
Table of Contents

Introduction	1.1
Overview of Apache Spark	1.2

Spark SQL

Spark SQL — Queries Over Structured Data on Massive Scale	2.1
SparkSession — The Entry Point to Spark SQL	2.2
Builder — Building SparkSession using Fluent API	2.2.1
SharedState — Shared State Across SparkSessions	2.2.2
Dataset — Strongly-Typed Structured Query with Encoder	2.3
Encoders — Internal Row Converters	2.3.1
ExpressionEncoder — Expression-Based Encoder	2.3.2
DataFrame — Dataset of Rows	2.3.3
Row	2.3.3.1
RowEncoder — Encoder for DataFrames	2.3.3.2
Schema — Structure of Data	2.4
StructType	2.4.1
StructField	2.4.2
Data Types	2.4.3
Dataset Operators	2.5
Column Operators	2.5.1
Standard Functions — functions Object	2.5.2
Standard Functions for Date and Time	2.5.2.1
Window Aggregate Functions	2.5.2.2
User-Defined Functions (UDFs)	2.5.3
Basic Aggregation — Typed and Untyped Grouping Operators	2.5.4
Joins	2.5.5
Broadcast Joins (aka Map-Side Joins)	2.5.5.1
Multi-Dimensional Aggregation	2.5.6
UserDefinedAggregateFunction — User-Defined Aggregate Functions (UDAFs)	

Dataset Caching and Persistence	2.5.8	2.5.7
RelationalGroupedDataset — Untyped Row-based Grouping		2.5.9
DataSource API — Loading and Saving Datasets		2.6
DataFrameReader — Reading Datasets from External Data Sources		2.6.1
DataFrameWriter		2.6.2
DataSource — Pluggable Data Provider Framework		2.6.3
 CreatableRelationProvider — Data Sources That Save Rows Per Save Mode		
 RelationProvider — Data Sources With Schema Inference	2.6.3.2	2.6.3.1
 SchemaRelationProvider — Data Sources With Mandatory User-Defined Schema		2.6.3.3
 DataSourceRegister		2.6.4
 CSVFileFormat		2.6.4.1
 JdbcRelationProvider		2.6.4.2
 JsonFileFormat		2.6.4.3
 JsonDataSource		2.6.4.4
 ParquetFileFormat		2.6.4.5
 Custom Formats		2.6.5
BaseRelation — Collection of Tuples with Schema		2.7
 HadoopFsRelation		2.7.1
 JDBCRelation		2.7.2
QueryExecution — Query Execution of Dataset		2.8
Expression — Executable Node in Catalyst Tree		2.9
 AggregateExpression		2.9.1
 AggregateFunction		2.9.2
 Attribute Leaf Expression		2.9.3
 CallMethodViaReflection Expression		2.9.4
 Generator — Catalyst Expressions that Generate Zero Or More Rows		2.9.5
 JsonToStructs Unary Expression		2.9.6
 Literal Leaf Expression		2.9.7
 ScalaUDAF		2.9.8
 StaticInvoke Non-SQL Expression		2.9.9
 TimeWindow Unevaluable Unary Expression		2.9.10
 UnixTimestamp TimeZoneAware Binary Expression		2.9.11
 WindowExpression Unevaluable Expression		2.9.12

WindowSpecDefinition Unevaluable Expression	2.9.12.1
LogicalPlan — Logical Query Plan / Logical Operator	2.10
Aggregate Unary Logical Operator	2.10.1
BroadcastHint Unary Logical Operator	2.10.2
DeserializeToObject Logical Operator	2.10.3
Expand Unary Logical Operator	2.10.4
GroupingSets Unary Logical Operator	2.10.5
Hint Logical Operator	2.10.6
InMemoryRelation Leaf Logical Operator For Cached Query Plans	2.10.7
Join Logical Operator	2.10.8
LocalRelation Logical Operator	2.10.9
LogicalRelation Logical Operator — Adapter for BaseRelation	2.10.10
Pivot Unary Logical Operator	2.10.11
Repartition Logical Operators — Repartition and RepartitionByExpression	2.10.12
RunnableCommand — Generic Logical Command with Side Effects	2.10.13
AlterViewAsCommand Logical Command	2.10.13.1
ClearCacheCommand Logical Command	2.10.13.2
CreateDataSourceTableCommand Logical Command	2.10.13.3
CreateViewCommand Logical Command	2.10.13.4
ExplainCommand Logical Command	2.10.13.5
SubqueryAlias Logical Operator	2.10.14
UnresolvedFunction Logical Operator	2.10.15
UnresolvedRelation Logical Operator	2.10.16
Window Unary Logical Operator	2.10.17
Analyzer — Logical Query Plan Analyzer	2.11
CheckAnalysis — Analysis Validation	2.11.1
SparkOptimizer — Logical Query Optimizer	2.12
Optimizer — Base for Logical Query Plan Optimizers	2.12.1
ColumnPruning	2.12.2
CombineTypedFilters	2.12.3
ConstantFolding	2.12.4
CostBasedJoinReorder	2.12.5
DecimalAggregates	2.12.6
EliminateSerialization	2.12.7

GetCurrentDatabase / ComputeCurrentTime	2.12.8
LimitPushDown	2.12.9
NullPropagation — Nullability (NULL Value) Propagation	2.12.10
PropagateEmptyRelation	2.12.11
PushDownPredicate — Predicate Pushdown / Filter Pushdown Logical Plan Optimization	2.12.12
ReorderJoin	2.12.13
SimplifyCasts	2.12.14
SparkPlan — Physical Query Plan / Physical Operator	2.13
BroadcastExchangeExec Unary Operator for Broadcasting Joins	2.13.1
BroadcastHashJoinExec Binary Physical Operator	2.13.2
BroadcastNestedLoopJoinExec Binary Physical Operator	2.13.3
CoalesceExec Unary Physical Operator	2.13.4
DataSourceScanExec — Contract for Leaf Physical Operators with Code Generation	2.13.5
FileSourceScanExec Physical Operator	2.13.5.1
RowDataSourceScanExec Physical Operator	2.13.5.2
ExecutedCommandExec Physical Operator	2.13.6
HashAggregateExec Aggregate Physical Operator	2.13.7
InMemoryTableScanExec Physical Operator	2.13.8
LocalTableScanExec Physical Operator	2.13.9
ShuffleExchange Unary Physical Operator	2.13.10
ShuffledHashJoinExec Binary Physical Operator	2.13.11
SortMergeJoinExec Binary Physical Operator	2.13.12
InputAdapter Unary Physical Operator	2.13.13
WindowExec Physical Operator	2.13.14
WholeStageCodegenExec Unary Operator with Java Code Generation	2.13.15
SparkPlanner — Query Planner with no Hive Support	2.14
SparkStrategy — Base for Execution Planning Strategies	2.14.1
SparkStrategies — Container of Execution Planning Strategies	2.14.2
Aggregation Execution Planning Strategy	2.14.3
BasicOperators Execution Planning Strategy	2.14.4
DataSourceStrategy Execution Planning Strategy	2.14.5
FileSourceStrategy Execution Planning Strategy	2.14.6

InMemoryScans Execution Planning Strategy	2.14.7
JoinSelection Execution Planning Strategy	2.14.8
Physical Plan Preparations Rules	2.15
CollapseCodegenStages Physical Preparation Rule — Collapsing Physical Operators for Whole-Stage CodeGen	2.15.1
EnsureRequirements Physical Preparation Rule	2.15.2
SQL Parsing Framework	2.16
SparkSqlParser — Default SQL Parser	2.16.1
SparkSqlAstBuilder	2.16.1.1
CatalystSqlParser — DataTypes and StructTypes Parser	2.16.2
AstBuilder — ANTLR-based SQL Parser	2.16.3
AbstractSqlParser — Base SQL Parsing Infrastructure	2.16.4
ParserInterface — SQL Parser Contract	2.16.5
SQLMetric — Physical Operator Metric	2.17
Catalyst — Tree Manipulation Framework	2.18
TreeNode — Node in Catalyst Tree	2.18.1
QueryPlan — Structured Query Plan	2.18.2
RuleExecutor — Tree Transformation Rule Executor	2.18.3
GenericStrategy	2.18.4
QueryPlanner — Converting Logical Plan to Physical Trees	2.18.5
Catalyst DSL — Implicit Conversions for Catalyst Data Structures	2.18.6
ExchangeCoordinator and Adaptive Query Execution	2.19
ShuffledRowRDD	2.20
Debugging Query Execution	2.21
Datasets vs DataFrames vs RDDs	2.22
SQLConf	2.23
CatalystConf	2.23.1
Catalog	2.24
CatalogImpl	2.24.1
ExternalCatalog — System Catalog of Permanent Entities	2.25
SessionState	2.26
BaseSessionStateBuilder — Base for Builders of SessionState	2.26.1
SessionCatalog — Metastore of Session-Specific Relational Entities	2.27
UDFRegistration	2.28

FunctionRegistry	2.29
ExperimentalMethods	2.30
SQLExecution Helper Object	2.31
CatalystSerde	2.32
Tungsten Execution Backend (aka Project Tungsten)	2.33
Whole-Stage Code Generation (CodeGen)	2.33.1
CodegenSupport — Physical Operators with Optional Java Code Generation	
InternalRow — Abstract Binary Row Format	2.33.3 2.33.2
UnsafeRow — Mutable Raw-Memory Unsafe Binary Row Format	2.33.3.1
UnsafeProjection	2.33.4
CodeGenerator	2.33.5
GenerateUnsafeProjection	2.33.5.1
JdbcDialect	2.34
KafkaWriter — Writing Dataset to Kafka	2.35
KafkaSourceProvider	2.35.1
KafkaWriteTask	2.35.2
Hive Integration	2.36
Spark SQL CLI - spark-sql	2.36.1
DataSinks Strategy	2.36.2
CacheManager — In-Memory Cache for Tables and Views	2.37
Thrift JDBC/ODBC Server — Spark Thrift Server (STS)	2.38
SparkSQLEnv	2.38.1
(obsolete) SQLContext	2.39
Settings	2.40

Spark MLlib

Spark MLlib — Machine Learning in Spark	3.1
ML Pipelines and PipelineStages (spark.ml)	3.2
ML Pipeline Components — Transformers	3.2.1
Tokenizer	3.2.1.1
ML Pipeline Components — Estimators	3.2.2
ML Pipeline Models	3.2.3
Evaluators	3.2.4

CrossValidator	3.2.5
Params and ParamMaps	3.2.6
ML Persistence — Saving and Loading Models and Pipelines	3.2.7
Example — Text Classification	3.2.8
Example — Linear Regression	3.2.9
Latent Dirichlet Allocation (LDA)	3.3
Vector	3.4
LabeledPoint	3.5
Streaming MLlib	3.6
GeneralizedLinearRegression	3.7

Structured Streaming

Spark Structured Streaming — Streaming Datasets	4.1
---	-----

Spark Core / Tools

Spark Shell — spark-shell shell script	5.1
Web UI — Spark Application's Web Console	5.2
Jobs Tab	5.2.1
Stages Tab — Stages for All Jobs	5.2.2
Stages for All Jobs	5.2.2.1
Stage Details	5.2.2.2
Pool Details	5.2.2.3
Storage Tab	5.2.3
BlockStatusListener Spark Listener	5.2.3.1
Environment Tab	5.2.4
EnvironmentListener Spark Listener	5.2.4.1
Executors Tab	5.2.5
ExecutorsListener Spark Listener	5.2.5.1
SQL Tab	5.2.6
SQLListener Spark Listener	5.2.6.1
JobProgressListener Spark Listener	5.2.7
StorageStatusListener Spark Listener	5.2.8

StorageListener — Spark Listener for Tracking Persistence Status of RDD Blocks	
RDDOperationGraphListener Spark Listener	5.2.10 5.2.9
SparkUI	5.2.11
Spark Submit — spark-submit shell script	5.3
SparkSubmitArguments	5.3.1
SparkSubmitOptionParser — spark-submit's Command-Line Parser	5.3.2
SparkSubmitCommandBuilder Command Builder	5.3.3
spark-class shell script	5.4
AbstractCommandBuilder	5.4.1
SparkLauncher — Launching Spark Applications Programmatically	5.5

Spark Core / Architecture

Spark Architecture	6.1
Driver	6.2
Executor	6.3
TaskRunner	6.3.1
ExecutorSource	6.3.2
Master	6.4
Workers	6.5

Spark Core / RDD

Anatomy of Spark Application	7.1
SparkConf — Programmable Configuration for Spark Applications	7.2
Spark Properties and spark-defaults.conf Properties File	7.2.1
Deploy Mode	7.2.2
SparkContext	7.3
HeartbeatReceiver RPC Endpoint	7.3.1
Inside Creating SparkContext	7.3.2
ConsoleProgressBar	7.3.3
SparkStatusTracker	7.3.4
Local Properties — Creating Logical Job Groups	7.3.5
RDD — Resilient Distributed Dataset	7.4

RDD Lineage — Logical Execution Plan	7.4.1
TaskLocation	7.4.2
ParallelCollectionRDD	7.4.3
MapPartitionsRDD	7.4.4
OrderedRDDFunctions	7.4.5
CoGroupedRDD	7.4.6
SubtractedRDD	7.4.7
HadoopRDD	7.4.8
NewHadoopRDD	7.4.9
ShuffledRDD	7.4.10
BlockRDD	7.4.11
Operators	7.5
Transformations	7.5.1
PairRDDFunctions	7.5.1.1
Actions	7.5.2
Caching and Persistence	7.6
StorageLevel	7.6.1
Partitions and Partitioning	7.7
Partition	7.7.1
Partitioner	7.7.2
HashPartitioner	7.7.2.1
Shuffling	7.8
Checkpointing	7.9
CheckpointRDD	7.9.1
RDD Dependencies	7.10
NarrowDependency — Narrow Dependencies	7.10.1
ShuffleDependency — Shuffle Dependencies	7.10.2
Map/Reduce-side Aggregator	7.11

Spark Core / Optimizations

Broadcast variables	8.1
Accumulators	8.2
AccumulatorContext	8.2.1

Spark Core / Services

SerializerManager	9.1
MemoryManager — Memory Management	9.2
UnifiedMemoryManager	9.2.1
SparkEnv — Spark Runtime Environment	9.3
DAGScheduler — Stage-Oriented Scheduler	9.4
Jobs	9.4.1
Stage — Physical Unit Of Execution	9.4.2
ShuffleMapStage — Intermediate Stage in Execution DAG	9.4.2.1
ResultStage — Final Stage in Job	9.4.2.2
StageInfo	9.4.2.3
DAGScheduler Event Bus	9.4.3
JobListener	9.4.4
JobWaiter	9.4.4.1
TaskScheduler — Spark Scheduler	9.5
Tasks	9.5.1
ShuffleMapTask — Task for ShuffleMapStage	9.5.1.1
ResultTask	9.5.1.2
TaskDescription	9.5.2
FetchFailedException	9.5.3
MapStatus — Shuffle Map Output Status	9.5.4
TaskSet — Set of Tasks for Stage	9.5.5
TaskSetManager	9.5.6
Schedulable	9.5.6.1
Schedulable Pool	9.5.6.2
Schedulable Builders	9.5.6.3
FIFOSchedulableBuilder	9.5.6.3.1
FairSchedulableBuilder	9.5.6.3.2
Scheduling Mode — spark.scheduler.mode Spark Property	9.5.6.4
TaskInfo	9.5.6.5
TaskSchedulerImpl — Default TaskScheduler	9.5.7
Speculative Execution of Tasks	9.5.7.1
TaskResultGetter	9.5.7.2

TaskContext	9.5.8
TaskContextImpl	9.5.8.1
TaskResults — DirectTaskResult and IndirectTaskResult	9.5.9
TaskMemoryManager	9.5.10
MemoryConsumer	9.5.10.1
TaskMetrics	9.5.11
ShuffleWriteMetrics	9.5.11.1
TaskSetBlacklist — Blacklisting Executors and Nodes For TaskSet	9.5.12
SchedulerBackend — Pluggable Scheduler Backends	9.6
CoarseGrainedSchedulerBackend	9.6.1
DriverEndpoint — CoarseGrainedSchedulerBackend RPC Endpoint	9.6.1.1
ExecutorBackend — Pluggable Executor Backends	9.7
CoarseGrainedExecutorBackend	9.7.1
MesosExecutorBackend	9.7.2
BlockManager — Key-Value Store for Blocks	9.8
MemoryStore	9.8.1
DiskStore	9.8.2
BlockDataManager	9.8.3
ShuffleClient	9.8.4
BlockTransferService — Pluggable Block Transfers	9.8.5
NettyBlockTransferService — Netty-Based BlockTransferService	9.8.5.1
NettyBlockRpcServer	9.8.5.2
BlockManagerMaster — BlockManager for Driver	9.8.6
BlockManagerMasterEndpoint — BlockManagerMaster RPC Endpoint	9.8.6.1
DiskBlockManager	9.8.7
BlockInfoManager	9.8.8
BlockInfo	9.8.8.1
BlockManagerSlaveEndpoint	9.8.9
DiskBlockObjectWriter	9.8.10
BlockManagerSource — Metrics Source for BlockManager	9.8.11
StorageStatus	9.8.12
MapOutputTracker — Shuffle Map Output Registry	9.9
MapOutputTrackerMaster — MapOutputTracker For Driver	9.9.1
MapOutputTrackerMasterEndpoint	9.9.1.1

MapOutputTrackerWorker — MapOutputTracker for Executors	9.9.2
ShuffleManager — Pluggable Shuffle Systems	9.10
SortShuffleManager — The Default Shuffle System	9.10.1
ExternalShuffleService	9.10.2
OneForOneStreamManager	9.10.3
ShuffleBlockResolver	9.10.4
IndexShuffleBlockResolver	9.10.4.1
ShuffleWriter	9.10.5
BypassMergeSortShuffleWriter	9.10.5.1
SortShuffleWriter	9.10.5.2
UnsafeShuffleWriter — ShuffleWriter for SerializedShuffleHandle	9.10.5.3
BaseShuffleHandle — Fallback Shuffle Handle	9.10.6
BypassMergeSortShuffleHandle — Marker Interface for Bypass Merge Sort Shuffle Handles	9.10.7
SerializedShuffleHandle — Marker Interface for Serialized Shuffle Handles	9.10.8
ShuffleReader	9.10.9
BlockStoreShuffleReader	9.10.9.1
ShuffleBlockFetcherIterator	9.10.10
ShuffleExternalSorter — Cache-Efficient Sorter	9.10.11
ExternalSorter	9.10.12
Serialization	9.11
Serializer — Task SerDe	9.11.1
SerializerInstance	9.11.2
SerializationStream	9.11.3
DeserializationStream	9.11.4
ExternalClusterManager — Pluggable Cluster Managers	9.12
BroadcastManager	9.13
BroadcastFactory — Pluggable Broadcast Variable Factories	9.13.1
TorrentBroadcastFactory	9.13.1.1
TorrentBroadcast	9.13.1.2
CompressionCodec	9.13.2
ContextCleaner — Spark Application Garbage Collector	9.14
CleanerListener	9.14.1
Dynamic Allocation (of Executors)	9.15

ExecutorAllocationManager — Allocation Manager for Spark Core	9.15.1
ExecutorAllocationClient	9.15.2
ExecutorAllocationListener	9.15.3
ExecutorAllocationManagerSource	9.15.4
HTTP File Server	9.16
Data Locality	9.17
Cache Manager	9.18
OutputCommitCoordinator	9.19
RpcEnv — RPC Environment	9.20
RpcEndpoint	9.20.1
RpcEndpointRef	9.20.2
RpcEnvFactory	9.20.3
Netty-based RpcEnv	9.20.4
TransportConf — Transport Configuration	9.21

(obsolete) Spark Streaming

Spark Streaming — Streaming RDDs	10.1
--	------

Spark Deployment Environments

Deployment Environments — Run Modes	11.1
Spark local (pseudo-cluster)	11.2
LocalSchedulerBackend	11.2.1
LocalEndpoint	11.2.2
Spark on cluster	11.3

Spark on YARN

Spark on YARN	12.1
YarnShuffleService — ExternalShuffleService on YARN	12.2
ExecutorRunnable	12.3
Client	12.4

YarnRMClient	12.5
ApplicationMaster	12.6
AMEndpoint — ApplicationMaster RPC Endpoint	12.6.1
YarnClusterManager — ExternalClusterManager for YARN	12.7
TaskSchedulers for YARN	12.8
YarnScheduler	12.8.1
YarnClusterScheduler	12.8.2
SchedulerBackends for YARN	12.9
YarnSchedulerBackend	12.9.1
YarnClientSchedulerBackend	12.9.2
YarnClusterSchedulerBackend	12.9.3
YarnSchedulerEndpoint RPC Endpoint	12.9.4
YarnAllocator	12.10
Introduction to Hadoop YARN	12.11
Setting up YARN Cluster	12.12
Kerberos	12.13
ConfigurableCredentialManager	12.13.1
ClientDistributedCacheManager	12.14
YarnSparkHadoopUtil	12.15
Settings	12.16

Spark Standalone

Spark Standalone	13.1
Standalone Master	13.2
Standalone Worker	13.3
web UI	13.4
Submission Gateways	13.5
Management Scripts for Standalone Master	13.6
Management Scripts for Standalone Workers	13.7
Checking Status	13.8
Example 2-workers-on-1-node Standalone Cluster (one executor per worker)	13.9
StandaloneSchedulerBackend	13.10

Spark on Mesos

Spark on Mesos	14.1
MesosCoarseGrainedSchedulerBackend	14.2
About Mesos	14.3

Execution Model

Execution Model	15.1
-----------------	------

Security

Spark Security	16.1
Securing Web UI	16.2

Spark Core / Data Sources

Data Sources in Spark	17.1
Using Input and Output (I/O)	17.2
Parquet	17.2.1
Spark and Cassandra	17.3
Spark and Kafka	17.4
Couchbase Spark Connector	17.5

(obsolete) Spark GraphX

Spark GraphX—Distributed Graph Computations	18.1
Graph Algorithms	18.2

Monitoring, Tuning and Debugging

Unified Memory Management	19.1
Spark History Server	19.2
HistoryServer	19.2.1

SQLHistoryListener	19.2.2
FsHistoryProvider	19.2.3
HistoryServerArguments	19.2.4
Logging	19.3
Performance Tuning	19.4
MetricsSystem	19.5
MetricsConfig — Metrics System Configuration	19.5.1
Metrics Source	19.5.2
Metrics Sink	19.5.3
SparkListener — Intercepting Events from Spark Scheduler	19.6
LiveListenerBus	19.6.1
ReplayListenerBus	19.6.2
SparkListenerBus — Internal Contract for Spark Event Buses	19.6.3
EventLoggingListener — Spark Listener for Persisting Events	19.6.4
StatsReportListener — Logging Summary Statistics	19.6.5
JsonProtocol	19.7
Debugging Spark using sbt	19.8

Varia

Building Apache Spark from Sources	20.1
Spark and Hadoop	20.2
SparkHadoopUtil	20.2.1
Spark and software in-memory file systems	20.3
Spark and The Others	20.4
Distributed Deep Learning on Spark	20.5
Spark Packages	20.6

Interactive Notebooks

Interactive Notebooks	21.1
Apache Zeppelin	21.1.1
Spark Notebook	21.1.2

Spark Tips and Tricks

Spark Tips and Tricks	22.1
Access private members in Scala in Spark shell	22.2
SparkException: Task not serializable	22.3
Running Spark Applications on Windows	22.4

Exercises

One-liners using PairRDDFunctions	23.1
Learning Jobs and Partitions Using take Action	23.2
Spark Standalone - Using ZooKeeper for High-Availability of Master	23.3
Spark's Hello World using Spark shell and Scala	23.4
WordCount using Spark shell	23.5
Your first complete Spark application (using Scala and sbt)	23.6
Spark (notable) use cases	23.7
Using Spark SQL to update data in Hive using ORC files	23.8
Developing Custom SparkListener to monitor DAGScheduler in Scala	23.9
Developing RPC Environment	23.10
Developing Custom RDD	23.11
Working with Datasets using JDBC (and PostgreSQL)	23.12
Causing Stage to Fail	23.13

Further Learning

Courses	24.1
Books	24.2

Spark Distributions

DataStax Enterprise	25.1
MapR Sandbox for Hadoop (Spark 1.5.2 only)	25.2

Spark Workshop

Spark Advanced Workshop	26.1
Requirements	26.1.1
Day 1	26.1.2
Day 2	26.1.3

Spark Talk Ideas

Spark Talks Ideas (STI)	27.1
10 Lesser-Known Tidbits about Spark Standalone	27.2
Learning Spark internals using groupBy (to cause shuffle)	27.3

Mastering Apache Spark 2

Welcome to Mastering Apache Spark 2 (aka #SparkLikePro)!

I'm [Jacek Laskowski](#), an **independent consultant** who is passionate about **Apache Spark**, Apache Kafka, Scala and sbt (with some flavour of Apache Mesos, Hadoop YARN, and quite recently DC/OS). I lead [Warsaw Scala Enthusiasts](#) and [Warsaw Spark](#) meetups in Warsaw, Poland.

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

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

Tip

I'm also writing [Apache Kafka Notebook](#), [Spark Structured Streaming Notebook](#) and [Spark Streaming Notebook](#).

Mastering Apache Spark 2 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 Apache 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 (as do you!).

The collection of notes serves as **the study material** for my 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](#) (and other publishers), press releases, conferences, [YouTube](#) or 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 (mostly) **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.

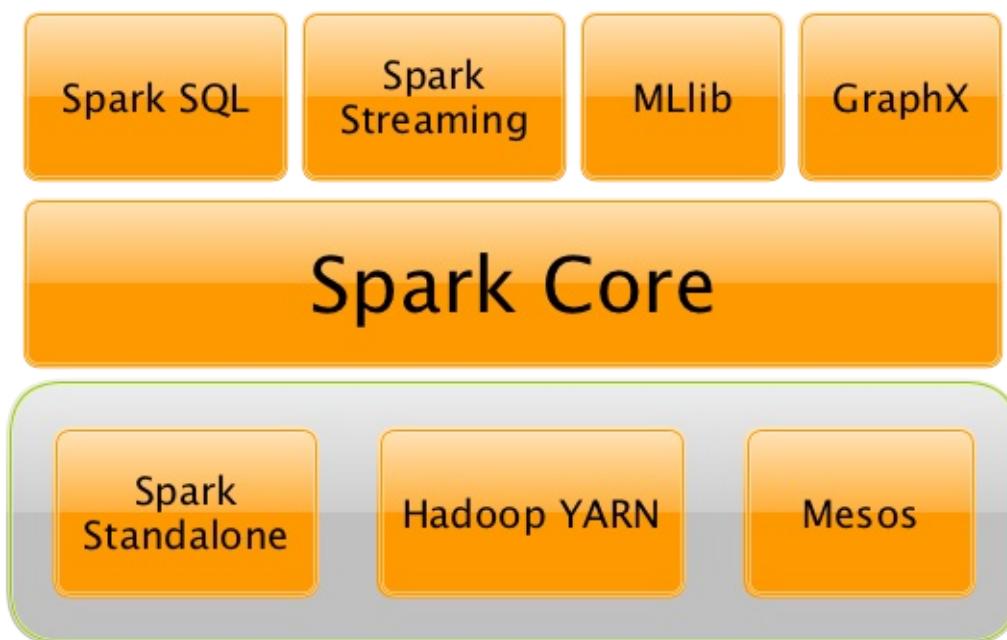


Figure 1. The Spark Platform

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 computation engine, Spark's multi-stage (mostly) in-memory computing engine allows for running most computations in memory, and hence most of the time provides better performance for certain applications, e.g. iterative algorithms or interactive data mining (read [Spark officially sets a new record in large-scale sorting](#)).

Spark aims at speed, ease of use, extensibility 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 provides developer API for languages like Java, Python, and R.

Note	<p>Microsoft's Mobius project provides C# API for Spark "<i>enabling the implementation of Spark driver program and data processing operations in the languages supported in the .NET framework like C# or F#.</i>"</p>
------	---

If you have large amounts of data that requires low latency processing that a typical MapReduce program cannot provide, Spark is a viable 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 Datasets), [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 Spark Core or the Apache Spark open source project which is an "umbrella" term for Spark Core and the accompanying Spark Application Frameworks, i.e. Spark SQL , Spark Streaming , Spark MLlib and Spark GraphX that sit on top of Spark Core and the main data abstraction in Spark called RDD - Resilient Distributed Dataset .
------	--

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 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](#)

Spark SQL — Batch and Streaming Queries Over Structured Data on Massive Scale

Like Apache Spark in general, **Spark SQL** in particular is all about distributed in-memory computations on massive scale.

The primary difference between Spark SQL's and the "bare" Spark Core's RDD computation models is the framework for loading, querying and persisting structured and semi-structured data using **structured queries** that can be expressed using *good ol' SQL*, **HiveQL** and the custom high-level SQL-like, declarative, type-safe [Dataset API](#) called **Structured Query DSL**.

Note	Semi- and structured datasets are collections of records that can be described using schema implicitly or explicitly, respectively.
------	---

Spark SQL supports structured queries in **batch** and **streaming** modes (with the latter as a separate module of Spark SQL called [Structured Streaming](#)).

Note	Under the covers, structured queries are automatically compiled into corresponding RDD operations.
------	--

Regardless of the query language you choose queries all end up as a [tree of Catalyst expressions](#) with [further optimizations](#) along the way to your large distributed data sets.

As of Spark 2.0, Spark SQL is now *de facto* the primary and feature-rich interface to Spark's underlying in-memory distributed platform (hiding Spark Core's RDDs behind higher-level abstractions).

```
// Define the schema using a case class
case class Person(name: String, age: Int)

// you could read people from a CSV file
// It's been a while since you saw RDDs, hasn't it?
// Excuse me for bringing you the old past.
import org.apache.spark.rdd.RDD
val peopleRDD: RDD[Person] = sc.parallelize(Seq(Person("Jacek", 10)))

// Convert RDD[Person] to Dataset[Person] and run a query

// Automatic schema inference from existing RDDs
scala> val people = peopleRDD.toDS
people: org.apache.spark.sql.Dataset[Person] = [name: string, age: int]

// Query for teenagers using Scala Query DSL
scala> val teenagers = people.where('age >= 10).where('age <= 19).select('name).as[String]
teenagers: org.apache.spark.sql.Dataset[String] = [name: string]

scala> teenagers.show
+---+
| name|
+---+
| Jacek|
+---+

// You could however want to use good ol' SQL, couldn't you?

// 1. Register people Dataset as a temporary view in Catalog
people.createOrReplaceTempView("people")

// 2. Run SQL query
val teenagers = sql("SELECT * FROM people WHERE age >= 10 AND age <= 19")
scala> teenagers.show
+---+---+
| name|age|
+---+---+
| Jacek| 10|
+---+---+
```

When the Hive support is enabled, Spark developers can read and write data located in existing Apache Hive deployments using HiveQL.

```

sql("CREATE OR REPLACE TEMPORARY VIEW v1 (key INT, value STRING) USING csv OPTIONS ('path'='people.csv', 'header'='true')")

// Queries are expressed in HiveQL
sql("FROM v1").show

scala> sql("desc EXTENDED v1").show(false)
+-----+-----+-----+
|col_name|data_type|comment|
+-----+-----+-----+
|# col_name|data_type|comment|
|key      |int       |null    |
|value    |string    |null    |
+-----+-----+-----+

```

Like SQL and NoSQL databases, Spark SQL offers performance query optimizations using [Logical Query Plan Optimizer](#), [code generation](#) (that could often be better than your own custom hand-written code!) and [Tungsten execution engine](#) with its own [Internal Binary Row Format](#).

Spark SQL introduces a tabular data abstraction called [Dataset](#) (that was previously [DataFrame](#)). `Dataset` data abstraction is designed to make processing large amount of structured tabular data on Spark infrastructure simpler and faster.

Note

Quoting [Apache Drill](#) which applies 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.

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("name", "score")
  .where($"score" > 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](#).

```

import org.apache.spark.sql.types._
val schema = StructType(
  StructField("id", LongType, nullable = false) :: 
  StructField("name", StringType, nullable = false) :: 
  StructField("score", DoubleType, nullable = false) :: Nil)

spark.readStream
  .format("json")
  .schema(schema)
  .load("input-json")
  .select("name", "score")
  .where('score > 15)
  .writeStream
  .format("console")
  .start

// -----
// Batch: 1
// -----
// +---+---+
// | name|score|
// +---+---+
// | Jacek| 20.5|
// +---+---+

```

As of Spark 2.0, the main data abstraction of Spark SQL is [Dataset](#). It represents a **structured data** which are 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 the following types of functions:

- [standard 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.
- [basic aggregate functions](#) that operate on a group of rows and calculate a single return value per group.
- [window aggregate functions](#) 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](#) property.

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)

You can parse data from external data sources and let the *schema inferencer* to deduct the schema.

```
// Example 1
val df = Seq(1 -> 2).toDF("i", "j")
val query = df.groupBy('i)
  .agg(max('j).as("aggOrdering"))
  .orderBy(sum('j))
  .as[(Int, Int)]
query.collect contains (1, 2) // true

// Example 2
val df = Seq((1, 1), (-1, 1)).toDF("key", "value")
df.createOrReplaceTempView("src")
scala> sql("SELECT IF(a > 0, a, 0) FROM (SELECT key a FROM src) temp").show
+-----+
|(IF((a > 0), a, 0))|
+-----+
|          1 |
|          0 |
+-----+
```

Further reading or watching

1. [Spark SQL home page](#)
2. (video) [Spark's Role in the Big Data Ecosystem - Matei Zaharia](#)
3. [Introducing Apache Spark 2.0](#)

SparkSession — The Entry Point to Spark SQL

`SparkSession` is the entry point to Spark SQL. It is the very first object you have to create while developing Spark SQL applications using the fully-typed `Dataset` (or untyped `Row`-based `DataFrame`) data abstractions.

Note	<code>SparkSession</code> has merged <code>SQLContext</code> and <code>HiveContext</code> in one object in Spark 2.0.
------	---

You use the `SparkSession.builder` method to create an instance of `SparkSession`.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = SparkSession.builder
  .appName("My Spark Application") // optional and will be autogenerated if not specified
  .master("local[*]")           // avoid hardcoding the deployment environment
  .enableHiveSupport()          // self-explanatory, isn't it?
  .config("spark.sql.warehouse.dir", "target/spark-warehouse")
  .getOrCreate
```

And stop the current `SparkSession` using `stop` method.

```
spark.stop
```

You can have as many `SparkSessions` as you want in a single Spark application. The common use case is to keep relational entities separate per `sparkSession` (see `catalog` attribute).

```
scala> spark.catalog.listTables.show
+-----+-----+-----+-----+
|       name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|my_permanent_table| default|      null|  MANAGED|     false|
|          strs|    null|      null|TEMPORARY|      true|
+-----+-----+-----+-----+
```

Internally, `sparkSession` requires a `SparkContext` and an optional `SharedState` (that represents the shared state across `sparkSession` instances).

Table 1. SparkSession's Class and Instance Methods

Method	Description
builder	"Opens" a builder to get or create a <code>SparkSession</code> instance
version	Returns the current version of Spark.
implicits	Use <code>import spark.implicits._</code> to import the implicits conversions and create <code>Datasets</code> from (almost arbitrary) Scala objects.
emptyDataset[T]	Creates an empty <code>Dataset[T]</code> .
range	Creates a <code>Dataset[Long]</code> .
sql	Executes a SQL query (and returns a <code>DataFrame</code>).
udf	Access to user-defined functions (UDFs).
table	Creates a <code>DataFrame</code> from a table.
catalog	Access to the catalog of the entities of structured queries
read	Access to <code>DataFrameReader</code> to read a <code>DataFrame</code> from external files and storage systems.
conf	Access to the current runtime configuration.
readStream	Access to <code>DataStreamReader</code> to read streaming datasets.
streams	Access to <code>StreamingQueryManager</code> to manage structured streaming queries.
newSession	Creates a new <code>SparkSession</code> .
stop	Stops the <code>SparkSession</code> .
Tip	<p>Use <code>spark.sql.warehouse.dir</code> Spark property to change the location of Hive's <code>hive.metastore.warehouse.dir</code> property, i.e. the location of the Hive local/embedded metastore database (using Derby).</p> <p>Refer to SharedState to learn about (the low-level details of) Spark SQL support for Apache Hive.</p> <p>See also the official Hive Metastore Administration document.</p>

Table 2. SparkSession's (Lazily-Initialized) Attributes (in alphabetical order)

Name	Type	Description
sessionState	SessionState	<p>Internally, <code>sessionState</code> clones the optional <code>parent SessionState</code> (if given when creating <code>SparkSession</code>) or creates a new <code>SessionState</code> using <code>BaseSessionStateBuilder</code> as defined by <code>spark.sql.catalogImplementation</code> property:</p> <ul style="list-style-type: none"> • in-memory (default) for <code>org.apache.spark.sql.internal.SessionStateBuilder</code> • hive for <code>org.apache.spark.sql.hive.HiveSessionStateBuilder</code>
sharedState	SharedState	

Note

`baseRelationToDataFrame` acts as a mechanism to plug `BaseRelation` object hierarchy in into `LogicalPlan` object hierarchy that `SparkSession` uses to bridge them.

Creating SparkSession Instance

Caution

FIXME

Creating SparkSession Using Builder Pattern — `builder` Method

```
builder(): Builder
```

`builder` creates a new `Builder` that you 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.

Accessing Version of Spark — `version` Method

```
version: String
```

`version` returns the version of Apache Spark in use.

Internally, `version` uses `spark.SPARK_VERSION` value that is the `version` property in `spark-version-info.properties` properties file on CLASSPATH.

Implicit Conversions — `implicits` object

The `implicits` object is a helper class with the Scala implicit methods (aka *conversions*) to convert Scala objects to [Datasets](#), [DataFrames](#) and [Columns](#). It also defines [Encoders](#) for Scala's "primitive" types, e.g. `Int`, `Double`, `String`, and their products and collections.

Note Import the `implicits` by `import spark.implicits._`.

```
val spark = SparkSession.builder.getOrCreate()
import spark.implicits._
```

`implicits` object offers support for creating `Dataset` from `RDD` of any type (for which an [encoder](#) exists in scope), or case classes or tuples, and `Seq`.

`implicits` object 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 only possible to call `toDF` methods on `RDD` objects of `Int`, `Long`, and `String` "primitive" types.

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)
```

`emptyDataset` creates a [LocalRelation](#) logical query plan.

Creating Dataset from Local Collections and RDDs

— `createDataset` methods

```
createDataset[T : Encoder](data: Seq[T]): Dataset[T]
createDataset[T : Encoder](data: RDD[T]): Dataset[T]
```

`createDataset` is an experimental API to create a [Dataset](#) from a local Scala collection, i.e. `Seq[T]`, Java's `List[T]`, or a distributed `RDD[T]`.

```
scala> val one = spark.createDataset(Seq(1))
one: org.apache.spark.sql.Dataset[Int] = [value: int]

scala> one.show
+---+
|value|
+---+
|   1|
+---+
```

`createDataset` creates a [LocalRelation](#) logical query plan (for the input `data` collection) or [LogicalRDD](#) (for the input `RDD[T]`).

You'd be better off using [Scala implicits](#) and `toDS` method instead (that does this conversion automatically for you).

Tip

```
val spark: SparkSession = ...
import spark.implicits._

scala> val one = Seq(1).toDS
one: org.apache.spark.sql.Dataset[Int] = [value: int]
```

Internally, `createDataset` first looks up the implicit [expression encoder](#) in scope to access the `AttributeReference`s (of the [schema](#)).

Note

Only unresolved [expression encoders](#) are currently supported.

The expression encoder is then used to map elements (of the input `seq[T]`) into a collection of [InternalRow](#)s. With the references and rows, `createDataset` returns a [Dataset](#) with a [LocalRelation](#) logical query plan.

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.

Creating Empty DataFrame — `emptyDataFrame` method

```
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)`.

Creating DataFrames from RDDs with Explicit Schema — `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`.

Executing SQL Queries (aka SQL Mode) — `sql` Method

```
sql(sqlText: String): DataFrame
```

`sql` executes the `sqlText` SQL statement and creates a [DataFrame](#).

Note

`sql` is imported in [spark-shell](#) so you can execute SQL statements as if `sql` were a part of the environment.

