelineStage](#) and thus can be a part of a [Pipeline](#).

A few available implementations of `Transformer` :

- [StopWordsRemover](#)
- [Binarizer](#)
- [SQLTransformer](#)
- [VectorAssembler](#) — a feature transformer that assembles (merges) multiple columns into a (feature) vector column.
- [UnaryTransformer](#)
  - [Tokenizer](#)
  - [RegexTokenizer](#)
  - [NGram](#)
  - [HashingTF](#)
  - [OneHotEncoder](#)
- [Model](#)

See [Custom UnaryTransformer](#) section for a custom `Transformer` implementation.

## StopWordsRemover

`StopWordsRemover` is a machine learning feature transformer that takes a string array column and outputs a string array column with all defined stop words removed. The transformer comes with a standard set of [English stop words](#) as default (that are the same as scikit-learn uses, i.e. [from the Glasgow Information Retrieval Group](#)).

Note	It works as if it were a <a href="#">UnaryTransformer</a> but it has not been migrated to extend the class yet.
------	---

`StopwordsRemover` class belongs to `org.apache.spark.ml.feature` package.

```
import org.apache.spark.ml.feature.StopWordsRemover
val stopWords = new StopWordsRemover
```

It accepts the following parameters:

```
scala> println(stopWords.explainParams)
caseSensitive: whether to do case-sensitive comparison during filtering (default: false
)
inputCol: input column name (undefined)
outputCol: output column name (default: stopWords_9c2c0fdd8a68__output)
stopWords: stop words (default: [Ljava.lang.String;@5dabe7c8)
```

Note	null values from the input array are preserved unless adding null to stopwords explicitly.
------	--

```

import org.apache.spark.ml.feature.RegexTokenizer
val regexTok = new RegexTokenizer("regexTok")
  .setInputCol("text")
  .setPattern("\\\\W+")

import org.apache.spark.ml.feature.StopWordsRemover
val stopWords = new StopWordsRemover("stopWords")
  .setInputCol(regexTok.getOutputCol)

val df = Seq("please find it done (and empty)", "About to be rich!", "empty")
  .zipWithIndex
  .toDF("text", "id")

scala> stopWords.transform(regexTok.transform(df)).show(false)
+-----+-----+-----+
|text          |id   |regexTok__output           |stopWords__o
|utput|
+-----+-----+-----+
-----+
|please find it done (and empty)|0  |[please, find, it, done, and, empty]||[]
 |
|About to be rich!            |1  |[about, to, be, rich]        |[rich]
 |
|empty                        |2  |[empty]                      |[]
 |
+-----+-----+-----+
-----+

```

## Binarizer

`Binarizer` is a `Transformer` that splits the values in the input column into two groups - "ones" for values larger than the `threshold` and "zeros" for the others.

It works with `DataFrames` with the input column of `DoubleType` or `VectorUDT`. The type of the result output column matches the type of the input column, i.e. `DoubleType` or `VectorUDT`.

```

import org.apache.spark.ml.feature.Binarizer
val bin = new Binarizer()
  .setInputCol("rating")
  .setOutputCol("label")
  .setThreshold(3.5)

scala> println(bin.explainParams)
inputCol: input column name (current: rating)
outputCol: output column name (default: binarizer_dd9710e2a831__output, current: label
)
threshold: threshold used to binarize continuous features (default: 0.0, current: 3.5)

val doubles = Seq((0, 1d), (1, 1d), (2, 5d)).toDF("id", "rating")

scala> bin.transform(doubles).show
+---+-----+
| id|rating|label|
+---+-----+
|  0|    1.0|   0.0|
|  1|    1.0|   0.0|
|  2|    5.0|   1.0|
+---+-----+

import org.apache.spark.mllib.linalg.Vectors
val denseVec = Vectors.dense(Array(4.0, 0.4, 3.7, 1.5))
val vectors = Seq((0, denseVec)).toDF("id", "rating")

scala> bin.transform(vectors).show
+---+-----+-----+
| id|      rating|      label|
+---+-----+-----+
|  0|[4.0,0.4,3.7,1.5]| [1.0,0.0,1.0,0.0]|
+---+-----+-----+

```

## SQLTransformer

`SQLTransformer` is a `Transformer` that does transformations by executing `SELECT ... FROM THIS` with `THIS` being the underlying temporary table registered for the input dataset.

Internally, `THIS` is replaced with a random name for a temporary table (using `registerTempTable`).

Note	It has been available since Spark 1.6.0.
------	--

It requires that the `SELECT` query uses `THIS` that corresponds to a temporary table and simply executes the mandatory `statement` using `sql` method.

You have to specify the mandatory `statement` parameter using `setStatement` method.

```
import org.apache.spark.ml.feature.SQLTransformer
val sql = new SQLTransformer()

// dataset to work with
val df = Seq((0, s"""hello\tworld"""), (1, "two  spaces inside")).toDF("label", "sentence")

scala> sql.setStatement("SELECT sentence FROM __THIS__ WHERE label = 0").transform(df)
.show
+-----+
| sentence|
+-----+
|hello world|
+-----+

scala> println(sql.explainParams)
statement: SQL statement (current: SELECT sentence FROM __THIS__ WHERE label = 0)
```

## VectorAssembler

`VectorAssembler` is a **feature transformer** that assembles (merges) multiple columns into a (feature) vector column.

It supports columns of the types `NumericType`, `BooleanType`, and `VectorUDT`. Doubles are passed on untouched. Other numeric types and booleans are `cast` to doubles.

```

import org.apache.spark.ml.feature.VectorAssembler
val vecAssembler = new VectorAssembler()

scala> print(vecAssembler.explainParams)
inputCols: input column names (undefined)
outputCol: output column name (default: vecAssembler_5ac31099dbeec__output)

final case class Record(id: Int, n1: Int, n2: Double, flag: Boolean)
val ds = Seq(Record(0, 4, 2.0, true)).toDS

scala> ds.printSchema
root
|-- id: integer (nullable = false)
|-- n1: integer (nullable = false)
|-- n2: double (nullable = false)
|-- flag: boolean (nullable = false)

val features = vecAssembler
  .setInputCols(Array("n1", "n2", "flag"))
  .setOutputCol("features")
  .transform(ds)

scala> features.printSchema
root
|-- id: integer (nullable = false)
|-- n1: integer (nullable = false)
|-- n2: double (nullable = false)
|-- flag: boolean (nullable = false)
|-- features: vector (nullable = true)

scala> features.show
+---+---+---+-----+
| id| n1| n2|flag|    features|
+---+---+---+-----+
| 0|  4|2.0|true|[4.0,2.0,1.0]|
+---+---+---+-----+

```

## 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 and so are *chainable*):

- `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	<code>createTransformFunc</code> function is abstract and defined by concrete <code>UnaryTransformer</code> objects.
------	--

Internally, `transform` method 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 <code>udf</code> and <code>withColumn</code> methods from Spark SQL demonstrates an excellent integration between the Spark modules: MLlib and SQL.
------	---

The following are `UnaryTransformer` implementations in spark.ml:

- [Tokenizer](#) that converts the input string to lowercase and then splits it by white spaces.
- [RegexTokenizer](#) that extracts tokens.
- [NGram](#) that converts the input array of strings into an array of n-grams.
- [HashingTF](#) that maps a sequence of terms to their term frequencies (cf. [SPARK-13998](#)  
[HashingTF should extend UnaryTransformer](#))
- [OneHotEncoder](#) that maps a numeric input column of label indices onto a column of binary vectors.

## Tokenizer

`Tokenizer` is a [UnaryTransformer](#) that converts the input string to lowercase and then splits it by white spaces.

```

import org.apache.spark.ml.feature.Tokenizer
val tok = new Tokenizer()

// dataset to transform
val df = Seq((1, "Hello world!"), (2, "Here is yet another sentence.")).toDF("label",
"sentence")

val tokenized = tok.setInputCol("sentence").transform(df)

scala> tokenized.show(false)
+-----+-----+
|label|sentence           |tok_b66af4001c8d__output   |
+-----+-----+
|1    |Hello world!        |[hello, world!]          |
|2    |Here is yet another sentence.|[here, is, yet, another, sentence.]|
+-----+-----+

```

## RegexTokenizer

`RegexTokenizer` is a [UnaryTransformer](#) that tokenizes a `String` into a collection of `String`.

```

import org.apache.spark.ml.feature.RegexTokenizer
val regexTok = new RegexTokenizer()

// dataset to transform with tabs and spaces
val df = Seq((0, s"""hello\tworld"""), (1, "two  spaces inside")).toDF("label", "sentence")

val tokenized = regexTok.setInputCol("sentence").transform(df)

scala> tokenized.show(false)
+-----+-----+
|label|sentence           |regexTok_810b87af9510__output|
+-----+-----+
|0    |hello    world      |[hello, world]          |
|1    |two  spaces inside|[two, spaces, inside]     |
+-----+-----+

```

Note	Read the official scaladoc for <a href="#">org.apache.spark.ml.feature.RegexTokenizer</a> .
------	---

It supports `minTokenLength` parameter that is the minimum token length that you can change using `setMinTokenLength` method. It simply filters out smaller tokens and defaults to `1`.

```
// see above to set up the vals

scala> rt.setInputCol("line").setMinTokenLength(6).transform(df).show
+-----+
|label|          line|regexTok_8c74c5e8b83a__output|
+-----+
|  1|      hello world|          []
|  2|yet another sentence| [another, sentence]|
+-----+
```

It has `gaps` parameter that indicates whether regex splits on gaps (`true`) or matches tokens (`false`). You can set it using `setGaps`. It defaults to `true`.

When set to `true` (i.e. splits on gaps) it uses `Regex.split` while `Regex.findAllIn` for `false`.

```
scala> rt.setInputCol("line").setGaps(false).transform(df).show
+-----+
|label|          line|regexTok_8c74c5e8b83a__output|
+-----+
|  1|      hello world|          []
|  2|yet another sentence| [another, sentence]|
+-----+

scala> rt.setInputCol("line").setGaps(false).setPattern("\\\\w").transform(df).show(false)
)
+-----+
|label|line          |regexTok_8c74c5e8b83a__output|
+-----+
| 1  |hello world    |[]           |
| 2  |yet another sentence|[another, sentence]|
+-----+
```

It has `pattern` parameter that is the regex for tokenizing. It uses Scala's `.r` method to convert the string to regex. Use `setPattern` to set it. It defaults to `\\\\s+`.

It has `toLowerCase` parameter that indicates whether to convert all characters to lowercase before tokenizing. Use `setToLowercase` to change it. It defaults to `true`.

## NGram

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]|
+---+-----+

```

## HashingTF

Another example of a transformer is [org.apache.spark.ml.feature.HashingTF](#) that works on a `Column of ArrayType`.

It transforms the rows for the input column into a sparse term frequency vector.

```

import org.apache.spark.ml.feature.HashingTF
val hashingTF = new HashingTF()
  .setInputCol("words")
  .setOutputCol("features")
  .setNumFeatures(5000)

// see above for regexTok transformer
val regexedDF = regexTok.transform(df)

// Use HashingTF
val hashedDF = hashingTF.transform(regexedDF)

scala> hashedDF.show(false)
+---+-----+-----+-----+
|id |text          |words           |features          |
+---+-----+-----+-----+
|0  |hello        |[hello, world]  |(5000,[2322,3802],[1.0,1.0])|
|1  |two spaces inside|[two, spaces, inside]|(5000,[276,940,2533],[1.0,1.0,1.0])|
+---+-----+-----+-----+

```

The name of the output column is optional, and if not specified, it becomes the identifier of a `HashingTF` object with the `_output` suffix.

```
scala> hashingTF.uid
res7: String = hashingTF_fe3554836819

scala> hashingTF.transform(regexDF).show(false)
+---+-----+-----+
---+-----+-----+
|id |text           |words           |hashingTF_fe3554836819__output
|   |
+---+-----+-----+
---+
|0  |hello      world      |[hello, world]      |([262144,[71890,72594],[1.0,1.0])
|   |
|1  |two spaces inside|[two, spaces, inside]|([262144,[53244,77869,115276],[1.0,1.0,1.0
])|
+---+-----+-----+
---+
```

## OneHotEncoder

`OneHotEncoder` is a `Tokenizer` that maps a numeric input column of label indices onto a column of binary vectors.

```
// dataset to transform
val df = Seq(
  (0, "a"), (1, "b"),
  (2, "c"), (3, "a"),
  (4, "a"), (5, "c"))
  .toDF("label", "category")
import org.apache.spark.ml.feature.StringIndexer
val indexer = new StringIndexer().setInputCol("category").setOutputCol("cat_index").fit(df)
val indexed = indexer.transform(df)

import org.apache.spark.sql.types.NumericType

scala> indexed.schema("cat_index").dataType.newInstance[NumericType]
res0: Boolean = true

import org.apache.spark.ml.feature.OneHotEncoder
val oneHot = new OneHotEncoder()
  .setInputCol("cat_index")
  .setOutputCol("cat_vec")

val oneHotted = oneHot.transform(indexed)

scala> oneHotted.show(false)
+---+---+---+---+
|label|category|cat_index|cat_vec      |
+---+---+---+---+
|0    |a        |0.0       |(2,[0],[1.0])|
|1    |b        |2.0       |(2,[],[])   |
|2    |c        |1.0       |(2,[1],[1.0])|
|3    |a        |0.0       |(2,[0],[1.0])|
|4    |a        |0.0       |(2,[0],[1.0])|
|5    |c        |1.0       |(2,[1],[1.0])|
+---+---+---+---+
scala> oneHotted.printSchema
root
 |-- label: integer (nullable = false)
 |-- category: string (nullable = true)
 |-- cat_index: double (nullable = true)
 |-- cat_vec: vector (nullable = true)

scala> oneHotted.schema("cat_vec").dataType.newInstance[VectorUDT]
res1: Boolean = true
```

## Custom UnaryTransformer

The following class is a custom `UnaryTransformer` that transforms words using upper letters.

```
package pl.japila.spark

import org.apache.spark.ml._
import org.apache.spark.ml.util.Identifiable
import org.apache.spark.sql.types._

class UpperTransformer(override val uid: String)
    extends UnaryTransformer[String, String, UpperTransformer] {

  def this() = this(Identifiable.randomUUID("upper"))

  override protected def validateInputType(inputType: DataType): Unit = {
    require(inputType == StringType)
  }

  protected def createTransformFunc: String => String = {
    _.toUpperCase
  }

  protected def outputDataType: DataType = StringType
}
```

Given a `DataFrame` you could use it as follows:

```
val upper = new UpperTransformer

scala> upper.setInputCol("text").transform(df).show
+---+-----+
| id| text|upper_0b559125fd61__output|
+---+-----+
|  0|hello|          HELLO|
|  1|world|          WORLD|
+---+-----+
```

# Estimators

An **estimator** is an abstraction of a **learning algorithm** that **fits a model** on a dataset.

**Note**

That was so machine learning to explain an estimator this way, *wasn't it?* It is that the more I spend time with Pipeline API the often I use the terms and phrases from this space. Sorry.

Technically, an `Estimator` produces a `Model` (i.e. a `Transformer`) for a given `DataFrame` and parameters (as `ParamMap`). It fits a model to the input `DataFrame` and `ParamMap` to produce a `Transformer` (a `Model`) that can calculate predictions for any `DataFrame`-based input datasets.

It is basically a function that maps a `DataFrame` onto a `Model` through `fit` method, i.e. it takes a `DataFrame` and produces a `Transformer` as a `Model`.

```
estimator: DataFrame =[fit]=> Model
```

Estimators are instances of `org.apache.spark.ml.Estimator` abstract class that comes with `fit` method (with the return type `M` being a `Model`):

```
fit(dataset: DataFrame): M
```

An `Estimator` is a `PipelineStage` (so it can be a part of a `Pipeline`).

**Note**

Pipeline considers `Estimator` special and executes `fit` method before `transform` (as for other `Transformer` objects in a pipeline). Consult [Pipeline](#) document.

As an example you could use [LinearRegression](#) learning algorithm estimator to train a [LinearRegressionModel](#).

Some of the direct specialized implementations of the `Estimator` abstract class are as follows:

- [StringIndexer](#)
- [KMeans](#)
- [TrainValidationSplit](#)
- [Predictors](#)

## StringIndexer

```
org.apache.spark.ml.feature.StringIndexer is an Estimator that produces
StringIndexerModel .

val df = ('a' to 'a' + 9).map(_.toString)
.zip(0 to 9)
.map(_.swap)
.toDF("id", "label")

import org.apache.spark.ml.feature.StringIndexer
val strIdx = new StringIndexer()
.setInputCol("label")
.setOutputCol("index")

scala> println(strIdx.explainParams)
handleInvalid: how to handle invalid entries. Options are skip (which will filter out
rows with bad values), or error (which will throw an error). More options may be added
later (default: error)
inputCol: input column name (current: label)
outputCol: output column name (default: strIdx_ded89298e014__output, current: index)

val model = strIdx.fit(df)
val indexed = model.transform(df)

scala> indexed.show
+---+-----+
| id|label|index|
+---+-----+
|  0|    a|  3.0|
|  1|    b|  5.0|
|  2|    c|  7.0|
|  3|    d|  9.0|
|  4|    e|  0.0|
|  5|    f|  2.0|
|  6|    g|  6.0|
|  7|    h|  8.0|
|  8|    i|  4.0|
|  9|    j|  1.0|
+---+-----+
```

## KMeans

`KMeans` class is an implementation of the K-means clustering algorithm in machine learning with support for **k-means||** (aka **k-means parallel**) in Spark MLlib.

Roughly, k-means is an unsupervised iterative algorithm that groups input data in a predefined number of `k` clusters. Each cluster has a **centroid** which is a cluster center. It is a highly iterative machine learning algorithm that measures the distance (between a vector

and centroids) as the nearest mean. The algorithm steps are repeated till the convergence of a specified number of steps.

Note	K-Means algorithm uses <a href="#">Lloyd's algorithm</a> in computer science.
------	---

It is an `Estimator` that produces a `KMeansModel`.

Tip	Do <code>import org.apache.spark.ml.clustering.KMeans</code> to work with <code>KMeans</code> algorithm.
-----	--

`KMeans` defaults to use the following values:

- Number of clusters or centroids (`k`): `2`
- Maximum number of iterations (`maxIter`): `20`
- Initialization algorithm (`initMode`): `k-means||`
- Number of steps for the k-means|| (`initSteps`): `5`
- Convergence tolerance (`tol`): `1e-4`

```
import org.apache.spark.ml.clustering._
val kmeans = new KMeans()

scala> println(kmeans.explainParams)
featuresCol: features column name (default: features)
initMode: initialization algorithm (default: k-means||)
initSteps: number of steps for k-means|| (default: 5)
k: number of clusters to create (default: 2)
maxIter: maximum number of iterations (>= 0) (default: 20)
predictionCol: prediction column name (default: prediction)
seed: random seed (default: -1689246527)
tol: the convergence tolerance for iterative algorithms (default: 1.0E-4)
```

`KMeans` assumes that `featuresCol` is of type `VectorUDT` and appends `predictionCol` of type `IntegerType`.

Internally, `fit` method "unwraps" the feature vector in `featuresCol` column in the input `DataFrame` and creates an `RDD[Vector]`. It then hands the call over to the MLlib variant of `KMeans` in `org.apache.spark.mllib.clustering.KMeans`. The result is copied to `KMeansModel` with a calculated `KMeansSummary`.

Each item (row) in a data set is described by a numeric vector of attributes called `features`. A single feature (a dimension of the vector) represents a word (token) with a value that is a metric that defines the importance of that word or term in the document.

Enable `INFO` logging level for `org.apache.spark.mllib.clustering.KMeans` logger to see what happens inside a `KMeans`.

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

**Tip**

```
log4j.logger.org.apache.spark.mllib.clustering.KMeans=INFO
```

Refer to [Logging](#).

## KMeans Example

You can represent a text corpus (document collection) using the vector space model. In this representation, the vectors have dimension that is the number of different words in the corpus. It is quite natural to have vectors with a lot of zero values as not all words will be in a document. We will use an optimized memory representation to avoid zero values using [sparse vectors](#).

This example shows how to use k-means to classify emails as a spam or not.

```
// NOTE Don't copy and paste the final case class with the other lines
// It won't work with paste mode in spark-shell
final case class Email(id: Int, text: String)

val emails = Seq(
  "This is an email from your lovely wife. Your mom says...", 
  "SPAM SPAM spam",
  "Hello, We'd like to offer you").zipWithIndex.map(_.swap).toDF("id", "text").as[Email]

// Prepare data for k-means
// Pass emails through a "pipeline" of transformers
import org.apache.spark.ml.feature._
val tok = new RegexTokenizer()
  .setInputCol("text")
  .setOutputCol("tokens")
  .setPattern("\\w+")

val hashTF = new HashingTF()
  .setInputCol("tokens")
  .setOutputCol("features")
  .setNumFeatures(20)

val preprocess = (tok.transform _).andThen(hashTF.transform)

val features = preprocess(emails.toDF)

scala> features.select('text, 'features).show(false)
```

```
+-----+-----+
|text |features
|-----+-----+
|This is an email from your lovely wife. Your mom says...|(20,[0,3,6,8,10,11,17,19],[1
.0,2.0,1.0,1.0,2.0,1.0,2.0,1.0])|
|SPAM SPAM spam |(20,[13],[3.0])
|Hello, We'd like to offer you |(20,[0,2,7,10,11,19],[2.0,1.0
,1.0,1.0,1.0,1.0])|
+-----+-----+
-----+-----+
```

```
import org.apache.spark.ml.clustering.KMeans
val kmeans = new KMeans
```

```
scala> val kmModel = kmeans.fit(features.toDF)
16/04/08 15:57:37 WARN KMeans: The input data is not directly cached, which may hurt p
erformance if its parent RDDs are also uncached.
16/04/08 15:57:37 INFO KMeans: Initialization with k-means|| took 0.219 seconds.
16/04/08 15:57:37 INFO KMeans: Run 0 finished in 1 iterations
16/04/08 15:57:37 INFO KMeans: Iterations took 0.030 seconds.
16/04/08 15:57:37 INFO KMeans: KMeans converged in 1 iterations.
16/04/08 15:57:37 INFO KMeans: The cost for the best run is 5.000000000000002.
16/04/08 15:57:37 WARN KMeans: The input data was not directly cached, which may hurt
performance if its parent RDDs are also uncached.
kmModel: org.apache.spark.ml.clustering.KMeansModel = kmeans_7a13a617ce0b
```

```
scala> kmModel.clusterCenters.map(_.toSparse)
res36: Array[org.apache.spark.mllib.linalg.SparseVector] = Array((20,[13],[3.0]), (20,[
0,2,3,6,7,8,10,11,17,19],[1.5,0.5,1.0,0.5,0.5,0.5,1.5,1.0,1.0,1.0]))
```

```
val email = Seq("hello mom").toDF("text")
val result = kmModel.transform(preprocess(email))
```

```
scala> .show(false)
+-----+-----+-----+-----+
|text |tokens |features |prediction|
+-----+-----+-----+-----+
|hello mom|[hello, mom]|(20,[2,19],[1.0,1.0))|1 |
+-----+-----+-----+-----+
```

## TrainValidationSplit

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

## Predictors

A `Predictor` is a specialization of `Estimator` for a `PredictionModel` with its own abstract `train` method.

```
train(dataset: DataFrame): M
```

The `train` method is supposed to ease dealing with schema validation and copying parameters to a trained `PredictionModel` model. It also sets the parent of the model to itself.

A `Predictor` is basically a function that maps a `DataFrame` onto a `PredictionModel`.

```
predictor: DataFrame =[train]=> PredictionModel
```

It implements the abstract `fit(dataset: DataFrame)` of the `Estimator` abstract class that validates and transforms the schema of a dataset (using a custom `transformSchema` of `PipelineStage`), and then calls the abstract `train` method.

Validation and transformation of a schema (using `transformSchema`) makes sure that:

1. `features` column exists and is of correct type (defaults to `Vector`).
2. `label` column exists and is of `Double` type.

As the last step, it adds the `prediction` column of `Double` type.

The following is a list of `Predictor` examples for different learning algorithms:

- [DecisionTreeClassifier](#)
- [LinearRegression](#)
- [RandomForestRegressor](#)

## DecisionTreeClassifier

`DecisionTreeClassifier` is a `ProbabilisticClassifier` that...

Caution	<a href="#">FIXME</a>
---------	-----------------------

## LinearRegression

`LinearRegression` is an example of `Predictor` (indirectly through the specialized `Regressor` private abstract class), and hence a `Estimator`, that represents the `linear regression` algorithm in Machine Learning.

`LinearRegression` belongs to `org.apache.spark.ml.regression` package.

## Tip

Read the scaladoc of [LinearRegression](#).

It expects `org.apache.spark.mllib.linalg.Vector` as the input type of the column in a dataset and produces [LinearRegressionModel](#).

```
import org.apache.spark.ml.regression.LinearRegression
val lr = new LinearRegression
```

The acceptable parameters:

```
scala> println(lr.explainParams)
elasticNetParam: the ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the
penalty is an L2 penalty. For alpha = 1, it is an L1 penalty (default: 0.0)
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 va
lue is 'auto' (default: auto)
standardization: whether to standardize the training features before fitting the model
(default: true)
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 weig
hts as 1.0 (default: )
```

## LinearRegression.train

```
train(dataset: DataFrame): LinearRegressionModel
```

`train` (protected) method of `LinearRegression` expects a `dataset` `DataFrame` with two columns:

1. `label` of type `DoubleType` .
2. `features` of type `Vector`.

It returns `LinearRegressionModel` .

It first counts the number of elements in features column (usually `features` ). The column has to be of `mllib.linalg.Vector` type (and can easily be prepared using [HashingTF transformer](#)).

```
val spam = Seq(
  0, "Hi Jacek. Wanna more SPAM? Best!"),
```

```
(1, "This is SPAM. This is SPAM")).toDF("id", "email")

import org.apache.spark.ml.feature.RegexTokenizer
val regexTok = new RegexTokenizer()
val spamTokens = regexTok.setInputCol("email").transform(spam)

scala> spamTokens.show(false)
+---+-----+-----+
| id | email | regexTok_646b6bcc4548__output |
+---+-----+-----+
| 0 | Hi Jacek. Wanna more SPAM? Best!|[hi, jacek., wanna, more, spam?, best!]|
| 1 | This is SPAM. This is SPAM |[this, is, spam., this, is, spam] |
+---+-----+-----+

import org.apache.spark.ml.feature.HashingTF
val hashTF = new HashingTF()
.setInputCol(regexTok.getOutputCol)
.setOutputCol("features")
.setNumFeatures(5000)

val spamHashed = hashTF.transform(spamTokens)

scala> spamHashed.select("email", "features").show(false)
+-----+-----+
| email | features |
+-----+-----+
| Hi Jacek. Wanna more SPAM? Best!|(5000,[2525,2943,3093,3166,3329,3980],[1.0,1.0,1.0,1.0,1.0,1.0])|
| This is SPAM. This is SPAM |(5000,[1713,3149,3370,4070],[1.0,1.0,2.0,2.0])|
+-----+-----+

// Create labeled datasets for spam (1)
val spamLabeled = spamHashed.withColumn("label", lit(1))

scala> spamLabeled.show
+---+-----+-----+-----+
| id | email | regexTok_646b6bcc4548__output | features | label |
+---+-----+-----+-----+
| 0 | Hi Jacek. Wanna m... | [hi, jacek., wann...] | (5000,[2525,2943,...] | 1.0 |
| 1 | This is SPAM. Thi... | [this, is, spam.,...] | (5000,[1713,3149,...] | 1.0 |
+---+-----+-----+-----+

val regular = Seq(
  (2, "Hi Jacek. I hope this email finds you well. Spark up!"),
  (3, "Welcome to Apache Spark project")).toDF("id", "email")
val regularTokens = regexTok.setInputCol("email").transform(regular)
val regularHashed = hashTF.transform(regularTokens)
// Create labeled datasets for non-spam regular emails (0)
```

# RandomForestRegressor

`RandomForestRegressor` is a concrete [Predictor](#) for Random Forest learning algorithm. It trains [RandomForestRegressionModel](#) (a subtype of [PredictionModel](#)) using `DataFrame` with features `column` of `Vector` type.

## Caution | [FIXME](#)

```

import org.apache.spark.mllib.linalg.Vectors
val features = Vectors.sparse(10, Seq((2, 0.2), (4, 0.4)))

val data = (0.0 to 4.0 by 1).map(d => (d, features)).toDF("label", "features")
// data.as[LabeledPoint]

scala> data.show(false)
+-----+-----+
|label|features           |
+-----+-----+
|0.0  |(10,[2,4,6],[0.2,0.4,0.6])|
|1.0  |(10,[2,4,6],[0.2,0.4,0.6])|
|2.0  |(10,[2,4,6],[0.2,0.4,0.6])|
|3.0  |(10,[2,4,6],[0.2,0.4,0.6])|
|4.0  |(10,[2,4,6],[0.2,0.4,0.6])|
+-----+-----+

import org.apache.spark.ml.regression.{ RandomForestRegressor, RandomForestRegressionM
odel }
val rfr = new RandomForestRegressor
val model: RandomForestRegressionModel = rfr.fit(data)

scala> model.trees.foreach(println)
DecisionTreeRegressionModel (uid=dtr_247e77e2f8e0) of depth 1 with 3 nodes
DecisionTreeRegressionModel (uid=dtr_61f8eachb2b61) of depth 2 with 7 nodes
DecisionTreeRegressionModel (uid=dtr_63fc5bde051c) of depth 2 with 5 nodes
DecisionTreeRegressionModel (uid=dtr_64d4e42de85f) of depth 2 with 5 nodes
DecisionTreeRegressionModel (uid=dtr_693626422894) of depth 3 with 9 nodes
DecisionTreeRegressionModel (uid=dtr_927f8a0bc35e) of depth 2 with 5 nodes
DecisionTreeRegressionModel (uid=dtr_82da39f6e4e1) of depth 3 with 7 nodes
DecisionTreeRegressionModel (uid=dtr_cb94c2e75bd1) of depth 0 with 1 nodes
DecisionTreeRegressionModel (uid=dtr_29e3362adfb2) of depth 1 with 3 nodes
DecisionTreeRegressionModel (uid=dtr_d6d896abcc75) of depth 3 with 7 nodes
DecisionTreeRegressionModel (uid=dtr_aacb22a9143d) of depth 2 with 5 nodes
DecisionTreeRegressionModel (uid=dtr_18d07dad5b9) of depth 2 with 7 nodes
DecisionTreeRegressionModel (uid=dtr_f0615c28637c) of depth 2 with 5 nodes
DecisionTreeRegressionModel (uid=dtr_4619362d02fc) of depth 2 with 5 nodes
DecisionTreeRegressionModel (uid=dtr_d39502f828f4) of depth 2 with 5 nodes
DecisionTreeRegressionModel (uid=dtr_896f3a4272ad) of depth 3 with 9 nodes
DecisionTreeRegressionModel (uid=dtr_891323c29838) of depth 3 with 7 nodes
DecisionTreeRegressionModel (uid=dtr_d658fe871e99) of depth 2 with 5 nodes
DecisionTreeRegressionModel (uid=dtr_d91227b13d41) of depth 2 with 5 nodes
DecisionTreeRegressionModel (uid=dtr_4a7976921f4b) of depth 2 with 5 nodes

scala> model.treeWeights
res12: Array[Double] = Array(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)

scala> model.featureImportances
res13: org.apache.spark.mllib.linalg.Vector = (1,[0],[1.0])

```

## Example

The following example uses [LinearRegression](#) estimator.