```
scala> spark.version
res0: String = 2.2.0-SNAPSHOT

scala> :imports
1) import spark.implicits._          (72 terms, 43 are implicit)
2) import spark.sql                 (1 terms)
```

```
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, `sql` requests the current [ParserInterface](#) to [execute a SQL query](#) that gives a [LogicalPlan](#).

Note

`sql` uses `SessionState` to access the current [ParserInterface](#).

`sql` then creates a [DataFrame](#) using the current [SparkSession](#) (itself) and the [LogicalPlan](#).

Tip

[spark-sql](#) is the main SQL environment in Spark to work with pure SQL statements (where you do not have to use Scala to execute them).

```
spark-sql> show databases;
default
Time taken: 0.028 seconds, Fetched 1 row(s)
```

Accessing UDF Registration Interface — `udf` Attribute

```
udf: UDFRegistration
```

`udf` attribute gives access to [UDFRegistration](#) that allows registering [user-defined functions](#) for SQL-based 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) UPPER FROM strs").show
+-----+-----+
|value|UPPER|
+-----+-----+
|    a|    A|
|    b|    B|
|    c|    C|
+-----+-----+
```

Internally, it is simply an alias for [SessionState.udfRegistration](#).

Creating DataFrame for Table — `table` method

```
table(tableName: String): DataFrame
```

`table` creates a [DataFrame](#) from records in the `tableName` table (if exists).

```
val df = spark.table("mytable")
```

Accessing Metastore — `catalog` Attribute

```
catalog: Catalog
```

`catalog` attribute is a (lazy) interface to the current metastore, i.e. [data catalog](#) (of relational entities like databases, tables, functions, table columns, and temporary views).

Tip	All methods in <code>Catalog</code> return <code>Datasets</code> .
-----	--

```
scala> spark.catalog.listTables.show
+-----+-----+-----+-----+
|       name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|my_permanent_table| default|      null|  MANAGED|    false|
|          strs|     null|      null|TEMPORARY|     true|
+-----+-----+-----+-----+
```

Internally, `catalog` creates a [CatalogImpl](#) (that uses the current `SparkSession`).

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](#)

`readStream` method

`readStream: DataStreamReader`

`readStream` returns a new [DataStreamReader](#).

`streams` Attribute

`streams: StreamingQueryManager`

`streams` attribute gives access to [StreamingQueryManager](#) (through `SessionState`).

```
val spark: SparkSession = ...
spark.streams.active.foreach(println)
```

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 their own extra strategies and optimization rules.

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
```

Stopping SparkSession — stop Method

`stop(): Unit`

`stop` stops the `SparkSession`, i.e. stops the underlying `SparkContext`.

Create DataFrame from BaseRelation — baseRelationToDataFrame Method

`baseRelationToDataFrame(baseRelation: BaseRelation): DataFrame`

Internally, `baseRelationToDataFrame` creates a `DataFrame` from the input `BaseRelation` wrapped inside `LogicalRelation`.

Note	<code>LogicalRelation</code> is an logical plan adapter for <code>BaseRelation</code> (so <code>BaseRelation</code> can be part of a <code>logical plan</code>).
------	---

Note	<code>baseRelationToDataFrame</code> is used when: <ul style="list-style-type: none"> • <code>DataFrameReader</code> loads data from a data source that supports multiple paths • <code>DataFrameReader</code> loads data from an external table using JDBC • <code>TextInputCSVDataSource</code> creates a base <code>Dataset</code> (of Strings) • <code>TextInputJsonDataSource</code> creates a base <code>Dataset</code> (of Strings)
------	--

Building SessionState — instantiateSessionState Internal Method

```
instantiateSessionState(className: String, sparkSession: SparkSession): SessionState
```

`instantiateSessionState` finds the `className` that is then used to `create` and immediately `build` a `BaseSessionStateBuilder`.

```
instantiateSessionState reports a IllegalArgumentException while constructing a  
SessionState :
```

```
Error while instantiating '[className]'
```

Note

`instantiateSessionState` is used exclusively when `SparkSession` is requested for `SessionState` (and one is not available yet).

Builder—Building SparkSession using Fluent API

`Builder` is the fluent API to build a fully-configured `SparkSession`.

Table 1. Builder Methods

Method	Description
<code>getOrCreate</code>	Gets the current <code>SparkSession</code> or creates a new one.
<code>enableHiveSupport</code>	Enables Hive support

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = SparkSession.builder
  .appName("My Spark Application") // optional and will be autogenerated if not specified
  .master("local[*]")           // avoid hardcoding the deployment environment
  .enableHiveSupport()          // self-explanatory, isn't it?
  .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 <code>SparkSession</code> s in a single Spark application for different data catalogs (through relational entities).
------	--

getOrCreate Method

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

config Method

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

Enabling Hive Support—enableHiveSupport Method

When [creating a `SparkSession`](#), you can optionally enable Hive support using `enableHiveSupport` method.

```
enableHiveSupport(): Builder
```

`enableHiveSupport` enables Hive support (with connectivity to a persistent Hive metastore, support for Hive serdes, and Hive user-defined functions).

Note	You do not need any existing Hive installation to use Spark's Hive support. <code>SparkSession</code> context will automatically create <code>metastore_db</code> in the current directory of a Spark application and a directory configured by spark.sql.warehouse.dir . Refer to SharedState .
------	---

Internally, `enableHiveSupport` makes sure that the Hive classes are on CLASSPATH, i.e. Spark SQL's `org.apache.hadoop.hive.conf.HiveConf`, and sets `spark.sql.catalogImplementation` property to `hive`.

SharedState — Shared State Across SparkSessions

`SharedState` is an internal class that holds the [shared state](#) across active [SparkSessions](#).

Table 1. SessionState's Attributes (Shared State)

Name	Type	Description
<code>cacheManager</code>	CacheManager	
<code>externalCatalog</code>	ExternalCatalog	
<code>globalTempViewManager</code>	GlobalTempViewManager	
<code>jarClassLoader</code>	NonClosableMutableURLClassLoader	
<code>listener</code>	SQLListener	
<code>sparkContext</code>	SparkContext	
<code>warehousePath</code>		

`SharedState` takes a [SparkContext](#) when created. It also adds `hive-site.xml` to [Hadoop's Configuration](#) in the current [SparkContext](#) if found on CLASSPATH.

Note	<code>hive-site.xml</code> is an optional Hive configuration file when working with Hive in Spark.
------	--

`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.

When created, `SharedState` sets `hive.metastore.warehouse.dir` to `spark.sql.warehouse.dir` if `hive.metastore.warehouse.dir` is not set or `spark.sql.warehouse.dir` is set. Otherwise, when `hive.metastore.warehouse.dir` is set and `spark.sql.warehouse.dir` is not, `spark.sql.warehouse.dir` gets set to `hive.metastore.warehouse.dir`.

You should see the following INFO message in the logs:

```
INFO spark.sql.warehouse.dir is not set, but hive.metastore.warehouse.dir is set. Setting spark.sql.warehouse.dir to the value of hive.metastore.warehouse.dir ('[hiveWarehouseDir]').
```

You should see the following INFO message in the logs:

```
INFO SharedState: Warehouse path is '[warehousePath]'.
```

Tip

Enable `INFO` logging level for `org.apache.spark.sql.internal.SharedState` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.sql.internal.SharedState=INFO
```

Refer to [Logging](#).

Dataset — Strongly-Typed Structured Query with Encoder

Dataset is a strongly-typed data structure in Spark SQL that represents a structured query with [encoders](#).

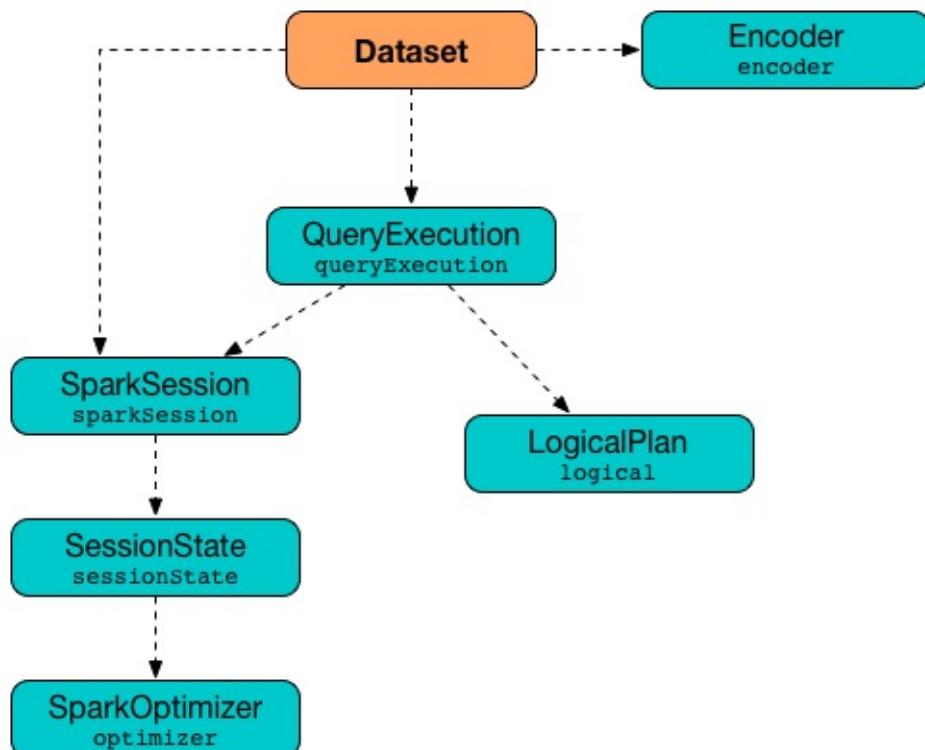


Figure 1. Dataset's Internals

Note

Given the picture above, one could say that a `Dataset` is a pair of an [Encoder](#) and [QueryExecution](#) (that in turn is a [LogicalPlan](#) in a [SparkSession](#))

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 (in a given [session](#)).

A Dataset is a result of executing a query expression against data storage like files, Hive tables or JDBC 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.

```

import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

scala> val dataset = spark.range(5)
dataset: org.apache.spark.sql.Dataset[Long] = [id: bigint]

// 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](#) (for fast serialization to and deserialization from [InternalRow](#)).

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	<p>SparkSession and QueryExecution are transient attributes of a <code>Dataset</code> and therefore do not participate in Dataset serialization. The only <i>firmly-tied</i> feature of a <code>Dataset</code> is the Encoder.</p>
------	--

You can [convert a type-safe Dataset to a "untyped" DataFrame](#) or access the [RDD](#) that is generated after executing the query. 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. You can however [persist a Dataset](#).

Note	<p>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.</p>
------	---

A `Dataset` is [local](#) if it was created from local collections using [SparkSession.emptyDataset](#) or [SparkSession.createDataset](#) methods and their derivatives like [toDF](#). If so, the queries on the Dataset can be optimized and run locally, i.e. without using Spark executors.

Note	<p><code>Dataset</code> makes sure that the underlying <code>queryExecution</code> is analyzed and checked.</p>
------	---

Table 1. Dataset's Properties

Name	Description				
boundEnc	ExpressionEncoder Used when...FIXME				
exprEnc	Implicit ExpressionEncoder Used when...FIXME				
logicalPlan	Logical plan				
rdd	<p>Content of the <code>Dataset</code> as a (lazily-created) <code>RDD</code> with rows of the JVM type T.</p> <p><code>rdd: RDD[T]</code></p> <p>Internally, <code>rdd</code> first creates a new logical plan to deserialize the Dataset's logical plan.</p> <p><code>rdd</code> then requests <code>SessionState</code> to execute the logical plan and gets the <code>RDD</code> of binary rows.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">Note</td> <td style="padding: 5px;"><code>rdd</code> uses <code>SparkSession</code> to access <code>SessionState</code>.</td> </tr> </table> </div> <p><code>rdd</code> then requests the Dataset's <code>ExpressionEncoder</code> for the <code>data type</code> of the rows (using <code>deserializer</code> expression) and maps over them (per partition) to create records of the expected type T.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">Note</td> <td style="padding: 5px;"><code>rdd</code> is at the "boundary" between the internal binary row format and the JVM type of the dataset.</td> </tr> </table> </div>	Note	<code>rdd</code> uses <code>SparkSession</code> to access <code>SessionState</code> .	Note	<code>rdd</code> is at the "boundary" between the internal binary row format and the JVM type of the dataset.
Note	<code>rdd</code> uses <code>SparkSession</code> to access <code>SessionState</code> .				
Note	<code>rdd</code> is at the "boundary" between the internal binary row format and the JVM type of the dataset.				
sqlContext	Lazily-created <code>SQLContext</code> Used when...FIXME				

Creating Dataset Instance

`Dataset` takes the following when created:

- `SparkSession`
- `QueryExecution`
- `Encoder` for the type T of the records

Note

You can also create a `Dataset` using `LogicalPlan` that is immediately `executed` using `SessionState`.

Internally, `Dataset` requests `QueryExecution` to `analyze` itself.

`Dataset` initializes the `internal registries and counters`.

Implicit Type Conversions to Datasets — `toDS` and `toDF` methods

`DatasetHolder` case class offers three methods that do the conversions from `Seq[T]` or `RDD[T]` types to a `Dataset[T]`:

- `toDS(): Dataset[T]`
- `toDF(): DataFrame`
- `toDF(colNames: String*): DataFrame`

Note

`DataFrame` is a *mere* type alias for `Dataset[Row]` since Spark 2.0.0.

`DatasetHolder` is used by `SQLImplicits` that is available to use after importing `implicits` object of `SparkSession`.

```
val spark: SparkSession = ...
import spark.implicits._

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 of `implicits` object's values is automatically executed in [Spark Shell](#) and so you don't need to do anything but use the conversions.

```
scala> spark.version
res11: String = 2.0.0

scala> :imports
1) import spark.implicits._  (59 terms, 38 are implicit)
2) import spark.sql        (1 terms)
```

```
val spark: SparkSession = ...
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
// https://issues.apache.org/jira/browse/SPARK-13456
val ds = data.toDS

// ds: org.apache.spark.sql.Dataset[Token] = [name: string, productId: int ... 1 more field]

// 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|
+-----+

```

Internals of toDS

Internally, the Scala compiler makes `toDS` implicitly available to any `Seq[T]` (using `SQLImplicits.localSeqToDatasetHolder` implicit method).

Note

This and other implicit methods are in scope whenever you do `import spark.implicits._`.

The input `Seq[T]` is converted into `Dataset[T]` by means of `SQLContext.createDataset` that in turn passes all calls on to `SparkSession.createDataset`. Once created, the `Dataset[T]` is wrapped in `DatasetHolder[T]` with `toDS` that just returns the input `ds`.

Queryable

Caution

FIXME

Tracking Multi-Job SQL Query Executions — `withNewExecutionId` Internal Method

```
withNewExecutionId[U](body: => U): U
```

`withNewExecutionId` is a `private[sql]` operator that executes the input `body` action using `SQLExecution.withNewExecutionId` that sets the **execution id** local property set.

Note

`withNewExecutionId` is used in `foreach`, `foreachPartition`, and (private) `collect`.

Creating DataFrame— `ofRows` Internal Method

```
ofRows(sparkSession: SparkSession, logicalPlan: LogicalPlan): DataFrame
```

Note

`ofRows` is a `private[sql]` operator that can only be accessed from code in `org.apache.spark.sql` package. It is not a part of `Dataset`'s public API.

`ofRows` returns `DataFrame` (which is the type alias for `Dataset[Row]`). `ofRows` uses `RowEncoder` to convert the schema (based on the input `logicalPlan` logical plan).

Internally, `ofRows` prepares the input `logicalPlan` for execution and creates a `Dataset[Row]` with the current `SparkSession`, the `QueryExecution` and `RowEncoder`.

Further reading or watching

- (video) Structuring Spark: DataFrames, Datasets, and Streaming

Encoders — Internal Row Converters

Encoder is the fundamental concept in the **serialization and deserialization (SerDe) framework** in Spark SQL 2.0. Spark SQL uses the SerDe framework for IO to make it efficient time- and space-wise.

Tip

Spark has borrowed the idea from the [Hive SerDe library](#) so it might be worthwhile to get familiar with Hive a little bit, too.

Encoders are modelled in Spark SQL 2.0 as `Encoder[T]` trait.

```
trait Encoder[T] extends Serializable {
  def schema: StructType
  def clsTag: ClassTag[T]
}
```

The type `T` stands for the type of records a `Encoder[T]` can deal with. An encoder of type `T`, i.e. `Encoder[T]`, is used to convert (*encode* and *decode*) any JVM object or primitive of type `T` (that could be your domain object) to and from Spark SQL's [InternalRow](#) which is the internal binary row format representation (using Catalyst expressions and code generation).

Note

`Encoder` is also called "*a container of serde expressions in Dataset*".

Note

The one and only implementation of the `Encoder` trait in Spark SQL 2 is [ExpressionEncoder](#).

Encoders are integral (and internal) part of any [Dataset\[T\]](#) (of records of type `T`) with a `Encoder[T]` that is used to serialize and deserialize the records of this dataset.

Note

`Dataset[T]` type is a Scala type constructor with the type parameter `T`. So is `Encoder[T]` that handles serialization and deserialization of `T` to the internal representation.

Encoders know the `schema` of the records. This is how they offer significantly faster serialization and deserialization (comparing to the default Java or Kryo serializers).

```
// The domain object for your records in a large dataset
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@1]: st
```

```

ring]

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]

// ExpressionEncoders may or may not be flat
scala> personExprEncoder.flat
res2: Boolean = false

// The Serializer part of the encoder
scala> personExprEncoder.serializer
res3: Seq[org.apache.spark.sql.catalyst.expressions.Expression] = List(assertnonnull(i
nput[0, Person, true], top level non-flat input object).id AS id#0L, staticinvoke(class
org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnonnull(inpu
t[0, Person, true], top level non-flat input object).name, true) AS name#1)

// The Deserializer part of the encoder
scala> personExprEncoder.deserializer
res4: org.apache.spark.sql.catalyst.expressions.Expression = newInstance(class Person)

scala> personExprEncoder.namedExpressions
res5: Seq[org.apache.spark.sql.catalyst.expressions.NamedExpression] = List(assertnonnull(i
nput[0, Person, true], top level non-flat input object).id AS id#2L, staticinvoke(
class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnonnull(
input[0, Person, true], top level non-flat input object).name, true) AS name#3)

// A record in a Dataset[Person]
// A mere instance of Person case class
// There could be a thousand of Person in a large dataset
val jacek = Person(0, "Jacek")

// Serialize a record to the internal representation, i.e. InternalRow
scala> val row = personExprEncoder.toRow(jacek)
row: org.apache.spark.sql.catalyst.InternalRow = [0,0,1800000005,6b6563614a]

// Spark uses InternalRows internally for IO
// Let's deserialize it to a JVM object, i.e. a Scala object
import org.apache.spark.sql.catalyst.dsl.expressions._

// in spark-shell there are competing implicits
// That's why DslSymbol is used explicitly in the following line
scala> val attrs = Seq(DslSymbol('id).long, DslSymbol('name).string)
attrs: Seq[org.apache.spark.sql.catalyst.expressions.AttributeReference] = List(id#8L,

```

```

name#9)

scala> val jacekReborn = personExprEncoder.resolveAndBind(attrs).fromRow(row)
jacekReborn: Person = Person(0, Jacek)

// Are the jacek instances same?
scala> jacek == jacekReborn
res6: Boolean = true

```

You can [create custom encoders using static methods of `Encoders` object](#). Note however that encoders for common Scala types and their product types are already available in `implicits` object.

```

val spark = SparkSession.builder.getOrCreate()
import spark.implicits._

```

Tip	The default encoders are already imported in spark-shell .
-----	--

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.

Note	In Spark SQL 2.0 <code>DataFrame</code> type is a mere type alias for <code>Dataset[Row]</code> with RowEncoder being 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]
```

Further reading or watching

- (video) [Modern Spark DataFrame and Dataset \(Intermediate Tutorial\)](#) by Adam Breindel from Databricks.

ExpressionEncoder — Expression-Based Encoder

`ExpressionEncoder[T]` is a generic [Encoder](#) for JVM objects of the type `T` that uses [Catalyst expressions](#) to [serialize](#) to and [deserialize](#) from the [internal binary row format](#) (i.e. `InternalRow`).

Note	<code>ExpressionEncoder</code> is the only supported implementation of <code>Encoder</code> which is explicitly enforced when <code>Dataset</code> is created (even though <code>Dataset</code> accepts a bare <code>Encoder[T]</code>).
------	---

```
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
val stringEncoder = ExpressionEncoder[String]
scala> val row = stringEncoder.toRow("hello world")
row: org.apache.spark.sql.catalyst.InternalRow = [0,100000000b,6f77206f6c6c6568,646c72
]

import org.apache.spark.sql.catalyst.expressions.UnsafeRow
scala> val unsafeRow = row match { case ur: UnsafeRow => ur }
unsafeRow: org.apache.spark.sql.catalyst.expressions.UnsafeRow = [0,100000000b,6f77206
f6c6c6568,646c72]
```

`ExpressionEncoder` uses [serializer expressions](#) to encode (aka [serialize](#)) a JVM object of type `T` to an [internal binary row format](#) (i.e. `InternalRow`).

Note	It is assumed that all serializer expressions contain at least one and the same <code>BoundReference</code> .
------	---

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

`ExpressionEncoder` uses a [deserializer expression](#) to decode (aka [deserialize](#)) a JVM object of type `T` from [internal binary row format](#).

`ExpressionEncoder` is [flat](#) when there is only one expression for the [serializer](#).

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.

Table 1. ExpressionEncoder's (Lazily-Initialized) Internal Properties

Property	Description
<code>constructProjection</code>	<code>Projection</code> generated for the <code>deserializer</code> expression Used exclusively when <code>ExpressionEncoder</code> is requested for a <code>JVM object from a Spark SQL row</code> (i.e. <code>InternalRow</code>).
<code>extractProjection</code>	<code>UnsafeProjection</code> generated for the <code>serializer</code> expressions Used exclusively when <code>ExpressionEncoder</code> is requested for an <code>encoded version of a JVM object as a Spark SQL row</code> (i.e. <code>InternalRow</code>).
<code>inputRow</code>	<code>GenericInternalRow</code> (with the underlying storage array) of size 1 (i.e. it can only store a single JVM object of any type). Used... <code>FIXME</code>
Note	<code>Encoders</code> object contains the default <code>ExpressionEncoders</code> for Scala and Java primitive types, e.g. <code>boolean</code> , <code>long</code> , <code>String</code> , <code>java.sql.Date</code> , <code>java.sql.Timestamp</code> , <code>Array[Byte]</code> .

resolveAndBind Method

Caution

`FIXME`

Creating ExpressionEncoder Instance

`ExpressionEncoder` takes the following when created:

- Schema
- Flag whether `ExpressionEncoder` is flat or not
- Serializer `expressions`
- Deserializer `expression`
- Scala's `ClassTag` for the JVM type `T`

Creating Deserialize Expression

— `ScalaReflection.deserializeFor` Method

`deserializeFor[T: TypeTag]: Expression`

`deserializerFor` creates an expression to deserialize from internal binary row format to a Scala object of type `T`.

```
import org.apache.spark.sql.catalystScalaReflection.deserializerFor
val timestampDeExpr = deserializerFor[java.sql.Timestamp]
scala> println(timestampDeExpr.numberedTreeString)
00 staticinvoke(class org.apache.spark.sql.catalyst.util.DateTimeUtils$, ObjectType(clazz java.sql.Timestamp), toJavaTimestamp, upcast(getcolumnbyordinal(0, TimestampType), TimestampType, - root class: "java.sql.Timestamp"), true)
01 +- upcast(getcolumnbyordinal(0, TimestampType), TimestampType, - root class: "java.sql.Timestamp")
02   +- getcolumnbyordinal(0, TimestampType)

val tuple2DeExpr = deserializerFor[(java.sql.Timestamp, Double)]
scala> println(tuple2DeExpr.numberedTreeString)
00 newInstance(class scala.Tuple2)
01 :- staticinvoke(class org.apache.spark.sql.catalyst.util.DateTimeUtils$, ObjectType(class java.sql.Timestamp), toJavaTimestamp, upcast(getcolumnbyordinal(0, TimestampType), TimestampType, - field (class: "java.sql.Timestamp", name: "_1"), - root class: "scala.Tuple2"), true)
02 : +- upcast(getcolumnbyordinal(0, TimestampType), TimestampType, - field (class: "java.sql.Timestamp", name: "_1"), - root class: "scala.Tuple2")
03 :   +- getcolumnbyordinal(0, TimestampType)
04 +- upcast(getcolumnbyordinal(1, DoubleType), DoubleType, - field (class: "scala.Double", name: "_2"), - root class: "scala.Tuple2")
05   +- getcolumnbyordinal(1, DoubleType)
```

Internally, `deserializerFor` calls the recursive internal variant of `deserializerFor` with a single-element walked type path with `- root class: "[clsName]"`

Tip

Read up on Scala's `TypeTags` in [TypeTags and Manifests](#).

Note

`deserializerFor` is used exclusively when `ExpressionEncoder` is created for a Scala type `T`.

Recursive Internal `deserializerFor` Method

```
deserializerFor(
  tpe: `Type`,
  path: Option[Expression],
  walkedTypePath: Seq[String]): Expression
```

Table 2. JVM Types and Deserialize Expressions (in evaluation order)

JVM Type (Scala or Java)	Deserialize Expressions
<code>Option[T]</code>	
<code>java.lang.Integer</code>	
<code>java.lang.Long</code>	
<code>java.lang.Double</code>	
<code>java.lang.Float</code>	
<code>java.lang.Short</code>	
<code>java.lang.Byte</code>	
<code>java.lang.Boolean</code>	
<code>java.sql.Date</code>	
<code>java.sql.Timestamp</code>	
<code>java.lang.String</code>	
<code>java.math.BigDecimal</code>	
<code>scala.BigDecimal</code>	
<code>java.math.BigInteger</code>	
<code>scala.math.BigInt</code>	
<code>Array[T]</code>	
<code>Seq[T]</code>	
<code>Map[K, V]</code>	
<code>SQLUserDefinedType</code>	
User Defined Types (UDTs)	
Product (including <code>Tuple</code>) or DefinedByConstructorParams	

Creating Serialize Expression — `ScalaReflection.serializerFor` Method

```
serializerFor[T: TypeTag](inputObject: Expression): CreateNamedStruct
```

`serializerFor` creates a `CreateNamedStruct` [expression](#) to serialize a Scala object of type `T` to [internal binary row format](#).

```
import org.apache.spark.sql.catalystScalaReflection.serializerFor

import org.apache.spark.sql.catalyst.expressions.BoundReference
import org.apache.spark.sql.types.TimestampType
val boundRef = BoundReference(ordinal = 0, dataType = TimestampType, nullable = true)

val timestampSerExpr = serializerFor[java.sql.Timestamp](boundRef)
scala> println(timestampSerExpr.numberedTreeString)
00 named_struct(value, input[0, timestamp, true])
01 :- value
02 +- input[0, timestamp, true]
```

Internally, `serializerFor` calls the recursive internal variant of `serializerFor` with a single-element walked type path with `- root class: "[clsName]"` and *pattern match* on the result [expression](#).

Caution	FIXME the pattern match part
---------	--

Tip	Read up on Scala's <code>TypeTags</code> in TypeTags and Manifests .
-----	--

Note	<code>serializerFor</code> is used exclusively when <code>ExpressionEncoder</code> is created for a Scala type <code>T</code> .
------	---

Recursive Internal `serializerFor` Method

```
serializerFor(
  inputObject: Expression,
  tpe: `Type`,
  walkedTypePath: Seq[String],
  seenTypeSet: Set[`Type`] = Set.empty): Expression
```

`serializerFor` creates an [expression](#) for serializing an object of type `T` to an internal row.

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

Encoding JVM Object to Internal Binary Row Format — `toRow` Method

```
toRow(t: T): InternalRow
```

`toRow` encodes (aka *serializes*) a JVM object `t` as a [Spark SQL row](#).

Internally, `toRow` sets the only JVM object to be `t` in `inputRow` and converts the `inputRow` to a [unsafe binary row](#) (using `extractProjection`).

In case of any exception while serializing, `toRow` reports a `RuntimeException`:

```
Error while encoding: [initial exception]
[multi-line serializer]
```

Note

`toRow` is *mostly* used when `SparkSession` is requested for:

- [Dataset from a local dataset](#)
- [DataFrame from RDD\[Row\]](#)

Decoding JVM Object From Internal Binary Row Format — `fromRow` Method

```
fromRow(row: InternalRow): T
```

`fromRow` decodes (aka *deserializes*) a JVM object from a `row` [InternalRow](#) (with the required values only).

Internally, `fromRow` uses `constructProjection` with `row` and gets the 0th element of type `ObjectType` that is then cast to the output type `T`.

In case of any exception while deserializing, `fromRow` reports a `RuntimeException`:

```
Error while decoding: [initial exception]
[deserializer]
```

Note

`fromRow` is used for:

- `Dataset` operators, i.e. `head`, `collect`, `collectAsList`, `toLocalIterator`
- Structured Streaming's `ForeachSink`

DataFrame — Dataset of Rows

Spark SQL introduces a tabular data abstraction called `DataFrame`. It is designed to ease processing large amount of structured tabular data on Spark infrastructure.

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` that is a result of a structured query it describes.

It uses the immutable, in-memory, resilient, distributed and parallel capabilities of `RDD`, and applies a structure called schema to the data.

Note

In Spark 2.0.0 `DataFrame` is a *mere* type alias for `Dataset[Row]`.

```
type DataFrame = Dataset[Row]
```

See [org.apache.spark.package.scala](#).

`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).

Note

Hey, big data consultants, time to help teams migrate the code from pandas' DataFrame into Spark's DataFrames (at least to PySpark's DataFrame) and offer services to set up large clusters!

DataFrames in Spark SQL strongly rely on [the features of RDD](#) - it's basically a `RDD` exposed as 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]
```

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 |
+-----+----+
| dennisisthemenace1 | 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|
+---+-----+---+

```

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")
```

Example Datasets

- [eBay online auctions](#)
- [SFPD Crime Incident Reporting system](#)

Row

`Row` is a generic row object with 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](#).

Note

`Row` is also called **Catalyst Row**.

`Row` may have an optional [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
```

Creating Row — `apply` Factory Method

Caution

FIXME

Field Access by Index — `apply` and `get` methods

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 `apply` or `get`) returns a value of type `Any`.

Get Field As Type — `getAs` method

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
}
```

RowEncoder — Encoder for DataFrames

`RowEncoder` is a part of the [Encoder framework](#) and acts as the encoder for [DataFrames](#), i.e. `Dataset[Row]` — [Datasets of Rows](#).

Note

`DataFrame` type is a mere type alias for `Dataset[Row]` that expects a `Encoder[Row]` available in scope which is indeed `RowEncoder` itself.

`RowEncoder` is an `object` in Scala with [apply](#) and other factory methods.

`RowEncoder` can create `ExpressionEncoder[Row]` from a [schema](#) (using [apply method](#)).

```
import org.apache.spark.sql.types._
val schema = StructType(
  StructField("id", LongType, nullable = false) ::
  StructField("name", StringType, nullable = false) :: Nil)

import org.apache.spark.sql.catalyst.encoders.RowEncoder
scala> val encoder = RowEncoder(schema)
encoder: org.apache.spark.sql.catalyst.encoders.ExpressionEncoder[org.apache.spark.sql.
Row] = class[id[0]: bigint, name[0]: string]

// RowEncoder is never flat
scala> encoder.flat
res0: Boolean = false
```

`RowEncoder` object belongs to `org.apache.spark.sql.catalyst.encoders` package.

Creating ExpressionEncoder of Rows — `apply` method

```
apply(schema: StructType): ExpressionEncoder[Row]
```

`apply` builds [ExpressionEncoder](#) of [Row](#), i.e. `ExpressionEncoder[Row]`, from the input `StructType` (as `schema`).

Internally, `apply` creates a `BoundReference` for the `Row` type and returns a `ExpressionEncoder[Row]` for the input `schema`, a `CreateNamedStruct` serializer (using `serializerFor` internal method), a deserializer for the schema, and the `Row` type.

serializerFor Internal Method

```
serializerFor(inputObject: Expression, inputType: DataType): Expression
```

`serializerFor` creates an `Expression` that is assumed to be `CreateNamedStruct`.

`serializerFor` takes the input `inputType` and:

1. Returns the input `inputObject` as is for native types, i.e. `NullType`, `BooleanType`, `ByteType`, `ShortType`, `IntegerType`, `LongType`, `FloatType`, `DoubleType`, `BinaryType`, `CalendarIntervalType`.

Caution	FIXME What does being native type mean?
---------	---

2. For `UserDefinedType`s, it takes the UDT class from the `SQLUserDefinedType` annotation or `UDTRegistration` object and returns an expression with `Invoke` to call `serialize` method on a `NewInstance` of the UDT class.
3. For `TimestampType`, it returns an expression with a `StaticInvoke` to call `fromJavaTimestamp` on `DateTimeUtils` class.
4. ...[FIXME](#)

Caution	FIXME Describe me.
---------	------------------------------------

Schema — Structure of Data

A **schema** is the description of the structure of your data (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 (that in turn are tuples of names, types, and `nullability` classifier).

`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 [CatalystSqlParser](#) that is responsible for parsing data types.

It is however recommended to use the singleton [DataTypes](#) class with static methods to create schema types.

```
import org.apache.spark.sql.types.DataTypes._
val schemaWithMap = StructType(
  StructField("map", createMapType(LongType, StringType), false) :: Nil)
```

`StructType` offers [printTreeString](#) that makes presenting the schema more user-friendly.

```

scala> schemaTyped.printTreeString
root
|-- a: integer (nullable = true)
|-- b: string (nullable = true)

scala> schemaWithMap.printTreeString
root
|-- map: map (nullable = false)
|   |-- key: long
|   |-- value: string (valueContainsNull = true)

// You can use prettyJson method on any DataType
scala> println(schema1.prettyJson)
{
  "type" : "struct",
  "fields" : [ {
    "name" : "a",
    "type" : "integer",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "b",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  } ]
}

```

As of Spark 2.0, you can describe the schema of your strongly-typed datasets using [encoders](#).

```

import org.apache.spark.sql.Encoders

scala> Encoders.INT.schema.printTreeString
root
|-- value: integer (nullable = true)

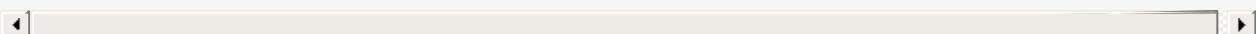
scala> Encoders.product[(String, java.sql.Timestamp)].schema.printTreeString
root
|-- _1: string (nullable = true)
|-- _2: timestamp (nullable = true)

case class Person(id: Long, name: String)
scala> Encoders.product[Person].schema.printTreeString
root
|-- id: long (nullable = false)
|-- name: string (nullable = true)

```

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
```



StructType — Data Type for Schema Definition

`StructType` is a built-in [data type](#) in Spark SQL to represent a collection of [StructField](#)s that together define a schema or its part.

Note

`structType` is a `Seq[StructField]` and therefore all things `Seq` apply equally here.

```
scala> schemaTyped.foreach(println)
StructField(a, IntegerType, true)
StructField(b, StringType, true)
```

Read the official documentation of [scala.collection.Seq](#).

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

```

Data Type 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
```

Displaying Schema As Tree — `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)
```

Data Types

`DataType` abstract class is the base type of all built-in data types in Spark SQL, e.g. strings, longs.

Table 1. Standard Data Types

Type Family	Data Type	Scala Types
Atomic Types <small>(except fractional and integral types)</small>	BinaryType	
	BooleanType	
	DateType	
	StringType	
	TimestampType	java.sql.Timestamp
Fractional Types	DecimalType	
	DoubleType	
	FloatType	
Integral Types	ByteType	
	IntegerType	
	LongType	
	ShortType	
	ArrayType	
	CalendarIntervalType	
	MapType	
	NullType	
	ObjectType	
	StructType	
	UserDefinedType	
	AnyDataType	Matches any concrete data type

Caution

[**FIXME**](#) What about AbstractDataType?

You can extend the type system and create your own [user-defined types \(UDTs\)](#).

The [DataType Contract](#) defines methods to build SQL, JSON and string representations.

Note

`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

```
import org.apache.spark.sql.types.DataTypes._

val maps = StructType(
  StructField("longs2strings", createMapType(LongType, StringType), false) :: Nil)

scala> maps.prettyJson
res0: String =
{
  "type" : "struct",
  "fields" : [ {
    "name" : "longs2strings",
    "type" : {
      "type" : "map",
      "keyType" : "long",
      "valueType" : "string",
      "valueContainsNull" : true
    },
    "nullable" : false,
    "metadata" : { }
  } ]
}

scala> maps.defaultSize
res1: Int = 2800

scala> maps.simpleString
res2: String = struct<longs2strings:map<bigint,string>>

scala> maps.catalogString
res3: String = struct<longs2strings:map<bigint,string>>

scala> maps.sql
res4: String = STRUCT<`longs2strings`: MAP<BIGINT, STRING>>
```

DataTypes — Factory Methods for Data Types

`DataTypes` is a Java class with methods to access simple or create complex `DataType` types in Spark SQL, i.e. arrays and maps.

Tip

It is recommended to use `DataTypes` class to define `DataType` types in a schema.

`DataTypes` 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)
```

Note

Simple `DataType` types themselves, i.e. `StringType` or `CalendarIntervalType`, come with their own Scala's `case object`'s alongside their definitions.

You may also import the `types` package and have access to the types.

```
import org.apache.spark.sql.types._
```

UDTs — User-Defined Types

Caution

FIXME

Dataset Operators

You can group the set of all operators to use with `Datasets` per their target, i.e. the part of a `Dataset` they are applied to.

1. Column Operators
2. Standard Functions (from `functions` object)
3. User-Defined Functions (UDFs)
4. Basic Aggregation — Typed and Untyped Grouping Operators
5. Window Aggregate Functions
6. User-Defined Aggregate Functions (UDAFs)
7. Joins
8. Caching

Beside the above operators, there are the following ones working with a `Dataset` as a whole.

Table 1. Dataset Operators

Operator	Description
<code>as</code>	Converting a <code>Dataset</code> to a <code>Dataset</code>
<code>coalesce</code>	Repartitioning a <code>Dataset</code> with shuffle disabled.
<code>count</code>	Counts the number of rows
<code>createGlobalTempView</code>	
<code>createOrReplaceTempView</code>	
<code>createTempView</code>	
<code>explain</code>	Explain logical and physical plans of a <code>Dataset</code>
<code>filter</code>	
<code>flatMap</code>	

<code>foreach</code>	Internally, <code>foreach</code> executes <code>foreach</code> action on the Dataset's RDD .
<code>foreachPartition</code>	Internally, <code>foreachPartition</code> executes <code>foreachPartition</code> action on the Dataset's RDD .
<code>isLocal</code>	
<code>isStreaming</code>	
<code>mapPartition</code>	
<code>randomSplit</code>	Randomly split a <code>Dataset</code> into two <code>Dataset</code> s
<code>rdd</code>	
<code>reduce</code>	Reduces the elements of a <code>Dataset</code> using the specified binary function. Internally, <code>reduce</code> executes <code>reduce</code> action on the Dataset's RDD .
<code>repartition</code>	Repartitioning a <code>Dataset</code> with shuffle enabled.
<code>schema</code>	
<code>select</code>	
<code>selectExpr</code>	
<code>show</code>	
<code>take</code>	
<code>toDF</code>	Converts a <code>Dataset</code> to a <code>DataFrame</code>
<code>toJSON</code>	
<code>transform</code>	Transforms a <code>Dataset</code>
<code>where</code>	
<code>write</code>	
<code>writeStream</code>	

count Operator

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

toLocalIterator Operator

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

createTempViewCommand Internal Operator

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

createGlobalTempView Operator

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

createOrReplaceTempView Operator

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

createTempView Operator

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

Transforming Datasets — transform Operator

```
transform[U](t: Dataset[T] => Dataset[U]): Dataset[U]
```

`transform` applies `t` function to the source `Dataset[T]` to produce a result `Dataset[U]`. It is for chaining custom transformations.

```

val dataset = spark.range(5)

// Transformation t
import org.apache.spark.sql.Dataset
def withDoubled(longs: Dataset[java.lang.Long]) = longs.withColumn("doubled", 'id * 2)

scala> dataset.transform(withDoubled).show
+---+-----+
| id|doubled|
+---+-----+
|  0|      0|
|  1|      2|
|  2|      4|
|  3|      6|
|  4|      8|
+---+-----+

```

Internally, `transform` executes `t` function on the current `Dataset[T]`.