```

import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
val data = (0.0 to 9.0 by 1)                                // create a collection of Doubles
    .map(n => (n, n))                                         // make it pairs
    .map { case (label, features) =>
      LabeledPoint(label, Vectors.dense(features)) } // create labeled points of dense vectors
    .toDF                                                       // make it a DataFrame

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]|
+-----+


import org.apache.spark.ml.regression.LinearRegression
val lr = new LinearRegression

val model = lr.fit(data)

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 field]

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|
+---+-----+-----+

```

```

import org.apache.spark.ml.evaluation.RegressionEvaluator

// rmse is the default metric
// We're explicit here for learning purposes
val regEval = new RegressionEvaluator().setMetricName("rmse")
val rmse = regEval.evaluate(predictions)

scala> println(s"Root Mean Squared Error: $rmse")
Root Mean Squared Error: 0.0

import org.apache.spark.mllib.linalg.DenseVector
// NOTE Follow along to learn spark.ml-way (not RDD-way)
predictions.rdd.map { r =>
  (r(0).asInstanceOf[Double], r(1).asInstanceOf[DenseVector](0).toDouble, r(2).asInstanceOf[Double]))
  .toDF("label", "feature0", "prediction").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|
+---+-----+-----+

```

```

// 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|
+---+-----+-----+
```

# Models

`Model` abstract class is a [Transformer](#) with the optional [Estimator](#) that has produced it (as a transient parent field).

```
model: DataFrame =[predict]=> DataFrame (with predictions)
```

Note	An <code>Estimator</code> is optional and is available only after <code>fit</code> (of an <a href="#">Estimator</a> ) has been executed whose result a model is.
------	--

As a `Transformer` it takes a `DataFrame` and transforms it to a result `DataFrame` with `prediction` column added.

There are two direct implementations of the `Model` class that are not directly related to a concrete ML algorithm:

- [PipelineModel](#)
- [PredictionModel](#)

## PipelineModel

Caution	<code>PipelineModel</code> is a <code>private[ml]</code> class.
---------	---

`PipelineModel` is a `Model` of [Pipeline](#) estimator.

Once fit, you can use the result model as any other models to transform datasets (as `DataFrame` ).

A very interesting use case of `PipelineModel` is when a `Pipeline` is made up of [Transformer](#) instances.

```
// Transformer #1
import org.apache.spark.ml.feature.Tokenizer
val tok = new Tokenizer().setInputCol("text")

// Transformer #2
import org.apache.spark.ml.feature.HashingTF
val hashingTF = new HashingTF().setInputCol(tok.getOutputCol).setOutputCol("features")

// Fuse the Transformers in a Pipeline
import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline().setStages(Array(tok, hashingTF))

val dataset = Seq((0, "hello world")).toDF("id", "text")

// Since there's no fitting, any dataset works fine
val featurize = pipeline.fit(dataset)

// Use the pipelineModel as a series of Transformers
scala> featurize.transform(dataset).show(false)
+---+-----+-----+
| id | text      | tok_8aec9bfad04a__output|features
+---+-----+-----+
| 0  | hello world|[hello, world]          | [(262144,[71890,72594],[1.0,1.0])|
+---+-----+-----+
```

## PredictionModel

`PredictionModel` is an abstract class to represent a model for prediction algorithms like regression and classification (that have their own specialized models - details coming up below).

`PredictionModel` is basically a `Transformer` with `predict` method to calculate predictions (that end up in `prediction` column).

`PredictionModel` belongs to `org.apache.spark.ml` package.

```
import org.apache.spark.ml.PredictionModel
```

The contract of `PredictionModel` class requires that every custom implementation defines `predict` method (with `FeaturesType` type being the type of `features` ).

```
predict(features: FeaturesType): Double
```

The direct less-algorithm-specific extensions of the `PredictionModel` class are:

- `RegressionModel`

- [ClassificationModel](#)
- [RandomForestRegressionModel](#)

As a custom `Transformer` it comes with its own custom `transform` method.

Internally, `transform` first ensures that the type of the `features` column matches the type of the model and adds the `prediction` column of type `Double` to the schema of the result `DataFrame`.

It then creates the result `DataFrame` and adds the `prediction` column with a `predictUDF` function applied to the values of the `features` column.

#### Caution

**FIXME** A diagram to show the transformation from a dataframe (on the left) and another (on the right) with an arrow to represent the transformation method.

#### Tip

Enable `DEBUG` logging level for a `PredictionModel` implementation, e.g. `LinearRegressionModel`, to see what happens inside.

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

```
log4j.logger.org.apache.spark.ml.regression.LinearRegressionModel=DEBUG
```

Refer to [Logging](#).

## ClassificationModel

`ClassificationModel` is a `PredictionModel` that transforms a `DataFrame` with mandatory `features`, `label`, and `rawPrediction` (of type `Vector`) columns to a DataFrame with `prediction` column added.

#### Note

A Model with `ClassifierParams` parameters, e.g. `ClassificationModel`, requires that a DataFrame have the mandatory `features`, `label` (of type `Double`), and `rawPrediction` (of type `Vector`) columns.

`ClassificationModel` comes with its own `transform` (as `Transformer`) and `predict` (as `PredictionModel`).

The following is a list of the known `ClassificationModel` custom implementations (as of March, 24th):

- `ProbabilisticClassificationModel` (the abstract parent of the following classification models)
  - `DecisionTreeClassificationModel` (final)

- LogisticRegressionModel
- NaiveBayesModel
- RandomForestClassificationModel ( final )

## RegressionModel

`RegressionModel` is a [PredictionModel](#) that transforms a `DataFrame` with mandatory `label`, `features`, and `prediction` columns.

It comes with no own methods or values and so is more a *marker abstract class* (to combine different features of regression models under one type).

## LinearRegressionModel

`LinearRegressionModel` represents a model produced by a [LinearRegression](#) estimator. It transforms the required `features` column of type [org.apache.spark.mllib.linalg.Vector](#).

Note	It is a <code>private[ml]</code> class so what you, a developer, may eventually work with is the more general <code>RegressionModel</code> , and since <code>RegressionModel</code> is just a <a href="#">marker no-method abstract class</a> , it is more a <a href="#">PredictionModel</a> .
------	--

As a linear regression model that extends `LinearRegressionParams` it expects the following schema of an input `DataFrame`:

- `label` (required)
- `features` (required)
- `prediction`
- `regParam`
- `elasticNetParam`
- `maxIter` (Int)
- `tol` (Double)
- `fitIntercept` (Boolean)
- `standardization` (Boolean)
- `weightCol` (String)
- `solver` (String)

(New in 1.6.0) `LinearRegressionModel` is also a `MLWritable` (so you can save it to a persistent storage for later reuse).

With `DEBUG` logging enabled (see above) you can see the following messages in the logs when `transform` is called and transforms the schema.

```
16/03/21 06:55:32 DEBUG LinearRegressionModel: Input schema: {"type":"struct","fields": [{"name":"label","type":"double","nullable":false,"metadata":{}}, {"name":"features","type":{"type":"udt","class":"org.apache.spark.mllib.linalg.VectorUDT","pyClass":"pyspark.mllib.linalg.VectorUDT","sqlType":{"type":"struct","fields":[{"name":"type","type":"byte","nullable":false,"metadata":{}}, {"name":"size","type":"integer","nullable":true,"metadata":{}}, {"name":"indices","type":{"type":"array","elementType":"integer","containsNull":false}, "nullable":true,"metadata":{}}, {"name":"values","type":{"type":"array","elementType":"double","containsNull":false}, "nullable":true,"metadata":{}]}]}, "nullable":true,"metadata":{}}]}
16/03/21 06:55:32 DEBUG LinearRegressionModel: Expected output schema: {"type":"struct","fields": [{"name":"label","type":"double","nullable":false,"metadata":{}}, {"name":"features","type":{"type":"udt","class":"org.apache.spark.mllib.linalg.VectorUDT","pyClass":"pyspark.mllib.linalg.VectorUDT","sqlType":{"type":"struct","fields":[{"name":"type","type":"byte","nullable":false,"metadata":{}}, {"name":"size","type":"integer","nullable":true,"metadata":{}}, {"name":"indices","type":{"type":"array","elementType":"integer","containsNull":false}, "nullable":true,"metadata":{}}, {"name":"values","type":{"type":"array","elementType":"double","containsNull":false}, "nullable":true,"metadata":{}]}]}, {"name":"prediction","type":"double","nullable":false,"metadata":{}}]}
```

The implementation of `predict` for `LinearRegressionModel` calculates `dot(v1, v2)` of two Vectors - `features` and `coefficients` - (of `DenseVector` or `SparseVector` types) of the same size and adds `intercept`.

Note

The `coefficients` Vector and `intercept` Double are the integral part of `LinearRegressionModel` as the required input parameters of the constructor.

## LinearRegressionModel Example

```

// Create a (sparse) Vector
import org.apache.spark.mllib.linalg.Vectors
val indices = 0 to 4
val elements = indices.zip(Stream.continually(1.0))
val sv = Vectors.sparse(elements.size, elements)

// Create a proper DataFrame
val ds = sc.parallelize(Seq((0.5, sv))).toDF("label", "features")

import org.apache.spark.ml.regression.LinearRegression
val lr = new LinearRegression

// Importing LinearRegressionModel and being explicit about the type of model value
// is for learning purposes only
import org.apache.spark.ml.regression.LinearRegressionModel
val model: LinearRegressionModel = lr.fit(ds)

// Use the same ds - just for learning purposes
scala> model.transform(ds).show
+-----+-----+
|label|      features|prediction|
+-----+-----+
| 0.5|(5,[0,1,2,3,4],[1...|      0.5|
+-----+-----+

```

## RandomForestRegressionModel

`RandomForestRegressionModel` is a [PredictionModel](#) with `features` column of type [Vector](#).

Interestingly, `DataFrame` transformation (as part of [Transformer](#) contract) uses `SparkContext.broadcast` to send itself to the nodes in a Spark cluster and calls calculates predictions (as `prediction` column) on `features`.

## KMeansModel

`KMeansModel` is a `Model` of [KMeans](#) algorithm.

It belongs to `org.apache.spark.ml.clustering` package.

```
// See spark-mllib-estimators.adoc#KMeans
val kmeans: KMeans = ???
val trainingDF: DataFrame = ???
val kmModel = kmeans.fit(trainingDF)

// Know the cluster centers
scala> kmModel.clusterCenters
res0: Array[org.apache.spark.mllib.linalg.Vector] = Array([0.1,0.3], [0.1,0.1])

val inputDF = Seq((0.0, Vectors.dense(0.2, 0.4))).toDF("label", "features")

scala> kmModel.transform(inputDF).show(false)
+---+-----+-----+
|label|features |prediction|
+---+-----+-----+
|0.0 |[0.2,0.4]|0          |
+---+-----+-----+
```

# Evaluators

A **evaluator** is a transformation that maps a `DataFrame` into a metric indicating how good a model is.

```
evaluator: DataFrame =[evaluate]=> Double
```

`Evaluator` is an abstract class with `evaluate` methods.

```
evaluate(dataset: DataFrame): Double
evaluate(dataset: DataFrame, paramMap: ParamMap): Double
```

It employs `isLargerBetter` method to indicate whether the `Double` metric should be maximized (`true`) or minimized (`false`). It considers a larger value better (`true`) by default.

```
isLargerBetter: Boolean = true
```

The following is a list of some of the available `Evaluator` implementations:

- [MulticlassClassificationEvaluator](#)
- [BinaryClassificationEvaluator](#)
- [RegressionEvaluator](#)

## MulticlassClassificationEvaluator

`MulticlassClassificationEvaluator` is a concrete `Evaluator` that expects `DataFrame` datasets with the following two columns:

- `prediction` of `DoubleType`
- `label` of `float` or `double` values

## BinaryClassificationEvaluator

`BinaryClassificationEvaluator` is a concrete `Evaluator` for binary classification that expects datasets (of `DataFrame` type) with two columns:

- `rawPrediction` being `DoubleType` or `VectorUDT`.

- `label` being `NumericType`

Note	It can cross-validate models <code>LogisticRegression</code> , <code>RandomForestClassifier</code> et al.
------	---

## RegressionEvaluator

`RegressionEvaluator` is a concrete `Evaluator` for regression that expects datasets (of `DataFrame` type) with the following two columns:

- `prediction` of `float` or `double` values
- `label` of `float` or `double` values

When executed (via `evaluate`) it prepares a `RDD[Double, Double]` with `(prediction, label)` pairs and passes it on to `org.apache.spark.mllib.evaluation.RegressionMetrics` (from the "old" Spark MLlib).

`RegressionEvaluator` can evaluate the following metrics:

- `rmse` (default; larger is better? no) is the **root mean squared error**.
- `mse` (larger is better? no) is the **mean squared error**.
- `r2` (larger is better?: yes)
- `mae` (larger is better? no) is the **mean absolute error**.

```
// prepare a fake input dataset using transformers
import org.apache.spark.ml.feature.Tokenizer
val tok = new Tokenizer().setInputCol("text")

import org.apache.spark.ml.feature.HashingTF
val hashTF = new HashingTF()
.setInputCol(tok.getOutputCol) // it reads the output of tok
.setOutputCol("features")

// Scala trick to chain transform methods
// It's of little to no use since we've got Pipelines
// Just to have it as an alternative
val transform = (tok.transform _).andThen(hashTF.transform _)

val dataset = Seq((0, "hello world", 0.0)).toDF("id", "text", "label")

// we're using Linear Regression algorithm
import org.apache.spark.ml.regression.LinearRegression
val lr = new LinearRegression

import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline().setStages(Array(tok, hashTF, lr))

val model = pipeline.fit(dataset)

// Let's do prediction
// Note that we're using the same dataset as for fitting the model
// Something you'd definitely not be doing in prod
val predictions = model.transform(dataset)

// Now we're ready to evaluate the model
// Evaluator works on datasets with predictions

import org.apache.spark.ml.evaluation.RegressionEvaluator
val regEval = new RegressionEvaluator

// check the available parameters
scala> println(regEval.explainParams)
labelCol: label column name (default: label)
metricName: metric name in evaluation (mse|rmse|r2|mae) (default: rmse)
predictionCol: prediction column name (default: prediction)

scala> regEval.evaluate(predictions)
res0: Double = 0.0
```

# CrossValidator

**Caution**

**FIXME** Needs more love to be finished.

`CrossValidator` is an [Estimator](#) to produce a [CrossValidatorModel](#), i.e. it can fit a `CrossValidatorModel` for a given input dataset.

It belongs to `org.apache.spark.ml.tuning` package.

```
import org.apache.spark.ml.tuning.CrossValidator
```

`CrossValidator` accepts `numFolds` parameter (amongst the others).