Converting "Typed" Dataset to "Untyped" DataFrame — `toDF` Methods

```

toDF(): DataFrame
toDF(colNames: String*): DataFrame

```

`toDF` converts a `Dataset` into a `DataFrame`.

Internally, the empty-argument `toDF` creates a `Dataset[Row]` using the `Dataset`'s `SparkSession` and `QueryExecution` with the encoder being `RowEncoder`.

Caution

[FIXME](#) Describe `toDF(colNames: String*)`

Converting to Dataset — `as` Method

Caution

[FIXME](#)

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

```
show(): Unit
show(numRows: Int): Unit
show(truncate: Boolean): Unit
show(numRows: Int, truncate: Boolean): Unit
show(numRows: Int, truncate: Int): Unit
```

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

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 Action

```
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](#).

foreachPartition Action

```
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	<code>foreachPartition</code> is used to save a DataFrame to a JDBC table (indirectly through <code>JdbcUtils.saveTable</code>) and ForeachSink .
------	--

mapPartitions Operator

```
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 Operator

```
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	<code>flatMap</code> can create new records. It deprecated <code>explode</code> .
------	---

```

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)` .

Repartitioning Dataset with Shuffle Disabled — `coalesce` Operator

```
coalesce(numPartitions: Int): Dataset[T]
```

`coalesce` operator repartitions the `dataset` to exactly `numPartitions` partitions.

Internally, `coalesce` creates a `Repartition` logical operator with `shuffle` disabled (which is marked as `false` in the below `explain`'s output).

```

scala> spark.range(5).coalesce(1).explain(extended = true)
== Parsed Logical Plan ==
Repartition 1, false
+- Range (0, 5, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
Repartition 1, false
+- Range (0, 5, step=1, splits=Some(8))

== Optimized Logical Plan ==
Repartition 1, false
+- Range (0, 5, step=1, splits=Some(8))

== Physical Plan ==
Coalesce 1
+- *Range (0, 5, step=1, splits=Some(8))

```

Repartitioning Dataset (Shuffle Enabled) — `repartition` Operator

```
repartition(numPartitions: Int): Dataset[T]
repartition(numPartitions: Int, partitionExprs: Column*): Dataset[T]
repartition(partitionExprs: Column*): Dataset[T]
```

`repartition` operators repartition the `dataset` to exactly `numPartitions` partitions or using `partitionExprs` expressions.

Internally, `repartition` creates a [Repartition](#) or [RepartitionByExpression](#) logical operators with `shuffle` enabled (which is `true` in the below `explain`'s output beside `Repartition`).

```
scala> spark.range(5).repartition(1).explain(extended = true)
== Parsed Logical Plan ==
Repartition 1, true
+- Range (0, 5, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
Repartition 1, true
+- Range (0, 5, step=1, splits=Some(8))

== Optimized Logical Plan ==
Repartition 1, true
+- Range (0, 5, step=1, splits=Some(8))

== Physical Plan ==
Exchange RoundRobinPartitioning(1)
+- *Range (0, 5, step=1, splits=Some(8))
```

Note

`repartition` methods correspond to SQL's [DISTRIBUTE BY](#) or [CLUSTER BY](#) clauses.

Projecting Columns — `select` Operator

```

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**FIXME**

filter Operator

Caution**FIXME**

where Operator

```

where(condition: Column): Dataset[T]
where(conditionExpr: String): Dataset[T]

```

`where` is a synonym for `filter` operator, i.e. it simply passes the parameters on to `filter`.

Projecting Columns using Expressions — `selectExpr` Operator

```
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.
------	-------------------------------

Randomly Split Dataset— `randomSplit` Operator

```
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 TrainValidationSplit 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 2.0.0.
------	-------------------------------

Displaying Logical and Physical Plans, Their Cost and Codegen — `explain` Operator

```

explain(): Unit
explain(extended: Boolean): Unit

```

`explain` prints the [logical](#) and (with `extended` flag enabled) [physical](#) plans, their cost and codegen to the console.

Tip	Use <code>explain</code> to review the structured queries and optimizations applied.
-----	--

Internally, `explain` creates a [ExplainCommand](#) logical command and requests `SessionState` to [execute it](#) (to get a [QueryExecution](#) back).

Note	<code>explain</code> uses ExplainCommand logical command that, when executed , gives different text representations of QueryExecution (for the Dataset's LogicalPlan) depending on the flags (e.g. <code>extended</code> , <code>codegen</code> , and <code>cost</code> which are disabled by default).
------	--

`explain` then requests `QueryExecution` for [SparkPlan](#) and collects the records (as [InternalRow](#) objects).

Note	<code>explain</code> uses Dataset's SparkSession to access the current <code>SessionState</code> .
------	--

In the end, `explain` goes over the `InternalRow` records and converts them to lines to display to console.

Note	<code>explain "converts"</code> an <code>InternalRow</code> record to a line using getString at position <code>0</code> .
------	---

Tip	If you are serious about query debugging you could also use the Debugging Query Execution facility .
-----	--

```
scala> spark.range(10).explain(extended = true)
== Parsed Logical Plan ==
Range (0, 10, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
Range (0, 10, step=1, splits=Some(8))

== Optimized Logical Plan ==
Range (0, 10, step=1, splits=Some(8))

== Physical Plan ==
*Range (0, 10, step=1, splits=Some(8))
```

toJSON Method

`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"}|
+-----+
```

Internally, `toJSON` grabs the `RDD[InternalRow]` (of the [QueryExecution](#) of the `Dataset`) and maps the records (per RDD partition) into JSON.

Note

`toJSON` uses Jackson's JSON parser—[jackson-module-scala](#).

Accessing Schema— `schema` Method

A `dataset` has a **schema**.

`schema: StructType`

Tip

You may also use the following methods to learn about the schema:

- `printSchema(): Unit`
- [explain](#)

Converting Dataset into RDD— `rdd` Attribute

`rdd: RDD[T]`

Whenever you are in need to convert a `Dataset` into a `RDD`, executing `rdd` method gives you the RDD of the proper input object type (not [Row as in DataFrames](#)) that sits behind the `Dataset`.

```
scala> val rdd = tokens.rdd
rdd: org.apache.spark.rdd.RDD[Token] = MapPartitionsRDD[11] at rdd at <console>:30
```

Internally, it looks [ExpressionEncoder](#) (for the `Dataset`) up and accesses the `deserializer` expression. That gives the `DataType` of the result of evaluating the expression.

Note

A deserializer expression is used to decode an [InternalRow](#) to an object of type `T`. See [ExpressionEncoder](#).

It then executes a [DeserializeToObject](#) logical operator that will produce a `RDD[InternalRow]` that is converted into the proper `RDD[T]` using the `DataType` and `T`.

Note

It is a lazy operation that "produces" a `RDD[T]`.

`isStreaming` Method

`isStreaming` returns `true` when `Dataset` contains [StreamingRelation](#) or [StreamingExecutionRelation](#) **streaming sources**.

Note	Streaming datasets are created using <code>DataFrameReader.stream</code> method (for <code>StreamingRelation</code>) and contain <code>StreamingExecutionRelation</code> after <code>DataStreamWriter.start</code> .
------	---

```
val reader = spark.read
val helloStream = reader.stream("hello")

scala> helloStream.isStreaming
res9: Boolean = true
```

Note	A new feature in Spark 2.0.0.
------	-------------------------------

Is Dataset Local — `isLocal` Method

```
isLocal: Boolean
```

`isLocal` is a flag that says whether operators like `collect` or `take` could be run locally, i.e. without using executors.

Internally, `isLocal` checks whether the logical query plan of a `Dataset` is `LocalRelation`.

Dataset Columns

`Column` type represents a column in a `Dataset` that is the values of records for a given field.

Note

A `Column` is a value generator for records of a `Dataset`.

With the `implicits` conversions imported, you can create "free" column references using Scala's symbols.

```
val spark: SparkSession = ...
import spark.implicits._

import org.apache.spark.sql.Column
scala> val nameCol: Column = 'name
nameCol: org.apache.spark.sql.Column = name
```

Note

"Free" column references are `Column`s with no association to a `Dataset`.

You can also create free column references from `$`-prefixed strings.

```
// Note that $ alone creates a ColumnName
scala> val idCol = $"id"
idCol: org.apache.spark.sql.ColumnName = id

import org.apache.spark.sql.Column

// The target type triggers the implicit conversion to Column
scala> val idCol: Column = $"id"
idCol: org.apache.spark.sql.Column = id
```

Beside using the `implicits` conversions to create columns, you can use `col` and `column` methods from `functions` object.

```
import org.apache.spark.sql.functions._

scala> val nameCol = col("name")
nameCol: org.apache.spark.sql.Column = name

scala> val cityCol = column("city")
cityCol: org.apache.spark.sql.Column = city
```

Finally, you can create a `Column` reference using the `Dataset` it belongs to using `Dataset.apply` factory method or `Dataset.col` method. You can only use such `Column` references for the `Dataset`s they were created from.

```
scala> val textCol = dataset.col("text")
textCol: org.apache.spark.sql.Column = text

scala> val idCol = dataset.apply("id")
idCol: org.apache.spark.sql.Column = id

scala> val idCol = dataset("id")
idCol: org.apache.spark.sql.Column = id
```

You can reference nested columns using `.` (dot).

Note

`Column` has a reference to Catalyst's `Expression` it was created for using `expr` m

```
scala> window('time, "5 seconds").expr
res0: org.apache.spark.sql.catalyst.expressions.Expression = timewindow('time,
```

Adding Column to Dataset— `withColumn` Method

```
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|
+---+---+

```

You can add new columns do a `dataset` using `withColumn` method.

```

val spark: SparkSession = ...
val dataset = spark.range(5)

// Add a new column called "group"
scala> dataset.withColumn("group", 'id % 2).show
+---+---+
| id|group|
+---+---+
|  0|    0|
|  1|    1|
|  2|    0|
|  3|    1|
|  4|    0|
+---+---+

```

Referencing Column — `apply` Method

```

val spark: SparkSession = ...
case class Word(id: Long, text: String)
val dataset = Seq(Word(0, "hello"), Word(1, "spark")).toDS

scala> val idCol = dataset.apply("id")
idCol: org.apache.spark.sql.Column = id

// or using Scala's magic a little bit
// the following is equivalent to the above explicit apply call
scala> val idCol = dataset("id")
idCol: org.apache.spark.sql.Column = id

```

Creating Column — `col` method

```

val spark: SparkSession = ...
case class Word(id: Long, text: String)
val dataset = Seq(Word(0, "hello"), Word(1, "spark")).toDS

scala> val textCol = dataset.col("text")
textCol: org.apache.spark.sql.Column = text

```

like Operator

Caution

[FIXME](#)

```

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|
+---+

```

Defining Windowing Column — `over` Operator

```
over(window: expressions.WindowSpec): Column
```

`over` creates a **windowing column** that allows for window computations over a `window` (i.e. a group of records that are in *some* relation to the current record).

Tip

Read up on windowed aggregation in Spark SQL in [Window Aggregate Functions](#).

cast Operator

`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

```

`cast` 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|
+-----+
```

Standard Functions — functions Object

`org.apache.spark.sql.functions` object defines many built-in functions to work with [Columns](#) in [Datasets](#).

You can access the functions using the following `import` statement:

```
import org.apache.spark.sql.functions._
```

There are over 200 functions in the `functions` object.

```
scala> spark.catalog.listFunctions.count
res1: Long = 251
```

Table 1. (Subset of) Standard Functions in Spark SQL

	Name	Description
Aggregate functions	<code>count</code>	
	<code>grouping</code>	Indicates whether a specified column is aggregated or not
	<code>grouping_id</code>	Computes the level of grouping
Collection functions	<code>explode</code>	
	<code>explode_outer</code>	(new in 2.2.0) Creates a new row for each element in the given array or map column. If the array/map is <code>null</code> or empty then <code>null</code> is produced.
	<code>from_json</code>	Parses a column with a JSON string into a StructType or ArrayType of <code>structType</code> elements with the specified schema.
Date and time functions	<code>current_timestamp</code>	
	<code>to_date</code>	
	<code>to_timestamp</code>	

	unix_timestamp	Converts current or specified time to Unix timestamp (in seconds)
	window	Generates tumbling time windows
Math functions	bin	Converts the value of a long column to binary format
Regular functions	broadcast	
	col and column	Creating Columns
	expr	
	struct	
String functions	split	
	upper	
UDF functions	udf	Creating UDFs
Window functions	rank , dense_rank , percent_rank	Ranking records per window partition
	ntile	Gives the ntile group if (from <code>1</code> to <code>n</code> inclusive) in an ordered window partition
	row_number	Sequential numbering per window partition
	cume_dist	Cumulative distribution of records across window partitions
	lag	
	lead	

Tip	The page gives only a brief overview of the many functions available in <code>functions</code> object and so you should read the official documentation of the <code>functions</code> object .
-----	--

count Function

Caution

FIXME

explode_outer Function

```
explode_outer(e: Column): Column
```

`explode_outer` generates a new row for each element in `e` array or map column.

Note

Unlike `explode`, `explode_outer` generates `null` when the array or map is `null` or empty.

```
val arrays = Seq((1, Seq.empty[String])).toDF("id", "array")
scala> arrays.printSchema
root
|-- id: integer (nullable = false)
|-- array: array (nullable = true)
|   |-- element: string (containsNull = true)
scala> arrays.select(explode_outer($"array")).show
+---+
| col|
+---+
| null|
+---+
```

Internally, `explode_outer` creates a `Column` with `GeneratorOuter` and `Explode Catalyst` expressions.

```
val explodeOuter = explode_outer($"array").expr
scala> println(explodeOuter.numberedTreeString)
00 generatorouter(explode('array))
01 +- explode('array)
02   +- 'array
```

explode Function

Caution

FIXME

```
scala> Seq(Array(0,1,2)).toDF("array").withColumn("num", explode('array)).show
+-----+---+
|    array|num|
+-----+---+
|[0, 1, 2]|  0|
|[0, 1, 2]|  1|
|[0, 1, 2]|  2|
+-----+---+
```

Note

`explode` function is an equivalent of `flatMap` operator for `Dataset`.

Ranking Records per Window Partition — `rank` Function

```
rank(): Column  
dense_rank(): Column  
percent_rank(): Column
```

`rank` functions assign the sequential rank of each distinct value per window partition. They are equivalent to `RANK`, `DENSE_RANK` and `PERCENT_RANK` functions in the good ol' SQL.

```

val dataset = spark.range(9).withColumn("bucket", 'id % 3)

import org.apache.spark.sql.expressions.Window
val byBucket = Window.partitionBy('bucket).orderBy('id)

scala> dataset.withColumn("rank", rank over byBucket).show
+---+-----+
| id|bucket|rank|
+---+-----+
| 0|    0|  1|
| 3|    0|  2|
| 6|    0|  3|
| 1|    1|  1|
| 4|    1|  2|
| 7|    1|  3|
| 2|    2|  1|
| 5|    2|  2|
| 8|    2|  3|
+---+-----+

scala> dataset.withColumn("percent_rank", percent_rank over byBucket).show
+---+-----+
| id|bucket|percent_rank|
+---+-----+
| 0|    0|      0.0|
| 3|    0|      0.5|
| 6|    0|      1.0|
| 1|    1|      0.0|
| 4|    1|      0.5|
| 7|    1|      1.0|
| 2|    2|      0.0|
| 5|    2|      0.5|
| 8|    2|      1.0|
+---+-----+

```

`rank` function assigns the same rank for duplicate rows with a gap in the sequence (similarly to Olympic medal places). `dense_rank` is like `rank` for duplicate rows but compacts the ranks and removes the gaps.

```

// rank function with duplicates
// Note the missing/sparse ranks, i.e. 2 and 4
scala> dataset.union(dataset).withColumn("rank", rank over byBucket).show
+---+-----+
| id|bucket|rank|
+---+-----+
| 0|    0|  1|
| 0|    0|  1|
| 3|    0|  3|
| 3|    0|  3|
| 6|    0|  5|
+---+-----+

```

```

| 6| 0| 5|
| 1| 1| 1|
| 1| 1| 1|
| 4| 1| 3|
| 4| 1| 3|
| 7| 1| 5|
| 7| 1| 5|
| 2| 2| 1|
| 2| 2| 1|
| 5| 2| 3|
| 5| 2| 3|
| 8| 2| 5|
| 8| 2| 5|
+---+---+---+
// dense_rank function with duplicates
// Note that the missing ranks are now filled in
scala> dataset.union(dataset).withColumn("dense_rank", dense_rank over byBucket).show
+---+---+---+
| id|bucket|dense_rank|
+---+---+---+
| 0| 0| 1|
| 0| 0| 1|
| 3| 0| 2|
| 3| 0| 2|
| 6| 0| 3|
| 6| 0| 3|
| 1| 1| 1|
| 1| 1| 1|
| 4| 1| 2|
| 4| 1| 2|
| 7| 1| 3|
| 7| 1| 3|
| 2| 2| 1|
| 2| 2| 1|
| 5| 2| 2|
| 5| 2| 2|
| 8| 2| 3|
| 8| 2| 3|
+---+---+---+
// percent_rank function with duplicates
scala> dataset.union(dataset).withColumn("percent_rank", percent_rank over byBucket).show
+---+---+---+
| id|bucket|percent_rank|
+---+---+---+
| 0| 0| 0.0|
| 0| 0| 0.0|
| 3| 0| 0.4|
| 3| 0| 0.4|
| 6| 0| 0.8|
| 6| 0| 0.8|

```

```

| 1| 1| 0.0|
| 1| 1| 0.0|
| 4| 1| 0.4|
| 4| 1| 0.4|
| 7| 1| 0.8|
| 7| 1| 0.8|
| 2| 2| 0.0|
| 2| 2| 0.0|
| 5| 2| 0.4|
| 5| 2| 0.4|
| 8| 2| 0.8|
| 8| 2| 0.8|
+---+---+-----+

```

Cumulative Distribution of Records Across Window Partitions — `cume_dist` Function

```
cume_dist(): Column
```

`cume_dist` computes the cumulative distribution of the records in window partitions. This is equivalent to SQL's `CUME_DIST` function.

```

val buckets = spark.range(9).withColumn("bucket", 'id % 3)
// Make duplicates
val dataset = buckets.union(buckets)

import org.apache.spark.sql.expressions.Window
val windowSpec = Window.partitionBy('bucket).orderBy('id)
scala> dataset.withColumn("cume_dist", cume_dist over windowSpec).show
+---+-----+-----+
| id|bucket|      cume_dist|
+---+-----+-----+
| 0|    0|0.3333333333333333|
| 3|    0|0.6666666666666666|
| 6|    0|        1.0|
| 1|    1|0.3333333333333333|
| 4|    1|0.6666666666666666|
| 7|    1|        1.0|
| 2|    2|0.3333333333333333|
| 5|    2|0.6666666666666666|
| 8|    2|        1.0|
+---+-----+-----+

```

lag Function

```
lag(e: Column, offset: Int): Column
lag(columnName: String, offset: Int): Column
lag(columnName: String, offset: Int, defaultValue: Any): Column
lag(e: Column, offset: Int, defaultValue: Any): Column
```

`lag` returns the value in `e` / `columnName` column that is `offset` records before the current record. `lag` returns `null` value if the number of records in a window partition is less than `offset` or `defaultValue`.

```
val buckets = spark.range(9).withColumn("bucket", 'id % 3)
// Make duplicates
val dataset = buckets.union(buckets)

import org.apache.spark.sql.expressions.Window
val windowSpec = Window.partitionBy('bucket).orderBy('id)
scala> dataset.withColumn("lag", lag('id, 1) over windowSpec).show
+---+-----+
| id|bucket| lag|
+---+-----+
|  0|     0| null|
|  3|     0|  0|
|  6|     0|  3|
|  1|     1| null|
|  4|     1|  1|
|  7|     1|  4|
|  2|     2| null|
|  5|     2|  2|
|  8|     2|  5|
+---+-----+

scala> dataset.withColumn("lag", lag('id, 2, "<default_value>") over windowSpec).show
+---+-----+
| id|bucket| lag|
+---+-----+
|  0|     0| null|
|  3|     0| null|
|  6|     0|  0|
|  1|     1| null|
|  4|     1| null|
|  7|     1|  1|
|  2|     2| null|
|  5|     2| null|
|  8|     2|  2|
+---+-----+
```

Caution

FIXME It looks like `lag` with a default value has a bug — the default value's not used at all.

lead Function

```
lead(columnName: String, offset: Int): Column
lead(e: Column, offset: Int): Column
lead(columnName: String, offset: Int, defaultValue: Any): Column
lead(e: Column, offset: Int, defaultValue: Any): Column
```

`lead` returns the value that is `offset` records after the current records, and `defaultValue` if there is less than `offset` records after the current record. `lag` returns `null` value if the number of records in a window partition is less than `offset` or `defaultValue`.

```
val buckets = spark.range(9).withColumn("bucket", 'id % 3)
// Make duplicates
val dataset = buckets.union(buckets)

import org.apache.spark.sql.expressions.Window
val windowSpec = Window.partitionBy('bucket).orderBy('id)
scala> dataset.withColumn("lead", lead('id, 1) over windowSpec).show
+---+-----+
| id|bucket|lead|
+---+-----+
|  0|     0|   0|
|  0|     0|   3|
|  3|     0|   3|
|  3|     0|   6|
|  6|     0|   6|
|  6|     0|null|
|  1|     1|   1|
|  1|     1|   4|
|  4|     1|   4|
|  4|     1|   7|
|  7|     1|   7|
|  7|     1|null|
|  2|     2|   2|
|  2|     2|   5|
|  5|     2|   5|
|  5|     2|   8|
|  8|     2|   8|
|  8|     2|null|
+---+-----+

scala> dataset.withColumn("lead", lead('id, 2, "<default_value>") over windowSpec).sho
w
+---+-----+
| id|bucket|lead|
+---+-----+
|  0|     0|   3|
|  0|     0|   3|
|  3|     0|   6|
|  3|     0|   6|
```

```
| 6| 0|null|
| 6| 0|null|
| 1| 1| 4|
| 1| 1| 4|
| 4| 1| 7|
| 4| 1| 7|
| 7| 1|null|
| 7| 1|null|
| 2| 2| 5|
| 2| 2| 5|
| 5| 2| 8|
| 5| 2| 8|
| 8| 2|null|
| 8| 2|null|
+---+-----+---+
```

Caution

FIXME It looks like `lead` with a default value has a bug — the default value's not used at all.

Sequential numbering per window partition

— `row_number` Function

```
row_number(): Column
```

`row_number` returns a sequential number starting at `1` within a window partition.

```
val buckets = spark.range(9).withColumn("bucket", 'id % 3)
// Make duplicates
val dataset = buckets.union(buckets)

import org.apache.spark.sql.expressions.Window
val windowSpec = Window.partitionBy('bucket).orderBy('id)
scala> dataset.withColumn("row_number", row_number() over windowSpec).show
+---+-----+
| id|bucket|row_number|
+---+-----+
|  0|     0|        1|
|  0|     0|        2|
|  3|     0|        3|
|  3|     0|        4|
|  6|     0|        5|
|  6|     0|        6|
|  1|     1|        1|
|  1|     1|        2|
|  4|     1|        3|
|  4|     1|        4|
|  7|     1|        5|
|  7|     1|        6|
|  2|     2|        1|
|  2|     2|        2|
|  5|     2|        3|
|  5|     2|        4|
|  8|     2|        5|
|  8|     2|        6|
+---+-----+
```

ntile Function

```
ntile(n: Int): Column
```

`ntile` computes the ntile group id (from `1` to `n` inclusive) in an ordered window partition.

```

val dataset = spark.range(7).select('*', 'id % 3 as "bucket")

import org.apache.spark.sql.expressions.Window
val byBuckets = Window.partitionBy('bucket).orderBy('id)
scala> dataset.select('*', ntile(3) over byBuckets as "ntile").show
+---+-----+-----+
| id|bucket|ntile|
+---+-----+-----+
|  0|      0|     1|
|  3|      0|     2|
|  6|      0|     3|
|  1|      1|     1|
|  4|      1|     2|
|  2|      2|     1|
|  5|      2|     2|
+---+-----+-----+

```

Caution

FIXME How is `ntile` different from `rank`? What about performance?

Creating Columns — `col` and `column` Functions

```

col(colName: String): Column
column(colName: String): Column

```

`col` and `column` methods create a [Column](#) that you can later use to reference a column in a dataset.

```

import org.apache.spark.sql.functions._

scala> val nameCol = col("name")
nameCol: org.apache.spark.sql.Column = name

scala> val cityCol = column("city")
cityCol: org.apache.spark.sql.Column = city

```

Defining UDFs — `udf` Function

```
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|
+---+---+

```

split Function

```
split(str: Column, pattern: String): Column
```

`split` function splits `str` column using `pattern`. It returns a new `column`.

Note	split UDF uses java.lang.String.split(String regex, int limit) 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	.\$ ()[{^?*+\` 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 <code>Column</code> and return another to showcase how to chain them.
------	--

```
val df = Seq((0,1,"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|
+---+-----+-----+
```

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

Read up on [Broadcast Joins \(aka Map-Side Joins\)](#).

```

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#123 AS id#126, _2#124 AS token#127]
: +- LocalRelation [_1#123, _2#124]
+- BroadcastHint
  +- Project [_1#136 AS token#139, _2#137 AS prob#140]
    +- LocalRelation [_1#136, _2#137]

== Analyzed Logical Plan ==
token: string, id: int, prob: double
Project [token#127, id#126, prob#140]
+- Join Inner, (token#127 = token#139)
  :- Project [_1#123 AS id#126, _2#124 AS token#127]
  : +- LocalRelation [_1#123, _2#124]
  +- BroadcastHint
    +- Project [_1#136 AS token#139, _2#137 AS prob#140]
      +- LocalRelation [_1#136, _2#137]

== Optimized Logical Plan ==
Project [token#127, id#126, prob#140]
+- Join Inner, (token#127 = token#139)
  :- Project [_1#123 AS id#126, _2#124 AS token#127]
  : +- Filter isnotnull(_2#124)
  :   +- LocalRelation [_1#123, _2#124]
  +- BroadcastHint
    +- Project [_1#136 AS token#139, _2#137 AS prob#140]
      +- Filter isnotnull(_1#136)
        +- LocalRelation [_1#136, _2#137]

== Physical Plan ==
*Project [token#127, id#126, prob#140]
+- *BroadcastHashJoin [token#127], [token#139], Inner, BuildRight
  :- *Project [_1#123 AS id#126, _2#124 AS token#127]
  : +- *Filter isnotnull(_2#124)
  :   +- LocalTableScan [_1#123, _2#124]
  +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
    +- *Project [_1#136 AS token#139, _2#137 AS prob#140]
      +- *Filter isnotnull(_1#136)
        +- LocalTableScan [_1#136, _2#137]

```

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.

grouping Aggregate Function

```

grouping(e: Column): Column
grouping(columnName: String): Column  (1)

```

1. Calls the first `grouping` with `columnName` as a `column`

`grouping` is an aggregate function that indicates whether a specified column is aggregated or not and:

- returns `1` if the column is in a subtotal and is `NULL`
- returns `0` if the underlying value is `NULL` or any other value

Note	<code>grouping</code> can only be used with <code>cube</code> , <code>rollup</code> or <code>GROUPING SETS</code> multi-dimensional aggregate operators (and is verified when <code>Analyzer</code> does check analysis).
------	---

From [Hive's documentation about Grouping_ID function](#) (that can somehow help to understand `grouping`):

When aggregates are displayed for a column its value is `null`. This may conflict in case the column itself has some `null` values. There needs to be some way to identify `NULL` in column, which means aggregate and `NULL` in column, which means value. `GROUPING_ID` function is the solution to that.

```
val tmpWorkshops = Seq(
  ("Warsaw", 2016, 2),
  ("Toronto", 2016, 4),
  ("Toronto", 2017, 1)).toDF("city", "year", "count")

// there seems to be a bug with nulls
// and so the need for the following union
val cityNull = Seq(
  (null.asInstanceOf[String], 2016, 2)).toDF("city", "year", "count")

val workshops = tmpWorkshops union cityNull

scala> workshops.show
+-----+----+----+
|   city|year|count|
+-----+----+----+
| Warsaw|2016|    2|
| Toronto|2016|    4|
| Toronto|2017|    1|
|   null|2016|    2|
+-----+----+----+

val q = workshops
  .cube("city", "year")
  .agg(grouping("city"), grouping("year")) // <-- grouping here
  .sort($"city".desc_nulls_last, $"year".desc_nulls_last)

scala> q.show
+-----+----+-----+-----+
|   city|year|grouping(city)|grouping(year)|
+-----+----+-----+-----+
| Warsaw|2016|          0|          0|
| Warsaw|null|         0|          1|
| Toronto|2017|         0|          0|
| Toronto|2016|         0|          0|
| Toronto|null|         0|          1|
|   null|2017|         1|          0|
|   null|2016|         1|          0|
|   null|2016|         0|          0| <-- null is city
|   null|null|        0|          1| <-- null is city
|   null|null|        1|          1|
+-----+----+-----+-----+
```

Internally, `grouping` creates a `Column` with `Grouping` expression.

```

val q = workshops(cube("city", "year").agg(grouping("city")))
scala> println(q.queryExecution.logical)
'Aggregate [cube(city#182, year#183)], [city#182, year#183, grouping('city) AS groupin
g(city)#705]
+- Union
  :- Project [_1#178 AS city#182, _2#179 AS year#183, _3#180 AS count#184]
    :  +- LocalRelation [_1#178, _2#179, _3#180]
  +- Project [_1#192 AS city#196, _2#193 AS year#197, _3#194 AS count#198]
    +- LocalRelation [_1#192, _2#193, _3#194]

scala> println(q.queryExecution.analyzed)
Aggregate [city#724, year#725, spark_grouping_id#721], [city#724, year#725, cast((shif
tright(spark_grouping_id#721, 1) & 1) as tinyint) AS grouping(city)#720]
+- Expand [List(city#182, year#183, count#184, city#722, year#723, 0), List(city#182,
year#183, count#184, city#722, null, 1), List(city#182, year#183, count#184, null, yea
r#723, 2), List(city#182, year#183, count#184, null, null, 3)], [city#182, year#183, c
ount#184, city#724, year#725, spark_grouping_id#721]
  +- Project [city#182, year#183, count#184, city#182 AS city#722, year#183 AS year#7
23]
    +- Union
      :- Project [_1#178 AS city#182, _2#179 AS year#183, _3#180 AS count#184]
        :  +- LocalRelation [_1#178, _2#179, _3#180]
      +- Project [_1#192 AS city#196, _2#193 AS year#197, _3#194 AS count#198]
        +- LocalRelation [_1#192, _2#193, _3#194]

```

Note

grouping was added to Spark SQL in [\[SPARK-12706\]](#) support grouping/grouping_id function together group set.

grouping_id Aggregate Function

```

grouping_id(cols: Column*): Column
grouping_id(colName: String, colNames: String*): Column (1)

```

1. Calls the first `grouping_id` with `colName` and `colNames` as objects of type `Column`
`grouping_id` is an aggregate function that computes the level of grouping:

- `0` for combinations of each column
- `1` for subtotals of column 1
- `2` for subtotals of column 2
- And so on...

```

val tmpworkshops = Seq(
  ("Warsaw", 2016, 2),
  ("Toronto", 2016, 4),

```

```
("Toronto", 2017, 1)).toDF("city", "year", "count")

// there seems to be a bug with nulls
// and so the need for the following union
val cityNull = Seq(
  (null.asInstanceOf[String], 2016, 2)).toDF("city", "year", "count")

val workshops = tmpWorkshops union cityNull

scala> workshops.show
+---+---+---+
| city|year|count|
+---+---+---+
| Warsaw|2016|    2|
| Toronto|2016|    4|
| Toronto|2017|    1|
| null|2016|    2|
+---+---+---+

val query = workshops
  .cube("city", "year")
  .agg(grouping_id()) // <-- all grouping columns used
  .sort($"city".desc_nulls_last, $"year".desc_nulls_last)
scala> query.show
+---+---+-----+
| city|year|grouping_id()|
+---+---+-----+
| Warsaw|2016|      0|
| Warsaw|null|     1|
| Toronto|2017|      0|
| Toronto|2016|      0|
| Toronto|null|     1|
| null|2017|      2|
| null|2016|      0|
| null|null|     1|
| null|null|      3|
+---+---+-----+

scala> spark.catalog.listFunctions.filter(_.name.contains("grouping_id")).show(false)
+-----+-----+-----+
+-----+
|name      |database|description|className
|isTemporary|
+-----+-----+-----+
+-----+
|grouping_id|null      |null          |org.apache.spark.sql.catalyst.expressions.GroupingID|
|true       |
+-----+-----+-----+
+-----+
// bin function gives the string representation of the binary value of the given long
column
```

```
scala> query.withColumn("bitmask", bin($"grouping_id()")).show
+-----+-----+-----+
| city|year|grouping_id()|bitmask|
+-----+-----+-----+
| Warsaw|2016|          0|      0|
| Warsaw|null|        1|      1|
| Toronto|2017|        0|      0|
| Toronto|2016|        0|      0|
| Toronto|null|        1|      1|
| null|2017|        2|     10|
| null|2016|        2|     10|
| null|2016|        0|      0| <- null is city
| null|null|        3|     11|
| null|null|        1|      1|
+-----+-----+-----+
```

The list of columns of `grouping_id` should match grouping columns (in `cube` or `rollup`) exactly, or empty which means all the grouping columns (which is exactly what the function expects).

Note	<code>grouping_id</code> can only be used with <code>cube</code> , <code>rollup</code> or <code>GROUPING SETS</code> multi-dimensional aggregate operators (and is verified when Analyzer does check analysis).
------	---

Note	Spark SQL's <code>grouping_id</code> function is known as <code>grouping_id</code> in Hive.
------	---

From [Hive's documentation about Grouping_ID function](#):

When aggregates are displayed for a column its value is `null`. This may conflict in case the column itself has some `null` values. There needs to be some way to identify `NULL` in column, which means aggregate and `NULL` in column, which means value. `GROUPING_ID` function is the solution to that.

Internally, `grouping_id()` creates a [Column](#) with `GroupingID` unevaluable expression.

Note	Unevaluable expressions are expressions replaced by some other expressions during analysis or optimization .
------	--

```
// workshops dataset was defined earlier
val q = workshops
  .cube("city", "year")
  .agg(grouping_id())

// grouping_id function is spark_grouping_id virtual column internally
// that is resolved during analysis - see Analyzed Logical Plan
scala> q.explain(true)
== Parsed Logical Plan ==
'Aggregate [cube(city#182, year#183)], [city#182, year#183, grouping_id() AS grouping_
```

```

id()#742]
+- Union
  :- Project [_1#178 AS city#182, _2#179 AS year#183, _3#180 AS count#184]
    :  +- LocalRelation [_1#178, _2#179, _3#180]
  +- Project [_1#192 AS city#196, _2#193 AS year#197, _3#194 AS count#198]
    +- LocalRelation [_1#192, _2#193, _3#194]

== Analyzed Logical Plan ==
city: string, year: int, grouping_id(): int
Aggregate [city#757, year#758, spark_grouping_id#754], [city#757, year#758, spark_grouping_id#754 AS grouping_id()#742]
+- Expand [List(city#182, year#183, count#184, city#755, year#756, 0), List(city#182, year#183, count#184, city#755, null, 1), List(city#182, year#183, count#184, null, year#756, 2), List(city#182, year#183, count#184, null, null, 3)], [city#182, year#183, count#184, city#757, year#758, spark_grouping_id#754]
  +- Project [city#182, year#183, count#184, city#182 AS city#755, year#183 AS year#756]
    +- Union
      :- Project [_1#178 AS city#182, _2#179 AS year#183, _3#180 AS count#184]
        :  +- LocalRelation [_1#178, _2#179, _3#180]
      +- Project [_1#192 AS city#196, _2#193 AS year#197, _3#194 AS count#198]
        +- LocalRelation [_1#192, _2#193, _3#194]

== Optimized Logical Plan ==
Aggregate [city#757, year#758, spark_grouping_id#754], [city#757, year#758, spark_grouping_id#754 AS grouping_id()#742]
+- Expand [List(city#755, year#756, 0), List(city#755, null, 1), List(null, year#756, 2), List(null, null, 3)], [city#757, year#758, spark_grouping_id#754]
  +- Union
    :- LocalRelation [city#755, year#756]
    +- LocalRelation [city#755, year#756]

== Physical Plan ==
*HashAggregate(keys=[city#757, year#758, spark_grouping_id#754], functions=[], output=[city#757, year#758, grouping_id()#742])
+- Exchange hashpartitioning(city#757, year#758, spark_grouping_id#754, 200)
  +- *HashAggregate(keys=[city#757, year#758, spark_grouping_id#754], functions=[], output=[city#757, year#758, spark_grouping_id#754])
    +- *Expand [List(city#755, year#756, 0), List(city#755, null, 1), List(null, year#756, 2), List(null, null, 3)], [city#757, year#758, spark_grouping_id#754]
      +- Union
        :- LocalTableScan [city#755, year#756]
        +- LocalTableScan [city#755, year#756]

```

Note

`grouping_id` was added to Spark SQL in [\[SPARK-12706\]](#) support `grouping/grouping_id` function together group set.

Parsing Column With JSON-Encoded Records — `from_json` Functions

```
from_json(e: Column, schema: DataType): Column (1)
from_json(
  e: Column,
  schema: DataType,
  options: Map[String, String]): Column
```

1. Relays to the other `from_json` with empty `options`

Parses a column with a JSON string into a `StructType` or `ArrayType` of `StructType` elements with the specified schema.

Note	<code>options</code> controls how a JSON is parsed and contains the same options as the json data source .
------	--

Internally, `from_json` creates a `Column` with `JsonToStructs` unary expression.

```
val jsons = Seq("""{ "id": 0 }""").toDF("json")

import org.apache.spark.sql.types._

val schema = StructType(
  StructField("id", IntegerType, nullable = false) :: Nil)

scala> jsons.select(from_json($"json", schema) as "ids").show
+---+
|ids|
+---+
|[0]|
+---+
```

Note	<code>from_json</code> corresponds to SQL's <code>from_json</code> .
------	--

Converting Long to Binary Format (in String Representation) — `bin` Function

```
bin(e: Column): Column
bin(columnName: String): Column (1)
```

1. Calls the first `bin` with `columnName` as a `Column`

`bin` converts the long value in a column to its binary format (i.e. as an unsigned integer in base 2) with no extra leading 0s.

```

scala> spark.range(5).withColumn("binary", bin('id)).show
+---+-----+
| id|binary|
+---+-----+
| 0 |    0|
| 1 |    1|
| 2 |   10|
| 3 |   11|
| 4 |  100|
+---+-----+

val withBin = spark.range(5).withColumn("binary", bin('id))
scala> withBin.printSchema
root
|-- id: long (nullable = false)
|-- binary: string (nullable = false)

```

Internally, `bin` creates a [Column](#) with `Bin` unary expression.