```
import org.apache.spark.ml.tuning.CrossValidator
val cv = new CrossValidator

scala> println(cv.explainParams)
estimator: estimator for selection (undefined)
estimatorParamMaps: param maps for the estimator (undefined)
evaluator: evaluator used to select hyper-parameters that maximize the validated metric (undefined)
numFolds: number of folds for cross validation (>= 2) (default: 3)
seed: random seed (default: -1191137437)
```

**Tip**

What makes `CrossValidator` a very useful tool for *model selection* is its ability to work with any [Estimator](#) instance, [Pipelines](#) including, that can preprocess datasets before passing them on. This gives you a way to work with any dataset and preprocess it before a new (possibly better) model could be fit to it.

## Example — CrossValidator in Pipeline

**Caution**

**FIXME** The example below does **NOT** work. Being investigated.

**Caution**

**FIXME** Can k-means be crossvalidated? Does it make any sense? Does it only applies to supervised learning?

```
// Let's create a pipeline with transformers and estimator
import org.apache.spark.ml.feature._

val tok = new Tokenizer().setInputCol("text")

val hashTF = new HashingTF()
.setInputCol(tok.getOutputCol)
```

```

.setOutputCol("features")
.setNumFeatures(10)

import org.apache.spark.ml.classification.RandomForestClassifier
val rfc = new RandomForestClassifier

import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline()
.setStages(Array(tok, hashTF, rfc))

// CAUTION: label must be double
// 0 = scientific text
// 1 = non-scientific text
val trainDS = Seq(
  (0L, "[science] hello world", 0d),
  (1L, "long text", 1d),
  (2L, "[science] hello all people", 0d),
  (3L, "[science] hello hello", 0d)).toDF("id", "text", "label").cache

// Check out the train dataset
// Values in label and prediction columns should be alike
val sampleModel = pipeline.fit(trainDS)
sampleModel
  .transform(trainDS)
  .select('text, 'label, 'features, 'prediction)
  .show(truncate = false)

+-----+-----+-----+
|text           |label|features          |prediction|
+-----+-----+-----+
|[science] hello world|0.0 |((10,[0,8],[2.0,1.0]))|0.0 |
|long text      |1.0 |((10,[4,9],[1.0,1.0]))|1.0 |
|[science] hello all people|0.0 |((10,[0,6,8],[1.0,1.0,2.0]))|0.0 |
|[science] hello hello|0.0 |((10,[0,8],[1.0,2.0]))|0.0 |
+-----+-----+-----+


val input = Seq("Hello ScienCE").toDF("text")
sampleModel
  .transform(input)
  .select('text, 'rawPrediction, 'prediction)
  .show(truncate = false)

+-----+-----+-----+
|text           |rawPrediction          |prediction|
+-----+-----+-----+
|Hello ScienCE|[12.666666666666668,7.333333333333333]|0.0 |
+-----+-----+-----+


import org.apache.spark.ml.tuning.ParamGridBuilder
val paramGrid = new ParamGridBuilder().build

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
val binEval = new BinaryClassificationEvaluator

```

```

import org.apache.spark.ml.tuning.CrossValidator
val cv = new CrossValidator()
.setEstimator(pipeline) // <-- pipeline is the estimator
.setEvaluator(binEval) // has to match the estimator
.setEstimatorParamMaps(paramGrid)

// WARNING: It does not work!!!
val cvModel = cv.fit(trainDS)

```

## Example (no Pipeline)

```

import org.apache.spark.mllib.linalg.Vectors
val features = Vectors.sparse(3, Array(1), Array(1d))
val df = Seq(
  (0, "hello world", 0.0, features),
  (1, "just hello", 1.0, features)).toDF("id", "text", "label", "features")

import org.apache.spark.ml.classification.LogisticRegression
val lr = new LogisticRegression

import org.apache.spark.ml.evaluation.RegressionEvaluator
val regEval = new RegressionEvaluator

import org.apache.spark.ml.tuning.ParamGridBuilder
// Parameterize the only estimator used, i.e. LogisticRegression
// Use println(lr.explainParams) to learn about the supported parameters
val paramGrid = new ParamGridBuilder()
.addGrid(lr.regParam, Array(0.1, 0.01))
.build()

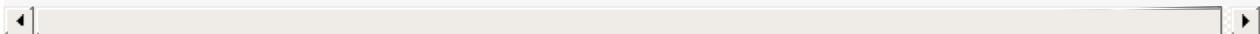
import org.apache.spark.ml.tuning.CrossValidator
val cv = new CrossValidator()
.setEstimator(lr) // just LogisticRegression not Pipeline
.setEvaluator(regEval)
.setEstimatorParamMaps(paramGrid)

// FIXME

scala> val cvModel = cv.fit(df)
java.lang.IllegalArgumentException: requirement failed: Nothing has been added to this
summarizer.
  at scala.Predef$.require(Predef.scala:219)
  at org.apache.spark.mllib.stat.MultivariateOnlineSummarizer.normL2(MultivariateOnlin
eSummarizer.scala:270)
  at org.apache.spark.mllib.evaluation.RegressionMetrics.SSerr$lzycompute(RegressionMe
trics.scala:65)
  at org.apache.spark.mllib.evaluation.RegressionMetrics.SSerr(RegressionMetrics.scala:
65)
  at org.apache.spark.mllib.evaluation.RegressionMetrics.meanSquaredError(RegressionMe
trics.scala:99)

```

```
    at org.apache.spark.mllib.evaluation.RegressionMetrics.rootMeanSquaredError(RegressionMetrics.scala:108)
    at org.apache.spark.ml.evaluation.RegressionEvaluator.evaluate(RegressionEvaluator.scala:94)
    at org.apache.spark.ml.tuning.CrossValidator$$anonfun$fit$1.apply(CrossValidator.scala:115)
    at org.apache.spark.ml.tuning.CrossValidator$$anonfun$fit$1.apply(CrossValidator.scala:105)
    at scala.collection.IndexedSeqOptimized$class.foreach(IndexedSeqOptimized.scala:33)
    at scala.collection.mutable.ArrayOps$ofRef.foreach(ArrayOps.scala:186)
    at org.apache.spark.ml.tuning.CrossValidator.fit(CrossValidator.scala:105)
    ... 61 elided
```



# 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 is a .

**Note**

The example uses a case class `LabeledText` to have the schema described nicely.

```
import spark.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")).toDF

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 spark.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!"),
  Article(3, "sci.math", "Hello, Math Revised!"),
  Article(4, "sci.math", "Better Math"),
  Article(5, "alt.religion", "TGIF")).toDS
```

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> articles.show
+---+-----+-----+
| id|      topic|          text|
+---+-----+-----+
| 0|sci.math|Hello, Math!|
| 1|alt.religion|Hello, Religion!|
| 2|sci.physics|Hello, Physics!|
| 3|sci.math|Hello, Math Revised!|
| 4|sci.math|Better Math|
| 5|alt.religion|TGIF|
+---+-----+-----+

val topic2Label: Boolean => Double = isSci => if (isSci) 1 else 0
val toLabel = udf(topic2Label)

val labelled = articles.withColumn("label", toLabel($"topic".like("sci%"))).cache

val Array(trainDF, testDF) = labelled.randomSplit(Array(0.75, 0.25))

scala> trainDF.show
+---+-----+-----+-----+
| id|      topic|          text|label|
+---+-----+-----+-----+
| 1|alt.religion|Hello, Religion!| 0.0|
| 3|sci.math|Hello, Math Revised!| 1.0|
+---+-----+-----+-----+

scala> testDF.show
+---+-----+-----+-----+
| id|      topic|          text|label|
+---+-----+-----+-----+
| 0|sci.math|Hello, Math!| 1.0|
| 2|sci.physics|Hello, Physics!| 1.0|
| 4|sci.math|Better Math| 1.0|
| 5|alt.religion|TGIF| 0.0|
+---+-----+-----+-----+

```

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).

```
import org.apache.spark.ml.feature.RegexTokenizer
val tokenizer = new RegexTokenizer()
  .setInputCol("text")
  .setOutputCol("words")

import org.apache.spark.ml.feature.HashingTF
val hashingTF = new HashingTF()
  .setInputCol(tokenizer.getOutputCol) // it does not wire transformers -- it's just
a column name
  .setOutputCol("features")
  .setNumFeatures(5000)

import org.apache.spark.ml.classification.LogisticRegression
val lr = new LogisticRegression().setMaxIter(20).setRegParam(0.01)

import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline().setStages(Array(tokenizer, hashingTF, lr))
```

It uses two columns, namely `label` and `features` vector to build a logistic regression model to make predictions.

```

val model = pipeline.fit(trainDF)

val trainPredictions = model.transform(trainDF)
val testPredictions = model.transform(testDF)

scala> trainPredictions.select('id, 'topic, 'text, 'label, 'prediction).show
+---+-----+-----+-----+
| id|      topic|          text|label|prediction|
+---+-----+-----+-----+
| 1|alt.religion|Hello, Religion!|  0.0|     0.0|
| 3|sci.math|Hello, Math Revised!|  1.0|     1.0|
+---+-----+-----+-----+

// Notice that the computations add new columns
scala> trainPredictions.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)

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
val evaluator = new BinaryClassificationEvaluator().setMetricName("areaUnderROC")

import org.apache.spark.ml.param.ParamMap
val evaluatorParams = ParamMap(evaluator.metricName -> "areaUnderROC")

scala> val areaTrain = evaluator.evaluate(trainPredictions, evaluatorParams)
areaTrain: Double = 1.0

scala> val areaTest = evaluator.evaluate(testPredictions, evaluatorParams)
areaTest: Double = 0.6666666666666666

```

Let's tune the model's hyperparameters (using "tools" from [org.apache.spark.ml.tuning package](#)).

Caution	<a href="#">FIXME</a> Review the available classes in the <a href="#">org.apache.spark.ml.tuning package</a> .
---------	--

```

import org.apache.spark.ml.tuning.ParamGridBuilder
val paramGrid = new ParamGridBuilder()
  .addGrid(hashingTF.numFeatures, Array(100, 1000))
  .addGrid(lr.regParam, Array(0.05, 0.2))
  .addGrid(lr.maxIter, Array(5, 10, 15))
  .build

// That gives all the combinations of the parameters

paramGrid: Array[org.apache.spark.ml.param.ParamMap] =
Array({
  logreg_cdb8970c1f11-maxIter: 5,
  hashingTF_8d7033d05904-numFeatures: 100,
  logreg_cdb8970c1f11-regParam: 0.05
}, {
  logreg_cdb8970c1f11-maxIter: 5,
  hashingTF_8d7033d05904-numFeatures: 1000,
  logreg_cdb8970c1f11-regParam: 0.05
}, {
  logreg_cdb8970c1f11-maxIter: 10,
  hashingTF_8d7033d05904-numFeatures: 100,
  logreg_cdb8970c1f11-regParam: 0.05
}, {
  logreg_cdb8970c1f11-maxIter: 10,
  hashingTF_8d7033d05904-numFeatures: 1000,
  logreg_cdb8970c1f11-regParam: 0.05
}, {
  logreg_cdb8970c1f11-maxIter: 15,
  hashingTF_8d7033d05904-numFeatures: 100,
  logreg_cdb8970c1f11-regParam: 0.05
}, {
  logreg_cdb8970c1f11-maxIter: 15,
  hashingTF_8d7033d05904-numFeatures: 1000,
  logreg_cdb8970c1f11-...
}

import org.apache.spark.ml.tuning.CrossValidator
import org.apache.spark.ml.param._
val cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEstimatorParamMaps(paramGrid)
  .setEvaluator(evaluator)
  .setNumFolds(10)

val cvModel = cv.fit(trainDF)

```

Let's use the cross-validated model to calculate predictions and evaluate their precision.

```
val cvPredictions = cvModel.transform(testDF)

scala> cvPredictions.select('topic, 'text, 'prediction).show
+-----+-----+-----+
|      topic|        text|prediction|
+-----+-----+-----+
|  sci.math|Hello, Math!|     0.0|
| sci.physics|Hello, Physics!|     0.0|
|  sci.math|    Better Math|     1.0|
|alt.religion|       TGIF|     0.0|
+-----+-----+-----+

scala> evaluator.evaluate(cvPredictions, evaluatorParams)
res26: Double = 0.6666666666666666

scala> val bestModel = cvModel.bestModel
bestModel: org.apache.spark.ml.Model[_] = pipeline_8873b744aac7
```

**Caution****FIXME Review**

<https://github.com/apache/spark/blob/master/mllib/src/test/scala/org/apache/spark/ml/PipelineTest.scala#L101>

You can eventually save the model for later use.

```
cvModel.write.overwrite.save("model")
```

Congratulations! You're done.

# 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
penalty is an L2 penalty. For alpha = 1, it is an L1 penalty (default: 0.0)
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 va
lue is 'auto' (default: auto)
standardization: whether to standardize the training features before fitting the model
(default: true)
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 weig
hts as 1.0 (default: )
```

**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
org.apache.spark.mllib.linalg.VectorUDT@f71b0bce but was actually IntegerType.
  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.sc
ala:51)
  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
```

# Latent Dirichlet Allocation (LDA)

Note	Information here are based almost exclusively from the blog post <a href="#">Topic modeling with LDA: MLlib meets GraphX</a> .
------	--

**Topic modeling** is a type of model that can be very useful in identifying hidden thematic structure in documents. Broadly speaking, it aims to find structure within an unstructured collection of documents. Once the structure is "discovered", you may answer questions like:

- What is document X about?
- How similar are documents X and Y?
- If I am interested in topic Z, which documents should I read first?

Spark MLlib offers out-of-the-box support for **Latent Dirichlet Allocation (LDA)** which is the first MLlib algorithm built upon [GraphX](#).

**Topic models** automatically infer the topics discussed in a collection of documents.

## Example

Caution	<a href="#">FIXME</a> Use Tokenizer, StopWordsRemover, CountVectorizer, and finally LDA in a pipeline.
---------	--

# Vector

`Vector` sealed trait represents a **numeric vector** of values (of `Double` type) and their indices (of `Int` type).

It belongs to `org.apache.spark.mllib.linalg` package.

Note

To Scala and Java developers:

`vector` class in Spark MLlib belongs to `org.apache.spark.mllib.linalg` package.

It is **not** the `Vector` type in Scala or Java. Train your eyes to see two types of the same name. You've been warned.

A `vector` object knows its `size`.

A `vector` object can be converted to:

- `Array[Double]` using `toArray`.
- a **dense vector** as `DenseVector` using `toDense`.
- a **sparse vector** as `SparseVector` using `toSparse`.
- (1.6.0) a JSON string using `toJson`.
- (*internal*) a **breeze vector** as `BV[Double]` using `toBreeze`.