```

scala> withBin.queryExecution.logical
res2: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
'Project [*, bin('id) AS binary#14]
+- Range (0, 5, step=1, splits=Some(8))

```

Note	<code>Bin</code> unary expression uses java.lang.Long.toBinaryString for the conversion.
------	--

Note	<code>Bin</code> expression supports code generation (aka <code>CodeGen</code>). <pre> val withBin = spark.range(5).withColumn("binary", bin('id)) scala> withBin.queryExecution.debugcodegen Found 1 WholeStageCodegen subtrees. == Subtree 1 / 1 == *Project [id#19L, bin(id#19L) AS binary#22] +- *Range (0, 5, step=1, splits=Some(8)) ... /* 103 */ UTF8String project_value1 = null; /* 104 */ project_value1 = UTF8String.fromString(java.lang.Long.toBir </pre>
------	---

Standard Functions for Date and Time

Table 1. (Subset of) Standard Functions for Date and Time

Name	Description
<code>current_timestamp</code>	
<code>to_date</code>	
<code>to_timestamp</code>	
<code>unix_timestamp</code>	Converts current or specified time to Unix timestamp (in seconds)
<code>window</code>	Generates tumbling time windows

current_date Function

```
to_date(e: Column, fmt: String): Column
```

Caution

[FIXME](#)

current_timestamp Function

```
current_timestamp(): Column
```

Caution

[FIXME](#)

Note

`current_timestamp` is also `now` function in SQL.

to_date Function

```
to_date(e: Column, fmt: String): Column
```

Caution

[FIXME](#)

to_timestamp Function

```
to_timestamp(s: Column): Column
to_timestamp(s: Column, fmt: String): Column
```

Caution

FIXME

Converting Current or Specified Time to Unix Timestamp

— `unix_timestamp` Function

```
unix_timestamp(): Column (1)
unix_timestamp(time: Column): Column (2)
unix_timestamp(time: Column, format: String): Column
```

1. Gives current timestamp (in seconds)
2. Converts `time` string in format `yyyy-MM-dd HH:mm:ss` to Unix timestamp (in seconds)

`unix_timestamp` converts the current or specified `time` in the specified `format` to a Unix timestamp (in seconds).

`unix_timestamp` supports a column of type `Date`, `Timestamp` or `String`.

```
// no time and format => current time
scala> spark.range(1).select(unix_timestamp as "current_timestamp").show
+-----+
|current_timestamp|
+-----+
|      1493362850|
+-----+


// no format so yyyy-MM-dd HH:mm:ss assumed
scala> Seq("2017-01-01 00:00:00").toDF("time").withColumn("unix_timestamp", unix_timestamp($"time")).show
+-----+-----+
|          time|unix_timestamp|
+-----+-----+
|2017-01-01 00:00:00|      1483225200|
+-----+-----+


scala> Seq("2017/01/01 00:00:00").toDF("time").withColumn("unix_timestamp", unix_timestamp($"time", "yyyy/MM/dd")).show
+-----+-----+
|          time|unix_timestamp|
+-----+-----+
|2017/01/01 00:00:00|      1483225200|
+-----+-----+
```

`unix_timestamp` returns `null` if conversion fails.

```
// note slashes as date separators
scala> Seq("2017/01/01 00:00:00").toDF("time").withColumn("unix_timestamp", unix_timestamp($"time")).show
+-----+-----+
|       time|unix_timestamp|
+-----+-----+
|2017/01/01 00:00:00|          null|
+-----+-----+
```

Note

`unix_timestamp` is also supported in [SQL mode](#).

```
scala> spark.sql("SELECT unix_timestamp() as unix_timestamp").show
+-----+
|unix_timestamp|
+-----+
| 1493369225|
+-----+
```

Internally, `unix_timestamp` creates a [Column](#) with [UnixTimestamp](#) binary expression (possibly with `currentTimestamp`).

Generating Tumbling Time Windows — `window` Function

```
window(
  timeColumn: Column,
  windowDuration: String): Column  (1)
window(
  timeColumn: Column,
  windowDuration: String,
  slideDuration: String): Column  (2)
window(
  timeColumn: Column,
  windowDuration: String,
  slideDuration: String,
  startTime: String): Column
```

1. With `slideDuration` as `windowDuration` and `0` second for `startTime`
2. With `0` second for `startTime`

`window` generates tumbling time windows of `windowDuration` duration given a `timeColumn` timestamp specifying column.

```
scala> val timeColumn = window('time, "5 seconds")
timeColumn: org.apache.spark.sql.Column = timewindow(time, 5000000, 5000000, 0) AS `window`
```

`timeColumn` has to be of [TimestampType](#), i.e. with `java.sql.Timestamp` values.

Tip	Use <code>java.sql.Timestamp.from</code> or <code>java.sql.Timestamp.valueOf</code> factory methods to create <code>Timestamp</code> instances.
-----	---

`windowDuration` and `slideDuration` are strings specifying the width of the window for duration and sliding identifiers, respectively.

Tip	Use <code>CalendarInterval</code> for valid window identifiers.
-----	---

Note	<code>window</code> is available as of Spark 2.0.0 .
------	---

Internally, `window` creates a [Column](#) (with [TimeWindow](#) expression) as `window`.

```
scala> timeColumn.expr.numberedTreeString
res9: String =
00 timewindow('time, 5000000, 5000000, 0) AS window#1
01 +- timewindow('time, 5000000, 5000000, 0)
02   +- 'time

import org.apache.spark.sql.catalyst.expressions.TimeWindow
scala> val timeWindow = timeColumn.expr.children(0).asInstanceOf[TimeWindow]
timeWindow: org.apache.spark.sql.catalyst.expressions.TimeWindow = timewindow('time, 5
000000, 5000000, 0)
```

Window Aggregate Functions

Window aggregate functions (aka **window functions** or **windowed aggregates**) are functions that perform a calculation over a group of records called **window** that are in *some* relation to the current record (i.e. can be in the same partition or frame as the current row).

In other words, when executed, a window function computes a value for each and every row in a window (per [window specification](#)).

Note	Window functions are also called over functions due to how they are applied using over operator.
------	---

Spark SQL supports three kinds of window functions:

- **ranking** functions
- **analytic** functions
- **aggregate** functions

Table 1. Window Aggregate Functions in Spark SQL

	Function	Purpose
Ranking functions	rank	
	dense_rank	
	percent_rank	
	ntile	
	row_number	
Analytic functions	cume_dist	
	lag	
	lead	

For aggregate functions, you can use the existing [aggregate functions](#) as window functions, e.g. `sum` , `avg` , `min` , `max` and `count` .

```
// Borrowed from 3.5. Window Functions in PostgreSQL documentation
// Example of window functions using Scala API
//

case class Salary(depName: String, empNo: Long, salary: Long)
val empsalary = Seq(
  Salary("sales", 1, 5000),
  Salary("personnel", 2, 3900),
  Salary("sales", 3, 4800),
  Salary("sales", 4, 4800),
  Salary("personnel", 5, 3500),
  Salary("develop", 7, 4200),
  Salary("develop", 8, 6000),
  Salary("develop", 9, 4500),
  Salary("develop", 10, 5200),
  Salary("develop", 11, 5200)).toDS

import org.apache.spark.sql.expressions.Window
// Windows are partitions of depName
scala> val byDepName = Window.partitionBy('depName)
byDepName: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@1a711314

scala> empsalary.withColumn("avg", avg('salary) over byDepName).show
+-----+-----+-----+
| depName|empNo|salary|           avg|
+-----+-----+-----+
| develop|    7|  4200|      5020.0|
| develop|    8|  6000|      5020.0|
| develop|    9|  4500|      5020.0|
| develop|   10|  5200|      5020.0|
| develop|   11|  5200|      5020.0|
| sales|    1|  5000| 4866.666666666667|
| sales|    3|  4800| 4866.666666666667|
| sales|    4|  4800| 4866.666666666667|
| personnel|  2|  3900|      3700.0|
| personnel|  5|  3500|      3700.0|
+-----+-----+-----+
```

You describe a window using the convenient factory methods in [Window object](#) that create a [window specification](#) that you can further refine with **partitioning**, **ordering**, and **frame boundaries**.

After you describe a window you can apply [window aggregate functions](#) like **ranking** functions (e.g. `RANK`), **analytic** functions (e.g. `LAG`), and the regular [aggregate functions](#), e.g. `sum`, `avg`, `max`.

Note

Window functions are supported in structured queries using [SQL](#) and [Column-based expressions](#).

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.

Note

The main difference between window aggregate functions and [aggregate functions](#) with [grouping operators](#) is that the former calculate values for every row in a window while the latter gives you at most the number of input rows, one value per group.

Tip

See [Examples](#) section in this document.

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(...)`.

Note

Window functions belong to [Window functions group](#) in Spark's Scala API.

Note

Window-based framework is available as an experimental feature since Spark **1.4.0**.

WindowSpec — Window Specification

A window function needs a window specification which is an instance of `WindowSpec` class.

Note

`WindowSpec` class is marked as experimental since **1.4.0**.

Tip

Consult [org.apache.spark.sql.expressions.WindowSpec](#) 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 [Window object](#) to create a `WindowSpec`.

```
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 records are in the same partition. With no partition defined, all records belong to a single partition.

2. **Ordering Specification** defines how records in a partition are ordered that in turn defines the position of a record in a partition. The ordering could be ascending (`ASC` in SQL or `asc` in Scala) or descending (`DESC` or `desc`).
3. **Frame Specification** (unsupported in Hive; see [Why do Window functions fail with "Window function X does not take a frame specification"?](#)) defines the records 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`

Partitioning Records — `partitionBy` Methods

```
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|
+---+-----+-----+
```

Ordering in Windows — `orderBy` Methods

```
orderBy(colName: String, colNames: String*): WindowSpec
orderBy(cols: Column*): WindowSpec
```

`orderBy` allows you to control the order of records in a window.

```

import org.apache.spark.sql.expressions.Window
val byDepnameSalaryDesc = Window.partitionBy('depname).orderBy('salary desc)

// a numerical rank within the current row's partition for each distinct ORDER BY value

scala> val rankByDepname = rank().over(byDepnameSalaryDesc)
rankByDepname: org.apache.spark.sql.Column = RANK() OVER (PARTITION BY depname ORDER BY
salary DESC UnspecifiedFrame)

scala> empsalary.select('*', rankByDepname as 'rank).show
+-----+-----+-----+
| depName|empNo|salary|rank|
+-----+-----+-----+
| develop|    8|  6000|    1|
| develop|   10|  5200|    2|
| develop|   11|  5200|    2|
| develop|    9|  4500|    4|
| develop|    7|  4200|    5|
| sales|    1|  5000|    1|
| sales|    3|  4800|    2|
| sales|    4|  4800|    2|
| personnel|   2|  3900|    1|
| personnel|   5|  3500|    2|
+-----+-----+-----+

```

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.

Window Operators in SQL Queries

The grammar of windows operators in SQL accepts the following:

1. CLUSTER BY or PARTITION BY or DISTRIBUTE BY for partitions,
2. ORDER BY or SORT BY for sorting order,
3. RANGE , ROWS , RANGE BETWEEN , and ROWS BETWEEN for window frame types,
4. UNBOUNDED PRECEDING , UNBOUNDED FOLLOWING , CURRENT ROW for frame bounds.

Tip

Consult [withWindows](#) helper in `AstBuilder`.

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 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
+	-----+	-----+	-----+	-----+	

Calculate rank of row

See "[Explaining" Query Plans of Windows](#)" for an elaborate example.

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?

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.

"Explaining" Query Plans of Windows

```

import org.apache.spark.sql.expressions.Window
val byDepnameSalaryDesc = Window.partitionBy('depname).orderBy('salary desc)

scala> val rankByDepname = rank().over(byDepnameSalaryDesc)
rankByDepname: org.apache.spark.sql.Column = RANK() OVER (PARTITION BY depname ORDER BY salary DESC UnspecifiedFrame)

// empsalary defined at the top of the page
scala> empsalary.select('*', rankByDepname as 'rank).explain(extended = true)
== Parsed Logical Plan ==
'Project [* , rank() windowspecdefinition('depname, 'salary DESC, UnspecifiedFrame) AS rank#9]
+- LocalRelation [depName#5, empNo#6L, salary#7L]

== Analyzed Logical Plan ==
depName: string, empNo: bigint, salary: bigint, rank: int
Project [depName#5, empNo#6L, salary#7L, rank#9]
+- Project [depName#5, empNo#6L, salary#7L, rank#9, rank#9]
  +- Window [rank(salary#7L) windowspecdefinition(depname#5, salary#7L DESC, ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#9], [depname#5], [salary#7L DESC]
    +- Project [depName#5, empNo#6L, salary#7L]
    +- LocalRelation [depName#5, empNo#6L, salary#7L]

== Optimized Logical Plan ==
Window [rank(salary#7L) windowspecdefinition(depname#5, salary#7L DESC, ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#9], [depname#5], [salary#7L DESC]
+- LocalRelation [depName#5, empNo#6L, salary#7L]

== Physical Plan ==
Window [rank(salary#7L) windowspecdefinition(depname#5, salary#7L DESC, ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#9], [depname#5], [salary#7L DESC]
+- *Sort [depname#5 ASC, salary#7L DESC], false, 0
  +- Exchange hashpartitioning(depname#5, 200)
    +- LocalTableScan [depName#5, empNo#6L, salary#7L]

```

Further reading or watching

- [Introducing Window Functions in Spark SQL](#)
- [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](#)

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 for transforming [Datasets](#).

Tip

Use the [higher-level standard Column-based functions](#) with [Dataset operators](#) whenever possible before reverting to using your own custom UDF functions since [UDFs are a blackbox](#) for Spark and so it does not even try to optimize them.

As Reynold once said on Spark's dev mailing list:

There are simple cases in which we can analyze the UDFs byte code and infer what it is doing, but it is pretty difficult to do in general.

You define a new UDF by defining a Scala function as an input parameter of `udf` function. It accepts Scala functions of up to 10 input parameters.

```
val dataset = Seq((0, "hello"), (1, "world")).toDF("id", "text")

// Define a regular Scala function
val upper: String => String = _.toUpperCase

// Define a UDF that wraps the upper Scala function defined above
// You could also define the function in place, i.e. inside udf
// but separating Scala functions from Spark SQL's UDFs allows for easier testing
import org.apache.spark.sql.functions.udf
val upperUDF = udf(upper)

// Apply the UDF to change the source dataset
scala> dataset.withColumn("upper", upperUDF('text)).show
+---+----+-----+
| id|text|upper|
+---+----+-----+
|  0|hello|HELLO|
|  1|world|WORLD|
+---+----+-----+
```

You can register UDFs to use in [SQL-based query expressions](#) via [UDFRegistration](#) (that is available through `SparkSession.udf` attribute).

```
val spark: SparkSession = ...
scala> spark.udf.register("myUpper", (input: String) => input.toUpperCase)
```

You can query for available [standard](#) and user-defined functions using the [Catalog](#) interface (that is available through `SparkSession.catalog` attribute).

```
val spark: SparkSession = ...
scala> spark.catalog.listFunctions.filter('name like "%upper%").show(false)
+-----+-----+-----+
|name  |database|description|className          |isTempor
ary|
+-----+-----+-----+-----+
|myupper|null    |null      |null           |true
 |
|upper  |null    |null      |org.apache.spark.sql.catalyst.expressions.Upper|true
 |
+-----+-----+-----+-----+
---+
```

Note

UDFs play a vital role in Spark MLlib to define new [Transformers](#) that are function objects that transform `DataFrames` into `DataFrames` by introducing new columns.

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. `toUpperCaseUDF`) 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](#).

UDFs are Blackbox

Let's review an example with an UDF. This example is converting strings of size 7 characters only and uses the `Dataset` standard operators first and then custom UDF to do the same transformation.

```

scala> spark.conf.get("spark.sql.parquet.filterPushdown")
res0: String = true

```

You are going to use the following `cities` dataset that is based on Parquet file (as used in [Predicate Pushdown / Filter Pushdown for Parquet Data Source](#) section). The reason for parquet is that it is an external data source that does support optimization Spark uses to optimize itself like predicate pushdown.

```
// no optimization as it is a more involved Scala function in filter
// 08/30 Asked on dev@spark mailing list for explanation
val cities6chars = cities.filter(_.name.length == 6).map(_.name.toUpperCase)

cities6chars.explain(true)

// or simpler when only concerned with PushedFilters attribute in Parquet
scala> cities6chars.queryExecution.optimizedPlan
res33: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.String, true], true) AS value#248]
+- MapElements <function1>, class City, [StructField(id,LongType,false), StructField(name,StringType,true)], obj#247: java.lang.String
  +- Filter <function1>.apply
    +- DeserializeToObject newInstance(class City), obj#246: City
      +- Relation[id#236L,name#237] parquet

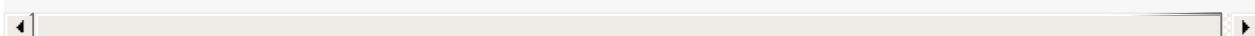
// no optimization for Dataset[City]!?
// 08/30 Asked on dev@spark mailing list for explanation
val cities6chars = cities.filter(_.name == "Warsaw").map(_.name.toUpperCase)

cities6chars.explain(true)

// The filter predicate is pushed down fine for Dataset's Column-based query in where
operator
scala> cities.where('name === "Warsaw").queryExecution.executedPlan
res29: org.apache.spark.sql.execution.SparkPlan =
*Project [id#128L, name#129]
+- *Filter (isnotnull(name#129) && (name#129 = Warsaw))
   +- *FileScan parquet [id#128L,name#129] Batched: true, Format: ParquetFormat, Input Paths: file:/Users/jacek/dev/oss/spark/cities.parquet, PartitionFilters: [], PushedFilters: [IsNotNull(name), EqualTo(name,Warsaw)], ReadSchema: struct<id:bigint,name:string>

// Let's define a UDF to do the filtering
val isWarsaw = udf { (s: String) => s == "Warsaw" }

// Use the UDF in where (replacing the Column-based query)
scala> cities.where(isWarsaw('name)).queryExecution.executedPlan
res33: org.apache.spark.sql.execution.SparkPlan =
*Filter UDF(name#129)
+- *FileScan parquet [id#128L,name#129] Batched: true, Format: ParquetFormat, InputPaths: file:/Users/jacek/dev/oss/spark/cities.parquet, PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id:bigint,name:string>
```



Basic Aggregation — Typed and Untyped Grouping Operators

You can group records in a [Dataset](#) by using [aggregate operators](#) and then executing [aggregate functions](#) to calculate aggregates (over a collection of grouped records).

Note	Aggregate functions without aggregate operators return a single value. If you want to find the aggregate values for each unique value (in a column), you should groupBy first (over this column) to build the groups.
------	---

Table 1. Aggregate Operators (in alphabetical order)

Operator	Return Type	Description
agg	RelationalGroupedDataset	Aggregates with or without grouping (i.e. entire Dataset)
groupBy	RelationalGroupedDataset	Used for untyped aggregations with DataFrames. Grouping is described using Column -based functions or column names.
groupByKey	KeyValueGroupedDataset	Used for type-preserving aggregations where records are grouped by a given key function.

Note	You can also use SparkSession to execute <i>good ol'</i> SQL with <code>GROUP BY</code> .
	<pre>val spark: SparkSession = ??? spark.sql("SELECT COUNT(*) FROM sales GROUP BY city")</pre>

Aggregates Over Subset Of or Whole Dataset — `agg` Operators

```
agg(expr: Column, exprs: Column*): DataFrame
agg(exprs: Map[String, String]): DataFrame
agg(aggExpr: (String, String), aggExprs: (String, String)*): DataFrame
```

`agg` applies an aggregate function on a subset or the entire `dataset` (i.e. considering the entire data set as one group).

Note	<code>agg</code> on a <code>Dataset</code> is simply a shortcut for groupBy().agg(...) .
------	--

```
scala> spark.range(10).agg(sum('id) as "sum").show
+---+
|sum|
+---+
| 45|
+---+
```

`agg` can compute aggregate expressions on all the records in a `Dataset`.

Untyped Grouping — `groupBy` Operator

```
groupBy(cols: Column*): RelationalGroupedDataset
groupBy(col1: String, cols: String*): RelationalGroupedDataset
```

`groupBy` methods group the records in a `Dataset` using the specified *discriminator* columns (as `Columns` or their text representation). It returns a `RelationalGroupedDataset` to execute aggregate functions or operators.

```
// 10^3-record large data set
val ints = 1 to math.pow(10, 3).toInt

scala> val dataset = ints.toDF("n").withColumn("m", 'n % 2)
dataset: org.apache.spark.sql.DataFrame = [n: int, m: int]

scala> dataset.count
res0: Long = 1000

scala> dataset.groupBy('m).agg(sum('n)).show
+---+-----+
| m|sum(n)|
+---+-----+
| 1|250000|
| 0|250500|
+---+-----+
```

Internally, it first `resolves columns` and then builds a `RelationalGroupedDataset`.

Note	The following session uses the data setup as described in Test Setup section below.
------	---

```
scala> dataset.show
+---+-----+-----+
|name|productId|score|
+---+-----+-----+
| aaa|      100| 0.12|
| aaa|      200| 0.29|
```

```

| bbb|    200| 0.53|
| bbb|    300| 0.42|
+---+-----+----+

```

```

scala> dataset.groupBy('name).avg().show
+-----+-----+
|name|avg(productId)|avg(score)|
+-----+-----+
| aaa|    150.0|    0.205|
| bbb|    250.0|    0.475|
+-----+-----+

```

```

scala> dataset.groupBy('name, 'productId).agg(Map("score" -> "avg")).show
+-----+-----+
|name|productId|avg(score)|
+-----+-----+
| aaa|    200|    0.29|
| bbb|    200|    0.53|
| bbb|    300|    0.42|
| aaa|    100|    0.12|
+-----+-----+

```

```

scala> dataset.groupBy('name).count.show
+-----+
|name|count|
+-----+
| aaa| 2|
| bbb| 2|
+-----+

```

```

scala> dataset.groupBy('name).max("score").show
+-----+
|name|max(score)|
+-----+
| aaa|    0.29|
| bbb|    0.53|
+-----+

```

```

scala> dataset.groupBy('name).sum("score").show
+-----+
|name|sum(score)|
+-----+
| aaa|    0.41|
| bbb|    0.95|
+-----+

```

```

scala> dataset.groupBy('productId).sum("score").show
+-----+
|productId|      sum(score)|
+-----+
|     300|        0.42|
|     100|        0.12|
|     200| 0.8200000000000001|
+-----+

```

```
+-----+-----+
```

Type-Preserving Grouping — `groupByKey` Operator

```
groupByKey[K: Encoder](func: T => K): KeyValueGroupedDataset[K, T]
```

`groupByKey` groups records (of type `T`) by the input `func`. It returns a `KeyValueGroupedDataset` to apply aggregation to.

Note	<code>groupByKey</code> is <code>Dataset</code> 's experimental API.
------	--

```
scala> dataset.groupByKey(_.productId).count.show
+-----+
|value|count(1)|
+-----+
| 300|      1|
| 100|      1|
| 200|      2|
+-----+


import org.apache.spark.sql.expressions.scalalang._
scala> dataset.groupByKey(_.productId).agg(typed.sum[Token](_.score)).toDF("productId"
, "sum").orderBy('productId).show
+-----+-----+
|productId|      sum|
+-----+-----+
|    100|      0.12|
|    200| 0.8200000000000001|
|    300|      0.42|
+-----+
```

KeyValueGroupedDataset

`KeyValueGroupedDataset` is an experimental interface to a result of executing the strongly-typed operator `groupByKey`.

```
scala> val tokensByName = dataset.groupByKey(_.name)
tokensByName: org.apache.spark.sql.KeyValueGroupedDataset[String,Token] = org.apache.s
park.sql.KeyValueGroupedDataset@1e3aad46
```

It holds `keys` that were used for the object.

```
scala> tokensByName.keys.show
+----+
|value|
+----+
| aaa|
| bbb|
+----+
```

The following methods are available for any `KeyValueGroupedDataset` to work on groups of records:

1. `agg` (of 1 to 4 types)
2. `mapGroups`
3. `flatMapGroups`
4. `reduceGroups`
5. `count` that is a special case of `agg` with `count` function applied.
6. `cogroup`

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 dataset = data.toDS.cache (1)
```

1. Cache the dataset so the following queries won't load/recompute data over and over again.

Join Operators

From PostgreSQL's [2.6. Joins Between Tables](#):

Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a **join query**.

You can join datasets using [join operators](#): `crossJoin`, `join`, and `joinWith`.

Table 1. Join Operators (in alphabetical order)

Operator	Return Type	Description
<code>crossJoin</code>	<code>DataFrame</code>	Untyped, <code>Row</code> -based cross join
<code>join</code>	<code>DataFrame</code>	Untyped, <code>Row</code> -based join
<code>joinWith</code>	<code>Dataset</code>	Used for type-preserving join with two output columns for records for which join condition holds

	You can also use SQL mode to join datasets using <i>good ol' SQL</i> .
Note	<pre>val spark: SparkSession = ... spark.sql("select * from t1, t2 where t1.id = t2.id")</pre>

You can specify a **join condition** (aka *join expression*) as part of join operators or using [where](#) operator.

```
df1.join(df2, $"df1Key" === $"df2Key")
df1.join(df2).where($"df1Key" === $"df2Key")
```

You can specify the **join type** as part of join operators (using `joinType` optional parameter).

```
df1.join(df2, $"df1Key" === $"df2Key", "inner")
```

Table 2. Join Types (in alphabetical order)

SQL	Name (joinType)	JoinType
CROSS	cross	Cross
INNER	inner	Inner
FULL OUTER	outer , full , fullouter	FullOuter
LEFT ANTI	leftanti	LeftAnti
LEFT OUTER	leftouter , left	LeftOuter
LEFT SEMI	leftsemi	LeftSemi
RIGHT OUTER	rightouter , right	RightOuter
NATURAL	Special case for Inner , LeftOuter , RightOuter , FullOuter	NaturalJoin
USING	Special case for Inner , LeftOuter , LeftSemi , RightOuter , FullOuter , LeftAnti	UsingJoin

Tip Name are case-insensitive and can use the underscore (`_`) at any position, i.e. `left_anti` and `LEFT_ANTI` are equivalent.

Note Spark SQL offers different [join strategies](#) with [Broadcast Joins \(aka Map-Side Joins\)](#) among them that are supposed to optimize your join queries over large distributed datasets.

join Operators

```
join(right: Dataset[_]): DataFrame (1)
join(right: Dataset[_], usingColumn: String): DataFrame (2)
join(right: Dataset[_], usingColumns: Seq[String]): DataFrame (3)
join(right: Dataset[_], usingColumns: Seq[String], joinType: String): DataFrame (4)
join(right: Dataset[_], joinExprs: Column): DataFrame (5)
join(right: Dataset[_], joinExprs: Column, joinType: String): DataFrame (6)
```

1. Condition-less inner join
2. Inner join with a single column that exists on both sides
3. Inner join with columns that exist on both sides

4. Equi-join with explicit join type
5. Inner join
6. Join with explicit join type. Self-joins are acceptable.

`join joins two Dataset S.`

```

val left = Seq((0, "zero"), (1, "one")).toDF("id", "left")
val right = Seq((0, "zero"), (2, "two"), (3, "three")).toDF("id", "right")

// Inner join
scala> left.join(right, "id").show
+---+---+
| id|left|right|
+---+---+
|  0|zero| zero|
+---+---+


scala> left.join(right, "id").explain
== Physical Plan ==
*Project [id#50, left#51, right#61]
+- *BroadcastHashJoin [id#50], [id#60], Inner, BuildRight
  :- LocalTableScan [id#50, left#51]
  +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as
bigint)))
    +- LocalTableScan [id#60, right#61]

// Full outer
scala> left.join(right, Seq("id"), "fullouter").show
+---+---+
| id|left|right|
+---+---+
|  1| one| null|
|  3|null|three|
|  2|null| two|
|  0|zero| zero|
+---+---+


scala> left.join(right, Seq("id"), "fullouter").explain
== Physical Plan ==
*Project [coalesce(id#50, id#60) AS id#85, left#51, right#61]
+- SortMergeJoin [id#50], [id#60], FullOuter
  :- *Sort [id#50 ASC NULLS FIRST], false, 0
  :  +- Exchange hashpartitioning(id#50, 200)
  :    +- LocalTableScan [id#50, left#51]
  +- *Sort [id#60 ASC NULLS FIRST], false, 0
    +- Exchange hashpartitioning(id#60, 200)
      +- LocalTableScan [id#60, right#61]

// Left anti
scala> left.join(right, Seq("id"), "leftanti").show

```

```

+---+----+
| id|left|
+---+----+
|  1| one|
+---+----+

scala> left.join(right, Seq("id"), "leftanti").explain
== Physical Plan ==
*BroadcastHashJoin [id#50], [id#60], LeftAnti, BuildRight
:- LocalTableScan [id#50, left#51]
+- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as big
int)))
+- LocalTableScan [id#60]

```

Internally, `join(right: Dataset[_])` creates a DataFrame with a condition-less Join logical operator (in the current SparkSession).

Note

`join(right: Dataset[_])` creates a logical plan with a condition-less Join operator with two child logical plans of the both sides of the join.

Note

`join(right: Dataset[_], usingColumns: Seq[String], joinType: String)` creates a logical plan with a condition-less Join operator with UsingJoin join type.

Note

`join(right: Dataset[_], joinExprs: Column, joinType: String)` accepts self-joins where `joinExprs` is of the form:

```
df("key") === df("key")
```

Note

That is usually considered a trivially true condition and refused as acceptable.

With `spark.sql.selfJoinAutoResolveAmbiguity` option enabled (which it is by default), `join` will automatically resolve ambiguous join conditions into ones that might make sense.

See [\[SPARK-6231\] Join on two tables \(generated from same one\) is broken.](#)

crossJoin Method

```
crossJoin(right: Dataset[_]): DataFrame
```

`crossJoin` joins two Datasets using Cross join type with no condition.

Note

`crossJoin` creates an explicit cartesian join that can be very expensive without an extra filter (that can be pushed down).

Type-Preserving Joins — `joinWith` Operators

```
joinWith[U](other: Dataset[U], condition: Column): Dataset[(T, U)] (1)
joinWith[U](other: Dataset[U], condition: Column, joinType: String): Dataset[(T, U)]
```

1. Type-safe inner join

`joinWith` creates a `Dataset` with two columns `_1` and `_2` that each contains records for which `condition` holds.

```
case class Person(id: Long, name: String, cityId: Long)
case class City(id: Long, name: String)

val people = Seq(Person(0, "Agata", 0), Person(1, "Iweta", 0)).toDS
val cities = Seq(City(0, "Warsaw"), City(1, "Washington")).toDS

val joined = people.joinWith(cities, people("cityId") === cities("id"))

scala> joined.printSchema
root
|-- _1: struct (nullable = false)
|   |-- id: long (nullable = false)
|   |-- name: string (nullable = true)
|   |-- cityId: long (nullable = false)
|-- _2: struct (nullable = false)
|   |-- id: long (nullable = false)
|   |-- name: string (nullable = true)

scala> joined.show
+-----+-----+
|      _1|      _2|
+-----+-----+
|[0,Agata,0]| [0,Warsaw]|
|[1,Iweta,0]| [0,Warsaw]|
+-----+-----+
```

Note	<code>joinWith</code> preserves type-safety with the original object types.
------	---

Note	<code>joinWith</code> creates a <code>Dataset</code> with <code>Join</code> logical plan.
------	---

Broadcast Joins (aka Map-Side Joins)

Spark SQL uses **broadcast join** (aka **broadcast hash join**) instead of hash join to optimize join queries when the size of one side data is below `spark.sql.autoBroadcastJoinThreshold`.

Broadcast join can be very efficient for joins between a large table (fact) with relatively small tables (dimensions) that could then be used to perform a **star-schema join**. It can avoid sending all data of the large table over the network.

You can use `broadcast` function or SQL's `broadcast hints` to mark a dataset to be broadcast when used in a join query.

Note	According to the article Map-Side Join in Spark , broadcast join is also called a replicated join (in the distributed system community) or a map-side join (in the Hadoop community).
------	--

`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.
------	--

[JoinSelection](#) execution planning strategy uses `spark.sql.autoBroadcastJoinThreshold` property (default: `10M`) to control the size of a dataset before broadcasting it to all worker nodes when performing a join.

```

val threshold = spark.conf.get("spark.sql.autoBroadcastJoinThreshold").toInt
scala> threshold / 1024 / 1024
res0: Int = 10

val q = spark.range(100).as("a").join(spark.range(100).as("b")).where($"a.id" === $"b.id")
scala> println(q.queryExecution.logical.numberedTreeString)
00 'Filter ('a.id = 'b.id)
01 +- Join Inner
02   :- SubqueryAlias a
03   :  +- Range (0, 100, step=1, splits=Some(8))
04   +- SubqueryAlias b
05     +- Range (0, 100, step=1, splits=Some(8))

scala> println(q.queryExecution.sparkPlan.numberedTreeString)
00 BroadcastHashJoin [id#0L], [id#4L], Inner, BuildRight
01 :- Range (0, 100, step=1, splits=8)
02 +- Range (0, 100, step=1, splits=8)

scala> q.explain
== Physical Plan ==
*BroadcastHashJoin [id#0L], [id#4L], Inner, BuildRight

```

```

:- *Range (0, 100, step=1, splits=8)
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
  +- *Range (0, 100, step=1, splits=8)

spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
scala> spark.conf.get("spark.sql.autoBroadcastJoinThreshold")
res1: String = -1

scala> q.explain
== Physical Plan ==
*SortMergeJoin [id#0L], [id#4L], Inner
:- *Sort [id#0L ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(id#0L, 200)
:    +- *Range (0, 100, step=1, splits=8)
+- *Sort [id#4L ASC NULLS FIRST], false, 0
  +- ReusedExchange [id#4L], Exchange hashpartitioning(id#0L, 200)

// Force BroadcastHashJoin with broadcast hint (as function)
val qBroadcast = spark.range(100).as("a").join(broadcast(spark.range(100)).as("b")).where($"a.id" === $"b.id")
scala> qBroadcast.explain
== Physical Plan ==
*BroadcastHashJoin [id#14L], [id#18L], Inner, BuildRight
:- *Range (0, 100, step=1, splits=8)
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
  +- *Range (0, 100, step=1, splits=8)

// Force BroadcastHashJoin using SQL's BROADCAST hint
// Supported hints: BROADCAST, BROADCASTJOIN or MAPJOIN
val qBroadcastLeft = """
SELECT /*+ BROADCAST (lf) */ *
FROM range(100) lf, range(1000) rt
WHERE lf.id = rt.id
"""

scala> sql(qBroadcastLeft).explain
== Physical Plan ==
*BroadcastHashJoin [id#34L], [id#35L], Inner, BuildRight
:- *Range (0, 100, step=1, splits=8)
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
  +- *Range (0, 1000, step=1, splits=8)

val qBroadcastRight = """
SELECT /*+ MAPJOIN (rt) */ *
FROM range(100) lf, range(1000) rt
WHERE lf.id = rt.id
"""

scala> sql(qBroadcastRight).explain
== Physical Plan ==
*BroadcastHashJoin [id#42L], [id#43L], Inner, BuildRight
:- *Range (0, 100, step=1, splits=8)
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
  +- *Range (0, 1000, step=1, splits=8)

```


Multi-Dimensional Aggregation

Multi-dimensional aggregate operators are enhanced variants of `groupBy` operator that allow you to create queries for subtotals, grand totals and superset of subtotals in one go.

```
val sales = Seq(
    ("Warsaw", 2016, 100),
    ("Warsaw", 2017, 200),
    ("Boston", 2015, 50),
    ("Boston", 2016, 150),
    ("Toronto", 2017, 50)
).toDF("city", "year", "amount")

// very labor-intense
// groupBy's unioned
val groupByCityAndYear = sales
    .groupBy("city", "year") // <-- subtotals (city, year)
    .agg(sum("amount") as "amount")
val groupByCityOnly = sales
    .groupBy("city")         // <-- subtotals (city)
    .agg(sum("amount") as "amount")
    .select($"city", lit(null) as "year", $"amount") // <-- year is null
val withUnion = groupByCityAndYear
    .union(groupByCityOnly)
    .sort($"city".desc_nulls_last, $"year".asc_nulls_last)
scala> withUnion.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|    300|
| Toronto|2017|     50|
| Toronto|null|     50|
| Boston|2015|     50|
| Boston|2016|    150|
| Boston|null|    200|
+-----+-----+
```

Multi-dimensional aggregate operators are semantically equivalent to `union` operator (or SQL's `UNION ALL`) to combine single grouping queries.

```
// Roll up your sleeves!
val withRollup = sales
  .rollup("city", "year")
  .agg(sum("amount") as "amount", grouping_id() as "gid")
  .sort($"city".desc_nulls_last, $"year".asc_nulls_last)
  .filter(grouping_id() != 3)
  .select("city", "year", "amount")
scala> withRollup.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|   300|
| Toronto|2017|     50|
| Toronto|null|    50|
| Boston|2015|     50|
| Boston|2016|   150|
| Boston|null|   200|
+-----+-----+

// Be even more smarter?
// SQL only, alas.
sales.createOrReplaceTempView("sales")
val withGroupingSets = sql("""
  SELECT city, year, SUM(amount) as amount
  FROM sales
  GROUP BY city, year
  GROUPING SETS ((city, year), (city))
  ORDER BY city DESC NULLS LAST, year ASC NULLS LAST
  """)

scala> withGroupingSets.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|   300|
| Toronto|2017|     50|
| Toronto|null|    50|
| Boston|2015|     50|
| Boston|2016|   150|
| Boston|null|   200|
+-----+-----+
```

Note

It is *assumed* that using one of the operators is usually more efficient (than `union` and `groupBy`) as it gives more freedom for query optimization.

Table 1. Multi-dimensional Aggregate Operators (in alphabetical order)

Operator	Return Type	Description
cube	RelationalGroupedDataset	Calculates subtotals and a grand total for every permutation of the columns specified.
rollup	RelationalGroupedDataset	Calculates subtotals and a grand total over (ordered) combination of groups.

Beside `cube` and `rollup` multi-dimensional aggregate operators, Spark SQL supports `GROUPING SETS` clause in SQL mode only.

Note	SQL's <code>GROUPING SETS</code> is the most general aggregate "operator" and can generate the same dataset as using a simple <code>groupBy</code> , <code>cube</code> and <code>rollup</code> operators.
------	---

```

import java.time.LocalDate
import java.sql.Date
val expenses = Seq(
  ((2012, Month.DECEMBER, 12), 5),
  ((2016, Month.AUGUST, 13), 10),
  ((2017, Month.MAY, 27), 15))
  .map { case ((yy, mm, dd), a) => (LocalDate.of(yy, mm, dd), a) }
  .map { case (d, a) => (d.toString, a) }
  .map { case (d, a) => (Date.valueOf(d), a) }
  .toDF("date", "amount")
scala> expenses.show
+-----+-----+
|      date|amount|
+-----+-----+
|2012-12-12|     5|
|2016-08-13|    10|
|2017-05-27|    15|
+-----+-----+

// rollup time!
val q = expenses
  .rollup(year($"date") as "year", month($"date") as "month")
  .agg(sum("amount") as "amount")
  .sort($"year".asc_nulls_last, $"month".asc_nulls_last)
scala> q.show
+----+----+-----+
|year|month|amount|
+----+----+-----+
|2012|   12|     5|
|2012| null|     5|
|2016|    8|    10|
|2016| null|    10|
|2017|    5|    15|
|2017| null|    15|
|null| null|    30|
+----+----+-----+

```

Tip	Review the examples per operator in the following sections.
-----	---

Note	Support for multi-dimensional aggregate operators was added in [SPARK-6356] Support the ROLLUP/CUBE/GROUPING SETS/grouping() in SQLContext.
------	--

rollup Operator

```

rollup(cols: Column*): RelationalGroupedDataset
rollup(col1: String, cols: String*): RelationalGroupedDataset

```

`rollup` multi-dimensional aggregate operator is an extension of `groupBy` operator that calculates subtotals and a grand total across specified group of `n + 1` dimensions (with `n` being the number of columns as `cols` and `col1` and `1` for where values become `null`, i.e. undefined).

Note

`rollup` operator is commonly used for analysis over hierarchical data; e.g. total salary by department, division, and company-wide total.

See PostgreSQL's [7.2.4. GROUPING SETS, CUBE, and ROLLUP](#)

Note

`rollup` operator is equivalent to `GROUP BY ... WITH ROLLUP` in SQL (which in turn is equivalent to `GROUP BY ... GROUPING SETS ((a,b,c),(a,b),(a),())` when used with 3 columns: `a`, `b`, and `c`).