There are exactly two available implementations of `vector` sealed trait (that also belong to `org.apache.spark.mllib.linalg` package):

- `DenseVector`
- `SparseVector`

Tip

Use `Vectors` factory object to create vectors, be it `DenseVector` or `SparseVector`.

```

import org.apache.spark.mllib.linalg.Vectors

// You can create dense vectors explicitly by giving values per index
val denseVec = Vectors.dense(Array(0.0, 0.4, 0.3, 1.5))
val almostAllZeros = Vectors.dense(Array(0.0, 0.4, 0.3, 1.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0))

// You can however create a sparse vector by the size and non-zero elements
val sparse = Vectors.sparse(10, Seq((1, 0.4), (2, 0.3), (3, 1.5)))

// Convert a dense vector to a sparse one
val fromSparse = sparse.toDense

scala> almostAllZeros == fromSparse
res0: Boolean = true

```

**Note** The factory object is called `vectors` (plural).

```

import org.apache.spark.mllib.linalg._

// prepare elements for a sparse vector
// NOTE: It is more Scala rather than Spark
val indices = 0 to 4
val elements = indices.zip(Stream.continually(1.0))
val sv = Vectors.sparse(elements.size, elements)

// Notice how Vector is printed out
scala> sv
res4: org.apache.spark.mllib.linalg.Vector = (5,[0,1,2,3,4],[1.0,1.0,1.0,1.0,1.0])

scala> sv.size
res0: Int = 5

scala> sv.toArray
res1: Array[Double] = Array(1.0, 1.0, 1.0, 1.0, 1.0)

scala> sv == sv.copy
res2: Boolean = true

scala> sv.toJson
res3: String = {"type":0,"size":5,"indices":[0,1,2,3,4],"values":[1.0,1.0,1.0,1.0,1.0]}

```

# LabeledPoint

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

`LabeledPoint` is a convenient class for declaring a schema for DataFrames that are used as input data for [Linear Regression](#) in Spark MLlib.

# Streaming MLlib

The following Machine Learning algorithms have their streaming variants in MLlib:

- [k-means](#)
- [Linear Regression](#)
- [Logistic Regression](#)

They can train models and predict on streaming data.

Note	The streaming algorithms belong to <code>spark.mllib</code> (the older RDD-based API).
------	--

## Streaming k-means

```
org.apache.spark.mllib.clustering.StreamingKMeans
```

## Streaming Linear Regression

```
org.apache.spark.mllib.regression.StreamingLinearRegressionWithSGD
```

## Streaming Logistic Regression

```
org.apache.spark.mllib.classification.StreamingLogisticRegressionWithSGD
```

## Sources

- [Streaming Machine Learning in Spark- Jeremy Freeman \(HHMI Janelia Research Center\)](#)

# Spark GraphX - Distributed Graph Computations

**Spark GraphX** is a graph processing framework built on top of Spark.

GraphX models graphs as **property graphs** where vertices and edges can have properties.

Caution

[FIXME](#) Diagram of a graph with friends.

GraphX comes with its own package `org.apache.spark.graphx`.

Tip

Import `org.apache.spark.graphx` package to work with GraphX.

```
import org.apache.spark.graphx._
```

## Graph

`Graph` abstract class represents a collection of `vertices` and `edges`.

```
abstract class Graph[VD: ClassTag, ED: ClassTag]
```

`vertices` attribute is of type `VertexRDD` while `edges` is of type `EdgeRDD`.

`Graph` can also be described by `triplets` (that is of type `RDD[EdgeTriplet[VD, ED]]`).

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
val vertices: RDD[(VertexId, String)] =
  sc.parallelize(Seq(
    (0L, "Jacek"),
    (1L, "Agata"),
    (2L, "Julian")))

val edges: RDD[Edge[String]] =
  sc.parallelize(Seq(
    Edge(0L, 1L, "wife"),
    Edge(1L, 2L, "owner"))
))

scala> val graph = Graph(vertices, edges)
graph: org.apache.spark.graphx.Graph[String, String] = org.apache.spark.graphx.impl.Gra
phImpl@5973e4ec
```

## package object graphx

`package object graphx` defines two type aliases:

- `vertexId` (`Long`) that represents a unique 64-bit vertex identifier.
- `PartitionID` (`Int`) that is an identifier of a graph partition.

## Standard GraphX API

`Graph` class comes with a small set of API.

- Transformations

- `mapVertices`
- `mapEdges`
- `mapTriplets`
- `reverse`
- `subgraph`
- `mask`
- `groupEdges`

- Joins

- `outerJoinVertices`

- Computation

- `aggregateMessages`

## Creating Graphs (Graph object)

`Graph` object comes with the following factory methods to create instances of `Graph`:

- `fromEdgeTuples`
- `fromEdges`
- `apply`

Note	The default implementation of <code>Graph</code> is <code>GraphImpl</code> .
------	--

## GraphOps - Graph Operations

## GraphImpl

`GraphImpl` is the default implementation of `Graph` abstract class.

It lives in `org.apache.spark.graphx.impl` package.

## OLD - perhaps soon to be removed

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.

## Further reading or watching

- (video) [GraphX: Graph Analytics in Spark- Ankur Dave \(UC Berkeley\)](#)



# Graph Algorithms

GraphX comes with a set of built-in graph algorithms.

## PageRank

## Triangle Count

## Connected Components

Identifies independent disconnected subgraphs.

## Collaborative Filtering

*What kinds of people like what kinds of products.*

# Monitoring, Tuning and Debugging

Caution

[FIXME](#)

# Unified Memory Management

**Unified Memory Management** was introduced in [SPARK-10000: Consolidate storage and execution memory management](#).

It uses the custom memory manager [UnifiedMemoryManager](#).

## Further reading or watching

- (video) [Deep Dive: Apache Spark Memory Management](#)
- (video) [Deep Dive into Project Tungsten \(...WGI\)](#)
- (video) [Spark Performance: What's Next \(...WYX4\)](#)
- [SPARK-10000: Unified Memory Management](#)

# HistoryServer

HistoryServer is a web interface for completed and running (aka *incomplete*) Spark applications.

You can start a HistoryServer instance by executing `$SPARK_HOME/sbin/start-history-server.sh` script. See [Starting HistoryServer](#).

Tip

Use [EventLoggingListener](#) to collect events.

Enable `INFO` logging level for `org.apache.spark.deploy.history.HistoryServer` logger to see what happens inside.

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

Tip

```
log4j.logger.org.apache.spark.deploy.history.HistoryServer=INFO
```

Refer to [Logging](#).

## Starting HistoryServer

You can start a HistoryServer instance by executing `$SPARK_HOME/sbin/start-history-server.sh` script.

```
$ ./sbin/start-history-server.sh
starting org.apache.spark.deploy.history.HistoryServer, logging to .../spark/logs/spark-jacek-org.apache.spark.deploy.history.HistoryServer-1-japila.out
```

Note

To stop the server execute `stop-history-server.sh` or kill it.

When started, it prints out the following INFO message to the logs:

```
INFO HistoryServer: Started daemon with process name: [processName]
```

It registers signal handlers (using `SignalUtils`) for `TERM`, `HUP`, `INT` to log their execution:

```
ERROR HistoryServer: RECEIVED SIGNAL [signal]
```

It init security if enabled (using `spark.history.kerberos.enabled` setting).

Caution

[FIXME](#) Describe `initSecurity`

It creates a `SecurityManager`.

It creates a `ApplicationHistoryProvider` (by reading `spark.history.provider`).

It reads `spark.history.ui.port`.

It creates a `HistoryServer` and requests to bind.

It registers a shutdown hook to call `stop` on the `HistoryServer` instance.

## Creating HistoryServer Instance

Caution

FIXME

### Settings

- `spark.history.provider` (default: `FsHistoryProvider`) is a fully-qualified class name for a `ApplicationHistoryProvider` that comes with a single-arg constructor accepting `SparkConf`.
- `spark.history.ui.port` (default: `18080`) — the port of the History Server's UI.

# SQLHistoryListener

`SQLHistoryListener` is a custom [SQLListener](#) for [History Server](#). It attaches [SQL tab](#) to History Server's web UI only when the first [SparkListenerSQLExecutionStart](#) arrives and shuts [onExecutorMetricsUpdate](#) off. It also handles [ends of tasks in a slightly different way](#).

Note	Support for SQL UI in History Server was added in SPARK-11206 Support SQL UI on the history server.
------	---

Caution	<a href="#">FIXME</a> Add the link to the JIRA.
---------	---

## onOtherEvent

```
onOtherEvent(event: SparkListenerEvent): Unit
```

When `SparkListenerSQLExecutionStart` event comes, `onOtherEvent` attaches [SQL tab](#) to web UI and passes the call to the parent [SQLListener](#).

## onTaskEnd

Caution	<a href="#">FIXME</a>
---------	-----------------------

## Creating SQLHistoryListener Instance

`SQLHistoryListener` is created using a (`private[sql]`) `SQLHistoryListenerFactory` class (which is `SparkHistoryListenerFactory`).

The `SQLHistoryListenerFactory` class is registered when `SparkUI.createHistoryUI` as a Java service in `META-INF/services/org.apache.spark.scheduler.SparkHistoryListenerFactory`:

```
org.apache.spark.sql.execution.ui.SQLHistoryListenerFactory
```

Note	Loading the service uses Java's <code>ServiceLoader.load</code> method.
------	---

## onExecutorMetricsUpdate

`onExecutorMetricsUpdate` does nothing.



# FsHistoryProvider

`FsHistoryProvider` is the default [application history provider](#) for [HistoryServer](#). It uses [SparkConf](#) and `Clock` objects for its operation.

**Tip** Enable `INFO` or `DEBUG` logging levels for `org.apache.spark.deploy.history.FsHistoryProvider` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.deploy.history.FsHistoryProvider=DEBUG
```

Refer to [Logging](#).

# ApplicationHistoryProvider

`ApplicationHistoryProvider` tracks the history of [Spark applications](#) with their [Spark UIs](#). It can be [stopped](#) and [write events to a stream](#).

It is an abstract class.

# ApplicationHistoryProvider Contract

Every `ApplicationHistoryProvider` offers the following:

- `getListing` to return a list of all known applications.

```
getListing(): Iterable[ApplicationHistoryInfo]
```

- `getAppUI` to return [Spark UI](#) for an application.

```
getAppUI(appId: String, attemptId: Option[String]): Option[LoadedAppUI]
```

- `stop` to stop the instance.

```
stop(): Unit
```

- `getConfig` to return configuration of...[FIXME](#)

```
getConfig(): Map[String, String] = Map()
```

- `writeEventLogs` to write events to a stream.

```
writeEventLogs(appId: String, attemptId: Option[String], zipStream: ZipOutputStream): Unit
```



# Logging

Spark uses [log4j](#) for logging.

## Logging Levels

The valid logging levels are [log4j's Levels](#) (from most specific to least):

- OFF (most specific, no logging)
- FATAL (most specific, little data)
- ERROR
- WARN
- INFO
- DEBUG
- TRACE (least specific, a lot of data)
- ALL (least specific, all data)

## 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 Programmatically

See [Setting Default Log Level Programmatically](#) in [SparkContext - the door to Spark](#).

## Setting Log Levels in Spark Applications

In standalone Spark applications or while in [Spark Shell](#) session, use the following:

```
import org.apache.log4j.{Level, Logger}

Logger.getLogger(classOf[RackResolver]).getLevel
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$A
ppClassLoader@1b6d3586.
log4j: Using URL [file:/Users/jacek/dev/oss/spark-activator/target/scala-2.11/classes/
log4j.properties] for automatic log4j configuration.
log4j: Reading configuration from URL file:/Users/jacek/dev/oss/spark-activator/target
/scala-2.11/classes/log4j.properties
```

## Disabling Logging

Use the following `conf/log4j.properties` to disable logging completely:

```
log4j.logger.org=OFF
```

# 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 behaviour of the components.

`org.apache.spark.metrics.source.Source` is the top-level class for the metric registries in Spark. They expose their internal status.

Spark uses [Metrics 3.1.0](#).

	<p><a href="#">FIXME</a> Review</p> <ul style="list-style-type: none"> <li>• How to use the metrics to monitor Spark using jconsole?</li> <li>• ApplicationSource</li> <li>• WorkerSource</li> <li>• ExecutorSource</li> <li>• JvmSource</li> <li>• MesosClusterSchedulerSource</li> <li>• StreamingSource</li> </ul>
Caution	

- 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"
}
```

# Spark Listeners

`SparkListener` is a developer API for custom Spark listeners. It is an abstract class that is a [SparkListenerInterface](#) with empty no-op implementations of all the *callback methods*.

With all the callbacks being no-ops, you can focus on events of your liking and process a subset of events.

Tip

Developing a custom `SparkListener` is an excellent introduction to low-level details of [Spark's Execution Model](#). Check out the exercise [Developing Custom SparkListener to monitor DAGScheduler in Scala](#).

## SparkListenerEvents

Caution

[FIXME](#) Give a less code-centric description of the times for the events.

### SparkListenerApplicationStart

```
SparkListenerApplicationStart(  
    appName: String,  
    appId: Option[String],  
    time: Long,  
    sparkUser: String,  
    appAttemptId: Option[String],  
    driverLogs: Option[Map[String, String]] = None)
```

`SparkListenerApplicationStart` is posted when `SparkContext` does `postApplicationStart`.

### SparkListenerJobStart

```
SparkListenerJobStart(  
    jobId: Int,  
    time: Long,  
    stageInfos: Seq[StageInfo],  
    properties: Properties = null)
```

`SparkListenerJobStart` is posted when `DAGScheduler` does `handleJobSubmitted` and `handleMapStageSubmitted`.

### SparkListenerStageSubmitted

```
SparkListenerStageSubmitted(stageInfo: StageInfo, properties: Properties = null)
```

`SparkListenerStageSubmitted` is posted when `DAGScheduler` does `submitMissingTasks`.

## SparkListenerTaskStart

```
SparkListenerTaskStart(stageId: Int, stageAttemptId: Int, taskInfo: TaskInfo)
```

`SparkListenerTaskStart` is posted when `DAGScheduler` does `handleBeginEvent`.

## SparkListenerTaskGettingResult

```
SparkListenerTaskGettingResult(taskInfo: TaskInfo)
```

`SparkListenerTaskGettingResult` is posted when `DAGScheduler` does `handleGetTaskResult`.

## SparkListenerTaskEnd

```
SparkListenerTaskEnd(  
    stageId: Int,  
    stageAttemptId: Int,  
    taskType: String,  
    reason: TaskEndReason,  
    taskInfo: TaskInfo,  
    // may be null if the task has failed  
    @Nullable taskMetrics: TaskMetrics)
```

`SparkListenerTaskEnd` is posted when `DAGScheduler` does `handleTaskCompletion`.

## SparkListenerStageCompleted

```
SparkListenerStageCompleted(stageInfo: StageInfo)
```

`SparkListenerStageCompleted` is posted when `DAGScheduler` does `markStageAsFinished`.

## SparkListenerJobEnd

```
SparkListenerJobEnd(  
    jobId: Int,  
    time: Long,  
    jobResult: JobResult)
```

`SparkListenerJobEnd` is posted when `DAGScheduler` does `cleanUpAfterSchedulerStop`, `handleTaskCompletion`, `failJobAndIndependentStages`, and `markMapStageJobAsFinished`.

## SparkListenerApplicationEnd

```
SparkListenerApplicationEnd(time: Long)
```

`SparkListenerApplicationEnd` is posted when `SparkContext` does `postApplicationEnd`.

## SparkListenerEnvironmentUpdate

```
SparkListenerEnvironmentUpdate(environmentDetails: Map[String, Seq[(String, String)]]))
```

`SparkListenerEnvironmentUpdate` is posted when `SparkContext` does `postEnvironmentUpdate`.

## SparkListenerBlockManagerAdded

```
SparkListenerBlockManagerAdded(  
    time: Long,  
    blockManagerId: BlockManagerId,  
    maxMem: Long)
```

`SparkListenerBlockManagerAdded` is posted when `BlockManagerMasterEndpoint` registers a `BlockManager`.

## SparkListenerBlockManagerRemoved

```
SparkListenerBlockManagerRemoved(  
    time: Long,  
    blockManagerId: BlockManagerId)
```

`SparkListenerBlockManagerRemoved` is posted when `BlockManagerMasterEndpoint` removes a `BlockManager`.

## SparkListenerBlockUpdated

```
SparkListenerBlockUpdated(blockUpdatedInfo: BlockUpdatedInfo)
```

`SparkListenerBlockUpdated` is posted when `BlockManagerMasterEndpoint` receives `UpdateBlockInfo` message.

## SparkListenerUnpersistRDD

```
SparkListenerUnpersistRDD(rddId: Int)
```

`SparkListenerUnpersistRDD` is posted when `SparkContext` does `unpersistRDD`.

## SparkListenerExecutorAdded

```
SparkListenerExecutorAdded(  
    time: Long,  
    executorId: String,  
    executorInfo: ExecutorInfo)
```

`SparkListenerExecutorAdded` is posted when `DriverEndpoint` RPC endpoint (of `CoarseGrainedSchedulerBackend`) handles `RegisterExecutor` message, `MesosFineGrainedSchedulerBackend` does `resourceOffers`, and `LocalSchedulerBackendEndpoint` starts.

## SparkListenerExecutorRemoved

```
SparkListenerExecutorRemoved(  
    time: Long,  
    executorId: String,  
    reason: String)
```

`SparkListenerExecutorRemoved` is posted when `DriverEndpoint` RPC endpoint (of `CoarseGrainedSchedulerBackend`) does `removeExecutor` and `MesosFineGrainedSchedulerBackend` does `removeExecutor`.

## Known Implementations

The following is the complete list of all known Spark listeners:

- [EventLoggingListener](#)
- `ExecutorsListener` that prepares information to be displayed on the **Executors** tab in [web UI](#).
- `SparkFirehoseListener` that allows users to receive all [SparkListenerEvent](#) events by overriding the single `onEvent` method only.
- `ExecutorAllocationListener`
- [HeartbeatReceiver](#)
- web UI and [EventLoggingListener](#) listeners

Caution	<a href="#">FIXME</a> Make it complete.
---------	---

## SparkListenerInterface

`SparkListenerInterface` is an internal interface for listeners of events from the Spark scheduler.

# LiveListenerBus

`LiveListenerBus` asynchronously passes [listener events](#) to registered [Spark listeners](#).

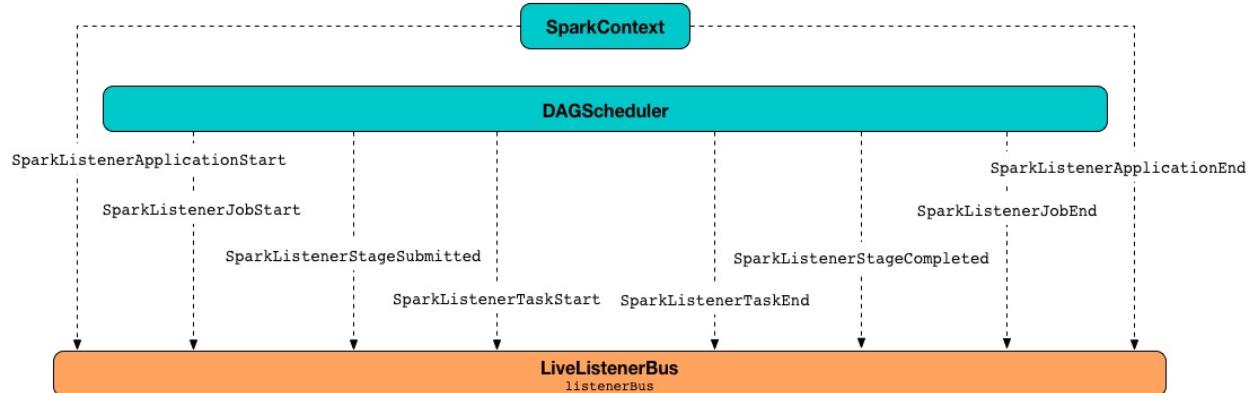


Figure 1. LiveListenerBus, SparkListenerEvents, and Senders

`LiveListenerBus` is a single-JVM `SparkListenerBus` that uses `listenerThread` to poll events. Emitters are supposed to use `post` method to post `SparkListenerEvent` events.

Note	The event queue is <code>java.util.concurrent.LinkedBlockingQueue</code> with capacity of 10000 <code>SparkListenerEvent</code> events.
------	---

Note	An instance of <code>LiveListenerBus</code> is created in <code>SparkContext</code> .
------	---

## Creating LiveListenerBus Instance

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

## Starting LiveListenerBus (start method)

```
start(sc: SparkContext): Unit
```

`start` starts processing events.

Internally, it saves the input `SparkContext` for later use and starts `listenerThread`. It makes sure that it only happens when `LiveListenerBus` has not been started before (i.e. `started` is disabled).

If however `LiveListenerBus` has already been started, a `IllegalStateException` is thrown:

```
[name] already started!
```

## Posting SparkListenerEvent Events (post method)

```
post(event: SparkListenerEvent): Unit
```

`post` puts the input `event` onto the internal `eventQueue` queue and releases the internal `eventLock` semaphore. If the event placement was not successful (and it could happen since it is tapped at 10000 events) [onDropEvent](#) method is called.

The event publishing is only possible when `stopped` flag has been enabled.

Caution

[FIXME](#) Who's enabling the `stopped` flag and when/why?

If `LiveListenerBus` has been stopped, the following ERROR appears in the logs:

```
ERROR [name] has already stopped! Dropping event [event]
```

## Event Dropped Callback (onDropEvent method)

```
onDropEvent(event: SparkListenerEvent): Unit
```

`onDropEvent` is called when no further events can be added to the internal `eventQueue` queue (while [posting a SparkListenerEvent event](#)).

It simply prints out the following ERROR message to the logs and ensures that it happens only once.

```
ERROR Dropping SparkListenerEvent because no remaining room in event queue. This likely means one of the SparkListeners is too slow and cannot keep up with the rate at which tasks are being started by the scheduler.
```

Note

It uses the internal `logDroppedEvent` atomic variable to track the state.

## Stopping LiveListenerBus

```
stop(): Unit
```

`stop` releases the internal `eventLock` semaphore and waits until [listenerThread](#) dies. It can only happen after all events were posted (and polling `eventQueue` gives nothing).

It checks that `started` flag is enabled (i.e. `true`) and throws a `IllegalStateException` otherwise.

```
Attempted to stop [name] that has not yet started!
```

`stopped` flag is enabled.

## listenerThread for Event Polling

`LiveListenerBus` uses `SparkListenerBus` single daemon thread that ensures that the polling events from the event queue is only after [the listener was started](#) and only one event at a time.

Caution

[`FIXME`](#) There is some logic around no events in the queue.

## Settings

### spark.extraListeners

`spark.extraListeners` (default: empty) is a comma-separated list of listener class names that should be registered with [LiveListenerBus](#) when [SparkContext is initialized](#).

## SparkListenerBus

`SparkListenerBus` is a [ListenerBus](#) that manages [SparkListenerInterface](#) listeners that process [SparkListenerEvent](#) events.

It comes with a custom `doPostEvent` method.

```
doPostEvent(listener: SparkListenerInterface, event: SparkListenerEvent): Unit
```

`doPostEvent` method simply relays `SparkListenerEvent` events to appropriate `SparkListenerInterface` methods as follows:

Table 1. SparkListenerEvent to SparkListenerInterface's Method "mapping"

SparkListenerEvent	SparkListenerInterface's Method
SparkListenerStageSubmitted	onStageSubmitted
SparkListenerStageCompleted	onStageCompleted
SparkListenerJobStart	onJobStart
SparkListenerJobEnd	onJobEnd
SparkListenerJobEnd	onJobEnd
SparkListenerTaskStart	onTaskStart
SparkListenerTaskGettingResult	onTaskGettingResult
SparkListenerTaskEnd	onTaskEnd
SparkListenerEnvironmentUpdate	onEnvironmentUpdate
SparkListenerBlockManagerAdded	onBlockManagerAdded
SparkListenerBlockManagerRemoved	onBlockManagerRemoved
SparkListenerUnpersistRDD	onUnpersistRDD
SparkListenerApplicationStart	onApplicationStart
SparkListenerApplicationEnd	onApplicationEnd
SparkListenerExecutorMetricsUpdate	onExecutorMetricsUpdate
SparkListenerExecutorAdded	onExecutorAdded
SparkListenerExecutorRemoved	onExecutorRemoved
SparkListenerBlockUpdated	onBlockUpdated
SparkListenerLogStart	<i>event ignored</i>
<i>other event types</i>	onOtherEvent

Note	There are two custom <code>SparkListenerBus</code> listeners: <a href="#">LiveListenerBus</a> and <a href="#">ReplayListenerBus</a> .
------	---

## ListenerBus

```
ListenerBus[L <: AnyRef, E]
```

`ListenerBus` is an event bus that post events (of type `E`) to all registered listeners (of type `L`).

It manages `listeners` of type `L`, i.e. it can add to and remove listeners from an internal `listeners` collection.

```
addListener(listener: L): Unit  
removeListener(listener: L): Unit
```

It can post events of type `E` to all registered listeners (using `postToAll` method). It simply iterates over the internal `listeners` collection and executes the abstract `doPostEvent` method.

```
doPostEvent(listener: L, event: E): Unit
```

Note	<code>doPostEvent</code> is provided by more specialized <code>ListenerBus</code> event buses.
------	--

In case of exception while posting an event to a listener you should see the following ERROR message in the logs and the exception.

```
ERROR Listener [listener] threw an exception
```

Note	There are three custom <code>ListenerBus</code> listeners: <a href="#">SparkListenerBus</a> , <a href="#">StreamingQueryListenerBus</a> , and <a href="#">StreamingListenerBus</a> .
------	--

	Enable <code>ERROR</code> logging level for <code>org.apache.spark.util.ListenerBus</code> logger to see what happens inside.
--	---

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

```
log4j.logger.org.apache.spark.util.ListenerBus=ERROR
```

Tip	Refer to <a href="#">Logging</a> .
-----	------------------------------------

# ReplayListenerBus

`ReplayListenerBus` is a custom `SparkListenerBus` that can replay JSON-encoded `SparkListenerEvent` events from a stream and post them to listeners.

Note	<code>ReplayListenerBus</code> is used in <code>FsHistoryProvider</code> .
------	--

Note	<code>ReplayListenerBus</code> is a <code>private[spark]</code> class in <code>org.apache.spark.scheduler</code> package.
------	---

## Replaying JSON-encoded SparkListenerEvents from Stream (replay method)

```
replay(  
  logData: InputStream,  
  sourceName: String,  
  maybeTruncated: Boolean = false): Unit
```

`replay` reads JSON-encoded `SparkListenerEvent` events from `logData` (one event per line) and posts them to all registered `SparkListenerInterface` listeners.

`replay` uses `JsonProtocol.sparkEventFromJson` to convert JSON-encoded events to `SparkListenerInterface` listeners objects.

Note	<code>replay</code> uses <b>jackson</b> from <code>json4s</code> library to parse the AST for JSON.
------	---

When there is an exception parsing a JSON event, you may see the following WARN message in the logs (for the last line) or a `JsonParseException`.

```
WARN Got JsonParseException from log file $sourceName at line [lineNumber], the file might not have finished writing cleanly.
```

Any other non-IO exceptions end up with the following ERROR messages in the logs:

```
ERROR Exception parsing Spark event log: [sourceName]  
ERROR Malformed line #[lineNumber]: [currentLine]
```

Note	The <code>sourceName</code> input argument is only used for messages.
------	---



# Persisting Events using EventLoggingListener

`EventLoggingListener` is a [SparkListener](#) that logs JSON-encoded events to a file.

When `enabled` it writes events to a log file under `spark.eventLog.dir` directory. All [Spark events](#) are logged.

Note	<code>SparkListenerBlockUpdated</code> and <code>SparkListenerExecutorMetricsUpdate</code> are not logged.
------	--

Events can optionally be [compressed](#).

In-flight log files are with `.inprogress` extension.

Tip	You can use <a href="#">History Server</a> to view the logs using a web interface.
-----	--

Note	It is a <code>private[spark]</code> class in <code>org.apache.spark.scheduler</code> package.
------	---

	Enable <code>INFO</code> logging level for <code>org.apache.spark.scheduler.EventLoggingListener</code> logger to see what happens inside <code>EventLoggingListener</code> .
--	---

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

Tip	<code>log4j.logger.org.apache.spark.scheduler.EventLoggingListener=INFO</code>
-----	--

Refer to [Logging](#).

## Creating EventLoggingListener Instance

`EventLoggingListener` requires an application id (`appId`), the application's optional attempt id (`appAttemptId`), `logBaseDir`, a `SparkConf` (as `sparkConf`) and Hadoop's `Configuration` (as `hadoopConf`).

Note	When initialized with no Hadoop's <code>Configuration</code> it calls <code>SparkHadoopUtil.get.newConfiguration(sparkConf)</code> .
------	--

## Starting EventLoggingListener (start method)

`start` checks whether `logBaseDir` is really a directory, and if it is not, it throws a `IllegalArgumentException` with the following message:

<code>Log directory [logBaseDir] does not exist.</code>
---

The log file's working name is created based on `appId` with or without the compression codec used and `appAttemptId`, i.e. `local-1461696754069`. It also uses `.inprogress` extension.

If [overwrite is enabled](#), you should see the WARN message:

```
WARN EventLoggingListener: Event log [path] already exists. Overwriting...
```

The working log `.inprogress` is attempted to be deleted. In case it could not be deleted, the following WARN message is printed out to the logs:

```
WARN EventLoggingListener: Error deleting [path]
```

The buffered output stream is created with metadata with Spark's version and `SparkListenerLogStart` class' name as the first line.

```
{"Event":"SparkListenerLogStart","Spark Version":"2.0.0-SNAPSHOT"}
```

At this point, `EventLoggingListener` is ready for event logging and you should see the following INFO message in the logs:

```
INFO EventLoggingListener: Logging events to [logPath]
```

## Logging Event (logEvent method)

```
logEvent(event: SparkListenerEvent, flushLogger: Boolean = false)
```

`logEvent` logs `event` as JSON using `org.apache.spark.util.JsonProtocol` object.

## Stopping EventLoggingListener (stop method)

`stop` closes `PrintWriter` for the log file and renames the file to be without `.inprogress` extension.

If the target log file exists (one without `.inprogress` extension), it overwrites the file if [spark.eventLog.overwrite](#) is enabled. You should see the following WARN message in the logs:

```
WARN EventLoggingListener: Event log [target] already exists. Overwriting...
```

If the target log file exists and overwrite is disabled, an `java.io.IOException` is thrown with the following message:

```
Target log file already exists ([logPath])
```

## Compressing Logged Events

If [event compression is enabled](#), `CompressionCodec.createCodec(sparkConf)` is executed to set up a compression codec.

Caution

[FIXME](#) What compression codecs are supported?

## Settings

### **spark.eventLog.enabled**

`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.dir**

`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. See [Creating a SparkContext](#). \* `spark.eventLog.buffer.kb` (default: `100`) - buffer size to use when writing to output streams.

### **spark.eventLog.overwrite**

`spark.eventLog.overwrite` (default: `false`) - whether to delete or at least overwrite an existing `.inprogress` log file.

### **spark.eventLog.compress**

`spark.eventLog.compress` (default: `false`) controls whether to compress events (`true`) or not (`false`).

See [Compressing Events](#).

### **spark.eventLog.testing**

`spark.eventLog.testing` (default: `false`) - internal flag for testing purposes to add JSON events to the internal `loggedEvents` array.

# 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](#) and [SparkListenerStageCompleted](#) messages.