```
val sales = Seq(
  ("Warsaw", 2016, 100),
  ("Warsaw", 2017, 200),
  ("Boston", 2015, 50),
  ("Boston", 2016, 150),
  ("Toronto", 2017, 50)
).toDF("city", "year", "amount")

val q = sales
  .rollup("city", "year")
  .agg(sum("amount") as "amount")
  .sort($"city".desc_nulls_last, $"year".asc_nulls_last)
scala> q.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100| <- subtotal for Warsaw in 2016
| Warsaw|2017|    200|
| Warsaw|null|    300| <- subtotal for Warsaw (across years)
| Toronto|2017|     50|
| Toronto|null|     50|
| Boston|2015|     50|
| Boston|2016|    150|
| Boston|null|    200|
|   null|null|    550| <- grand total
+-----+-----+

// The above query is semantically equivalent to the following
val q1 = sales
  .groupBy("city", "year") // <- subtotals (city, year)
  .agg(sum("amount") as "amount")
val q2 = sales
  .groupBy("city")        // <- subtotals (city)
  .agg(sum("amount") as "amount")
  .select($"city", lit(null) as "year", $"amount") // <- year is null
val q3 = sales
```

```
.groupBy()           // <-- grand total
.agg(sum("amount") as "amount")
.select(lit(null) as "city", lit(null) as "year", $"amount") // <-- city and year are null
val qq = q1
.union(q2)
.union(q3)
.sort($"city".desc_nulls_last, $"year".asc_nulls_last)
scala> qq.show
+-----+---+-----+
|   city|year|amount|
+-----+---+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|    300|
| Toronto|2017|     50|
| Toronto|null|     50|
| Boston|2015|     50|
| Boston|2016|    150|
| Boston|null|    200|
|    null|null|    550|
+-----+---+-----+
```

From [Using GROUP BY with ROLLUP, CUBE, and GROUPING SETS in Microsoft's TechNet](#):

The ROLLUP, CUBE, and GROUPING SETS operators are extensions of the GROUP BY clause. The ROLLUP, CUBE, or GROUPING SETS operators can generate the same result set as when you use UNION ALL to combine single grouping queries; however, using one of the GROUP BY operators is usually more efficient.

From [PostgreSQL's 7.2.4. GROUPING SETS, CUBE, and ROLLUP](#):

References to the grouping columns or expressions are replaced by null values in result rows for grouping sets in which those columns do not appear.

From [Summarizing Data Using ROLLUP in Microsoft's TechNet](#):

The ROLLUP operator is useful in generating reports that contain subtotals and totals. (...) ROLLUP generates a result set that shows aggregates for a hierarchy of values in the selected columns.

```
// Borrowed from Microsoft's "Summarizing Data Using ROLLUP" article
val inventory = Seq(
  ("table", "blue", 124),
  ("table", "red", 223),
  ("chair", "blue", 101),
  ("chair", "red", 210)).toDF("item", "color", "quantity")

scala> inventory.show
+---+---+-----+
| item|color|quantity|
+---+---+-----+
|chair| blue|    101|
|chair| red|    210|
|table| blue|    124|
|table| red|    223|
+---+---+-----+

// ordering and empty rows done manually for demo purposes
scala> inventory.rollup("item", "color").sum().show
+---+---+-----+
| item|color|sum(quantity)|
+---+---+-----+
|chair| blue|      101|
|chair| red|      210|
|chair| null|      311|
|   |   |      |
|table| blue|      124|
|table| red|      223|
|table| null|      347|
|   |   |      |
| null| null|      658|
+---+---+-----+
```

From Hive's Cubes and Rollups:

WITH ROLLUP is used with the GROUP BY only. ROLLUP clause is used with GROUP BY to compute the aggregate at the hierarchy levels of a dimension.

GROUP BY a, b, c with ROLLUP assumes that the hierarchy is "a" drilling down to "b" drilling down to "c".

GROUP BY a, b, c, WITH ROLLUP is equivalent to GROUP BY a, b, c GROUPING SETS ((a, b, c), (a, b), (a), ()).

Note	Read up on ROLLUP in Hive's LanguageManual in Grouping Sets, Cubes, Rollups, and the GROUPING_ID Function .
------	---

```
// Borrowed from http://stackoverflow.com/a/27222655/1305344
val quarterlyScores = Seq(
  ("winter2014", "Agata", 99),
  ("winter2014", "Jacek", 97),
  ("summer2015", "Agata", 100),
  ("summer2015", "Jacek", 63),
  ("winter2015", "Agata", 97),
  ("winter2015", "Jacek", 55),
  ("summer2016", "Agata", 98),
  ("summer2016", "Jacek", 97)).toDF("period", "student", "score")

scala> quarterlyScores.show
+-----+-----+-----+
| period|student|score|
+-----+-----+-----+
|winter2014|  Agata|   99|
|winter2014|  Jacek|   97|
|summer2015|  Agata|  100|
|summer2015|  Jacek|   63|
|winter2015|  Agata|   97|
|winter2015|  Jacek|   55|
|summer2016|  Agata|   98|
|summer2016|  Jacek|   97|
+-----+-----+-----+

// ordering and empty rows done manually for demo purposes
scala> quarterlyScores.rollup("period", "student").sum("score").show
+-----+-----+-----+
| period|student|sum(score)|
+-----+-----+-----+
|winter2014|  Agata|     99|
|winter2014|  Jacek|     97|
|winter2014|    null|    196|
|           |        |        |
|summer2015|  Agata|    100|
|summer2015|  Jacek|    63|
|summer2015|    null|    163|
|           |        |        |
|winter2015|  Agata|    97|
|winter2015|  Jacek|    55|
|winter2015|    null|    152|
|           |        |        |
|summer2016|  Agata|    98|
|summer2016|  Jacek|    97|
|summer2016|    null|    195|
|           |        |        |
|    null|    null|    706|
+-----+-----+-----+
```

From PostgreSQL's 7.2.4. GROUPING SETS, CUBE, and ROLLUP:

The individual elements of a CUBE or ROLLUP clause may be either individual expressions, or sublists of elements in parentheses. In the latter case, the sublists are treated as single units for the purposes of generating the individual grouping sets.

```
// given the above inventory dataset

// using struct function
scala> inventory.rollup(struct("item", "color") as "(item,color)").sum().show
+-----+-----+
|(item,color)|sum(quantity)|
+-----+-----+
| [table,red]|      223|
| [chair,blue]|     101|
|      null|     658|
| [chair,red]|     210|
| [table,blue]|     124|
+-----+-----+

// using expr function
scala> inventory.rollup(expr("(item, color)") as "(item, color)").sum().show
+-----+-----+
|(item, color)|sum(quantity)|
+-----+-----+
| [table,red]|      223|
| [chair,blue]|     101|
|      null|     658|
| [chair,red]|     210|
| [table,blue]|     124|
+-----+-----+
```

Internally, `rollup` converts the `Dataset` into a `DataFrame` (i.e. uses `RowEncoder` as the encoder) and then creates a `RelationalGroupedDataset` (with `RollupType` group type).

Note

`Rollup` expression represents `GROUP BY ... WITH ROLLUP` in SQL in Spark's Catalyst Expression tree (after `AstBuilder` parses a structured query with aggregation).

Tip

Read up on `rollup` in [Deeper into Postgres 9.5 - New Group By Options for Aggregation](#).

cube Operator

```
cube(cols: Column*): RelationalGroupedDataset
cube(col1: String, cols: String*): RelationalGroupedDataset
```

`cube` multi-dimensional aggregate operator is an extension of `groupBy` operator that allows calculating subtotals and a grand total across all combinations of specified group of `n + 1` dimensions (with `n` being the number of columns as `cols` and `col1` and `1` for where values become `null`, i.e. undefined).

`cube` returns `RelationalGroupedDataset` that you can use to execute aggregate function or operator.

Note

`cube` is more than `rollup` operator, i.e. `cube` does `rollup` with aggregation over all the missing combinations given the columns.

```
val sales = Seq(
  ("Warsaw", 2016, 100),
  ("Warsaw", 2017, 200),
  ("Boston", 2015, 50),
  ("Boston", 2016, 150),
  ("Toronto", 2017, 50)
).toDF("city", "year", "amount")

val q = sales.cube("city", "year")
  .agg(sum("amount") as "amount")
  .sort($"city".desc_nulls_last, $"year".asc_nulls_last)
scala> q.show
+-----+----+
| city|year|amount|
+-----+----+
| Warsaw|2016|   100| <-- total in Warsaw in 2016
| Warsaw|2017|   200| <-- total in Warsaw in 2017
| Warsaw|null|   300| <-- total in Warsaw (across all years)
| Toronto|2017|    50|
| Toronto|null|   50|
| Boston|2015|    50|
| Boston|2016|   150|
| Boston|null|   200|
|   null|2015|    50| <-- total in 2015 (across all cities)
|   null|2016|   250|
|   null|2017|   250|
|   null|null|  550| <-- grand total (across cities and years)
+-----+----+
```

GROUPING SETS SQL Clause

```
GROUP BY ... GROUPING SETS (...)
```

`GROUPING SETS` clause generates a dataset that is equivalent to `union` operator of multiple `groupBy` operators.

```

val sales = Seq(
  ("Warsaw", 2016, 100),
  ("Warsaw", 2017, 200),
  ("Boston", 2015, 50),
  ("Boston", 2016, 150),
  ("Toronto", 2017, 50)
).toDF("city", "year", "amount")
sales.createOrReplaceTempView("sales")

// equivalent to rollup("city", "year")
val q = sql("""
  SELECT city, year, sum(amount) as amount
  FROM sales
  GROUP BY city, year
  GROUPING SETS ((city, year), (city), ())
  ORDER BY city DESC NULLS LAST, year ASC NULLS LAST
""")

scala> q.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|    300|
| Toronto|2017|     50|
| Toronto|null|     50|
| Boston|2015|     50|
| Boston|2016|    150|
| Boston|null|    200|
| null|null|   550| <- grand total across all cities and years
+-----+-----+


// equivalent to cube("city", "year")
// note the additional (year) grouping set
val q = sql("""
  SELECT city, year, sum(amount) as amount
  FROM sales
  GROUP BY city, year
  GROUPING SETS ((city, year), (city), (year), ())
  ORDER BY city DESC NULLS LAST, year ASC NULLS LAST
""")

scala> q.show
+-----+-----+
| city|year|amount|
+-----+-----+
| Warsaw|2016|    100|
| Warsaw|2017|    200|
| Warsaw|null|    300|
| Toronto|2017|     50|
| Toronto|null|     50|
| Boston|2015|     50|
| Boston|2016|    150|

```

```
| Boston|null| 200|
| null|2015| 50| <-- total across all cities in 2015
| null|2016| 250| <-- total across all cities in 2016
| null|2017| 250| <-- total across all cities in 2017
| null|null| 550|
+-----+-----+
```

Internally, `GROUPING SETS` clause is parsed in [withAggregation](#) parsing handler (in `AstBuilder`) and becomes a [GroupingSets](#) logical operator internally.

Rollup GroupingSet with CodegenFallback Expression (for rollup Operator)

```
Rollup(groupByExprs: Seq[Expression])
extends GroupingSet
```

`Rollup` expression represents `rollup` operator in Spark's Catalyst Expression tree (after `AstBuilder` parses a structured query with aggregation).

Note	<code>GroupingSet</code> is an Expression with CodegenFallback support.
------	---

UserDefinedAggregateFunction — User-Defined Aggregate Functions (UDAFs)

Creating Column for UDAF — `apply` Method

Caution

[FIXME](#)

Creating Column for UDAF with Distinct Values — `distinct` Method

Caution

[FIXME](#)

Dataset Caching and Persistence

Table 1. Caching Operators (in alphabetical order)

Operator	Description
cache	
persist	
unpersist	

```
// Cache Dataset -- it is lazy
scala> val df = spark.range(1).cache
df: org.apache.spark.sql.Dataset[Long] = [id: bigint]

// Trigger caching
scala> df.show
+---+
| id|
+---+
|  0|
+---+

// Visit http://localhost:4040/storage to see the Dataset cached. It should.

// You may also use queryExecution or explain to see InMemoryRelation
// InMemoryRelation is used for cached queries
scala> df.queryExecution.withCachedData
res0: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
InMemoryRelation [id#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 repl
icas)
  +- *Range (0, 1, step=1, splits=Some(8))

// Use the cached Dataset in another query
// Notice InMemoryRelation in use for cached queries
scala> df.withColumn("newId", 'id).explain(extended = true)
== Parsed Logical Plan ==
'Project [* , 'id AS newId#16]
+- Range (0, 1, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint, newId: bigint
Project [id#0L, id#0L AS newId#16L]
+- Range (0, 1, step=1, splits=Some(8))

== Optimized Logical Plan ==
Project [id#0L, id#0L AS newId#16L]
+- InMemoryRelation [id#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 r
```

```

replicas)
    +- *Range (0, 1, step=1, splits=Some(8))

== Physical Plan ==
*Project [id#0L, id#0L AS newId#16L]
+- InMemoryTableScan [id#0L]
    +- InMemoryRelation [id#0L], true, 10000, StorageLevel(disk, memory, deserialize
d, 1 replicas)
        +- *Range (0, 1, step=1, splits=Some(8))

// Clear in-memory cache using SQL
// Equivalent to spark.catalog.clearCache
scala> sql("CLEAR CACHE").collect
res1: Array[org.apache.spark.sql.Row] = Array()

// Visit http://localhost:4040/storage to confirm the cleaning

```

You can also use SQL's `CACHE TABLE [tableName]` to cache `tableName` table in memory. Unlike `cache` and `persist` operators, `CACHE TABLE` is an eager operation which is executed as soon as the statement is executed.

```
sql("CACHE TABLE [tableName]")
```

You could however use `LAZY` keyword to make caching lazy.

Note

```
sql("CACHE LAZY TABLE [tableName]")
```

Use SQL's `REFRESH TABLE [tableName]` to refresh a cached table.

Use SQL's `UNCACHE TABLE (IF EXISTS)? [tableName]` to remove a table from cache.

Use SQL's `CLEAR CACHE` to remove all tables from cache.

Note

Be careful what you cache, i.e. what Dataset is cached, as it gives different queries cached.

```
// cache after range(5)
val q1 = spark.range(5).cache.filter($"id" % 2 === 0).select("id")
scala> q1.explain
== Physical Plan ==
*Filter ((id#0L % 2) = 0)
+- InMemoryTableScan [id#0L], [(id#0L % 2) = 0]
    +- InMemoryRelation [id#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
        +- *Range (0, 5, step=1, splits=8)

// cache at the end
val q2 = spark.range(1).filter($"id" % 2 === 0).select("id").cache
scala> q2.explain
== Physical Plan ==
InMemoryTableScan [id#17L]
    +- InMemoryRelation [id#17L], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
        +- *Filter ((id#17L % 2) = 0)
            +- *Range (0, 1, step=1, splits=8)
```

Tip

You can check whether a Dataset was cached or not using the following code:

```
scala> :type q2
org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]

val cache = spark.sharedState.cacheManager
scala> cache.lookupCachedData(q2.queryExecution.logical).isDefined
res0: Boolean = false
```

SQL's CACHE TABLE

SQL's `CACHE TABLE` corresponds to requesting the session-specific `catalog` to [caching the table](#).

Internally, `CACHE TABLE` becomes `CacheTableCommand` runnable command that...[FIXME](#)

Caching Dataset— `cache` Method

```
cache(): this.type
```

`cache` merely executes the no-argument [persist](#) method.

```
val ds = spark.range(5).cache
```

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` and returns it.

Internally, `persist` requests `CacheManager` [to cache the query](#) (that is accessible through `SharedState` of the current [SparkSession](#)).

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

Unpersisting Dataset— `unpersist` Method

```
unpersist(blocking: Boolean): this.type
```

`unpersist` uncache the `Dataset` possibly by `blocking` the call.

Internally, `unpersist` requests `CacheManager` [to uncache the query](#).

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

RelationalGroupedDataset — Untyped Row-based Grouping

`RelationalGroupedDataset` is an interface to compute aggregates over a grouped collection of records in a [DataFrame](#).

`RelationalGroupedDataset` is a result of executing the following aggregate operators:

- [groupBy](#)
- [rollup](#)
- [cube](#)
- [pivot](#) (after [groupBy](#) operator)

Table 1. RelationalGroupedDataset's Aggregate Operators (in alphabetical order)

Operator	Description
agg	
avg	
count	
max	
mean	
min	
pivot	Pivots on a column (with new columns per distinct value)
sum	

Note	<code>spark.sql.retainGroupColumns</code> property controls whether to retain columns used for aggregation or not (in <code>RelationalGroupedDataset</code> operators). Enabled by default.
------	---

Creating DataFrame from Aggregate Expressions — `toDF` Internal Method

```
toDF(aggExprs: Seq[Expression]): DataFrame
```

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

Internally, `toDF` branches off per group type.

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

For `PivotType`, `toDF` creates a DataFrame with `Pivot` unary logical operator.

Creating RelationalGroupedDataset Instance

`RelationalGroupedDataset` takes the following when created:

- `DataFrame`
- Grouping [expressions](#)
- Group type (to indicate what operation has created it), i.e. `GroupByType`, `CubeType`, `RollupType`, `PivotType`

agg Operator

```
agg(aggExpr: (String, String), aggExprs: (String, String)*): DataFrame
agg(exprs: Map[String, String]): DataFrame
agg(expr: Column, exprs: Column*): DataFrame
```

pivot Operator

```
pivot(pivotColumn: String): RelationalGroupedDataset (1)
pivot(pivotColumn: String, values: Seq[Any]): RelationalGroupedDataset (2)
```

1. Selects distinct and sorted values on `pivotColumn` and calls the other `pivot` (that results in 3 extra "scanning" jobs)
2. Preferred as more efficient because the unique values are already provided

`pivot` pivots on a `pivotColumn` column, i.e. adds new columns per distinct values in `pivotColumn`.

Note	<code>pivot</code> is only supported after <code>groupBy</code> operation.
------	--

Note	Only one <code>pivot</code> operation is supported on a <code>RelationalGroupedDataset</code> .
------	---

```

val visits = Seq(
  (0, "Warsaw", 2015),
  (1, "Warsaw", 2016),
  (2, "Boston", 2017)
).toDF("id", "city", "year")

val q = visits
  .groupBy("city") // <-- rows in pivot table
  .pivot("year")   // <-- columns (unique values queried)
  .count()         // <-- values in cells
scala> q.show
+-----+-----+-----+
| city|2015|2016|2017|
+-----+-----+-----+
|Warsaw|    1|    1| null|
|Boston| null| null|    1|
+-----+-----+-----+

scala> q.explain
== Physical Plan ==
HashAggregate(keys=[city#8], functions=[pivotfirst(year#9, count(1) AS `count`#222L, 2015, 2016, 2017, 0, 0)])
+- Exchange hashpartitioning(city#8, 200)
  +- HashAggregate(keys=[city#8], functions=[partial_pivotfirst(year#9, count(1) AS `count`#222L, 2015, 2016, 2017, 0, 0)])
    +- *HashAggregate(keys=[city#8, year#9], functions=[count(1)])
      +- Exchange hashpartitioning(city#8, year#9, 200)
        +- *HashAggregate(keys=[city#8, year#9], functions=[partial_count(1)])
          +- LocalTableScan [city#8, year#9]

scala> visits
  .groupBy('city)
  .pivot("year", Seq("2015")) // <-- one column in pivot table
  .count
  .show
+-----+
| city|2015|
+-----+
|Warsaw|    1|
|Boston| null|
+-----+

```

Important Use `pivot` with a list of distinct values to pivot on so Spark does not have to compute the list itself (and run three extra "scanning" jobs).

Completed Jobs (5)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
4	show at <console>:28	2017/04/19 09:58:23	74 ms	1/1 (2 skipped)	75/75 (203 skipped)
3	show at <console>:28	2017/04/19 09:58:23	0.1 s	1/1 (2 skipped)	100/100 (203 skipped)
2	show at <console>:28	2017/04/19 09:58:23	22 ms	1/1 (2 skipped)	20/20 (203 skipped)
1	show at <console>:28	2017/04/19 09:58:23	11 ms	1/1 (2 skipped)	4/4 (203 skipped)
0	show at <console>:28	2017/04/19 09:58:22	1 s	3/3	204/204

Figure 1. pivot in web UI (Distinct Values Defined Explicitly)

Completed Jobs (8)					
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
7	show at <console>:29	2017/04/19 09:51:20	36 ms	1/1 (2 skipped)	75/75 (203 skipped)
6	show at <console>:29	2017/04/19 09:51:20	94 ms	1/1 (2 skipped)	100/100 (203 skipped)
5	show at <console>:29	2017/04/19 09:51:20	12 ms	1/1 (2 skipped)	20/20 (203 skipped)
4	show at <console>:29	2017/04/19 09:51:20	6 ms	1/1 (2 skipped)	4/4 (203 skipped)
3	show at <console>:29	2017/04/19 09:51:19	0.6 s	3/3	204/204
2	pivot at <console>:28	2017/04/19 09:51:19	8 ms	1/1 (2 skipped)	3/3 (203 skipped)
1	pivot at <console>:28	2017/04/19 09:51:18	0.4 s	2/2 (1 skipped)	201/201 (3 skipped)
0	pivot at <console>:28	2017/04/19 09:51:17	0.8 s	2/2	203/203

Figure 2. pivot in web UI — Three Extra Scanning Jobs Due to Unspecified Distinct Values

Note	<code>spark.sql.pivotMaxValues</code> (default: 10000) controls the maximum number of (distinct) values that will be collected without error (when doing <code>pivot</code> without specifying the values for the pivot column).
------	--

Internally, `pivot` creates a `RelationalGroupedDataset` with `PivotType` group type and `pivotColumn` resolved using the `DataFrame`'s columns with `values as Literal` expressions.

Note	<p><code>toDF</code> internal method maps <code>PivotType</code> group type to a <code>DataFrame</code> with <code>Pivot</code> unary logical operator.</p> <pre>scala> q.queryExecution.logical res0: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan = Pivot [city#8], year#9: int, [2015, 2016, 2017], [count(1) AS count#24L] +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9] +- LocalRelation [_1#3, _2#4, _3#5]</pre>
------	---

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 [tables](#) using [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 [streaming](#) (part of Spark 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 — Reading Datasets from External Data Sources

`DataFrameReader` is an interface to read datasets from external data sources, e.g. [files](#), [Hive tables](#), [JDBC](#) or [Dataset\[String\]](#), into untyped `DataFrames` (mostly) or typed `Datasets`.

`DataFrameReader` is available using [SparkSession.read](#).

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

import org.apache.spark.sql.DataFrameReader
val r: DataFrameReader = spark.read
```

`DataFrameReader` supports many [file formats](#) natively and offers the [interface to define custom file formats](#).

Note	<code>DataFrameReader</code> assumes parquet file format by default that you can change using spark.sql.sources.default property.
------	---

After you have described the [reading pipeline](#) to read datasets from an external data source, you eventually trigger the loading using format-agnostic [load](#) or format-specific (e.g. [json](#), [csv](#)) operators that create untyped `DataFrames`.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

import org.apache.spark.sql.DataFrame

// Using format-agnostic load operator
val csvs: DataFrame = spark
  .read
  .format("csv")
  .option("header", true)
  .option("inferSchema", true)
  .load("*.csv")

// Using format-specific load operator
val jsons: DataFrame = spark
  .read
  .json("metrics/*.json")
```

(New in Spark 2.0) `DataFrameReader` can read text files using `textFile` methods that return typed `Datasets`.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

import org.apache.spark.sql.Dataset
val lines: Dataset[String] = spark
  .read
  .textFile("README.md")
```

(New in Spark 2.2) `DataFrameReader` can load datasets from `Dataset[String]` (with lines being complete "files") using format-specific `csv` and `json` operators.

```

val csvLine = "0,Warsaw,Poland"

import org.apache.spark.sql.Dataset
val cities: Dataset[String] = Seq(csvLine).toDS
scala> cities.show
+-----+
|      value|
+-----+
|0,Warsaw,Poland|
+-----+

// Define schema explicitly (as below)
// or
// option("header", true) + option("inferSchema", true)
import org.apache.spark.sql.types.StructType
val schema = new StructType()
  .add($"id".long.copy(nullable = false))
  .add($"city".string)
  .add($"country".string)
scala> schema.printTreeString
root
 |-- id: long (nullable = false)
 |-- city: string (nullable = true)
 |-- country: string (nullable = true)

import org.apache.spark.sql.DataFrame
val citiesDF: DataFrame = spark
  .read
  .schema(schema)
  .csv(cities)
scala> citiesDF.show
+---+---+---+
| id| city|country|
+---+---+---+
| 0|Warsaw| Poland|
+---+---+---+

```

Defining 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 [define your own custom file formats](#).

Defining Schema — `schema` method

```
schema(schema: StructType): DataFrameReader
```

You can specify a `schema` of the input data source.

```
import org.apache.spark.sql.types.StructType
val schema = new StructType()
  .add($"id".long.copy(nullable = false))
  .add($"city".string)
  .add($"country".string)

scala> schema.printTreeString
root
| -- id: long (nullable = false)
| -- city: string (nullable = true)
| -- country: string (nullable = true)

import org.apache.spark.sql.DataFrameReader
val r: DataFrameReader = spark.read.schema(schema)
```

Note

Some formats can infer schema from datasets, e.g. `csv`, using [options](#).

Tip

Read up on [Schema](#).

Defining Loading Options — `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 as of Spark 2.0

You can also use `options` method to describe different options in a single `Map`.

```
options(options: scala.collection.Map[String, String]): DataFrameReader
```

Loading Data from Data Sources with Multiple Files Support — `load` Method

```
load(): DataFrame
load(path: String): DataFrame
load(paths: String*): DataFrame
```

`load` loads a data from data sources that support multiple `paths` and represents it as an untyped [DataFrame](#).

Internally, `load` creates a [DataSource](#) (for the current [SparkSession](#), a user-specified [schema](#), a source [format](#) and [options](#)). It then immediately [resolves it](#) and [converts](#) [BaseRelation](#) into a [DataFrame](#).

Loading Datasets from Files (into DataFrames) Using Format-Specific Load Operators

`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:

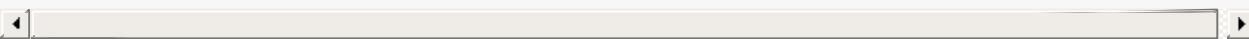
- `none` or `uncompressed`
 - `snappy` - the default codec in Spark **2.0.0**.
 - `gzip` - the default codec in Spark before **2.0.0**
 - `lzo`

```
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 codecs are uncompressed, gzip, lzo, snappy, none.
   at org.apache.spark.sql.execution.datasources.parquet.ParquetOptions.<init>(ParquetOptions.scala:43)
   at org.apache.spark.sql.execution.datasources.parquet.DefaultSource.prepareWrite(ParquetRelation.scala:77)
   at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$run$1$$anonfun$4.apply(InsertIntoHadoopFsRelation.scala:122)
   at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$run$1
```

```
n$1$$anonfun$4.apply(InsertIntoHadoopFsRelation.scala:122)
  at org.apache.spark.sql.execution.datasources.BaseWriterContainer.driverSideSetup(WriterContainer.scala:103)
  at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$run$1.apply$mcV$sp(InsertIntoHadoopFsRelation.scala:141)
  at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$run$1.apply(InsertIntoHadoopFsRelation.scala:116)
  at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$run$1.apply(InsertIntoHadoopFsRelation.scala:116)
  at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation$$anonfun$run$1.apply(InsertIntoHadoopFsRelation.scala:116)
  at org.apache.spark.sql.execution.SQLExecution$.withNewExecutionId(SQLExecution.scala:53)
  at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelation.run(InsertIntoHadoopFsRelation.scala:116)
  at org.apache.spark.sql.execution.command.ExecutedCommand.sideEffectResult$lzycompute(commands.scala:61)
  at org.apache.spark.sql.execution.command.ExecutedCommand.sideEffectResult(commands.scala:59)
  at org.apache.spark.sql.execution.command.ExecutedCommand.doExecute(commands.scala:73)
)
  at org.apache.spark.sql.execution.SparkPlan$$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$1.apply(SparkPlan.scala:137)
  at org.apache.spark.rdd.RDDOperationScope$.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.scala: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
```

Loading Datasets from Tables (into DataFrames) — table Method

```
table(tableName: String): DataFrame
```

`table` loads `tableName` table into an untyped [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?

Loading Data From External Table using JDBC — `jdbc` Method

```
jdbc(url: String, table: String, properties: 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` loads data from an external table using JDBC and represents it as an untyped [DataFrame](#).

Table 1. Options for JDBC Data Source (in alphabetical order)

Option	Description
<code>batchsize</code>	The minimum value is <code>1</code> Defaults to <code>1000</code>
<code>createTableColumnTypes</code>	

<code>createTableOptions</code>	
<code>dbtable</code>	(required)
<code>driver</code>	(recommended) JDBC driver's class name. When defined, the class will get registered with Java's java.sql.DriverManager
<code>fetchsize</code>	Defaults to <code>0</code>
<code>isolationLevel</code>	One of the following: <ul style="list-style-type: none">• <code>NONE</code>• <code>READ_UNCOMMITTED</code> (default)• <code>READ_COMMITTED</code>• <code>REPEATABLE_READ</code>• <code>SERIALIZABLE</code>
<code>lowerBound</code>	Lower bound of partition column
<code>numPartitions</code>	Number of partitions
<code>partitionColumn</code>	Name of the column used to partition dataset (using a <code>JDBCPartitioningInfo</code>). Used in <code>JdbcRelationProvider</code> to create a JDBCRelation (with proper <code>JDBCPartitions</code> with <code>WHERE</code> clause). When defined, <code>lowerBound</code> , <code>upperBound</code> and <code>numPartitions</code> options are required. When undefined, <code>lowerBound</code> and <code>upperBound</code> have to be undefined.
<code>truncate</code>	(used only for writing) Enables table truncation. Defaults to <code>false</code>
<code>upperBound</code>	Upper bound of the partition column
<code>url</code>	(required)

Internally, `jdbc` creates a `JDBCOptions` from `url`, `table` and `extraOptions` with `connectionProperties`.

`jdbc` then creates one `JDBCPartition` per `predicates`.

In the end, `jdbc` requests the `SparkSession` to [create a `DataFrame`](#) for a `JDBCRelation` (given `JDBCPartitions` and `JDBCOptions` created earlier).

Note

`jdbc` does not support a custom `schema` and reports an `AnalysisException` if defined:

```
User specified schema not supported with `'[jdbc]`
```

Note

`jdbc` method uses `java.util.Properties` (and appears overly Java-centric). Use [format\("jdbc"\)](#) instead.

Tip

Review the exercise [Creating DataFrames from Tables using JDBC and PostgreSQL](#).

Loading Datasets From Text Files — `textFile` Method

```
textFile(path: String): Dataset[String]
textFile(paths: String*): Dataset[String]
```

`textFile` loads one or many text files into a typed `Dataset[String]`.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

import org.apache.spark.sql.Dataset
val lines: Dataset[String] = spark
  .read
  .textFile("README.md")
```

Note

`textFile` are similar to `text` family of methods in that they both read text files but `text` methods return untyped `DataFrame` while `textFile` return typed `Dataset[String]`.

Internally, `textFile` passes calls on to `text` method and [selects](#) the only `value` column before it applies `Encoders.STRING encoder`.

Creating DataFrameReader Instance

`DataFrameReader` takes the following when created:

- `SparkSession`

DataFrameWriter

`DataFrameWriter` is an interface to persist a `Dataset` to an external storage system in a batch fashion.

Tip

Read [DataStreamWriter](#) for streamed writing.

You use `write` method on a `Dataset` to access `DataFrameWriter`.

```
import org.apache.spark.sql.DataFrameWriter

val nums: Dataset[Long] = ...
val writer: DataFrameWriter[Row] = nums.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 = "nums")

// Save dataset in JSON format
writer.format("json").save(path = "nums-json")
```

In the end, you trigger the actual saving of the content of a `Dataset` using `save` method.

```
writer.save
```

Note

Interestingly, a `DataFrameWriter` is really a type constructor in Scala. It keeps a reference to a source `DataFrame` during its lifecycle (starting right from the moment it was created).

runCommand Internal Method

```
runCommand
(session: SparkSession, name: String)
(command: LogicalPlan): Unit
```

Caution

[FIXME](#)

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
<code>source</code>	<code>format</code>
<code>mode</code>	<code>mode</code>
<code>extraOptions</code>	<code>option</code> , <code>options</code> , <code>save</code>
<code>partitioningColumns</code>	<code>partitionBy</code>
<code>bucketColumnNames</code>	<code>bucketBy</code>
<code>numBuckets</code>	<code>bucketBy</code>
<code>sortColumnNames</code>	<code>sortBy</code>

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	<code>Ignore</code>	Do nothing
yes	<code>ErrorIfExists</code>	Throws a <code>AnalysisException</code> exception with <code>Table [tableIdent] already exists.</code> error message.
<i>anything</i>	<i>anything</i>	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("int")
sql("show tables").show
```

Saving DataFrame — `save` Method

`save(): Unit`

`save` saves the result of a structured query (the content of a [Dataset](#)) to a data source.

Internally, `save` runs a [SaveIntoDataSourceCommand](#) runnable command under the name `save`.

Note	<code>save</code> does not support saving to Hive (when <code>source</code> is <code>hive</code>) and bucketing.
------	---

Caution	FIXME What does <code>bucketing</code> mean? What about <code>assertNotBucketed</code> ?
---------	--

Caution	FIXME What is <code>partitioningColumns</code> ?
---------	--

Note	<code>save</code> uses <code>source</code> , <code>partitioningColumns</code> , <code>extraOptions</code> , and <code>mode</code> internal properties.
------	--

jdbc Method

```
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

<code>partitionBy(colNames: String*): DataFrameWriter[T]</code>

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

Defining Write Behaviour Per Sink's Existence (aka Save Mode) — mode Method

<code>mode(saveMode: String): DataFrameWriter[T]</code>
<code>mode(saveMode: SaveMode): DataFrameWriter[T]</code>

`mode` defines the behaviour of `save` when an external file or table (Spark writes to) already exists, i.e. `SaveMode` .

Table 3. Types of SaveMode (in alphabetical order)

Name	Description
Append	Records are appended to existing data.
ErrorIfExists	Exception is thrown.
Ignore	Do not save the records and not change the existing data in any way.
Overwrite	Existing data is overwritten by new records.

Writer Configuration — option and options Methods

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

Writing DataFrames to Files

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

Specifying Alias or Fully-Qualified Class Name of DataSource — `format` Method

Caution	FIXME Compare to DataFrameReader.
---------	---

Parquet

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

Note	Parquet is the default data source format.
------	--

`insertInto` Method

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

DataSource — Pluggable Data Provider Framework

`DataSource` is among the main components of **Data Source API** in Spark SQL (together with `DataFrameReader` for loading datasets, `DataFrameWriter` for saving datasets and `StreamSourceProvider` for creating streaming sources).

`DataSource` models a **pluggable data provider framework** with [extension points](#) for Spark SQL integrators to expand the list of supported external data sources in Spark SQL.

Table 1. DataSource's Provider (and Format) Contracts

Extension Point	Description
<code>CreatableRelationProvider</code>	Data source that saves the result of a structured query per save mode and returns the schema
<code>FileFormat</code>	Used in: <ul style="list-style-type: none"> <code>sourceSchema</code> for streamed reading <code>write</code> for writing a <code>DataFrame</code> to a <code>DataSource</code> (as part of creating a table as <code>select</code>)
<code>RelationProvider</code>	Data source that supports schema inference and can be accessed using SQL's <code>USING</code> clause
<code>SchemaRelationProvider</code>	Data source that requires a user-defined schema
<code>StreamSourceProvider</code>	Used in: <ul style="list-style-type: none"> <code>sourceSchema</code> and <code>createSource</code> for streamed reading <code>createSink</code> for streamed writing <code>resolveRelation</code> for resolved <code>BaseRelation</code>.

As a user, you interact with `DataSource` by `DataFrameReader` (when you execute `spark.read` or `spark.readStream`) or SQL's `CREATE TABLE USING`.

```
// Batch reading
val people: DataFrame = spark.read
  .format("csv")
  .load("people.csv")

// Streamed reading
val messages: DataFrame = spark.readStream
  .format("kafka")
  .option("subscribe", "topic")
  .option("kafka.bootstrap.servers", "localhost:9092")
  .load
```

`DataSource` 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.

Note	Data source is also called a table provider .
------	--

Writing DataFrame to Data Source per Save Mode Followed by Reading Rows Back (as BaseRelation)

— `writeAndRead` Method

```
writeAndRead(mode: SaveMode, data: DataFrame): BaseRelation
```

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

Note	<code>writeAndRead</code> is used exclusively when CreateDataSourceTableAsSelectCommand is executed.
------	--

providingClass Property

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

Writing DataFrame to Data Source Per Save Mode

— `write` Method

```
write(mode: SaveMode, data: DataFrame): BaseRelation
```

`write` writes the result of executing a structured query (as [DataFrame](#)) to a data source per `save mode`.

Internally, `write` looks up the data source and branches off per `providingClass`.

Table 2. write's Branches per Supported providingClass (in execution order)

providingClass	Description
<code>CreatableRelationProvider</code>	Executes <code>CreatableRelationProvider.createRelation</code>
<code>FileFormat</code>	<code>writelnFileFormat</code>
<i>others</i>	Reports a <code>RuntimeException</code>

Note	<code>write</code> does not support the internal <code>CalendarIntervalType</code> in the schema of <code>data</code> <code>DataFrame</code> and throws a <code>AnalysisException</code> when there is one.
Note	<code>write</code> is used exclusively when <code>SaveIntoDataSourceCommand</code> is executed.

writelnFileFormat Internal Method

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

For `FileFormat` data sources, `write` takes all `paths` and `path` option and makes sure that there is only one.

Note	<code>write</code> uses Hadoop's <code>Path</code> to access the <code>FileSystem</code> and calculate the qualified output path.
------	---

`write` does `PartitioningUtils.validatePartitionColumn`.

Caution	FIXME What is <code>PartitioningUtils.validatePartitionColumn</code> for?
---------	---

When appending to a table, ...[FIXME](#)

In the end, `write` (for a `FileFormat` data source) prepares a `InsertIntoHadoopFsRelationCommand` logical plan with executes it.

Caution	FIXME Is <code>toRdd</code> a job execution?
---------	--

createSource Method

```
createSource(metadataPath: String): Source
```

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

createSink Method

Caution

[FIXME](#)

Creating DataSource Instance

```
class DataSource(
    sparkSession: SparkSession,
    className: String,
    paths: Seq[String] = Nil,
    userSpecifiedSchema: Option[StructType] = None,
    partitionColumns: Seq[String] = Seq.empty,
    bucketSpec: Option[BucketSpec] = None,
    options: Map[String, String] = Map.empty,
    catalogTable: Option[CatalogTable] = None)
```

When being created, `DataSource` first looks up the providing class given `className` (considering it an alias or a fully-qualified class name) and computes the name and schema of the data source.

Note

`DataSource` does the initialization lazily on demand and only once.

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. `FileFormat` and Structured Streaming's `StreamSourceProvider` data sources.

Internally, `sourceSchema` first creates an instance of the data source and...

Caution

[FIXME](#) Finish...

For Structured Streaming's `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 <code>DataSource.sourceSchema</code> and <code>DataSource.createSource</code> when <code>FileFormat</code> is processed.
------	--

lookupDataSource Internal Method

```
lookupDataSource(provider0: String): Class[]
```

Internally, `lookupDataSource` first searches the classpath for available `DataSourceRegister` providers (using Java's `ServiceLoader.load` method) to find the requested data source by short name (alias), e.g. `parquet` or `kafka`.

If a `DataSource` could not be found by short name, `lookupDataSource` tries to load the class given the input `provider0` or its variant `provider0.DefaultSource` (with `.DefaultSource` suffix).

Note	You can reference your own custom <code>DataSource</code> in your code by <code>DataFrameWriter.format</code> method which is the alias or fully-qualified class name.
------	--

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.
```

Creating BaseRelation — `resolveRelation` Method

```
resolveRelation(checkFilesExist: Boolean = true): BaseRelation
```

`resolveRelation` resolves (i.e. creates) a [BaseRelation](#).

Internally, `resolveRelation` creates an instance of [providingClass](#) (of a `DataSource`) and branches off per its type, i.e. [SchemaRelationProvider](#), [RelationProvider](#) or [FileFormat](#).

Table 3. Resolving BaseRelation per Providers

Provider	Behaviour
<code>SchemaRelationProvider</code>	Executes SchemaRelationProvider.createRelation with the provided schema
<code>RelationProvider</code>	Executes RelationProvider.createRelation
<code>FileFormat</code>	Creates a HadoopFsRelation

	<p><code>resolveRelation</code> is used when:</p> <ul style="list-style-type: none"> <code>DataSource</code> writes and reads the result of a structured query (when providingClass is of type <code>FileFormat</code>) <code>DataFrameReader</code> loads data from a data source that supports multiple paths <code>TextInputCSVDataSource</code> and <code>TextInputJsonDataSource</code> are requested to infer schema
Note	<ul style="list-style-type: none"> <code>CreateDataSourceTableCommand</code> runnable command is executed <code>CreateTempViewUsing</code> runnable command is executed <code>FindDataSourceTable</code> does <code>readDataSourceTable</code> <code>ResolvesQLOnFile</code> converts a logical plan (when providingClass is of type <code>FileFormat</code>) <code>HiveMetastoreCatalog</code> does <code>convertToLogicalRelation</code> Structured Streaming's <code>FileStreamSource</code> creates batches of records

CreatableRelationProvider — Data Sources That Save Rows Per Save Mode

`CreatableRelationProvider` is a [contract](#) for data source providers that [save the result of a structured query per save mode](#) and return the [schema](#).

Note

A structured query is a [DataFrame](#) while the result are [Rows](#).

`CreatableRelationProvider` is used when:

- `DataSource` is requested to [write the result of a structured query to data source per save mode](#) (after `DataFrameWriter` is requested to [save](#))
- `DataSource` is requested to [write the result of a structured query to data source per save mode followed by reading rows back](#) (after `DataFrameWriter` is requested to [save](#) to a non-Hive table or for [Create Table As Select SQL statements](#))

Table 1. `CreatableRelationProvider`'s Known Implementations

Name	Description
JdbcRelationProvider	
KafkaSourceProvider	

CreatableRelationProvider Contract

```
package org.apache.spark.sql.sources

trait CreatableRelationProvider {
  def createRelation(
    sqlContext: SQLContext,
    mode: SaveMode,
    parameters: Map[String, String],
    data: DataFrame): BaseRelation
}
```

Table 2. CreatableRelationProvider Contract

Method	Description
<code>createRelation</code>	<p>Saves the result of a structured query to a target relation per save mode and parameters. Creates a BaseRelation to describe the scheme.</p> <p>The save mode specifies what happens when the destination already exists:</p> <ul style="list-style-type: none">• Append• ErrorIfExists• Ignore• Overwrite

RelationProvider — Data Sources With Schema Inference

`RelationProvider` is a [contract](#) for [data source providers](#) that support schema inference (and also can be accessed using SQL's USING clause, i.e. in `CREATE TEMPORARY VIEW` and `DROP DATABASE` DDL operators).

Note

Schema inference is also called **schema discovery**.

`RelationProvider` is used exclusively when:

- `DataSource` [creates a BaseRelation](#) (with no user-defined schema or the user-defined schema matches `RelationProvider`'s)

Note

[BaseRelation](#) models a collection of tuples from an external data source with a schema.

Table 1. RelationProvider's Known Implementations

Name	Description
JdbcRelationProvider	
KafkaSourceProvider	

Tip

Use [SchemaRelationProvider](#) for relation providers that require a user-defined schema.

RelationProvider Contract

```
package org.apache.spark.sql.sources

trait RelationProvider {
  def createRelation(
    sqlContext: SQLContext,
    parameters: Map[String, String]): BaseRelation
}
```

Table 2. RelationProvider Contract

Method	Description
<code>createRelation</code>	Accepts optional parameters (from SQL's <code>OPTIONS</code> clause)

SchemaRelationProvider — Data Sources With Mandatory User-Defined Schema

`SchemaRelationProvider` is a contract for [data source providers](#) that require a user-defined schema.

`SchemaRelationProvider` is used exclusively when:

- `DataSource` is requested for a `BaseRelation` for a data source

Note	<code>BaseRelation</code> models a collection of tuples from an external data source with a schema.
------	---

Tip	Use RelationProvider for data source providers with schema inference.
-----	---

Tip	Use both <code>SchemaRelationProvider</code> and RelationProvider if a data source can support both schema inference and user-defined schemas.
-----	--

SchemaRelationProvider Contract

```
package org.apache.spark.sql.sources

trait SchemaRelationProvider {
  def createRelation(
    sqlContext: SQLContext,
    parameters: Map[String, String],
    schema: StructType): BaseRelation
}
```

Table 1. SchemaRelationProvider Contract

Method	Description
<code>createRelation</code>	Creates a <code>BaseRelation</code> for the <code>parameters</code> and user-defined <code>schema</code>

DataSourceRegister

`DataSourceRegister` is an interface to register [DataSources](#) under their `shortName` aliases (to [look them 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.

CSVFileFormat

CSVFileFormat is a TextBasedFileFormat that registers DataSources under the name csv .

```
spark.read.csv("people.csv")  
  
// or the same as above in a more verbose way  
spark.read.format("csv").load("people.csv")
```

JdbcRelationProvider

`JdbcRelationProvider` is a [CreatableRelationProvider](#) and [RelationProvider](#) that handles data sources for `jdbc` format.

```
val table = spark.read.jdbc(...)

// or in a more verbose way
val table = spark.read.format("jdbc").load(...)
```

Creating JDBCRelation — `createRelation` Method (from [RelationProvider](#))

```
createRelation(
    sqlContext: SQLContext,
    parameters: Map[String, String]): BaseRelation
```

`createRelation` creates a `JDBCPartitioningInfo` (using [JDBCOptions](#) and the input parameters that correspond to [Options for JDBC Data Source](#)).

Note	<code>createRelation</code> uses <code>partitionColumn</code> , <code>lowerBound</code> , <code>upperBound</code> and <code>numPartitions</code> .
------	--

In the end, `createRelation` creates a `JDBCRelation` using [column partitions](#) (and [JDBCOptions](#)).

Note	<code>createRelation</code> is a part of RelationProvider Contract .
------	--

Creating JDBCRelation After Preparing Table in Database — `createRelation` Method (from [CreatableRelationProvider](#))

```
createRelation(
    sqlContext: SQLContext,
    mode: SaveMode,
    parameters: Map[String, String],
    df: DataFrame): BaseRelation
```

Internally, `createRelation` creates a `JDBCOptions` (from the input `parameters`).

`createRelation` reads `caseSensitiveAnalysis` (using the input `sqlContext`).

`createRelation` checks whether the table (given `dbtable` and `url` `options` in the input parameters) exists.

Note

`createRelation` uses a database-specific `JdbcDialect` to check whether a table exists.

`createRelation` branches off per whether the table already exists in the database or not.

If the table **does not** exist, `createRelation` creates the table (by executing `CREATE TABLE` with `createTableColumnTypes` and `createTableOptions` `options` from the input parameters) and saves the records to the database in a single transaction.

If however the table **does** exist, `createRelation` branches off per [SaveMode](#) (see the following [createRelation and SaveMode](#)).

Table 1. `createRelation` and `SaveMode` (in alphabetical order)

Name	Description
Append	Saves the records to the table.
ErrorIfExists	Throws a <code>AnalysisException</code> with the message: <code>Table or view '[table]' already exists. SaveMode: ErrorIfExists.</code>
Ignore	Does nothing.
Overwrite	Truncates or drops the table <p>Note <code>createRelation</code> truncates the table only when <code>truncate</code> <code>option</code> is enabled and <code>isCascadingTruncateTable</code> is disabled.</p>

In the end, `createRelation` closes the JDBC connection to the database and [creates a JDBCRelation](#).

Note

`createRelation` is a part of [CreatableRelationProvider Contract](#).

JsonFileFormat

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

JsonDataSource

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

ParquetFileFormat

ParquetFileFormat is a FileFormat that registers DataSources under the name parquet .

Custom Formats

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

See [spark-mf-format](#) project at GitHub for a complete solution.

BaseRelation — Collection of Tuples with Schema

`BaseRelation` is a [contract](#) to model a collection of tuples (from a data source) with a [schema](#).

Note	A "data source" and "relation" appear as synonyms.
------	--

`BaseRelation` can optionally provide information about its estimated size in bytes (as `sizeInBytes`) that defaults to [spark.sql.defaultSizeInBytes](#) internal property (i.e. infinite).

`BaseRelation` whether it needs a conversion.

`BaseRelation` computes the list of `Filter` that this data source may not be able to handle.

Table 1. BaseRelation's Known Implementations (in alphabetical order)

BaseRelation	Description
HadoopFsRelation	
JDBCRelation	
KafkaRelation	Structured Streaming's <code>BaseRelation</code> for datasets from Apache Kafka

Note	<code>BaseRelation</code> is "created" using <code>DataSource</code> 's resolveRelation .
------	---

Note	<code>BaseRelation</code> is transformed into a <code>DataFrame</code> using <code>SparkSession</code> 's baseRelationToDataFrame .
------	---

BaseRelation Contract

```
package org.apache.spark.sql.sources

abstract class BaseRelation {
    // only required methods that have no implementation
    def schema: StructType
    def sqlContext: SQLContext
}
```

Table 2. (Subset of) BaseRelation Contract (in alphabetical order)

Method	Description
schema	StructType
sqlContext	SQLContext

HadoopFsRelation

```
case class HadoopFsRelation(  
    location: FileIndex,  
    partitionSchema: StructType,  
    dataSchema: StructType,  
    bucketSpec: Option[BucketSpec],  
    fileFormat: FileFormat,  
    options: Map[String, String])(val sparkSession: SparkSession)  
extends BaseRelation with FileRelation
```

`HadoopFsRelation` is a [BaseRelation](#) in a [SparkSession](#) (through which it gets to the current [SQLContext](#)).

`HadoopFsRelation` requires a schema (as [StructType](#)) that it expands with the input `partitionSchema` schema.

`sizeInBytes` and `inputFiles` (from the base [BaseRelation](#)) use the input `FileIndex` to compute the size and input files, respectively.

JDBCRelation

columnPartition Method

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

QueryExecution — Query Execution of Dataset

`QueryExecution` represents the [structured query execution pipeline](#) of a `Dataset`.

Note	When an action of a <code>Dataset</code> is executed, that triggers an execution of <code>queryExecution</code> (in the form of calling <code>toRdd</code>) which will morph itself into a <code>RDD</code> of binary rows , i.e. <code>RDD[InternalRow]</code> .
------	--

You can access the `QueryExecution` of a `Dataset` using `queryExecution` attribute.

```
val ds: Dataset[Long] = ...
val queryExec = ds.queryExecution
```

`QueryExecution` is the result of [executing a `LogicalPlan`](#) in a `SparkSession` (and so you could create a `Dataset` from a [logical operator](#) or use the `QueryExecution` after executing a logical operator).

Table 1. `QueryExecution`'s (Lazily-Initialized) Attributes (aka Structured Query Execution Pipeline)

Attribute / Phase	Description		
<code>analyzed</code>	Analyzed logical plan that has passed Analyzer 's check rules. <pre>val schema = queryExecution.analyzed.output</pre> <table border="1"> <tr> <td>Tip</td> <td>Use Dataset's <code>explain(extended = true)</code> or SQL's <code>EXPLAIN EXTENDED</code> to see the analyzed logical plan of a structured query.</td> </tr> </table>	Tip	Use Dataset's <code>explain(extended = true)</code> or SQL's <code>EXPLAIN EXTENDED</code> to see the analyzed logical plan of a structured query.
Tip	Use Dataset's <code>explain(extended = true)</code> or SQL's <code>EXPLAIN EXTENDED</code> to see the analyzed logical plan of a structured query.		
<code>withCachedData</code>	LogicalPlan that is the <code>analyzed</code> plan after being analyzed, checked (for unsupported operations) and replaced with cached segments.		
<code>optimizedPlan</code>	Optimized logical plan being the result of executing the session-owned Catalyst Query Optimizer to <code>withCachedData</code> .		
<code>sparkPlan</code>	Physical plan (after SparkPlanner has planned the optimized logical plan). <table border="1"> <tr> <td>Note</td> <td><code>sparkPlan</code> is the first physical plan from the collection of all possible physical plans.</td> </tr> </table>	Note	<code>sparkPlan</code> is the first physical plan from the collection of all possible physical plans.
Note	<code>sparkPlan</code> is the first physical plan from the collection of all possible physical plans.		

	Note	It is guaranteed that Catalyst's <code>QueryPlanner</code> (which <code>SparkPlanner</code> extends) will always generate at least one physical plan .
<code>executedPlan</code>	Note	Physical plan ready for execution (i.e. <code>sparkPlan</code> after physical optimization rules applied).
<code>toRdd</code>	Note	RDD of binary rows (after executing the <code>executedPlan</code>). <code>toRdd</code> is a "boundary" between two Spark modules: Spark SQL and Spark Core. After you have executed <code>toRdd</code> (directly or not), you basically "leave" Spark SQL's Datasets and "enter" Spark Core's RDD space.

You can access the lazy attributes as follows:

```
val dataset: Dataset[Long] = ...
dataset.queryExecution.executedPlan
```

Table 2. QueryExecution's Properties (in alphabetical order)

Name	Description
<code>planner</code>	SparkPlanner

`QueryExecution` uses the input `SparkSession` to access the current [SparkPlanner](#) (through `SessionState`) when it is created. It then computes a [SparkPlan](#) (a `PhysicalPlan` exactly) using the planner. It is available as the `sparkPlan` attribute.

A streaming variant of `QueryExecution` is [IncrementalExecution](#).

Tip	Use <code>explain</code> operator to know about the logical and physical plans of a <code>Dataset</code> .
-----	--

```

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]

```

Note	<code>QueryExecution</code> belongs to <code>org.apache.spark.sql.execution</code> package.
------	---

Note	<code>QueryExecution</code> is a transient feature of a <code>Dataset</code> , i.e. it is not preserved across serializations.
------	--

simpleString Method

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

debug Object

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

Building Complete Text Representation — completeString Internal Method

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

Creating QueryExecution Instance

`QueryExecution` takes the following when created:

- `SparkSession`
- `LogicalPlan`

Physical Plan Preparation Rules — `preparations` Method

`preparations` is a sequence of [physical plan](#) preparation rules (i.e. `Rule[SparkPlan]`).

Tip	A <code>SparkPlan</code> preparation rule transforms a physical plan to another (possibly more efficient).
-----	--

`preparations` is one of the final phases of query execution that Spark developers could use for further query 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 physical preparation rules are applied sequentially in order to the physical plan before execution, i.e. they generate a <code>SparkPlan</code> when <code>executedPlan</code> lazy value is first accessed (and is cached afterwards).
------	---

Executing preparations Physical Plan Rules — `prepareForExecution` Method

```
prepareForExecution(plan: SparkPlan): SparkPlan
```

`prepareForExecution` takes [preparations](#) rules and applies them one by one to the input `plan`.

Note	<code>prepareForExecution</code> is used exclusively when <code>QueryExecution</code> prepares physical plan for execution .
------	--

IncrementalExecution

`IncrementalExecution` is a custom `QueryExecution` with `OutputMode`, `checkpointLocation`, and `currentBatchId`.

It lives in `org.apache.spark.sql.execution.streaming` package.

Caution

[FIXME](#) What is `stateStrategy` ?

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

[FIXME](#) What does `IncrementalExecution` do? Where is it used?

Creating Analyzed Logical Plan and Checking Correctness

— `assertAnalyzed` Method

`assertAnalyzed(): Unit`

`assertAnalyzed` triggers initialization of `analyzed` (which is almost like executing it).

Note

`assertAnalyzed` executes `analyzed` by accessing it and throwing the result away. Since `analyzed` is a lazy value in Scala, it will then get initialized for the first time and stays so forever.

`assertAnalyzed` then requests `Analyzer` to check the correctness of the analysis of the `LogicalPlan` (i.e. `analyzed`).

Note

`assertAnalyzed` uses `SparkSession` to access the current `SessionState` that it then uses to access the `Analyzer`.

In Scala the access path looks as follows.

```
sparkSession.sessionState.analyzer
```

In case of any `AnalysisException`, `assertAnalyzed` creates a new `AnalysisException` to make sure that it holds `analyzed` and reports it.

Note

`assertAnalyzed` is used when:

- `Dataset` is created
- `QueryExecution` is requested for `LogicalPlan` with cached data
- `CreateViewCommand` and `AlterViewAsCommand` are executed

Building Extended Text Representation with Logical and Physical Plans — `toString` Method

```
toString: String
```

`toString` is a mere alias for `completeString` with `appendStats` flag disabled.

Note

`toString` is on the "other" side of `toStringWithStats` which has `appendStats` flag enabled.

Note

`toString` is used when...**FIXME**

Building Text Representation with Cost Stats — `toStringWithStats` Method

```
toStringWithStats: String
```

`toStringWithStats` is a mere alias for `completeString` with `appendStats` flag enabled.

Note

`toStringWithStats` is a custom `toString` with cost statistics.

```
// test dataset
val dataset = spark.range(20).limit(2)

// toStringWithStats in action - note Optimized Logical Plan section with Statistics
scala> dataset.queryExecution.toStringWithStats
res6: String =
== Parsed Logical Plan ==
GlobalLimit 2
+- LocalLimit 2
  +- Range (0, 20, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
GlobalLimit 2
+- LocalLimit 2
  +- Range (0, 20, step=1, splits=Some(8))

== Optimized Logical Plan ==
GlobalLimit 2, Statistics(sizeInBytes=32.0 B, rowCount=2, isBroadcastable=false)
+- LocalLimit 2, Statistics(sizeInBytes=160.0 B, isBroadcastable=false)
  +- Range (0, 20, step=1, splits=Some(8)), Statistics(sizeInBytes=160.0 B, isBroadcastable=false)

== Physical Plan ==
CollectLimit 2
+- *Range (0, 20, step=1, splits=Some(8))
```

Note

`toStringWithStats` is used exclusively when `ExplainCommand` is executed (only when `cost` attribute is enabled).

Transforming SparkPlan Execution Result to Hive-Compatible Output Format — `hiveResultString` Method

```
hiveResultString(): Seq[String]
```

`hiveResultString` returns the result as a Hive-compatible output format.

```
scala> spark.range(5).queryExecution.hiveResultString
res0: Seq[String] = ArrayBuffer(0, 1, 2, 3, 4)

scala> spark.read.csv("people.csv").queryExecution.hiveResultString
res4: Seq[String] = ArrayBuffer(id      name      age, 0  Jacek  42)
```

Internally, `hiveResultString` transformation the `SparkPlan`.

Table 3. `hiveResultString`'s SparkPlan Transformations (in execution order)

SparkPlan	Description
<code>ExecutedCommandExec</code> for <code>DescribeTableCommand</code>	Executes <code>DescribeTableCommand</code> and transforms every <code>Row</code> to a Hive-compatible output format.
<code>ExecutedCommandExec</code> for <code>ShowTablesCommand</code>	Executes <code>ExecutedCommandExec</code> and transforms the result to a collection of table names.
Any other <code>SparkPlan</code>	Executes <code>sparkPlan</code> and transforms the result to a Hive-compatible output format.
Note	<code>hiveResultString</code> is used exclusively when <code>SparksQLDriver</code> (of <code>ThriftServer</code>) runs a command.

Expression — Executable Node in Catalyst Tree

`Expression` is a executable [node](#) (in a Catalyst tree) that can be [evaluated](#) to a value given input values, i.e. can produce a JVM object per `InternalRow`.

Note

`Expression` is often called a **Catalyst expression** even though it is *merely* built using (not be part of) the [Catalyst — Tree Manipulation Framework](#).

```
// evaluating an expression
// Use Literal expression to create an expression from a Scala object
import org.apache.spark.sql.catalyst.expressions.Expression
import org.apache.spark.sql.catalyst.expressions.Literal
val e: Expression = Literal("hello")

import org.apache.spark.sql.catalyst.expressions.EmptyRow
val v: Any = e.eval(EmptyRow)

// Convert to Scala's String
import org.apache.spark.unsafe.types.UTF8String
scala> val s = v.asInstanceOf[UTF8String].toString
s: String = hello
```

`Expression` can [generate a Java source code](#) that is then used in evaluation.

Table 1. Specialized Expressions

Name	Scala Kind	Behaviour	Example
BinaryExpression	abstract class		<ul style="list-style-type: none"> • UnixTime
CodegenFallback	trait	Does not support code generation and falls back to interpreted mode	<ul style="list-style-type: none"> • CallMethod
ExpectsInputTypes	trait		
LeafExpression	abstract class	Has no child expressions (and hence "terminates" the expression tree).	<ul style="list-style-type: none"> • Attribute • Literal
NamedExpression		Can later be referenced in a dataflow graph.	
Nondeterministic	trait		

<code>NonSQLExpression</code>	<code>trait</code>	<p>Expression with no SQL representation</p> <p>Gives the only custom <code>sql</code> method that is non-overridable (i.e. <code>final</code>).</p> <p>When requested <code>SQL representation</code>, <code>NonSQLExpression</code> transforms <code>Attributes</code> to be <code>PrettyAttribute</code>s to build text representation.</p>	<ul style="list-style-type: none"> <code>ScalaUD</code> <code>StaticInv</code> <code>TimeWin</code>
<code>TernaryExpression</code>	<code>abstract class</code>		
<code>TimeZoneAwareExpression</code>	<code>trait</code>	Timezone-aware expressions	<ul style="list-style-type: none"> <code>UnixTime</code> <code>JsonToSt</code>
<code>UnaryExpression</code>	<code>abstract class</code>		<ul style="list-style-type: none"> <code>ExplodeE</code> <code>JsonToSt</code>
<code>Unevaluatable</code>	<code>trait</code>	<p>Cannot be evaluated, i.e. <code>eval</code> and <code>doGenCode</code> are not supported and report an <code>UnsupportedOperationException</code>.</p> <p>Replaced by some other expressions during <code>analysis</code> or <code>optimization</code>.</p>	<ul style="list-style-type: none"> <code>Aggregat</code> <code>CurrentD</code> <code>TimeWin</code> <code>WindowE</code> <code>WindowS</code>

Expression Contract

```
package org.apache.spark.sql.catalyst.expressions

abstract class Expression extends TreeNode[Expression] {
    // only required methods that have no implementation
    def dataType: DataType
    def doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode
    def eval(input: InternalRow = EmptyRow): Any
    def nullable: Boolean
}
```

Table 2. (Subset of) Expression Contract (in alphabetical

Method	Description
<code>canonicalized</code>	

checkInputDataTypes			
childrenResolved			
dataType			
deterministic			
doGenCode	<p>Code-generated evaluation that generates a Java source code (in a more optimized way not directly using <code>eval</code>).</p> <p>Used as part of <code>genCode</code>.</p>		
eval	<p>No-code-generated evaluation that evaluates the expression to (without generating a corresponding Java code.)</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="padding: 2px;">Note</td> <td style="padding: 2px;">By default accepts <code>EmptyRow</code>, i.e. <code>null</code>.</td> </tr> </table>	Note	By default accepts <code>EmptyRow</code> , i.e. <code>null</code> .
Note	By default accepts <code>EmptyRow</code> , i.e. <code>null</code> .		
foldable			
genCode	<p>Code-generated evaluation that generates a Java source code (in a more optimized way not directly using <code>eval</code>).</p> <p>Similar to <code>doGenCode</code> but supports expression reuse (aka <code>subexpr</code>).</p>		
nullable			
prettyName			
references			
resolved			
semanticEquals			
semanticHash			
sql	<p>SQL representation</p> <p><code>prettyName</code> followed by <code>sql</code> of <code>children</code> in the round brackets and e.g.</p> <pre> import org.apache.spark.sql.catalyst.dsl.expressions._ import org.apache.spark.sql.catalyst.expressions.Sentences val sentences = Sentences("Hi there! Good morning.", "en", "US") import org.apache.spark.sql.catalyst.expressions.Expression val expr: Expression = count("*") === 5 && count(sentences) === 1 scala> expr.sql res0: String = ((count('*') = 5) AND (count(sentences('Hi there! Good morning.')) = 1)) </pre>		

Nondeterministic Expression

`Nondeterministic` expressions are non-deterministic and non-`foldable`, i.e. `deterministic` and `foldable` properties are disabled (i.e. `false`). They require explicit initialization before evaluation.

`Nondeterministic` expressions have two additional methods:

1. `initInternal` for internal initialization (called before `eval`)
2. `evalInternal` to evaluate a `InternalRow` into a JVM object.

Note	<code>Nondeterministic</code> is a Scala trait.
------	---

`Nondeterministic` expressions have the additional `initialized` flag that is enabled (i.e. `true`) after the other additional `initInternal` method has been called.

Examples of `Nondeterministic` expressions are `InputFileName`, `MonotonicallyIncreasingID`, `SparkPartitionID` functions and the abstract `RDG` (that is the base for `Rand` and `Randn` functions).

Note	<code>Nondeterministic</code> expressions are the target of <code>PullOutNondeterministic</code> logical plan rule.
------	---

AggregateExpression — Unevaluable Expression with AggregateFunction

`AggregateExpression` is an [unevaluable](#) expression (i.e. with no support for `eval` and `doGenCode` methods) that holds the following:

- [AggregateFunction](#)
- `AggregateMode`
- `isDistinct` flag that...[FIXME](#)
- `ExprId`

Note

`AggregateExpression` is a mere container of [AggregateFunction](#) and relays all calls to it.

Table 1. AggregateExpression's Properties (in alphabetical order)

Name	Description
canonicalized	AggregateExpression with AggregateFunction expression canonicalized with the special ExprId as 0 .
children	AggregateFunction expression (for which AggregateExpression was created).
dataType	DataType of AggregateFunction expression
foldable	Disabled (i.e. false)
nullable	Whether or not AggregateFunction expression is nullable.
references	AttributeSet with the following: <ul style="list-style-type: none"> references of AggregateFunction when AggregateMode is Partial or Complete aggBufferAttributes of AggregateFunction when PartialMerge or Final
resultAttribute	Attribute that is: <ul style="list-style-type: none"> AttributeReference when AggregateFunction is itself resolved UnresolvedAttribute otherwise
sql	Requests AggregateFunction to generate SQL output (with isDistinct flag).
toString	AggregateMode prefix followed by AggregateFunction's toAggString (with isDistinct flag).

AggregateFunction

`AggregateFunction` is the [contract](#) for [expressions](#) that...[FIXME](#)

Note	Aggregate functions are not foldable , i.e. FIXME
------	---

Table 1. AggregateFunction Top-Level Expressions

Name	Behaviour	Examples
DeclarativeAggregate		
ImperativeAggregate		
TypedAggregateExpression		

AggregateFunction Contract

```
abstract class AggregateFunction extends Expression {
  def aggBufferSchema: StructType
  def aggBufferAttributes: Seq[AttributeReference]
  def inputAggBufferAttributes: Seq[AttributeReference]
  def defaultResult: Option[Literal] = None
}
```

Table 2. AggregateFunction Contract (in alphabetical order)

Method	Description
aggBufferSchema	<p>Schema of an aggregation buffer to hold partial aggregate results.</p> <p>Used mostly in ScalaUDAF and AggregationIterator (for <code>initializeAggregateFunctions</code>).</p>
aggBufferAttributes	<p>Collection of AttributeReference objects of an aggregation buffer to hold partial aggregate results.</p> <p>Used in:</p> <ul style="list-style-type: none"> • DeclarativeAggregateEvaluator • AggregateExpression for references • Expression -based aggregate's <code>bufferSchema</code> in DeclarativeAggregate • ...
inputAggBufferAttributes	
defaultResult	Defaults to <code>None</code> .

Attribute Leaf Expression

`Attribute` is a [leaf](#) (i.e. no children) [named](#) expression.

Note	A QueryPlan uses Attributes to build the schema of the query (it represents).
------	---

Table 1. Attribute's Properties and Their Behaviour (Inherited from Expression)

Property	Behaviour
<code>references</code>	A one-element collection with itself.
<code>toAttribute</code>	Self-reference

`Attribute` abstract class defines three additional "builder" methods.

Table 2. `Attribute` Expression Builder Methods

Name	Description
<code>withNullability</code>	Sets
<code>withQualifier</code>	Sets
<code>withName</code>	Sets

Note	<code>Attribute</code> is the base expression for <code>AttributeReference</code> , <code>UnresolvedAttribute</code> , and <code>PrettyAttribute</code> .
------	---

As an optimization, `Attribute` is marked as to not tolerate `nulls` , and when given a `null` input produces a `null` output.

CallMethodViaReflection Expression

`CallMethodViaReflection` is an [expression](#) that represents a static method call in Scala or Java using `reflect` and `java_method` functions.

Note

`reflect` and `java_method` functions are only supported in [SQL](#) and [expression](#) modes.

Table 1. CallMethodViaReflection's DataType to JVM Types Mapping

DataType	JVM Type
<code>BooleanType</code>	<code>java.lang.Boolean</code> / <code>scala.Boolean</code>
<code>ByteType</code>	<code>java.lang.Byte</code> / <code>Byte</code>
<code>ShortType</code>	<code>java.lang.Short</code> / <code>Short</code>
<code>IntegerType</code>	<code>java.lang.Integer</code> / <code>Int</code>
<code>LongType</code>	<code>java.lang.Long</code> / <code>Long</code>
<code>FloatType</code>	<code>java.lang.Float</code> / <code>Float</code>
<code>DoubleType</code>	<code>java.lang.Double</code> / <code>Double</code>
<code>StringType</code>	<code>String</code>

```

import org.apache.spark.sql.catalyst.expressions.CallMethodViaReflection
import org.apache.spark.sql.catalyst.expressions.Literal
scala> val expr = CallMethodViaReflection(
|   Literal("java.time.LocalDateTime") :::
|   Literal("now") :: Nil)
expr: org.apache.spark.sql.catalyst.expressions.CallMethodViaReflection = reflect(java
.time.LocalDateTime, now)
scala> println(expr.numberedTreeString)
00 reflect(java.time.LocalDateTime, now)
01 :- java.time.LocalDateTime
02 +- now

// CallMethodViaReflection as the expression for reflect SQL function
val q = """
  select reflect("java.time.LocalDateTime", "now") as now
"""
val plan = spark.sql(q).queryExecution.logical
// CallMethodViaReflection shows itself under "reflect" name
scala> println(plan.numberedTreeString)
00 Project [reflect(java.time.LocalDateTime, now) AS now#39]
01 +- OneRowRelation$
```

`CallMethodViaReflection` supports a [fallback mode for expression code generation](#).

Table 2. `CallMethodViaReflection`'s Properties (in alphabetical order)

Property	Description
<code>dataType</code>	<code>StringType</code>
<code>deterministic</code>	Disabled (i.e. <code>false</code>)
<code>nullable</code>	Enabled (i.e. <code>true</code>)
<code>prettyName</code>	<code>reflect</code>

Note

`CallMethodViaReflection` is very similar to [StaticInvoke](#) expression.

Generator — Catalyst Expressions that Generate Zero Or More Rows

`Generator` is a [contract](#) for [Catalyst expressions](#) that can [produce](#) zero or more rows given a single input row.

`Generator` is not [foldable](#) and not [nullable](#) by default.

`Generator` supports [whole-stage codegen](#) when not [CodegenFallback](#) by default.

Table 1. Generators (in alphabetical order)

Name	Description
CollectionGenerator	
ExplodeBase	
Explode	
GeneratorOuter	
HiveGenericUDTF	
Inline	Corresponds to <code>inline</code> and <code>inline_outer</code> functions.
JsonTuple	
PosExplode	
Stack	
UnresolvedGenerator	<p>Represents an unresolved generator.</p> <p>Created when <code>AstBuilder</code> creates <code>Generate</code> for <code>LATERAL VIEW</code> that corresponds to the following:</p> <pre>LATERAL VIEW (OUTER)? generatorFunctionName (arg1, arg2, ...) tblName AS? col1, col2, ...</pre> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>Note UnresolvedGenerator is resolved to Generator by ResolveFunctions (in Analyzer).</p> </div>
UserDefinedGenerator	Used exclusively in the now-deprecated <code>explode</code> operator

You can only have one generator per select clause that is enforced by [ExtractGenerator](#)

```
scala> xys.select(explode($"xs"), explode($"ys")).show
org.apache.spark.sql.AnalysisException: Only one generator allowed per select clause
    at org.apache.spark.sql.catalyst.analysis.Analyzer$ExtractGenerator$$anonfun$analyze$1.apply(Analyzer.scala:100)
    at org.apache.spark.sql.catalyst.analysis.Analyzer$ExtractGenerator$$anonfun$analyze$1.apply(Analyzer.scala:98)
    at org.apache.spark.sql.catalyst.plans.logical.LogicalPlan$$anonfun$resolveOperators$1.apply(LogicalPlan.scala:100)
    at org.apache.spark.sql.catalyst.plans.logical.LogicalPlan$$anonfun$resolveOperators$1.apply(LogicalPlan.scala:99)
```

If you want to have more than one generator in a structured query you should use e.g.

```
val arrayTuple = (Array(1,2,3), Array("a","b","c"))
val ncs = Seq(arrayTuple).toDF("ns", "cs")
```

```
scala> ncs.show
+-----+-----+
|      ns|      cs|
+-----+-----+
|[1, 2, 3]| [a, b, c]|
+-----+-----+
```

Note

```
scala> ncs.createOrReplaceTempView("ncs")
```

```
val q = """
  SELECT n, c FROM ncs
  LATERAL VIEW explode(ns) nsExpl AS n
  LATERAL VIEW explode(cs) csExpl AS c
"""


```

```
scala> sql(q).show
+---+---+
| n| c|
+---+---+
| 1| a|
| 1| b|
| 1| c|
| 2| a|
| 2| b|
| 2| c|
| 3| a|
| 3| b|
| 3| c|
+---+---+
```

Generator Contract

```
package org.apache.spark.sql.catalyst.expressions

trait Generator extends Expression {
  // only required methods that have no implementation
  def elementSchema: StructType
  def eval(input: InternalRow): TraversableOnce[InternalRow]
}
```

Table 2. (Subset of) Generator Contract (in alphabetical order)

Method	Description
elementSchema	<code>StructType</code> of the elements generated
eval	Used when...

Explode Generator Unary Expression

`Explode` is a unary expression that produces a sequence of records for each value in the array or map.

`Explode` is a result of executing `explode` function (in SQL and [functions](#))

```
scala> sql("SELECT explode(array(10,20))").explain
== Physical Plan ==
Generate explode([10,20]), false, false, [col#68]
+- Scan OneRowRelation[]

scala> sql("SELECT explode(array(10,20))").queryExecution.optimizedPlan.expressions(0)
res18: org.apache.spark.sql.catalyst.expressions.Expression = explode([10,20])

val arrayDF = Seq(Array(0,1)).toDF("array")
scala> arrayDF.withColumn("num", explode('array)).explain
== Physical Plan ==
Generate explode(array#93), true, false, [array#93, num#102]
+- LocalTableScan [array#93]
```

PosExplode

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

ExplodeBase Unary Expression

`ExplodeBase` is the base class for `Explode` and `PosExplode`.

`ExplodeBase` is [UnaryExpression](#) and [Generator](#) with [CodegenFallback](#).

JsonToStructs Unary Expression

`JsonToStructs` is a [unary](#) expression with [timezone](#) support and [CodegenFallback](#) that represents [from_json](#) function.

Parsing Table Schema for String Literals — `validateSchemaLiteral` Method

```
validateSchemaLiteral(exp: Expression): StructType
```

`validateSchemaLiteral` requests [CatalystSqlParser](#) to [parseTableSchema](#) for [Literal](#) of [StringType](#).

For any other non- [stringType](#) [types](#), `validateSchemaLiteral` reports a [AnalysisException](#):

```
Expected a string literal instead of [expression]
```

Literal Leaf Expression

`Literal` is `LeafExpression` that is created for a Scala `value` and `DataType`.

Table 1. Literal's Properties (in alphabetical order)

Property	Description
<code>foldable</code>	Enabled (i.e. <code>true</code>)
<code>nullable</code>	Enabled when <code>value</code> is <code>null</code>

ScalaUDAF

ScalaUDAF is a `ImperativeAggregate` expression with no representation in SQL.

ScalaUDAF is created when:

- `UDFRegistration` registers a user-defined aggregate function
- `UserDefinedAggregateFunction` creates a `Column` for a user-defined aggregate function using `all` and `distinct` values

StaticInvoke Non-SQL Expression

`StaticInvoke` is an [expression](#) with [no SQL representation](#) that represents a static method call in Scala or Java. It supports [generating Java code](#) to evaluate itself.

`StaticInvoke` is [created](#) when:

- `ScalaReflection` is requested for the [deserializer](#) or [serializer](#) for a Scala type
- `RowEncoder` is requested for `deserializerFor` or [serializer](#) for a Scala type
- `JavaTypeInference` is requested for `deserializerFor` or `serializerFor`

```
import org.apache.spark.sql.types.StructType
val schema = new StructType()
  .add($"id".long.copy(nullable = false))
  .add($"name".string.copy(nullable = false))

import org.apache.spark.sql.catalyst.encoders.RowEncoder
val encoder = RowEncoder(schema)
scala> println(encoder.serializer(0).numberedTreeString)
00 validateexternaltype(getexternalrowfield(assertnonnull(input[0, org.apache.spark.sql.Row, true]), 0, id), LongType) AS id#1640L
01 +- validateexternaltype(getexternalrowfield(assertnonnull(input[0, org.apache.spark.sql.Row, true]), 0, id), LongType)
02     +- getexternalrowfield(assertnonnull(input[0, org.apache.spark.sql.Row, true]), 0, id)
03         +- assertnonnull(input[0, org.apache.spark.sql.Row, true])
04             +- input[0, org.apache.spark.sql.Row, true]
```

Note	<code>StaticInvoke</code> is similar to <code>CallMethodViaReflection</code> expression.
------	--

Creating StaticInvoke Instance

`StaticInvoke` takes the following when created:

- Target object of the static call
- [Data type](#) of the return value of the [method](#)
- Name of the method to call on the [static object](#)
- Optional [expressions](#) to pass as input arguments to the [function](#)

- Flag to control whether to propagate `nulls` or not (enabled by default). If any of the arguments is `null`, `null` is returned instead of calling the [function](#)

TimeWindow Unevaluable Unary Expression

TimeWindow is an [unevaluable](#) and [non-SQL](#) unary expression that represents window function.

```
import org.apache.spark.sql.functions.window
val timeColumn = window('time, "5 seconds")

scala> timeColumn.expr.numberedTreeString
res9: String =
00 timewindow('time, 5000000, 5000000, 0) AS window#1
01 +- timewindow('time, 5000000, 5000000, 0)
02   +- 'time

import org.apache.spark.sql.catalyst.expressions.TimeWindow
scala> val timeWindow = timeColumn.expr.children(0).asInstanceOf[TimeWindow]
timeWindow: org.apache.spark.sql.catalyst.expressions.TimeWindow = timewindow('time, 5
000000, 5000000, 0)
```

TimeWindow can never be resolved as it is converted to [Filter](#) with [Expand](#) logical operators at [analysis phase](#).

Analysis Phase

TimeWindow is resolved to [Expand](#) logical operator in [TimeWindowing](#) logical evaluation rule.