```
$ ./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
.StageInfo@4d3956a4
15/11/04 15:39:45 INFO StatsReportListener: task runtime:(count: 8, mean: 36.625000, s
tdev: 5.893588, max: 52.000000, min: 33.000000)
15/11/04 15:39:45 INFO StatsReportListener:          0%      5%     10%     25%     50%
           75%     90%     95%     100%
15/11/04 15:39:45 INFO StatsReportListener:          33.0 ms 33.0 ms 33.0 ms 34.0 ms 35.0 m
s      36.0 ms 52.0 ms 52.0 ms 52.0 ms
15/11/04 15:39:45 INFO StatsReportListener: task result size:(count: 8, mean: 953.0000
00, stdev: 0.000000, max: 953.000000, min: 953.000000)
15/11/04 15:39:45 INFO StatsReportListener:          0%      5%     10%     25%     50%
           75%     90%     95%     100%
15/11/04 15:39:45 INFO StatsReportListener:          953.0 B 953.0 B 953.0 B 953.0 B 953.0
B      953.0 B 953.0 B 953.0 B 953.0 B
15/11/04 15:39:45 INFO StatsReportListener: executor (non-fetch) time pct: (count: 8,
mean: 17.660220, stdev: 1.948627, max: 20.000000, min: 13.461538)
15/11/04 15:39:45 INFO StatsReportListener:          0%      5%     10%     25%     50%
           75%     90%     95%     100%
15/11/04 15:39:45 INFO StatsReportListener:          13 %   13 %   13 %   17 %   18 %
           20 %   20 %   20 %   20 %
15/11/04 15:39:45 INFO StatsReportListener: other time pct: (count: 8, mean: 82.339780
, stdev: 1.948627, max: 86.538462, min: 80.000000)
15/11/04 15:39:45 INFO StatsReportListener:          0%      5%     10%     25%     50%
           75%     90%     95%     100%
15/11/04 15:39:45 INFO StatsReportListener:          80 %   80 %   80 %   82 %   82 %
           83 %   87 %   87 %   87 %
```

# 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-2.7 -Phive -Phive-thriftserver -DskipTests clean assembly
```

Using Java 8 to build Spark using sbt takes ca 10 minutes.

```
→ spark git:(master) ✘ ./build/sbt -Pyarn -Phadoop-2.7 -Phive -Phive-thriftserver -DskipTests clean assembly
...
[success] Total time: 496 s, completed Dec 7, 2015 8:24:41 PM
```

## Build Profiles

**Caution**

[FIXME](#) Describe yarn profile and others

## Using Apache Maven as the build tool

The build command with Apache Maven is as follows:

```
$ ./build/mvn -Pyarn -P 'hadoop-2.7' -Phive -Phive-thriftserver -DskipTests clean install
```

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_92"
Java(TM) SE Runtime Environment (build 1.8.0_92-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.92-b14, mixed mode)

→ spark git:(master) ✘ ./build/mvn -Pyarn -Phadoop-2.7 -Phive -Phive-thriftserver -DskipTests clean install
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=512M; support was removed in 8.0
Using `mvn` from path: /usr/local/bin/mvn
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=512M; support was removed in 8.0
[INFO] Scanning for projects...
[INFO] -----
[INFO] Reactor Build Order:
[INFO]
[INFO] Spark Project Parent POM
[INFO] Spark Project Tags
[INFO] Spark Project Sketch
[INFO] Spark Project Networking
[INFO] Spark Project Shuffle Streaming Service
[INFO] Spark Project Unsafe
[INFO] Spark Project Launcher
[INFO] Spark Project Core
[INFO] Spark Project GraphX
[INFO] Spark Project Streaming
[INFO] Spark Project Catalyst
[INFO] Spark Project SQL
[INFO] Spark Project ML Local Library
[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 Flume Sink
[INFO] Spark Project External Flume
[INFO] Spark Project External Flume Assembly
[INFO] Spark Integration for Kafka 0.8
[INFO] Spark Project Examples
[INFO] Spark Project External Kafka Assembly
[INFO] Spark Integration for Kafka 0.10
[INFO] Spark Integration for Kafka 0.10 Assembly
[INFO] Spark Project Java 8 Tests
[INFO]
[INFO] -----
[INFO] Building Spark Project Parent POM 2.0.0-SNAPSHOT
[INFO] -----
```

```

...
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] Spark Project Parent POM ..... SUCCESS [ 4.186 s]
[INFO] Spark Project Tags ..... SUCCESS [ 4.893 s]
[INFO] Spark Project Sketch ..... SUCCESS [ 5.066 s]
[INFO] Spark Project Networking ..... SUCCESS [ 11.108 s]
[INFO] Spark Project Shuffle Streaming Service ..... SUCCESS [ 7.051 s]
[INFO] Spark Project Unsafe ..... SUCCESS [ 7.650 s]
[INFO] Spark Project Launcher ..... SUCCESS [ 9.905 s]
[INFO] Spark Project Core ..... SUCCESS [02:09 min]
[INFO] Spark Project GraphX ..... SUCCESS [ 19.317 s]
[INFO] Spark Project Streaming ..... SUCCESS [ 42.077 s]
[INFO] Spark Project Catalyst ..... SUCCESS [01:32 min]
[INFO] Spark Project SQL ..... SUCCESS [01:47 min]
[INFO] Spark Project ML Local Library ..... SUCCESS [ 10.049 s]
[INFO] Spark Project ML Library ..... SUCCESS [01:36 min]
[INFO] Spark Project Tools ..... SUCCESS [ 3.520 s]
[INFO] Spark Project Hive ..... SUCCESS [ 52.528 s]
[INFO] Spark Project REPL ..... SUCCESS [ 7.243 s]
[INFO] Spark Project YARN Shuffle Service ..... SUCCESS [ 7.898 s]
[INFO] Spark Project YARN ..... SUCCESS [ 15.380 s]
[INFO] Spark Project Hive Thrift Server ..... SUCCESS [ 24.876 s]
[INFO] Spark Project Assembly ..... SUCCESS [ 2.971 s]
[INFO] Spark Project External Flume Sink ..... SUCCESS [ 7.377 s]
[INFO] Spark Project External Flume ..... SUCCESS [ 10.752 s]
[INFO] Spark Project External Flume Assembly ..... SUCCESS [ 1.695 s]
[INFO] Spark Integration for Kafka 0.8 ..... SUCCESS [ 13.013 s]
[INFO] Spark Project Examples ..... SUCCESS [ 31.728 s]
[INFO] Spark Project External Kafka Assembly ..... SUCCESS [ 3.472 s]
[INFO] Spark Integration for Kafka 0.10 ..... SUCCESS [ 12.297 s]
[INFO] Spark Integration for Kafka 0.10 Assembly ..... SUCCESS [ 3.789 s]
[INFO] Spark Project Java 8 Tests ..... SUCCESS [ 4.267 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 12:29 min
[INFO] Finished at: 2016-07-07T22:29:56+02:00
[INFO] Final Memory: 110M/913M
[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 2.11.8** (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-2.7 -Phive -Phive  
-thriftserver -DskipTests
```

Once finished, you will have the distribution in the current directory, i.e. `spark-2.0.0-  
SNAPSHOT-bin-2.7.2.tgz`.

# Spark and Hadoop

## SparkHadoopUtil

	Caution	FIXME
Tip	<p>Enable <code>DEBUG</code> logging level for <code>org.apache.spark.deploy.SparkHadoopUtil</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.deploy.SparkHadoopUtil=DEBUG</pre> <p>Refer to <a href="#">Logging</a>.</p>	

## runAsSparkUser

```
runAsSparkUser(func: () => Unit)
```

`runAsSparkUser` runs a `func` function block with Hadoop's `UserGroupInformation` of the current user.

Internally, it reads the current username (as `SPARK_USER` environment variable or the short user name from Hadoop's `UserGroupInformation` ).

Caution	FIXME How to use <code>SPARK_USER</code> to change the current user name?
---------	---

You should see the current username printed out in the following DEBUG message in the logs:

```
DEBUG YarnSparkHadoopUtil: running as user: [user]
```

It then creates a remote user for the current user (using `UserGroupInformation.createRemoteUser` ), [transfers credential tokens](#) and runs the input `func` function as the privileged user.

## transferCredentials

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

## newConfiguration

Caution

FIXME

## Creating SparkHadoopUtil Instance (get method)

Caution

FIXME

## Hadoop Storage Formats

The currently-supported Hadoop storage formats typically used with HDFS are:

- Parquet
- RCfile
- Avro
- ORC

Caution

FIXME What are the differences between the formats and how are they used in Spark.

## 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 partitions 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.

# TransportConf — Transport Configuration

`TransportConf` is a class for the transport-related network configuration for modules, e.g. [ExternalShuffleService](#) or [YarnShuffleService](#).

It exposes methods to access settings for a single module as `spark.module.prefix` or [general network-related settings](#).

## spark.module.prefix Settings

The settings can be in the form of `spark.[module].[prefix]` with the following prefixes:

- `io.mode` (default: `NIO`) — the IO mode: `nio` or `epoll`.
- `io.preferDirectBuffs` (default: `true`) — a flag to control whether Spark prefers allocating off-heap byte buffers within Netty (`true`) or not (`false`).
- `io.connectionTimeout` (default: `spark.network.timeout` or `120s`) — the connection timeout in milliseconds.
- `io.backLog` (default: `-1` for no backlog) — the requested maximum length of the queue of incoming connections.
- `io.numConnectionsPerPeer` (default: `1`) — the number of concurrent connections between two nodes for fetching data.
- `io.serverThreads` (default: `0` i.e. `2x#cores`) — the number of threads used in the server thread pool.
- `io.clientThreads` (default: `0` i.e. `2x#cores`) — the number of threads used in the client thread pool.
- `io.receiveBuffer` (default: `-1`) — the receive buffer size (SO\_RCVBUF).
- `io.sendBuffer` (default: `-1`) — the send buffer size (SO\_SNDBUF).
- `sasl.timeout` (default: `30s`) — the timeout (in milliseconds) for a single round trip of SASL token exchange.
- `io.maxRetries` (default: `3`) — the maximum number of times Spark will try IO exceptions (such as connection timeouts) per request. If set to `0`, Spark will not do any retries.
- `io.retryWait` (default: `5s`) — the time (in milliseconds) that Spark will wait in order to perform a retry after an `IOException`. Only relevant if `io.maxRetries > 0`.

- `io.lazyFD` (default: `true`)—controls whether to initialize `FileDescriptor` lazily (`true`) or not (`false`). If `true`, file descriptors are created only when data is going to be transferred. This can reduce the number of open files.

## General Network-Related Settings

### **spark.storage.memoryMapThreshold**

`spark.storage.memoryMapThreshold` (default: `2m`) is the minimum size of a block that we should start using memory map rather than reading in through normal IO operations.

This prevents Spark from memory mapping very small blocks. In general, memory mapping has high overhead for blocks close to or below the page size of the OS.

### **spark.network.sasl.maxEncryptedBlockSize**

`spark.network.sasl.maxEncryptedBlockSize` (default: `64k`) is the maximum number of bytes to be encrypted at a time when SASL encryption is enabled.

### **spark.network.sasl.serverAlwaysEncrypt**

`spark.network.sasl.serverAlwaysEncrypt` (default: `false`) controls whether the server should enforce encryption on SASL-authenticated connections (`true`) or not (`false`).

# Interactive Notebooks

This document aims at presenting and eventually supporting me to select the open-source web-based visualisation tool for [Apache Spark](#) with [Scala](#) 2.11 support.

## Requirements

1. Support for Apache Spark 2.0
2. Support for Scala 2.11 (the default Scala version for Spark 2.0)
3. Web-based
4. Open Source with [ASL 2.0](<http://www.apache.org/licenses/LICENSE-2.0>) or similar license
5. Notebook Sharing using GitHub
6. Active Development and Community (the number of commits per week and month, github, gitter)
7. Autocompletion

Optional Requirements:

1. Sharing SparkContext

## Candidates

- [Apache Zeppelin](#)
- [Spark Notebook](#)
- [Beaker](#)
- [Jupyter Notebook](#)
  - [Jupyter Scala](#) - Lightweight Scala kernel for [Jupyter Notebook](#)

## Jupyter Notebook

You can combine code execution, rich text, mathematics, plots and rich media

- [Jupyter Notebook](#) (formerly known as the [IPython Notebook](#))- open source, interactive data science and scientific computational environment supporting over 40 programming languages.

## Further reading or watching

- (Quora) Is there a preference in the data science/analyst community between the [iPython](#) Spark notebook and Zeppelin? It looks like both support Scala, Python and SQL. What are the shortcomings of one vs the other?

# Apache Zeppelin

Apache Zeppelin is a web-based notebook platform that enables interactive data analytics with interactive data visualizations and notebook sharing. You can make data-driven, interactive and collaborative documents with SQL, Scala, Python, R in a single notebook document.

It shares a single `SparkContext` between languages (so you can pass data between Scala and Python easily).

This is an excellent tool for prototyping Scala/Spark code with SQL queries to analyze data (by data visualizations) that could be used by non-Scala developers like data analysts using SQL and Python.

Note	Zeppelin aims at more analytics and business people (not necessarily for Spark/Scala developers for whom <a href="#">Spark Notebook</a> may appear a better fit).
------	---

Clients talk to the Zeppelin Server using HTTP REST or Websocket endpoints.

Available basic and advanced **display systems**:

- text (default)
- HTML
- table
- Angular

## Features

1. Apache License 2.0 licensed
2. Interactive
3. Web-Based
4. Data Visualization (charts)
5. Collaboration by Sharing Notebooks and Paragraphs
6. Multiple Language and Data Processing Backends called **Interpreters**, including the **built-in Apache Spark integration**, Apache Flink, Apache Hive, Apache Cassandra, Apache Tajo, Apache Phoenix, Apache Ignite, Apache Geode
7. Display Systems

8. Built-in Scheduler to run notebooks with cron expression

## Further reading or watching

1. (video) [Data Science Lifecycle with Zeppelin and Spark](#) from Spark Summit Europe 2015 with the creator of the Apache Zeppelin project — Moon soo Lee.

# Spark Notebook

[Spark Notebook](#) is a Scala-centric tool for interactive and reactive data science using Apache Spark.

This is an excellent tool for prototyping Scala/Spark code with SQL queries to analyze data (by data visualizations). It seems to have [more advanced data visualizations](#) (comparing to [Apache Zeppelin](#)), and seems rather focused on Scala, SQL and Apache Spark.

It can visualize the output of SQL queries directly as tables and charts (which [Apache Zeppelin](#) cannot yet).

Note	Spark Notebook is best suited for Spark/Scala developers. Less development-oriented people may likely find Apache Zeppelin a better fit.
------	--

Spark Notebook is best suited for Spark/Scala developers. Less development-oriented people may likely find Apache Zeppelin a better fit.

# 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 /Users/jacek/dev/oss/spark/conf/:/Users/jacek/dev/oss/spark/assembly/target/scala-2.11/spark-assembly-1.6.0-SNAPSHOT-hadoop2.7.1.jar:/Users/jacek/dev/oss/spark/lib_managed/jars/datanucleus-api-jdo-3.2.6.jar:/Users/jacek/dev/oss/spark/lib_managed/jars/datanucleus-core-3.2.10.jar:/Users/jacek/dev/oss/spark/lib_managed/jars/datanucleus-rdbms-3.2.9.jar -Dscala.usejavacp=true -Xms1g -Xmx1g org.apache.spark.deploy.SparkSubmit --master spark://localhost:7077 --class org.apache.spark.repl.Main --name Spark shell spark-shell
=====
```

## 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

   ____
  / _\|_ _ _ _ _ / /_
 _\ \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> 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$$clean(ClosureCleaner.scala:294)
  at org.apache.spark.util.ClosureCleaner$.clean(ClosureCleaner.scala:122)
  at org.apache.spark.SparkContext.clean(SparkContext.scala:2055)
  at org.apache.spark.rdd.RDD$$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(SerializationDebugger.scala:40)
at org.apache.spark.serializer.JavaSerializationStream.writeObject(JavaSerializer.scala:47)
at org.apache.spark.serializer.JavaSerializerInstance.serialize(JavaSerializer.scala:101)
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 different from other operating systems like Linux or Mac OS X, but there are few minor issues due to the way Hive works on Windows, among them is a permission error when running Spark Shell.