```
// https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html
import java.time.LocalDateTime
// https://docs.oracle.com/javase/8/docs/api/java/sql/Timestamp.html
import java.sql.Timestamp
val levels = Seq(
    // (year, month, dayOfMonth, hour, minute, second)
    ((2012, 12, 12, 12, 12, 12), 5),
    ((2012, 12, 12, 12, 12, 14), 9),
    ((2012, 12, 12, 13, 13, 14), 4),
    ((2016, 8, 13, 0, 0, 0), 10),
    ((2017, 5, 27, 0, 0, 0), 15))
    .map { case ((yy, mm, dd, h, m, s), a) => (LocalDateTime.of(yy, mm, dd, h, m, s), a)}
)
    .map { case (ts, a) => (Timestamp.valueOf(ts), a) }
    .toDF("time", "level")
scala> levels.show
+-----+----+
|          time|level|
+-----+----+
|2012-12-12 12:12:12|    5|
|2012-12-12 12:12:14|    9|
|2012-12-12 13:13:14|    4|
|2016-08-13 00:00:00|   10|
|2017-05-27 00:00:00|   15|
+-----+----+
val q = levels.select(window($"time", "5 seconds"))

// Before Analyzer
scala> println(q.queryExecution.logical.numberedTreeString)
00 'Project [timewindow('time, 5000000, 5000000, 0) AS window#57]
01 +- Project [_1#26 AS time#29, _2#27 AS level#30]

02   +- LocalRelation [_1#26, _2#27]
// After Analyzer
scala> println(q.queryExecution.analyzed.numberedTreeString)
00 Project [window#58 AS window#57]
01 +- Filter ((time#29 >= window#58.start) && (time#29 < window#58.end))
02   +- Expand [List(named_struct(start, (((CEIL((cast((precisetimestamp(time#29) - 0) as double) / cast(5000000 as double))) + cast(0 as bigint)) - cast(1 as bigint)) * 5000000) + 0), end, (((((CEIL((cast((precisetimestamp(time#29) - 0) as double) / cast(5000000 as double))) + cast(0 as bigint)) - cast(1 as bigint)) * 5000000) + 0) + 5000000), time#29, level#30), List(named_struct(start, (((CEIL((cast((precisetimestamp(ti me#29) - 0) as double) / cast(5000000 as double))) + cast(1 as bigint)) - cast(1 as bigint)) * 5000000) + 0), end, (((((CEIL((cast((precisetimestamp(time#29) - 0) as double) / cast(5000000 as double))) + cast(1 as bigint)) - cast(1 as bigint)) * 5000000) + 0) + 5000000)), time#29, level#30)], [window#58, time#29, level#30]
03     +- Project [_1#26 AS time#29, _2#27 AS level#30]
04         +- LocalRelation [_1#26, _2#27]
```


UnixTimestamp TimeZoneAware Binary Expression

`UnixTimestamp` is a [binary](#) expression with [timezone](#) support that represents `unix_timestamp` function (and indirectly [to_date](#) and [to_timestamp](#)).

```
import org.apache.spark.sql.functions.unix_timestamp
val c1 = unix_timestamp()

scala> c1.explain(true)
unix_timestamp(current_timestamp(), yyyy-MM-dd HH:mm:ss, None)

scala> println(c1.expr.numberedTreeString)
00 unix_timestamp(current_timestamp(), yyyy-MM-dd HH:mm:ss, None)
01 :- current_timestamp()
02 +- yyyy-MM-dd HH:mm:ss

import org.apache.spark.sql.catalyst.expressions.UnixTimestamp
scala> c1.expr.newInstanceOf[UnixTimestamp]
res0: Boolean = true
```

Note

`UnixTimestamp` is `UnixTime` expression internally (as is `ToUnixTimestamp` expression).

`UnixTimestamp` supports `StringType`, `DateType` and `TimestampType` as input types for a time expression and returns `LongType`.

```
scala> c1.expr.eval()
res1: Any = 1493354303
```

`UnixTimestamp` uses `DateTimeUtils.newDateFormat` for date/time format (as Java's [java.text.DateFormat](#)).

WindowExpression Unevaluable Expression

`WindowExpression` is an [unevaluable expression](#) (i.e. with no support for `eval` and `doGenCode` methods).

`WindowExpression` is created to represent `windowFunction` and `WindowSpecDefinition` expressions in a SQL query after `Analyzer` [resolves](#) `UnresolvedWindowExpressions`.

```
import org.apache.spark.sql.catalyst.expressions.WindowExpression
// relation - Dataset as a table to query
val table = spark.emptyDataset[Int]

scala> val windowExpr = table
    .selectExpr("count() OVER (PARTITION BY value) AS count")
    .queryExecution
    .logical      (1)
    .expressions
    .toList(0)
    .children(0)
    .asInstanceOf[WindowExpression]
windowExpr: org.apache.spark.sql.catalyst.expressions.WindowExpression = 'count() wind
owspecdefinition('value, UnspecifiedFrame)

scala> windowExpr.sql
res2: String = count() OVER (PARTITION BY `value` UnspecifiedFrame)
```

1. Use `sqlParser` directly as in the [WithWindowDefinition Example](#) below.

Note

`WindowExpression` is used in `Analyzer` in [ExtractWindowExpressions](#), [ResolveWindowOrder](#) [ResolveWindowFrame](#) rules.

Note

`WindowExpression` is also used in `Analyzer` for [analysis validation](#) for the following checks: [FIXME...](#)

Note

`WindowExpression` is used in [NullPropagation](#) optimization.

Table 1. WindowExpression's Properties (in alphabetical order)

Name	Description
children	Collection of two expressions , i.e. windowFunction and WindowSpecDefinition , for which WindowExpression was created.
dataType	DataType of windowFunction
foldable	Whether or not windowFunction is foldable.
nullable	Whether or not windowFunction is nullable.
sql	"[windowFunction].sql OVER [windowSpec].sql"
toString	"[windowFunction] [windowSpec]"

WithWindowDefinition Unary Logical Plan

`WithWindowDefinition` is a [unary logical plan](#) with a single `child` logical plan and a `windowDefinitions` lookup table of [WindowSpecDefinition](#) per name.

`WithWindowDefinition` is created exclusively when `AstBuilder` [parses window definitions](#).

Table 2. WithWindowDefinition's Properties (in alphabetical order)

Name	Description
output	Output of the <code>child</code> logical plan.

WithWindowDefinition Example

```

// Table for FROM clauses below
(0 to 5).toDF("id").createOrReplaceTempView("ids")

// Example with window specification alias and definition

val sqlText = """
SELECT count(*) OVER anotherWindowSpec
FROM ids
WINDOW
anotherWindowSpec AS myWindowSpec,
myWindowSpec AS (
    PARTITION BY id
    RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
)
"""

import spark.sessionState.{analyzer,sqlParser}

scala> sqlParser.parsePlan(sqlText)
res1: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
'WithWindowDefinition Map(anotherWindowSpec -> windowspecdefinition('id, RANGE BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW), myWindowSpec -> windowspecdefinition('id, RANGE
BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW))
+- 'Project [unresolvedalias(unresolvedwindowexpression('count(1), WindowSpecReference
(anotherWindowSpec)), None)]
    +- 'UnresolvedRelation `ids`

scala> sql(sqlText)
res2: org.apache.spark.sql.DataFrame = [count(1) OVER (PARTITION BY id RANGE BETWEEN U
NBOUNDED PRECEDING AND CURRENT ROW): bigint]

scala> analyzer.execute(sqlParser.parsePlan(sqlText))
res3: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Project [count(1) OVER (PARTITION BY id RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT
ROW)#12L]
+- Project [id#3, count(1) OVER (PARTITION BY id RANGE BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)#12L, count(1) OVER (PARTITION BY id RANGE BETWEEN UNBOUNDED PRECEDING AN
D CURRENT ROW)#12L]
    +- Window [count(1) windowspecdefinition(id#3, RANGE BETWEEN UNBOUNDED PRECEDING AN
D CURRENT ROW) AS count(1) OVER (PARTITION BY id RANGE BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)#12L], [id#3]
        +- Project [id#3]
            +- SubqueryAlias ids
                +- Project [value#1 AS id#3]
                    +- LocalRelation [value#1]

```

UnresolvedWindowExpression Unevaluable Expression — WindowExpression With Unresolved Window Specification Reference

`UnresolvedWindowExpression` is an [unevaluable expression](#) (i.e. with no support for `eval` and `doGenCode` methods).

`UnresolvedWindowExpression` is created to represent a `child expression` and `WindowSpecReference` (with an identifier for the window reference) when `AstBuilder` parses a function evaluated in a windowed context with a `WindowSpecReference`.

`UnresolvedWindowExpression` is resolved to a `WindowExpression` when `Analyzer` resolves `UnresolvedWindowExpressions`.

```
import spark.sessionState.sqlParser

scala> sqlParser.parseExpression("foo() OVER windowSpecRef")
res1: org.apache.spark.sql.catalyst.expressions.Expression = unresolvedwindowexpression('foo(), WindowSpecReference(windowSpecRef))
```

Table 3. `UnresolvedWindowExpression`'s Properties (in alphabetical order)

Name	Description
<code>dataType</code>	Reports a <code>unresolvedException</code>
<code>foldable</code>	Reports a <code>unresolvedException</code>
<code>nullable</code>	Reports a <code>unresolvedException</code>
<code>resolved</code>	Disabled (i.e. <code>false</code>)

WindowSpecDefinition Unevaluable Expression

`WindowSpecDefinition` is an [unevaluable expression](#) (i.e. with no support for `eval` and `doGenCode` methods).

`WindowSpecDefinition` is created for a [window specification](#) in a SQL query or `Column`'s `over` operator.

```
import org.apache.spark.sql.expressions.Window
val byValueDesc = Window.partitionBy("value").orderBy($"value".desc)

val query = table.withColumn(
  "count over window", count($"*") over byValueDesc)

import org.apache.spark.sql.catalyst.expressions.WindowExpression
val windowExpr = query.queryExecution
  .logical
  .expressions(1)
  .children(0)
  .asInstanceOf[WindowExpression]

scala> windowExpr.windowSpec
res0: org.apache.spark.sql.catalyst.expressions.WindowSpecDefinition = windowspecdefinition('value, 'value DESC NULLS LAST, UnspecifiedFrame)
```

`WindowSpecDefinition` contains the following:

- Window partition specifications (as [expressions](#))
- Window order specifications (as `SortOrder` objects)
- Window frame specification (as `WindowFrame`)

```
import org.apache.spark.sql.catalyst.expressions.WindowSpecDefinition

Seq((0, "hello"), (1, "windows"))
  .toDF("id", "token")
  .createOrReplaceTempView("mytable")

val sqlText = """
  SELECT count(*) OVER myWindowSpec
  FROM mytable
  WINDOW
    myWindowSpec AS (
      PARTITION BY token
      ORDER BY id
```

```

        RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    )
"""

import spark.sessionState.{analyzer,sqlParser}

scala> val parsedPlan = sqlParser.parsePlan(sqlText)
parsedPlan: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
'WithWindowDefinition Map(myWindowSpec -> windowspecdefinition('token, 'id ASC NULLS F
IRST, RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW))
+- 'Project [unresolvedalias(unresolvedwindowexpression('count(1), WindowSpecReference
(myWindowSpec)), None)]
    +- 'UnresolvedRelation `mytable`

import org.apache.spark.sql.catalyst.plans.logical.WithWindowDefinition
val myWindowSpec = parsedPlan.asInstanceOf[WithWindowDefinition].windowDefinitions("my
WindowSpec")

scala> println(myWindowSpec)
windowspecdefinition('token, 'id ASC NULLS FIRST, RANGE BETWEEN UNBOUNDED PRECEDING AN
D CURRENT ROW)

scala> println(myWindowSpec.sql)
(PARTITION BY `token` ORDER BY `id` ASC NULLS FIRST RANGE BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW)

scala> sql(sqlText)
res4: org.apache.spark.sql.DataFrame = [count(1) OVER (PARTITION BY token ORDER BY id
ASC NULLS FIRST RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW): bigint]

scala> println(analyzer.execute(sqlParser.parsePlan(sqlText)))
Project [count(1) OVER (PARTITION BY token ORDER BY id ASC NULLS FIRST RANGE BETWEEN U
NBOUNDED PRECEDING AND CURRENT ROW)#25L]
+- Project [token#13, id#12, count(1) OVER (PARTITION BY token ORDER BY id ASC NULLS F
IRST RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)#25L, count(1) OVER (PARTITION
BY token ORDER BY id ASC NULLS FIRST RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
)#25L]
    +- Window [count(1) windowspecdefinition(token#13, id#12 ASC NULLS FIRST, RANGE BET
WEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS count(1) OVER (PARTITION BY token ORDER B
Y id ASC NULLS FIRST RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)#25L], [token#1
3], [id#12 ASC NULLS FIRST]
        +- Project [token#13, id#12]
            +- SubqueryAlias mytable
                +- Project [_1#9 AS id#12, _2#10 AS token#13]
                    +- LocalRelation [_1#9, _2#10]

```

Table 1. WindowSpecDefinition's Properties (in alphabetical order)

Name	Description
children	Window <code>partition</code> and <code>order</code> specifications (for which <code>WindowExpression</code> was created).
dataType	Unsupported (i.e. reports a <code>UnsupportedOperationException</code>)
foldable	Disabled (i.e. <code>false</code>)
nullable	Enabled (i.e. <code>true</code>)
resolved	Enabled when <code>children</code> are valid and the input <code>DataType</code> is <code>SpecifiedWindowFrame</code> .
sql	Contains <code>PARTITION BY</code> with comma-separated elements of <code>partitionSpec</code> (if defined) with <code>ORDER BY</code> with comma-separated elements of <code>orderSpec</code> (if defined) followed by <code>frameSpecification</code> . <code>(PARTITION BY `token` ORDER BY `id` ASC NULLS FIRST RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)</code>

validate Method

```
validate: Option[String]
```

Caution

[FIXME](#)

LogicalPlan — Logical Query Plan / Logical Operator

`LogicalPlan` is a base Catalyst [query plan](#) for **logical operators** to build a **logical query plan** that, when [analyzed](#) and [resolved](#), can be resolved to a [physical query plan](#).

Tip

Use [QueryExecution](#) of a structured query to see the **logical plan**.

```
val q: DataFrame = ...
val plan = q.queryExecution.logical
```

`LogicalPlan` can be **analyzed** which is to say that the plan (including children) has gone through analysis and verification.

```
scala> plan.analyzed
res1: Boolean = true
```

A logical plan can also be **resolved** to a specific schema.

```
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
```

`LogicalPlan` can be [streaming](#) if it contains one or more [structured streaming sources](#).

Table 1. Logical Operators / Specialized Logical Plans

LogicalPlan	Description
LeafNode	Logical operator with no child operators
UnaryNode	Logical plan with a single child (logical plan).
BinaryNode	Logical operator with two child operators
Command	
RunnableCommand	

Table 2. LogicalPlan's Internal Registries and Counters (in alphabetical order)

Name	Description
statsCache	Cached plan statistics (as <code>statistics</code>) of the <code>LogicalPlan</code> . Computed and cached in <code>stats</code> . Used in <code>stats</code> and <code>verboseStringWithSuffix</code> . Reset in <code>invalidateStatsCache</code>

Getting Cached or Calculating Estimated Statistics — `stats` Method

```
stats(conf: CatalystConf): Statistics
```

`stats` returns the `cached plan statistics` or `computes a new one` (and caches it as `statsCache`).

Note

- `stats` is used when:
- A `LogicalPlan` computes statistics
 - `QueryExecution` builds complete text representation
 - `JoinSelection` checks whether a plan can be broadcast et al
 - `CostBasedJoinReorder` attempts to reorder inner joins
 - `LimitPushDown` is executed (for `FullOuter` join)
 - `AggregateEstimation` estimates statistics
 - `FilterEstimation` estimates child statistics
 - `InnerOuterEstimation` estimates statistics of the left and right sides of a join
 - `LeftSemiAntiEstimation` estimates statistics
 - `ProjectEstimation` estimates statistics

invalidateStatsCache method

Caution

FIXME

verboseStringWithSuffix method

Caution

FIXME

resolveQuoted method

Caution

FIXME

setAnalyzed method

Caution

FIXME

Command — Logical Commands

`Command` is the base for `leaf logical plans` that represent non-query commands to be executed by the system. It defines `output` to return an empty collection of `Attributes`.

Known commands are:

1. CreateTable
2. Any RunnableCommand

Is Operator Streaming? — `isStreaming` method

```
isStreaming: Boolean
```

`isStreaming` is a part of the public API of `LogicalPlan` and is enabled (i.e. `true`) when a logical plan is a [streaming source](#).

By default, it walks over subtrees and calls itself, i.e. `isStreaming`, on every child node to find a streaming source.

```
val spark: SparkSession = ...

// Regular dataset
scala> val ints = spark.createDataset(0 to 9)
ints: org.apache.spark.sql.Dataset[Int] = [value: int]

scala> ints.queryExecution.logical.isStreaming
res1: Boolean = false

// Streaming dataset
scala> val logs = spark.readStream.format("text").load("logs/*.out")
logs: org.apache.spark.sql.DataFrame = [value: string]

scala> logs.queryExecution.logical.isStreaming
res2: Boolean = true
```

Note	Streaming Datasets are part of Structured Streaming.
------	--

Computing Statistics Estimates (of All Child Logical Operators) for Cost-Based Optimizer — `computeStats` method

```
computeStats(conf: CatalystConf): Statistics
```

`computeStats` creates a `Statistics` with `sizeInBytes` as a product of `statistics` of all child logical plans.

For a no-children logical plan, `computeStats` reports a `UnsupportedOperationException`:

LeafNode [nodeName] must implement statistics.

Note `computeStats` is a `protected` method that logical operators are expected to override to provide their own custom plan statistics calculation.

Note `computeStats` is used exclusively when `LogicalPlan` is requested for logical plan statistics estimates.

Aggregate Unary Logical Operator

Aggregate is a unary logical operator that holds the following:

- Grouping expressions
- Aggregate named expressions
- Child logical plan

Aggregate is created to represent the following after a logical plan is analyzed:

- SQL's GROUP BY clause (possibly with WITH CUBE or WITH ROLLUP) in AstBuilder
- RelationalGroupedDataset aggregations (e.g. pivot)
- KeyValueGroupedDataset aggregations
- AnalyzeColumnCommand

Note	Aggregate logical operator is translated to one of HashAggregateExec, objectHashAggregateExec OR SortAggregateExec physical operators in Aggregation execution planning strategy.
------	---

Table 1. Aggregate's Properties (in alphabetical order)

Name	Description			
maxRows	<p>Child logical plan's <code>maxRows</code></p> <table border="1"> <tr> <td>Note</td><td>Part of LogicalPlan contract.</td></tr> </table>		Note	Part of LogicalPlan contract .
Note	Part of LogicalPlan contract .			
output	<p>Attributes of aggregate named expressions</p> <table border="1"> <tr> <td>Note</td><td>Part of QueryPlan contract.</td></tr> </table>		Note	Part of QueryPlan contract .
Note	Part of QueryPlan contract .			
resolved	<p>Enabled when:</p> <ul style="list-style-type: none"> • expressions and child logical plan are resolved • No WindowExpressions exist in aggregate named expressions <table border="1"> <tr> <td>Note</td><td>Part of LogicalPlan contract.</td></tr> </table>		Note	Part of LogicalPlan contract .
Note	Part of LogicalPlan contract .			
validConstraints	<p>The (expression) constraints of child logical plan and non-aggregate aggregate named expressions.</p> <table border="1"> <tr> <td>Note</td><td>Part of QueryPlan contract.</td></tr> </table>		Note	Part of QueryPlan contract .
Note	Part of QueryPlan contract .			

computeStats Method

Caution	FIXME
Note	<code>computeStats</code> is a part of LogicalPlan Contract to calculating statistics estimates (for cost-based optimizer).

Rule-Based Logical Optimization Phase

[PushDownPredicate](#) logical plan optimization applies so-called **filter pushdown** to a [Pivot](#) operator when under `Filter` operator and with all expressions deterministic.

```

import org.apache.spark.sql.catalyst.optimizer.PushDownPredicate

val q = visits
  .groupBy("city")
  .pivot("year")
  .count()
  .where($"city" === "Boston")

val pivotPlanAnalyzed = q.queryExecution.analyzed
scala> println(pivotPlanAnalyzed.numberedTreeString)
00 Filter (city#8 = Boston)
01 +- Project [city#8, __pivot_count(1) AS `count` AS `count(1) AS ``count````#142[0] AS
  2015#143L, __pivot_count(1) AS `count` AS `count(1) AS ``count````#142[1] AS 2016#144L
, __pivot_count(1) AS `count` AS `count(1) AS ``count````#142[2] AS 2017#145L]
02   +- Aggregate [city#8], [city#8, pivotfirst(year#9, count(1) AS `count`#134L, 2015
, 2016, 2017, 0, 0) AS __pivot_count(1) AS `count` AS `count(1) AS ``count````#142]
03     +- Aggregate [city#8, year#9], [city#8, year#9, count(1) AS count(1) AS `coun
t`#134L]
04       +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9]
05         +- LocalRelation [_1#3, _2#4, _3#5]

val afterPushDown = PushDownPredicate(pivotPlanAnalyzed)
scala> println(afterPushDown.numberedTreeString)
00 Project [city#8, __pivot_count(1) AS `count` AS `count(1) AS ``count````#142[0] AS 2
015#143L, __pivot_count(1) AS `count` AS `count(1) AS ``count````#142[1] AS 2016#144L,
__pivot_count(1) AS `count` AS `count(1) AS ``count````#142[2] AS 2017#145L]
01 +- Aggregate [city#8], [city#8, pivotfirst(year#9, count(1) AS `count`#134L, 2015,
2016, 2017, 0, 0) AS __pivot_count(1) AS `count` AS `count(1) AS ``count````#142]
02   +- Aggregate [city#8, year#9], [city#8, year#9, count(1) AS count(1) AS `count`#
134L]
03     +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9]
04       +- Filter (_2#4 = Boston)
05         +- LocalRelation [_1#3, _2#4, _3#5]

```

BroadcastHint Unary Logical Operator

`BroadcastHint` is a [unary logical operator](#) that acts as a hint for...[FIXME](#)

`BroadcastHint` is added to a [logical plan](#) when:

- Analyzer [resolves broadcast hints](#), i.e. `BROADCAST`, `BROADCASTJOIN` and `MAPJOIN` hints in SQL queries (see [the example](#))
- `broadcast` function is used (see [the example](#))

BroadcastHint and SQL's Hints

```
Seq((0, "aa"), (0, "bb"))
    .toDF("id", "token")
    .createOrReplaceTempView("left")

Seq(("aa", 0.99), ("bb", 0.57))
    .toDF("token", "prob")
    .createOrReplaceTempView("right")

scala> spark.catalog.listTables.filter('name.like("left") or 'name.like("right")).show
+-----+-----+-----+-----+
| name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
| left|    null|      null|TEMPORARY|     true|
| right|   null|      null|TEMPORARY|     true|
+-----+-----+-----+-----+

val query = """
    | EXPLAIN COST
    | SELECT /*+ BROADCAST (right) */ *
    | FROM left, right
    | WHERE left.token = right.token
    | """

val cost = sql(query).as[String].collect()(0)

scala> println(cost)
== Parsed Logical Plan ==
'Hint BROADCAST, [right]
+- 'Project [*]
  +- 'Filter ('left.token = 'right.token)
    +- 'Join Inner
      :- 'UnresolvedRelation `left`
      +- 'UnresolvedRelation `right`

== Analyzed Logical Plan ==
```

```

id: int, token: string, token: string, prob: double
Project [id#184, token#185, token#195, prob#196]
+- Filter (token#185 = token#195)
  +- Join Inner
    :- SubqueryAlias left
      :  +- Project [_1#181 AS id#184, _2#182 AS token#185]
      :    +- LocalRelation [_1#181, _2#182]
    +- BroadcastHint
      +- SubqueryAlias right
        +- Project [_1#192 AS token#195, _2#193 AS prob#196]
          +- LocalRelation [_1#192, _2#193]

== Optimized Logical Plan ==
Join Inner, (token#185 = token#195), Statistics(sizeInBytes=2.6 KB, isBroadcastable=false)
:- Project [_1#181 AS id#184, _2#182 AS token#185], Statistics(sizeInBytes=48.0 B, isBroadcastable=false)
:  +- Filter isnotnull(_2#182), Statistics(sizeInBytes=48.0 B, isBroadcastable=false)
:    +- LocalRelation [_1#181, _2#182], Statistics(sizeInBytes=48.0 B, isBroadcastable=false)
+- BroadcastHint, Statistics(sizeInBytes=56.0 B, isBroadcastable=true)
  +- Project [_1#192 AS token#195, _2#193 AS prob#196], Statistics(sizeInBytes=56.0 B, isBroadcastable=false)
    +- Filter isnotnull(_1#192), Statistics(sizeInBytes=56.0 B, isBroadcastable=false)
      +- LocalRelation [_1#192, _2#193], Statistics(sizeInBytes=56.0 B, isBroadcastable=false)

== Physical Plan ==
*BroadcastHashJoin [token#185], [token#195], Inner, BuildRight
:- *Project [_1#181 AS id#184, _2#182 AS token#185]
:  +- *Filter isnotnull(_2#182)
:    +- LocalTableScan [_1#181, _2#182]
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
  +- *Project [_1#192 AS token#195, _2#193 AS prob#196]
    +- *Filter isnotnull(_1#192)
      +- LocalTableScan [_1#192, _2#193]

```

BroadcastHint and broadcast function

```

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> println(left.join(broadcast(right), "token").queryExecution.toStringWithStats)
== Parsed Logical Plan ==
'Join UsingJoin(Inner, List(token))
:- Project [_1#123 AS id#126, _2#124 AS token#127]
:  +- LocalRelation [_1#123, _2#124]
+- BroadcastHint
  +- Project [_1#136 AS token#139, _2#137 AS prob#140]
    +- LocalRelation [_1#136, _2#137]

```

```

== Analyzed Logical Plan ==
token: string, id: int, prob: double
Project [token#127, id#126, prob#140]
+- Join Inner, (token#127 = token#139)
  :- Project [_1#123 AS id#126, _2#124 AS token#127]
  :   +- LocalRelation [_1#123, _2#124]
  +- BroadcastHint
    +- Project [_1#136 AS token#139, _2#137 AS prob#140]
      +- LocalRelation [_1#136, _2#137]

== Optimized Logical Plan ==
Project [token#127, id#126, prob#140], Statistics(sizeInBytes=1792.0 B, isBroadcastable=false)
+- Join Inner, (token#127 = token#139), Statistics(sizeInBytes=2.6 KB, isBroadcastable=false)
  :- Project [_1#123 AS id#126, _2#124 AS token#127], Statistics(sizeInBytes=48.0 B, isBroadcastable=false)
    :   +- Filter isnotnull(_2#124), Statistics(sizeInBytes=48.0 B, isBroadcastable=false)
  )
  :     +- LocalRelation [_1#123, _2#124], Statistics(sizeInBytes=48.0 B, isBroadcastable=false)
  +- BroadcastHint, Statistics(sizeInBytes=56.0 B, isBroadcastable=true)
    +- Project [_1#136 AS token#139, _2#137 AS prob#140], Statistics(sizeInBytes=56.0 B, isBroadcastable=false)
      +- Filter isnotnull(_1#136), Statistics(sizeInBytes=56.0 B, isBroadcastable=false)
    +- LocalRelation [_1#136, _2#137], Statistics(sizeInBytes=56.0 B, isBroadcastable=false)

== Physical Plan ==
*Project [token#127, id#126, prob#140]
+- *BroadcastHashJoin [token#127], [token#139], Inner, BuildRight
  :- *Project [_1#123 AS id#126, _2#124 AS token#127]
  :   +- *Filter isnotnull(_2#124)
  :     +- LocalTableScan [_1#123, _2#124]
  +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
    +- *Project [_1#136 AS token#139, _2#137 AS prob#140]
      +- *Filter isnotnull(_1#136)
        +- LocalTableScan [_1#136, _2#137]

```

computeStats Method

```
computeStats(conf: CatalystConf): Statistics
```

`computeStats` marks the parent as broadcast (i.e. `isBroadcastable` flag is enabled).

Note	<code>computeStats</code> is a part of LogicalPlan Contract .
------	---

DeserializeToObject Unary Logical Operator

```
case class DeserializeToObject(  
    deserializer: Expression,  
    outputObjAttr: Attribute,  
    child: LogicalPlan) extends UnaryNode with ObjectProducer
```

`DeserializeToObject` is a [unary logical operator](#) that takes the input row from the input `child` [logical plan](#) and turns it into the input `outputObjAttr` [attribute](#) using the given `deserializer` [expression](#).

`DeserializeToObject` is a `ObjectProducer` which produces domain objects as output. `DeserializeToObject`'s output is a single-field safe row containing the produced object.

Note

`DeserializeToObject` is the result of [CatalystSerde.deserialize](#).

Expand Unary Logical Operator

`Expand` is a [unary logical operator](#) that represents `Cube`, `Rollup`, [GroupingSets](#) and [TimeWindow](#) logical operators after they have been resolved at [analysis phase](#).

```
FIXME Examples for
1. Cube
2. Rollup
3. GroupingSets
4. See TimeWindow

val q = ...

scala> println(q.queryExecution.logical.numberedTreeString)
...
```

Note	<code>Expand</code> logical operator is translated to <code>ExpandExec</code> physical operator in BasicOperators execution planning strategy.
------	--

Table 1. Expand's Properties (in alphabetical order)

Name	Description
<code>references</code>	<code>AttributeSet</code> from projections
<code>validConstraints</code>	Empty set of expressions

Analysis Phase

`Expand` logical operator is resolved to at [analysis phase](#) in the following logical evaluation rules:

- [ResolveGroupingAnalytics](#) (for `Cube`, `Rollup`, [GroupingSets](#) logical operators)
- [TimeWindowing](#) (for [TimeWindow](#) logical operator)

Note	Aggregate → (Cube Rollup GroupingSets) → constructAggregate → constructExpand
------	---

```

val spark: SparkSession = ...
// using q from the example above
val plan = q.queryExecution.logical

scala> println(plan.numberedTreeString)
...FIXME

```

Rule-Based Logical Optimization Phase

- ColumnPruning
- FoldablePropagation
- RewriteDistinctAggregates

computeStats Method

Caution	FIXME
Note	computeStats is a part of LogicalPlan Contract to calculating statistics estimates (for cost-based optimizer).

Creating Expand Instance

`Expand` takes the following when created:

- Projection [expressions](#)
- Output attributes
- Child [logical plan](#)

GroupingSets Unary Logical Operator

`GroupingSets` is a [unary logical operator](#) that represents SQL's [GROUPING SETS](#) variant of `GROUP BY` clause.

```
val q = sql("""
    SELECT customer, year, SUM(sales)
    FROM VALUES ("abc", 2017, 30) AS t1 (customer, year, sales)
    GROUP BY customer, year
    GROUPING SETS ((customer), (year))
""")  

scala> println(q.queryExecution.logical.numberedTreeString)
00 'GroupingSets [ArrayBuffer('customer), ArrayBuffer('year)], ['customer, 'year], ['c
ustomer, 'year, unresolvedalias('SUM('sales), None)]
01 +- 'SubqueryAlias t1
02     +- 'UnresolvedInlineTable [customer, year, sales], [List(abc, 2017, 30)]
```

`GroupingSets` operator is resolved to an `Aggregate` logical operator at [analysis phase](#).

```
scala> println(q.queryExecution.analyzed.numberedTreeString)
00 Aggregate [customer#8, year#9, spark_grouping_id#5], [customer#8, year#9, sum(cast(
  sales#2 as bigint)) AS sum(sales)#4L]
01 +- Expand [List(customer#0, year#1, sales#2, customer#6, null, 1), List(customer#0,
  year#1, sales#2, null, year#7, 2)], [customer#0, year#1, sales#2, customer#8, year#9,
  spark_grouping_id#5]
02     +- Project [customer#0, year#1, sales#2, customer#0 AS customer#6, year#1 AS ye
  ar#7]
03         +- SubqueryAlias t1
04             +- LocalRelation [customer#0, year#1, sales#2]
```

Note	<code>GroupingSets</code> can only be created using SQL.
------	--

Note	<code>GroupingSets</code> is not supported on Structured Streaming's streaming Datasets .
------	---

`GroupingSets` is never resolved (as it can only be converted to an `Aggregate` logical operator).

`GroupingSets`'s output [attributes](#) are the attributes of [aggregate named expressions](#).

Analysis Phase

`GroupingSets` operator is resolved at [analysis phase](#) in the following logical evaluation rules:

- [ResolveAliases](#) for unresolved aliases in aggregate named expressions

- [ResolveGroupingAnalytics](#)

`GroupingSets` operator is resolved to an [Aggregate](#) with [Expand](#) logical operators.

```
val spark: SparkSession = ...
// using q from the example above
val plan = q.queryExecution.logical

scala> println(plan.numberedTreeString)
00 'GroupingSets [ArrayBuffer('customer), ArrayBuffer('year)], ['customer, 'year], ['customer, 'year, unresolvedalias('SUM('sales), None)]
01 +- 'SubqueryAlias t1
02   +- 'UnresolvedInlineTable [customer, year, sales], [List(abc, 2017, 30)]

// Note unresolvedalias for SUM expression
// Note UnresolvedInlineTable and SubqueryAlias

// FIXME Show the evaluation rules to get rid of the unresolvable parts
```

Creating GroupingSets Instance

`GroupingSets` takes the following when created:

- [Expressions](#) from `GROUPING SETS` clause
- Grouping [expressions](#) from `GROUP BY` clause
- Child [logical plan](#)
- Aggregate [named expressions](#)

Hint Logical Operator

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

InMemoryRelation Leaf Logical Operator For Cached Query Plans

`InMemoryRelation` is a [leaf logical operator](#) that represents a cached [physical query plan](#).

`InMemoryRelation` is created when `cacheManager` is requested to [cache a Dataset](#).

```
// Cache sample table range5 using pure SQL
// That registers range5 to contain the output of range(5) function
spark.sql("CACHE TABLE range5 AS SELECT * FROM range(5)")
val q1 = spark.sql("SELECT * FROM range5")
scala> q1.explain
== Physical Plan ==
InMemoryTableScan [id#0L]
+- InMemoryRelation [id#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1
replicas), `range5`
    +- *Range (0, 5, step=1, splits=8)

// you could also use optimizedPlan to see InMemoryRelation
scala> println(q1.queryExecution.optimizedPlan.numberedTreeString)
00 InMemoryRelation [id#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 r
eplicas), `range5`
01     +- *Range (0, 5, step=1, splits=8)

// Use Dataset's cache
val q2 = spark.range(10).groupBy('id % 5).count.cache
scala> println(q2.queryExecution.optimizedPlan.numberedTreeString)
00 InMemoryRelation [(id % 5)#84L, count#83L], true, 10000, StorageLevel(disk, memory,
deserialized, 1 replicas)
01     +- *HashAggregate(keys=[(id#77L % 5)#88L], functions=[count(1)], output=[(id % 5
)#84L, count#83L])
02         +- Exchange hashpartitioning((id#77L % 5)#88L, 200)
03             +- *HashAggregate(keys=[(id#77L % 5) AS (id#77L % 5)#88L], functions=[part
ial_count(1)], output=[(id#77L % 5)#88L, count#90L])
04                 +- *Range (0, 10, step=1, splits=8)
```

`InMemoryRelation` is a `MultiInstanceRelation` which means that the same instance will appear multiple times in a physical plan.

```
// Cache a Dataset
val q = spark.range(10).cache

// Make sure that q Dataset is cached
val cache = spark.sharedState.cacheManager
scala> cache.lookupCachedData(q.queryExecution.logical).isDefined
res0: Boolean = true

scala> q.explain
== Physical Plan ==
InMemoryTableScan [id#122L]
+- InMemoryRelation [id#122L], true, 10000, StorageLevel(disk, memory, deserialized
, 1 replicas)
    +- *Range (0, 10, step=1, splits=8)

val qCrossJoined = q.crossJoin(q)
scala> println(qCrossJoined.queryExecution.optimizedPlan.numberedTreeString)
00 Join Cross
01 :- InMemoryRelation [id#122L], true, 10000, StorageLevel(disk, memory, deserialized
, 1 replicas)
02 :   +- *Range (0, 10, step=1, splits=8)
03 +- InMemoryRelation [id#170L], true, 10000, StorageLevel(disk, memory, deserialized
, 1 replicas)
04       +- *Range (0, 10, step=1, splits=8)

// Use sameResult for comparison
// since the plans use different output attributes
// and have to be canonicalized internally
import org.apache.spark.sql.execution.columnar.InMemoryRelation
val optimizedPlan = qCrossJoined.queryExecution.optimizedPlan
scala> optimizedPlan.children(0).sameResult(optimizedPlan.children(1))
res1: Boolean = true
```

Note

`InMemoryRelation` can be created using `apply` factory method that has no [output attributes](#) (and uses `child` physical plan's `output`).

```
apply(
  useCompression: Boolean,
  batchSize: Int,
  storageLevel: StorageLevel,
  child: SparkPlan,
  tableName: Option[String]): InMemoryRelation
```

Creating InMemoryRelation Instance

`InMemoryRelation` takes the following when created:

- [Output attributes](#)

- `useCompression` flag
- batch size
- Storage level
- Child physical plan
- Optional table name

Join Logical Operator

`Join` is a [binary logical operator](#), i.e. works with two logical operators. `Join` has a join type and an optional expression condition for the join.

```
class Join(  
    left: LogicalPlan,  
    right: LogicalPlan,  
    joinType: JoinType,  
    condition: Option[Expression])  
extends BinaryNode
```

LocalRelation Logical Query Plan

`LocalRelation` is a [leaf logical plan](#) that allow functions like `collect` or `take` to be executed locally, i.e. without using Spark executors.

Note

When `Dataset` operators can be executed locally, the `Dataset` is considered [local](#).

`LocalRelation` represents `Datasets` that were created from local collections using [SparkSession.emptyDataset](#) or [SparkSession.createDataset](#) methods and their derivatives like [toDF](#).

```
val dataset = Seq(1).toDF
scala> dataset.explain(true)
== Parsed Logical Plan ==
LocalRelation [value#216]

== Analyzed Logical Plan ==
value: int
LocalRelation [value#216]

== Optimized Logical Plan ==
LocalRelation [value#216]

== Physical Plan ==
LocalTableScan [value#216]
```

It can only be constructed with the output attributes being all resolved.

The size of the objects (in `statistics`) is the sum of the default size of the attributes multiplied by the number of records.

When executed, `LocalRelation` is translated to [LocalTableScanExec](#) physical operator.

LogicalRelation Logical Operator — Adapter for BaseRelation

`LogicalRelation` is a [leaf logical operator](#) that acts as an adapter for [BaseRelation](#) in a [logical query plan](#).

```
val q1 = spark.read.option("header", true).csv("../datasets/people.csv")
scala> println(q1.queryExecution.logical.numberedTreeString)
00 Relation[id#72,name#73,age#74] csv

val q2 = sql("select * from `csv`..../datasets/people.csv`")
scala> println(q2.queryExecution.optimizedPlan.numberedTreeString)
00 Relation[_c0#175,_c1#176,_c2#177] csv
```

`LogicalRelation` is [created](#) when:

- `DataFrameReader` loads data from a data source that supports multiple paths (through `SparkSession.baseRelationToDataFrame`)
- `DataFrameReader` loads data from an external table using JDBC (through `SparkSession.baseRelationToDataFrame`)
- `TextInputCSVDataSource` and `TextInputJsonDataSource` are requested to infer schema
- `ResolveSQLOnFile` converts a logical plan
- `FindDataSourceTable` converts a logical plan
- `RelationConversions` converts a logical plan
- `CreateTempViewUsing` runnable command is executed
- Structured Streaming's `FileStreamSource` creates batches of records

<p>Note</p>	<p><code>LogicalRelation</code> is created using <code>apply</code> factory methods that accept <code>BaseRelation</code> with optional <code>CatalogTable</code>.</p>
--------------------	--

```
apply(relation: BaseRelation): LogicalRelation
apply(relation: BaseRelation, table: CatalogTable): LogicalRelation
```

Creating LogicalRelation Instance

`LogicalRelation` takes the following when created:

- **BaseRelation**
- **Output** AttributeReferences
- **Optional** CatalogTable

Pivot Unary Logical Operator

Pivot is a unary logical operator that represents pivot operator.