**Note**

The issue is due to the way Hive works on Windows. You need no changes if you need no Hive integration in Spark SQL.

```
15/01/29 17:21:27 ERROR Shell: Failed to locate the winutils binary in the hadoop binary path
java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.
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)
```

**Note**

You need to have Administrator rights on your laptop. All the following commands must be executed in a command-line window (`cmd`) ran as Administrator, i.e. using **Run As Administrator** option while executing `cmd`.

Download [winutils.exe](#) and save it to a directory of your choice, say `c:\hadoop\bin`.

Set `HADOOP_HOME` to reflect the directory with `winutils` (without `bin`).

```
set HADOOP_HOME=c:\hadoop
```

Set `PATH` environment variable to include `%HADOOP_HOME%\bin` as follows:

```
set PATH=%HADOOP_HOME%\bin;%PATH%
```

**Tip**

Define `HADOOP_HOME` and `PATH` environment variables in Control Panel.

Create `c:\tmp\hive` folder and execute the following command:

```
winutils.exe chmod -R 777 \tmp\hive
```

Check the permissions:

```
winutils.exe ls \tmp\hive
```

Open `spark-shell` and report SUCCESS!

# 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](#).

## 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, "user4"), (3, "user1"))

// 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>:18

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	<a href="#">FIXME</a> 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	<a href="#">FIXME</a> 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	<a href="#">FIXME</a> 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.conf
```

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
8080 -h localhost -p 17077 --webui-port 18080 --properties-file ha.conf
4808 org.apache.spark.deploy.master.Master --ip japila.local --port 7077 --webui-port
8080 -h localhost -p 7077 --webui-port 8080 --properties-file ha.conf
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-version),1)
(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/
sparkme-app/)
[info] Compiling 1 Scala source to /Users/jacek/dev/sandbox/sparkme-app/target/scala-2
.11/classes...
[info] Packaging /Users/jacek/dev/sandbox/sparkme-app/target/scala-2.11/sparkme-projec
t_2.11-1.0.jar ...
[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
.spark.SparkMeApp target/scala-2.11/sparkme-project_2.11-1.0.jar build.sbt
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
platform... using builtin-java classes where applicable
15/09/23 01:06:04 WARN MetricsSystem: Using default name DAGScheduler for source because
spark.app.id is not set.
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"><li>• <a href="#">FIXME</a> Offer a complete working solution in Scala</li><li>• <a href="#">FIXME</a> Load ORC files into dataframe<ul style="list-style-type: none"><li>◦ <code>val df = hiveContext.read.format("orc").load(to/path)</code></li></ul></li></ul>
---------	--

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 Spark Listener. You should read [Spark 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, SparkLi
stenerJobStart}

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 ${stageComplete
d.stageInfo.numTasks} tasks.")
  }
}

```

## 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/targ
et/scala-2.11/classes...
[info] Packaging /Users/jacek/dev/sandbox/custom-spark-listener/target/scala-2.11/cust
om-spark-listener_2.11-1.0.jar ...
[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.sp
ark.CustomSparkListener --driver-class-path /Users/jacek/dev/sandbox/custom-spark-list
ener/target/scala-2.11/custom-spark-listener_2.11-1.0.jar
```

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.spark.scheduler.StageInfo@4b08f37b),{})
...
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><b>FIXME</b></p> <ul style="list-style-type: none"> <li>• Create the exercise</li> <li>• 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.</li> </ul>
---------	---

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(AbstractFileC
lassLoader.scala:62)
    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$class.forName(Utils.scala:173)
    at org.apache.spark.rpc.RpcEnv$_.getRpcEnvFactory(RpcEnv.scala:38)
    at org.apache.spark.rpc.RpcEnv$.create(RpcEnv.scala:49)
    at org.apache.spark.SparkEnv$.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:6
2)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImp
1.java:43)
```

```

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$$anonfun$loadAndRunReq$1.ap-
ply(IMain.scala:636)
at scala.tools.nsc.interpreter.IMain$WrappedRequest$$anonfun$loadAndRunReq$1.ap-
ply(IMain.scala:635)
at scala.reflect.internal.util.ScalaClassLoader$class.asContext(ScalaClassLoad-
er.scala:31)
at scala.reflect.internal.util.AbstractFileClassLoader.asContext(AbstractFileC-
lassLoader.scala:19)
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$$anonfun$initializeSpark$1.apply$mcZ$sp(Sp-
arkILoop.scala:39)
at org.apache.spark.repl.SparkILoop$$anonfun$initializeSpark$1.apply(SparkILoo-
p.scala:38)
at org.apache.spark.repl.SparkILoop$$anonfun$initializeSpark$1.apply(SparkILoo-
p.scala:38)
at scala.tools.nsc.interpreter.IMain.beQuietDuring(IMain.scala:213)
at org.apache.spark.repl.SparkILoop.initializeSpark(SparkILoop.scala:38)
at org.apache.spark.repl.SparkILoop.loadFiles(SparkILoop.scala:94)
at scala.tools.nsc.interpreter.ILoop$$anonfun$process$1.apply$mcZ$sp(ILoop.sca-
la:922)
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(ScalaClas-
sLoader.scala:97)
at scala.tools.nsc.interpreter.ILoop.process(ILoop.scala:911)
at org.apache.spark.repl.Main$.main(Main.scala:49)
at org.apache.spark.repl.Main.main(Main.scala)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:6-
2)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImp-
l.java:43)
at java.lang.reflect.Method.invoke(Method.java:497)
at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit$$r-
unMain(SparkSubmit.scala:680)
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	<a href="#">FIXME</a>
---------	-----------------------

# Creating DataFrames from Tables using JDBC and PostgreSQL

Start `spark-shell` with the proper JDBC driver.

Note	Download PostgreSQL JDBC Driver JDBC 4.1 ( <code>postgresql-9.4.1208.jar</code> ) from <a href="#">the Maven repository</a> .
Tip	<p>Execute the command to have the jar downloaded into <code>~/.ivy2/jars</code> directory by <code>spark-shell</code> :</p> <pre>./bin/spark-shell --packages org.postgresql:postgresql:9.4.1208</pre> <p>The entire path to the driver file is like  <code>/Users/jacek/.ivy2/jars/org.postgresql_postgresql-9.4.1208.jar</code> .</p>

Start `./bin/spark-shell` with `--driver-class-path` command line option and the driver jar.

```
SPARK_PRINT_LAUNCH_COMMAND=1 ./bin/spark-shell --driver-class-path /Users/jacek/.m2/repository/org/postgresql/postgresql/9.4.1207.jre7/postgresql-9.4.1207.jre7.jar
```

It will give you the proper setup for accessing PostgreSQL using the JDBC driver.

Execute the following to access `projects` table in `sparkdb` .

```
val opts = Map(
  "url" -> "jdbc:postgresql:sparkdb",
  "dbtable" -> "projects")
val df = spark
  .read
  .format("jdbc")
  .options(opts)
  .load

scala> df.show(false)
+---+-----+-----+
|id |name      |website          |
+---+-----+-----+
|1  |Apache Spark|http://spark.apache.org|
|2  |Apache Hive  |http://hive.apache.org |
|3  |Apache Kafka|http://kafka.apache.org|
|4  |Apache Flink|http://flink.apache.org|
+---+-----+-----+
```

## Troubleshooting

If things can go wrong, they sooner or later go wrong. Here is a list of possible issues and their solutions.

### **java.sql.SQLException: No suitable driver**

Ensure that the JDBC driver sits on the CLASSPATH. Use `--driver-class-path` as described above.

```
scala> spark.read.format("jdbc").options(Map("url" -> "jdbc:postgresql:dbserver", "dbtable" -> "projects")).load
java.sql.SQLException: No suitable driver
  at java.sql.DriverManager.getDriver(DriverManager.java:315)
  at org.apache.spark.sql.execution.datasources.jdbc.JdbcUtils$$anonfun$2.apply(JdbcUtils.scala:50)
  at org.apache.spark.sql.execution.datasources.jdbc.JdbcUtils$$anonfun$2.apply(JdbcUtils.scala:50)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.sql.execution.datasources.jdbc.JdbcUtils$.createConnectionFactory(JdbcUtils.scala:49)
  at org.apache.spark.sql.execution.datasources.jdbc.JDBCRDD$.resolveTable(JDBCRDD.scala:123)
  at org.apache.spark.sql.execution.datasources.jdbc.JDBCRelation.<init>(JDBCRelation.scala:97)
  at org.apache.spark.sql.execution.datasources.jdbc.DefaultSource.createRelation(DataSource.scala:57)
  at org.apache.spark.sql.execution.datasources.DataSource.resolveRelation(DataSource.scala:225)
  at org.apache.spark.sql.DataFrameReader.load(DataFrameReader.scala:132)
  ... 48 elided
```

## PostgreSQL Setup

Note	I'm on Mac OS X so YMMV (aka <i>Your Mileage May Vary</i> ).
------	--

Use the sections to have a properly configured PostgreSQL database.

- [Installation](#)
- [Starting Database Server](#)
- [Create Database](#)
- [Accessing Database](#)
- [Creating Table](#)

- Dropping Database
- Stopping Database Server

## Installation

Install PostgreSQL as described in...TK

Caution	This page serves as a cheatsheet for the author so he does not have to search Internet to find the installation steps.
---------	--

```
$ initdb /usr/local/var/postgres -E utf8
The files belonging to this database system will be owned by user "jacek".
This user must also own the server process.

The database cluster will be initialized with locale "pl_pl.utf-8".
initdb: could not find suitable text search configuration for locale "pl_pl.utf-8"
The default text search configuration will be set to "simple".

Data page checksums are disabled.

creating directory /usr/local/var/postgres ... ok
creating subdirectories ... ok
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting dynamic shared memory implementation ... posix
creating configuration files ... ok
creating template1 database in /usr/local/var/postgres/base/1 ... ok
initializing pg_authid ... ok
initializing dependencies ... ok
creating system views ... ok
loading system objects' descriptions ... ok
creating collations ... ok
creating conversions ... ok
creating dictionaries ... ok
setting privileges on built-in objects ... ok
creating information schema ... ok
loading PL/pgSQL server-side language ... ok
vacuuming database template1 ... ok
copying template1 to template0 ... ok
copying template1 to postgres ... ok
syncing data to disk ... ok

WARNING: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.

Success. You can now start the database server using:

pg_ctl -D /usr/local/var/postgres -l logfile start
```

## Starting Database Server

**Note** Consult [17.3. Starting the Database Server](#) in the official documentation.

Start the database server using `pg_ctl`.

```
$ pg_ctl -D /usr/local/var/postgres -l logfile start
server starting
```

Alternatively, you can run the database server using `postgres`.

```
$ postgres -D /usr/local/var/postgres
```

## Create Database

```
$ createdb sparkdb
```

**Tip** Consult [createdb](#) in the official documentation.

## Accessing Database

Use `psql sparkdb` to access the database.

```
$ psql sparkdb
psql (9.5.2)
Type "help" for help.

sparkdb=#
```

Execute `SELECT version()` to know the version of the database server you have connected to.

```
sparkdb=# SELECT version();
              version
-----
-
-
PostgreSQL 9.5.2 on x86_64-apple-darwin14.5.0, compiled by Apple LLVM version 7.0.2 (
clang-700.1.81), 64-bit
(1 row)
```

Use `\h` for help and `\q` to leave a session.

## Creating Table

Create a table using `CREATE TABLE` command.

```
CREATE TABLE projects (
    id SERIAL PRIMARY KEY,
    name text,
    website text
);
```

Insert rows to initialize the table with data.

```
INSERT INTO projects (name, website) VALUES ('Apache Spark', 'http://spark.apache.org');
INSERT INTO projects (name, website) VALUES ('Apache Hive', 'http://hive.apache.org');
INSERT INTO projects VALUES (DEFAULT, 'Apache Kafka', 'http://kafka.apache.org');
INSERT INTO projects VALUES (DEFAULT, 'Apache Flink', 'http://flink.apache.org');
```

Execute `select * from projects;` to ensure that you have the following records in `projects` table:

```
sparkdb=# select * from projects;
+----+-----+-----+
| id | name | website |
+----+-----+-----+
| 1  | Apache Spark | http://spark.apache.org |
| 2  | Apache Hive | http://hive.apache.org |
| 3  | Apache Kafka | http://kafka.apache.org |
| 4  | Apache Flink | http://flink.apache.org |
+----+-----+-----+
(4 rows)
```

## Dropping Database

```
$ dropdb sparkdb
```

Tip

Consult [dropdb](#) in the official documentation.

## Stopping Database Server

```
pg_ctl -D /usr/local/var/postgres stop
```



# Exercise: Causing Stage to Fail

The example shows how Spark re-executes a stage in case of stage failure.

## Recipe

Start a Spark cluster, e.g. 1-node Hadoop YARN.

```
start-yarn.sh
```

```
// 2-stage job -- it _appears_ that a stage can be failed only when there is a shuffle
sc.parallelize(0 to 3e3.toInt, 2).map(n => (n % 2, n)).groupByKey.count
```

Use 2 executors at least so you can kill one and keep the application up and running (on one executor).

```
YARN_CONF_DIR=hadoop-conf ./bin/spark-shell --master yarn \
-c spark.shuffle.service.enabled=true \
--num-executors 2
```

## Spark courses

- [Spark Fundamentals I](#) from Big Data University.
- [Data Science and Engineering with Apache Spark](#) from University of California and Databricks (includes 5 edX courses):
  - [Introduction to Apache Spark](#)
  - [Distributed Machine Learning with Apache Spark](#)
  - [Big Data Analysis with Apache Spark](#)
  - [Advanced Apache Spark for Data Science and Data Engineering](#)
  - [Advanced Distributed Machine Learning with Apache Spark](#)

# 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 \(Case Studies with Hadoop, Scalding and Spark\)](#)

# DataStax Enterprise

[DataStax Enterprise](#)

# MapR Sandbox for Hadoop

[MapR Sandbox for Hadoop](#) is a Spark distribution from MapR.

The MapR Sandbox for Hadoop is a fully-functional single-node cluster that gently introduces business analysts, current and aspiring Hadoop developers, and administrators (database, system, and Hadoop) to the big data promises of Hadoop and its ecosystem. Use the sandbox to experiment with Hadoop technologies using the MapR Control System (MCS) and Hue.

The latest version of MapR Sandbox for Hadoop 5.1 uses Spark **1.5.2** and is *completely* unsuitable for any Spark development (given how old the Spark version is used in the box).

The documentation is available at <http://maprdocs.mapr.com/51/>.

# 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](mailto: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.

## Spark Core

- Don't fear the logs - Learn Spark by Logs
- Everything you always wanted to know about accumulators (and task metrics)
- Optimizing Spark using SchedulableBuilders
- [Learning Spark internals using groupBy \(to cause shuffle\)](#)

## Spark on Cluster

- [10 Lesser-Known Tidbits about Spark Standalone](#)

## Spark Streaming

- 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.