```
val visits = Seq(
  (0, "Warsaw", 2015),
  (1, "Warsaw", 2016),
  (2, "Boston", 2017)
).toDF("id", "city", "year")

val q = visits
  .groupBy("city")
  .pivot("year", Seq("2015", "2016", "2017"))
  .count()
scala> println(q.queryExecution.logical.numberedTreeString)
00 Pivot [city#8], year#9: int, [2015, 2016, 2017], [count(1) AS count#157L]
01 +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9]
02   +- LocalRelation [_1#3, _2#4, _3#5]
```

Pivot is created when RelationalGroupedDataset creates a DataFrame for an aggregate operator.

Analysis Phase

Pivot operator is resolved at analysis phase in the following logical evaluation rules:

- ResolveAliases
- ResolvePivot

```
val spark: SparkSession = ...

import spark.sessionState.analyzer.ResolveAliases
// see q in the example above
val plan = q.queryExecution.logical

scala> println(plan.numberedTreeString)
00 Pivot [city#8], year#9: int, [2015, 2016, 2017], [count(1) AS count#24L]
01 +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9]
02   +- LocalRelation [_1#3, _2#4, _3#5]

// FIXME Find a plan to show the effect of ResolveAliases
val planResolved = ResolveAliases(plan)
```

`Pivot` operator "disappears" behind (i.e. is converted to) a [Aggregate](#) logical operator (possibly under `Project` operator).

```
import spark.sessionState.analyzer.ResolvePivot
val planAfterResolvePivot = ResolvePivot(plan)
scala> println(planAfterResolvePivot.numberedTreeString)
00 Project [city#8, __pivot_count(1) AS `count` AS `count(1) AS ``count``#62[0] AS 20
15#63L, __pivot_count(1) AS `count` AS `count(1) AS ``count``#62[1] AS 2016#64L, __pi
pivot_count(1) AS `count` AS `count(1) AS ``count``#62[2] AS 2017#65L]
01 +- Aggregate [city#8], [city#8, pivotfirst(year#9, count(1) AS `count`#54L, 2015, 2
016, 2017, 0, 0) AS __pivot_count(1) AS `count` AS `count(1) AS ``count``#62]
02     +- Aggregate [city#8, year#9], [city#8, year#9, count(1) AS count#24L AS count(1
) AS `count`#54L]
03         +- Project [_1#3 AS id#7, _2#4 AS city#8, _3#5 AS year#9]
04             +- LocalRelation [_1#3, _2#4, _3#5]
```

Creating Pivot Instance

`Pivot` takes the following when created:

- Grouping [named expressions](#)
- Pivot column [expression](#)
- Pivot values [literals](#)
- Aggregation [expressions](#)
- Child [logical plan](#)

Repartition Logical Operators — Repartition and RepartitionByExpression

[Repartition](#) and [RepartitionByExpression](#) (**repartition operations** in short) are unary logical operators that create a new `RDD` that has exactly `numPartitions` partitions.

Note

`RepartitionByExpression` is also called **distribute** operator.

[Repartition](#) is the result of [coalesce](#) or [repartition](#) (with no partition expressions defined) operators.

```
val rangeAlone = spark.range(5)

scala> rangeAlone.rdd.getNumPartitions
res0: Int = 8

// Repartition the records

val withRepartition = rangeAlone.repartition(numPartitions = 5)

scala> withRepartition.rdd.getNumPartitions
res1: Int = 5

scala> withRepartition.explain(true)
== Parsed Logical Plan ==
Repartition 5, true
+- Range (0, 5, step=1, splits=Some(8))

// ...

== Physical Plan ==
Exchange RoundRobinPartitioning(5)
+- *Range (0, 5, step=1, splits=Some(8))

// Coalesce the records

val withCoalesce = rangeAlone.coalesce(numPartitions = 5)
scala> withCoalesce.explain(true)
== Parsed Logical Plan ==
Repartition 5, false
+- Range (0, 5, step=1, splits=Some(8))

// ...

== Physical Plan ==
Coalesce 5
+- *Range (0, 5, step=1, splits=Some(8))
```

[RepartitionByExpression](#) is the result of [repartition](#) operator with explicit partition expressions defined and SQL's [DISTRIBUTE BY clause](#).

```
// RepartitionByExpression
// 1) Column-based partition expression only
scala> rangeAlone.repartition(partitionExprs = 'id % 2).explain(true)
== Parsed Logical Plan ==
'RepartitionByExpression [('id % 2)], 200
+- Range (0, 5, step=1, splits=Some(8))

// ...

== Physical Plan ==
Exchange hashpartitioning((id#10L % 2), 200)
+- *Range (0, 5, step=1, splits=Some(8))

// 2) Explicit number of partitions and partition expression
scala> rangeAlone.repartition(numPartitions = 2, partitionExprs = 'id % 2).explain(true)
)
== Parsed Logical Plan ==
'RepartitionByExpression [('id % 2)], 2
+- Range (0, 5, step=1, splits=Some(8))

// ...

== Physical Plan ==
Exchange hashpartitioning((id#10L % 2), 2)
+- *Range (0, 5, step=1, splits=Some(8))
```

`Repartition` and `RepartitionByExpression` logical operators are described by:

- `shuffle` flag
- target number of partitions

Note	BasicOperators strategy maps <code>Repartition</code> to ShuffleExchange (with <code>RoundRobinPartitioning</code> expression) or CoalesceExec physical operators per shuffle — enabled or not, respectively.
------	---

Note	BasicOperators strategy maps <code>RepartitionByExpression</code> to ShuffleExchange physical operator with <code>HashPartitioning</code> expression.
------	---

Repartition Operation Optimizations

1. [CollapseRepartition](#) logical optimization collapses adjacent repartition operations.
2. Repartition operations allow [FoldablePropagation](#) and [PushDownPredicate](#) logical optimizations to "push through".

3. [PropagateEmptyRelation](#) logical optimization may result in an empty [LocalRelation](#) for repartition operations.

RunnableCommand — Generic Logical Command with Side Effects

`RunnableCommand` is the generic [logical command](#) that is [executed](#) for its side effects.

`RunnableCommand` defines one abstract method `run` that computes a collection of `Row` records with the side effect, i.e. the result of executing a command.

```
run(sparkSession: SparkSession): Seq[Row]
```

Note	<code>RunnableCommand</code> logical operator is translated to ExecutedCommandExec physical operator in BasicOperators strategy.
------	--

Note	<code>run</code> is executed when: <ul style="list-style-type: none"> • <code>ExecutedCommandExec</code> executes logical <code>RunnableCommand</code> and caches the result as InternalRows • <code>InsertIntoHadoopFsRelationCommand</code> runs • <code>QueryExecution</code> transforms the result of executing <code>DescribeTableCommand</code> to a Hive-compatible output format
------	---

Table 1. Available RunnableCommands (in alphabetical order)

RunnableCommand	Description
AddFileCommand	
AddJarCommand	
AlterDatabasePropertiesCommand	
AlterTableAddPartitionCommand	
AlterTableChangeColumnCommand	
AlterTableDropPartitionCommand	
AlterTableRecoverPartitionsCommand	
AlterTableRenameCommand	
AlterTableRenamePartitionCommand	

AlterTableSerDePropertiesCommand					
AlterTableSetLocationCommand					
AlterTableSetPropertiesCommand					
AlterTableUnsetPropertiesCommand					
AlterViewAsCommand					
AnalyzeColumnCommand					
AnalyzeTableCommand					
CacheTableCommand	<p>When executed, CacheTableCommand creates a DataFrame by registering a temporary view for the optional query.</p> <pre>CACHE LAZY? TABLE [table] (AS? [query])?</pre> <p>CacheTableCommand requests the session-specific catalog for the table.</p> <table border="1"> <tr> <td>Note</td> <td>CacheTableCommand uses SparkSession's Catalog.</td> </tr> </table> <p>If the caching is not LAZY (which is not by default), CacheTableCommand creates a DataFrame for the rows (that will trigger the caching).</p> <table border="1"> <tr> <td>Note</td> <td>CacheTableCommand uses a Spark SQL DataFrame caching by executing the query.</td> </tr> </table> <pre>val q = "CACHE TABLE ids AS SELECT * from random" scala> println(sql(q).queryExecution.logical 00 CacheTableCommand `ids`, false 01 +- 'Project [*] 02 +- 'UnresolvedTableValuedFunction random val q2 = "CACHE LAZY TABLE ids" scala> println(sql(q2).queryExecution.logical 17/05/17 06:16:39 WARN CacheManager: Asked to d data. 00 CacheTableCommand `ids`, true</pre>	Note	CacheTableCommand uses SparkSession's Catalog.	Note	CacheTableCommand uses a Spark SQL DataFrame caching by executing the query.
Note	CacheTableCommand uses SparkSession's Catalog.				
Note	CacheTableCommand uses a Spark SQL DataFrame caching by executing the query.				
ClearCacheCommand					
CreateDatabaseCommand					
	When executed, ...FIXME				

createDataSourceTableAsSelectCommand	Used exclusively when DataSourceAnalysis evaluates the <code>CreateTable</code> logical operator with queries using providers (which is when <code>DataFrameWriter save</code> to a non-Hive table or for Create Table As Select SQL)
CreateDataSourceTableCommand	
CreateFunctionCommand	
CreateHiveTableAsSelectCommand	
CreateTableCommand	
CreateTableLikeCommand	
CreateTempViewUsing	
CreateViewCommand	
DescribeDatabaseCommand	
DescribeFunctionCommand	
DescribeTableCommand	
DropDatabaseCommand	
DropFunctionCommand	
DropTableCommand	
ExplainCommand	
InsertIntoDataSourceCommand	
InsertIntoHadoopFsRelationCommand	
InsertIntoHiveTable	
ListFilesCommand	
ListJarsCommand	
LoadDataCommand	

RefreshResource	
RefreshTable	
ResetCommand	
SaveIntoDataSourceCommand	<p>When executed, requests <code>DataSource</code> to write a source per save mode.</p> <p>Used exclusively when <code>DataFrameWriter</code> is requ DataFrame to a data source.</p>
SetCommand	
SetDatabaseCommand	
ShowColumnsCommand	
ShowCreateTableCommand	
ShowDatabasesCommand	
ShowFunctionsCommand	
ShowPartitionsCommand	
ShowTablePropertiesCommand	
ShowTablesCommand	
StreamingExplainCommand	
TruncateTableCommand	
UncacheTableCommand	

AlterViewAsCommand Logical Command

`AlterViewAsCommand` is a [logical command](#) to alter a view.

`AlterViewAsCommand` works with a table identifier (as `TableIdentifier`), the original SQL text, and a [LogicalPlan](#) for the SQL query.

`AlterViewAsCommand` corresponds to `ALTER VIEW` in SQL.

Note

`AlterViewAsCommand` is described by `alterViewQuery` labeled alternative in statement expression in `SqlBase.g4` and parsed using [SparkSqlParser](#).

When executed, `AlterViewAsCommand` attempts to [alter a temporary view in the current SessionCatalog](#) first, and if that "fails", [alters the permanent view](#).

run Method

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

alterPermanentView Method

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

ClearCacheCommand Logical Command

`ClearCacheCommand` is a [logical command](#) to remove all cached tables from the in-memory cache.

`ClearCacheCommand` corresponds to `CLEAR CACHE` in SQL.

Note	<code>clearCacheCommand</code> is described by <code>clearCache</code> labeled alternative in statement expression in <code>SqlBase.g4</code> and parsed using SparkSqlParser .
------	---

CreateDataSourceTableCommand Logical Command

`CreateDataSourceTableCommand` is a [logical command](#) that [creates a new table](#) (in a session-scoped `SessionCatalog`).

`CreateDataSourceTableCommand` is created exclusively when [DataSourceAnalysis](#) evaluation rule resolves `createTable` logical operator for a non-Hive table provider with no query.

`CreateDataSourceTableCommand` takes a `table` metadata and `ignoreIfExists` flag.

run Method

```
run(sparkSession: SparkSession): Seq[Row]
```

`run` [creates a new table](#) in a session-scoped `SessionCatalog` .

Note	<code>run</code> uses the input <code>SparkSession</code> to access SessionState that in turn is used to access the current SessionCatalog .
------	--

Internally, `run` [creates a BaseRelation](#) to access the table's schema.

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

Note	<code>run</code> accepts tables only (not views) with the provider defined.
------	---

Note	<code>run</code> is a part of RunnableCommand Contract .
------	--

CreateViewCommand Logical Command

`CreateViewCommand` is a [logical command](#) for creating a view or a table.

`CreateViewCommand` is a result of parsing `CREATE VIEW` (and variants) in SQL and executing `Dataset` operators: `createTempView`, `createOrReplaceTempView`, and `createGlobalTempView`.

Tip	<code>CreateViewCommand</code> is described by <code>createView</code> labeled alternative in <code>statement</code> expression in <code>SqlBase.g4</code> and parsed using SparkSqlParser .
-----	--

Caution	FIXME What's the difference between <code>createTempViewUsing</code> ?
---------	--

`CreateViewCommand` works with different view types (aka `viewType`).

Table 1. `CreateViewCommand`'s View Types

View Type	Description / Side Effect
<code>LocalTempView</code>	<p>A session-scoped local temporary view. Available until the session that has created it stops.</p> <p>When executed, <code>CreateViewCommand</code> requests the current SessionCatalog to create a temporary view.</p>
<code>GlobalTempView</code>	<p>A cross-session global temporary view. Available until a Spark application stops.</p> <p>When executed, <code>CreateViewCommand</code> requests the current SessionCatalog to create a global view.</p>
<code>PersistedView</code>	<p>A cross-session persisted view. Available until you drop it.</p> <p>When executed, <code>CreateViewCommand</code> checks if the table exists. If it does and replace is enabled <code>CreateViewCommand</code> requests the current SessionCatalog to alter a table. Otherwise, when the table does not exist, <code>CreateViewCommand</code> requests the current SessionCatalog to create it.</p>

run Method

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

ExplainCommand Logical Command

`ExplainCommand` is a [logical command](#) with side effect that allows users to see how a structured query is structured and will eventually be executed, i.e. shows logical and physical plans with or without details about codegen and cost.

When [executed](#), `ExplainCommand` computes a `QueryExecution` that is then used to output a single-column `DataFrame` with the following:

1. **codegen explain**, i.e. [WholeStageCodegen](#) subtrees if `codegen` flag is enabled.
2. **extended explain**, i.e. the parsed, analyzed, optimized logical plans with the physical plan if [extended](#) flag is enabled.
3. **cost explain**, i.e. [optimized logical plan](#) with stats if `cost` flag is enabled.
4. **simple explain**, i.e. the physical plan only when no `codegen` and `extended` flags are enabled.

`ExplainCommand` is created by Dataset's `explain` operator and `EXPLAIN` SQL statement (accepting `EXTENDED` and `CODEGEN` options).

```
// Explain in SQL

scala> sql("EXPLAIN EXTENDED show tables").show(truncate = 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
^__^
...
```

codegenString Attribute

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

output Attribute

Caution

[FIXME](#)

Creating ExplainCommand Instance

`ExplainCommand` takes the following when created:

- `LogicalPlan`
- `extended` flag whether to include extended details in the output when `ExplainCommand` is [executed](#) (disabled by default)
- `codegen` flag whether to include codegen details in the output when `ExplainCommand` is [executed](#) (disabled by default)
- `cost` flag whether to include code in the output when `ExplainCommand` is [executed](#) (disabled by default)

`ExplainCommand` initializes `output` attribute.

Note

`ExplainCommand` is created when...[FIXME](#)

Computing Text Representation of QueryExecution (as Single Row) — run Method

```
run(sparkSession: SparkSession): Seq[Row]
```

`run` computes `QueryExecution` and returns its text representation in a single `Row`.

Note

`run` is a part of `RunnableCommand Contract` to execute commands.

Internally, `run` creates a `IncrementalExecution` for a streaming dataset directly or requests `SessionState` to [execute the](#) `LogicalPlan`.

Note

`Streaming Dataset` is a part of Spark Structured Streaming.

`run` then requests `QueryExecution` to build the output text representation, i.e. `codegened`, `extended` (with logical and physical plans), `with stats`, or `simple`.

In the end, `run` [creates a](#) `Row` with the text representation.

SubqueryAlias Logical Operator

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

UnresolvedFunction Logical Operator

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

UnresolvedRelation Logical Operator

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

Window Unary Logical Operator

`Window` is a [unary logical operator](#) that is created for:

- a collection of [named expressions](#) (for windows)
- a collection of [expressions](#) (for partitions)
- a collection of `sortOrder` (for sorting) and a child [logical plan](#).

The `output` (collection of [Attributes](#)) is the child's attributes and the window's.

`Window` logical plan is a subject of pruning unnecessary window expressions in [ColumnPruning](#) rule and pushing filter operators in [PushDownPredicate](#) rule.

Analyzer — Logical Query Plan Analyzer

`Analyzer` is a **logical query plan analyzer** in Spark SQL that semantically validates and transforms an [unresolved logical plan](#) to an **analyzed logical plan** (with fully-typed relational entities) using [logical evaluation rules](#).

```
Analyzer: Unresolved Logical Plan ==> Analyzed Logical Plan
```

You can access a session-specific `Analyzer` through [SessionState](#).

```
val spark: SparkSession = ...
spark.sessionState.analyzer
```

You can access the analyzed logical plan of a `dataset` using `explain` (with `extended` flag enabled) or SQL's `EXPLAIN EXTENDED` operators.

```
// sample Dataset
val inventory = spark.range(5)
  .withColumn("new_column", 'id + 5 as "plus5")

// Using explain operator (with extended flag enabled)
scala> inventory.explain(extended = true)
== Parsed Logical Plan ==
'Project [*, ('id + 5) AS plus5#81 AS new_column#82]
+- Range (0, 5, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint, new_column: bigint
Project [id#78L, (id#78L + cast(5 as bigint)) AS new_column#82L]
+- Range (0, 5, step=1, splits=Some(8))

== Optimized Logical Plan ==
Project [id#78L, (id#78L + 5) AS new_column#82L]
+- Range (0, 5, step=1, splits=Some(8))

== Physical Plan ==
*Project [id#78L, (id#78L + 5) AS new_column#82L]
+- *Range (0, 5, step=1, splits=8)
```

Alternatively, you can also access the analyzed logical plan through `queryExecution`'s [analyzed](#) attribute (that together with `numberedTreeString` method is a very good "debugging" tool).

```
// With numberedTreeString to...please your eyes :)
scala> println(inventory.queryExecution.analyzed.numberedTreeString)
00 Project [id#78L, (id#78L + cast(5 as bigint)) AS new_column#82L]
01 +- Range (0, 5, step=1, splits=Some(8))
```

`Analyzer` defines `extendedResolutionRules` extension point for additional logical evaluation rules that a custom `Analyzer` can use to extend the `Resolution` batch. The rules are added at the end of the `Resolution` batch.

Note	<code>SessionState</code> uses its own <code>Analyzer</code> with custom <code>extendedResolutionRules</code> , <code>postHocResolutionRules</code> , and <code>extendedCheckRules</code> extension methods.
------	--

`Analyzer` is created while its owning `SessionState` is.

Table 1. Analyzer's Internal Registries and Counters (in alphabetical order)

Name	Description
<code>extendedResolutionRules</code>	Additional rules for <code>Resolution</code> batch. Empty by default
<code>fixedPoint</code>	FixedPoint with <code>maxIterations</code> for <code>Hints</code> , <code>Substitution</code> , <code>Resolution</code> and <code>Cleanup</code> batches. Set when <code>Analyzer</code> is created (and can be defined explicitly or through <code>optimizerMaxIterations</code> configuration setting).
<code>postHocResolutionRules</code>	The only rules in Post-Hoc Resolution batch if defined (that are executed in one pass, i.e. <code>once</code> strategy). Empty by default

`Analyzer` is used by `QueryExecution` to resolve the managed `LogicalPlan` (and, as a sort of follow-up, assert that a structured query has already been properly analyzed, i.e. no failed or unresolved or somehow broken logical plan operators and expressions exist).

Enable `TRACE` or `DEBUG` logging levels for the respective session-specific loggers to see what happens inside `Analyzer`.

- `org.apache.spark.sql.internal.SessionState$$anon$1`
- `org.apache.spark.sql.hive.HiveSessionStateBuilder$$anon$1` when [Hive support is enabled](#)

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

```
# with no Hive support
log4j.logger.org.apache.spark.sql.internal.SessionState$$anon$1=TRACE

# with Hive support enabled
log4j.logger.org.apache.spark.sql.hive.HiveSessionStateBuilder$$anon$1=DEBUG
```

Refer to [Logging](#).

The reason for such weird-looking logger names is that `analyzer` attribute is created as an anonymous subclass of `Analyzer` class in the respective `SessionStates`.

Executing Logical Evaluation Rules — `execute` Method

`Analyzer` is a [RuleExecutor](#) that defines the logical evaluation rules (i.e. resolving, removing, and in general modifying it), e.g.

- Resolves unresolved [relations](#) and [functions](#) (including `unresolvedGenerators`) using provided [SessionCatalog](#)
- ...

Table 2. Analyzer's Batches

Batch Name	Strategy	Rules	
Hints	FixedPoint	ResolveBroadcastHints	Adds a Broadcast Hint to <code>Unresolved</code> relations
		RemoveAllHints	Removes all broadcast hints
Simple Sanity Check	Once	LookupFunctions	Checks whether there are any <code>NoSuchFunction</code> errors
		CTESubstitution	Resolves CTE substitutions
			Substitutes values in expressions

Substitution	FixedPoint	WindowsSubstitution	For every window expression: If a window function: Window sp	
			Note 1 C	
		EliminateUnions	Eliminates	
		SubstituteUnresolvedOrdinals	Replaces ordinal values	
		ResolveTableValuedFunctions	Replaces table-valued functions	
		ResolveRelations	Resolves relations	
		ResolveReferences		
		ResolveCreateNamedStruct		
		ResolveDeserializer		
		ResolveNewInstance		
		ResolveUpCast		
		Resolves grouping: <ul style="list-style-type: none">Cube ,FilterSort '	Resolves grouping: <ul style="list-style-type: none">Cube ,FilterSort '	
		Expects that some iteration is done	Expects that some iteration is done	
		Fails analysis	Fails analysis	
		ResolveGroupingAnalytics	scala> sq org.apache.spark.sql. at org.apache.spark. at org.apache.spark. at org.apache.spark. at org.apache.spark.	

		Note
Resolution	ResolvePivot	Resolves P a single Ag
	ResolveOrdinalInOrderByAndGroupBy	
	ResolveMissingReferences	
	ExtractGenerator	
	ResolveGenerate	
	ResolveFunctions	Resolves U If Generator [name] generat
	ResolveAliases	Replaces <ul style="list-style-type: none">NamecMultiAAlias
ResolveSubquery		
ResolveWindowOrder		
ResolveWindowFrame	Resolves W	
ResolveNaturalAndUsingJoin		
ExtractWindowExpressions		
GlobalAggregates	Replaces (a logical oper	
ResolveAggregateFunctions	Resolves a Note F	

		TimeWindowing	Resolves T
			Note
		ResolveInlineTables	Resolves I
		TypeCoercion.typeCoercionRules	
		extendedResolutionRules	
Post-Hoc Resolution	Once	postHocResolutionRules	
View	Once	AliasViewChild	
Nondeterministic	Once	PullOutNondeterministic	
UDF	Once	HandleNullInputsForUDF	
FixNullability	Once	FixNullability	
ResolveTimeZone	Once	ResolveTimeZone	Replaces
Cleanup	FixedPoint	CleanupAliases	

Tip Consult the sources of [Analyzer](#) for the up-to-date list of the evaluation rules.

Creating Analyzer Instance

`Analyzer` takes the following when created:

- [SessionCatalog](#)
- [CatalystConf](#)
- Number of iterations before [FixedPoint](#) rule batches have converged (i.e. [Hints](#), [Substitution](#), [Resolution](#) and [Cleanup](#))

`Analyzer` initializes the [internal registries and counters](#).

Note `Analyzer` can also be created without specifying the `maxIterations` which is then configured using `optimizerMaxIterations` configuration setting.

resolver Method

```
resolver: Resolver
```

resolver requests [CatalystConf](#) for Resolver.

Note

Resolver is a mere function of two String parameters that returns true if both refer to the same entity (i.e. for case insensitive equality).

CheckAnalysis — Analysis Validation

`CheckAnalysis` defines `checkAnalysis` method that `Analyzer` uses to check if a `logical plan` is correct (after all the transformations) by applying `validation rules` and in the end marking it as analyzed.

Note	An analyzed logical plan is correct and ready for execution.
------	--

`CheckAnalysis` defines `extendedCheckRules` extension point that allows for extra analysis check rules.

Checking Results of Analysis of Logical Plan and Marking Plan As Analyzed — `checkAnalysis` Method

```
checkAnalysis(plan: LogicalPlan): Unit
```

`checkAnalysis` recursively checks the correctness of the analysis of the input `LogicalPlan` and marks it as analyzed.

Note	<code>checkAnalysis</code> fails analysis when finds <code>UnresolvedRelation</code> in the input <code>LogicalPlan</code> ... <code>FIXME</code> What else?
------	--

Internally, `checkAnalysis` processes nodes in the input `plan` (starting from the leafs, i.e. nodes down the operator tree).

`checkAnalysis` skips logical plans that have already undergone analysis.

Table 1. `checkAnaly`

LogicalPlan/Operator	
<code>UnresolvedRelation</code>	Fails analysis with the error message: <pre>Table or view not found: [tableIdentifier]</pre>
<code>Unresolved Attribute</code>	Fails analysis with the error message: <pre>cannot resolve '[expr]' given input columns: [from]</pre>
<code>Expression</code> with incorrect input data types	Fails analysis with the error message: <pre>cannot resolve '[expr]' due to data type mismatch: [message]</pre>

	Fails analysis with the error message: invalid cast from [dataType] to [dataType]
Unresolved cast	
Grouping	Fails analysis with the error message: grouping() can only be used with GroupingSets/Cube/Rollup
GroupingID	Fails analysis with the error message: grouping_id() can only be used with GroupingSets/Cube/Rollup
WindowExpression with AggregateExpression with <code>isDistinct</code> flag enabled	Fails analysis with the error message: Distinct window functions are not supported: [w] Example: <pre>val windowedDistinctCountExpr = "COUNT(DISTINCT value#95)" scala> spark.emptyDataset[Int].selectExpr(windowedDistinctCountExpr) org.apache.spark.sql.AnalysisException: Distinct window functions are not supported. windowspecdefinition(value#95, ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) Project [COUNT(1) OVER (PARTITION BY value UnspecifiedFrame)] +- Project [value#95, COUNT(1) OVER (PARTITION BY value UnspecifiedFrame)] +- UnspecifiedFrame +- Window [count(distinct 1) windowspecdefinition(value#95, ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)] +- Project [value#95] +- LocalRelation <empty>, [value#95] at org.apache.spark.sql.catalyst.analysis.CheckAnalysis.assertValidWindowSpecDefinition at org.apache.spark.sql.catalyst.analysis.Analyzer.analyzeWindowedExpression at org.apache.spark.sql.catalyst.analysis.Analyzer.analyze at org.apache.spark.sql.catalyst.analysis.CheckAnalysis.assertValidWindowSpecDefinition at org.apache.spark.sql.catalyst.analysis.CheckAnalysis.assertValidWindowSpecDefinition at org.apache.spark.sql.catalyst.analysis.CheckAnalysis.assertValidWindowSpecDefinition</pre>
FIXME	FIXME

After the validations, `checkAnalysis` executes additional check rules for correct analysis.

`checkAnalysis` then checks if `plan` is analyzed correctly (i.e. no logical plans are left unresolved). If there is one, `checkAnalysis` fails the analysis with `AnalysisException` and the following error message:

```
unresolved operator [o.simpleString]
```

In the end, `checkAnalysis` marks the entire logical plan as analyzed.

Note

`checkAnalysis` is used when:

- `QueryExecution` creates analyzed logical plan and checks its correctness (which happens mostly when a `Dataset` is created)
- `ExpressionEncoder` does `resolveAndBind`
- `ResolveAggregateFunctions` is executed (for `sort` logical plan)

Extra Analysis Check Rules — `extendedCheckRules` Extension Point

```
extendedCheckRules: Seq[LogicalPlan => Unit]
```

`extendedCheckRules` is a collection of rules (functions) that `checkAnalysis` uses for custom analysis checks (after the main validations have been executed).

Note

When a condition of a rule does not hold the function throws an `AnalysisException` directly or using `failAnalysis` method.

SparkOptimizer — Logical Query Optimizer

`SparkOptimizer` is the one and only custom [logical query plan optimizer](#) in Spark SQL that comes with the [additional logical plan optimizations](#).

Note

You can extend the available logical plan optimizations and register yours using [ExperimentalMethods](#).

`SparkOptimizer` is available as `optimizer` attribute of `SessionState`.

```
sparkSession.sessionState.optimizer
```

The result of applying the `batches` of `SparkOptimizer` to a `LogicalPlan` is called **optimizedPlan**. Optimized logical plan of a structured query is available as `optimizedPlan` attribute

Note

```
// Applying two filter in sequence on purpose
// We want to kick CombineTypedFilters optimizer in
val dataset = spark.range(10).filter(_ % 2 == 0).filter(_ == 0)

// optimizedPlan is a lazy value
// Only at the first time you call it you will trigger optimizations
// Next calls end up with the cached already-optimized result
// Use explain to trigger optimizations again
scala> dataset.queryExecution.optimizedPlan
res0: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)
+- Range (0, 10, step=1, splits=Some(8))
```

Table 1. SparkOptimizer's Optimization Rules (in the order of execution)

Batch Name	Strategy	Rules	Description
Optimize Metadata Only Query	Once	OptimizeMetadataOnlyQuery	
Extract Python UDF from Aggregate	Once	ExtractPythonUDFFromAggregate	
Prune File Source Table Partitions	Once	PruneFileSourcePartitions	
User Provided Optimizers	FixedPoint	extraOptimizations	

Tip	Enable <code>DEBUG</code> or <code>TRACE</code> logging levels for <code>org.apache.spark.sql.execution.SparkOptimizer</code> logger to see what happens inside. Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.sql.execution.SparkOptimizer=TRACE</code> Refer to Logging .

Creating SparkOptimizer Instance

`SparkOptimizer` takes the following when created:

- [SessionCatalog](#)
- [SQLConf](#)
- [ExperimentalMethods](#)

Note	<code>SparkOptimizer</code> is created when <code>SessionState</code> is created (that initializes <code>optimizer</code> property).
------	--

Further reading or watching

1. [Deep Dive into Spark SQL's Catalyst Optimizer](#)

2. (video) [Modern Spark DataFrame and Dataset \(Intermediate Tutorial\)](#) by Adam Breindel from Databricks.

Optimizer—Base for Logical Query Plan Optimizers

`optimizer` is the base **rule-based logical query plan optimizer** in Spark SQL that uses Catalyst Framework to optimize [logical query plans](#) using [optimization rules](#).

Note	<code>SparkOptimizer</code> is the one and only custom <code>optimizer</code> .
------	---

`optimizer` is available as `optimizer` of a `SessionState`.

```
val spark: SparkSession = ...
spark.sessionState.optimizer
```

`optimizer` is a [RuleExecutor](#) that defines [collection of logical plan optimization rules](#).

Table 1. Optimizer's Logical Plan Optimization Rules (in the order of execution)

Batch Name	Strategy	Rules	D
Finish Analysis	Once	EliminateSubqueryAliases	
		EliminateView	
		ReplaceExpressions	
		ComputeCurrentTime	
		GetCurrentDatabase	
		RewriteDistinctAggregates	
		ReplaceDeduplicateWithAggregate	
Union	Once	CombineUnions	
Subquery	Once	OptimizeSubqueries	
Replace Operators	FixedPoint	ReplaceIntersectWithSemiJoin	
		ReplaceExceptWithAntiJoin	
		ReplaceDistinctWithAggregate	
		RemoveLiteralFromGroupExpressions	

Aggregate Operator Optimizations	FixedPoint FixedPoint	RemoveRepetitionFromGroupExpressions	
		PushProjectionThroughUnion	
		ReorderJoin	
		EliminateOuterJoin	
		PushPredicateThroughJoin	
		PushDownPredicate	
		LimitPushDown	
		ColumnPruning	
		InferFiltersFromConstraints	
		CollapseRepartition	Collapse and Repartit
		CollapseProject	
		CollapseWindow	
		CombineFilters	
		CombineLimits	
		CombineUnions	
		NullPropagation	
		FoldablePropagation	
		OptimizeIn	
		ConstantFolding	
		ReorderAssociativeOperator	
		LikeSimplification	

		BooleanSimplification	
		SimplifyConditionals	
		RemoveDisposableExpressions	
		SimplifyBinaryComparison	
		PruneFilters	
		EliminateSorts	
		SimplifyCasts	
		SimplifyCaseConversionExpressions	
		RewriteCorrelatedScalarSubquery	
		EliminateSerialization	
		RemoveRedundantAliases	
		RemoveRedundantProject	
		SimplifyCreateStructOps	
		SimplifyCreateArrayOps	
		SimplifyCreateMapOps	
Check Cartesian Products	Once	CheckCartesianProducts	
Join Reorder	Once	CostBasedJoinReorder	
Decimal Optimizations	FixedPoint	DecimalAggregates	
Typed Filter Optimization	FixedPoint	CombineTypedFilters	
LocalRelation	FixedPoint	ConvertToLocalRelation	
		PropagateEmptyRelation	

OptimizeCodegen	Once	OptimizeCodegen	
RewriteSubquery	Once	RewritePredicateSubquery	
		CollapseProject	

Tip Consult the [sources of `Optimizer`](#) for the up-to-date list of the optimization rules.

Note **Catalyst** is a Spark SQL framework for manipulating trees. It can work with trees of relational operators and expressions in [logical plans](#) before they end up as [physical execution plans](#).

```
scala> sql("select 1 + 1 + 1").explain(true)
== Parsed Logical Plan ==
'Project [unresolvedalias(((1 + 1) + 1), None)]
+- OneRowRelation$

== Analyzed Logical Plan ==
((1 + 1) + 1): int
Project [((1 + 1) + 1) AS ((1 + 1) + 1)#4]
+- OneRowRelation$

== Optimized Logical Plan ==
Project [3 AS ((1 + 1) + 1)#4]
+- OneRowRelation$

== Physical Plan ==
*Project [3 AS ((1 + 1) + 1)#4]
+- Scan OneRowRelation[]
```

Table 2. Optimizer's Properties (in alphabetical order)

Name	Initial Value	Description
fixedPoint	FixedPoint with the number of iterations as defined by spark.sql.optimizer.maxIterations	Used in Replace Operators , Aggregate , Operator Optimizations , Decimal Optimizations , Typed Filter Optimization and LocalRelation batches (and also indirectly in the User Provided Optimizers rule batch in SparkOptimizer).

Creating Optimizer Instance

`Optimizer` takes the following when created:

- [SessionCatalog](#)

- [CatalystConf](#)

`optimizer` initializes the [internal properties](#).

ColumnPruning Logical Plan Optimization

ColumnPruning is a logical optimization (aka Rule[LogicalPlan]) in Optimizer that...FIXME

ColumnPruning is a part of Operator Optimizations batch in the base rule-based logical query plan optimizer.

Example 1

```

val dataset = spark.range(10).withColumn("bucket", 'id % 3)

import org.apache.spark.sql.expressions.Window
val rankCol = rank over Window.partitionBy('bucket).orderBy('id) as "rank"

val ranked = dataset.withColumn("rank", rankCol)

scala> ranked.explain(true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.ColumnPruning ==
Project [id#73L, bucket#76L, rank#192]
                                         Project
[id#73L, bucket#76L, rank#192]
!+- Project [id#73L, bucket#76L, rank#82, rank#82 AS rank#192]
                                         +- Proj
ect [id#73L, bucket#76L, rank#82 AS rank#192]
    +- Window [rank(id#73L) windowspecdefinition(bucket#76L, id#73L ASC, ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#82], [bucket#76L], [id#73L ASC]      +- W
indow [rank(id#73L) windowspecdefinition(bucket#76L, id#73L ASC, ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW) AS rank#82], [bucket#76L], [id#73L ASC]
!      +- Project [id#73L, bucket#76L]
                                         +
- Project [id#73L, (id#73L % cast(3 as bigint)) AS bucket#76L]
!          +- Project [id#73L, (id#73L % cast(3 as bigint)) AS bucket#76L]

    +- Range (0, 10, step=1, splits=Some(8))
!          +- Range (0, 10, step=1, splits=Some(8))
...
TRACE SparkOptimizer: Fixed point reached for batch Operator Optimizations after 2 iterations.
DEBUG SparkOptimizer:
== Result of Batch Operator Optimizations ==
!Project [id#73L, bucket#76L, rank#192]
                                         Window
[rank(id#73L) windowspecdefinition(bucket#76L, id#73L ASC, ROWS BETWEEN UNBOUNDED PREC
EDING AND CURRENT ROW) AS rank#82], [bucket#76L], [id#73L ASC]
!+- Project [id#73L, bucket#76L, rank#82, rank#82 AS rank#192]
                                         +- Proj
ect [id#73L, (id#73L % 3) AS bucket#76L]
!    +- Window [rank(id#73L) windowspecdefinition(bucket#76L, id#73L ASC, ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#82], [bucket#76L], [id#73L ASC]      +- R
ange (0, 10, step=1, splits=Some(8))
!        +- Project [id#73L, bucket#76L]
!          +- Project [id#73L, (id#73L % cast(3 as bigint)) AS bucket#76L]
!            +- Range (0, 10, step=1, splits=Some(8))
...

```

Example 2

```

// the business object
case class Person(id: Long, name: String, city: String)

// the dataset to query over
val dataset = Seq(Person(0, "Jacek", "Warsaw")).toDS

// the query
// Note that we work with names only (out of 3 attributes in Person)
val query = dataset.groupBy(upper('name) as 'name).count

scala> query.explain(extended = true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.ColumnPruning ==
Aggregate [upper(name#126)], [upper(name#126) AS name#160, count(1) AS count#166L]
Aggregate [upper(name#126)], [upper(name#126) AS name#160, count(1) AS count#166L]
+- LocalRelation [id#125L, name#126, city#127]
+- Project [name#126]
!
+- LocalRelation [id#125L, name#126, city#127]
...
== Parsed Logical Plan ==
'Aggregate [upper('name) AS name#160], [upper('name) AS name#160, count(1) AS count#166
L]
+- LocalRelation [id#125L, name#126, city#127]

== Analyzed Logical Plan ==
name: string, count: bigint
Aggregate [upper(name#126)], [upper(name#126) AS name#160, count(1) AS count#166L]
+- LocalRelation [id#125L, name#126, city#127]

== Optimized Logical Plan ==
Aggregate [upper(name#126)], [upper(name#126) AS name#160, count(1) AS count#166L]
+- LocalRelation [name#126]

== Physical Plan ==
*HashAggregate(keys=[upper(name#126)#171], functions=[count(1)], output=[name#160, cou
nt#166L])
+- Exchange hashpartitioning(upper(name#126)#171, 200)
  +- *HashAggregate(keys=[upper(name#126) AS upper(name#126)#171], functions=[partial
  _count(1)], output=[upper(name#126)#171, count#173L])
    +- LocalTableScan [name#126]

```

CombineTypedFilters Logical Plan Optimization

`CombineTypedFilters` combines two back to back (typed) filters into one that ultimately ends up as a single method call.

```
val spark: SparkSession = ...
// Notice two consecutive filters
spark.range(10).filter(_ % 2 == 0).filter(_ == 0)
```

`CombineTypedFilters` is the only logical plan optimization rule in **Typed Filter Optimization batch** in the base [Optimizer](#).

```
val spark: SparkSession = ...

// Notice two consecutive filters
val dataset = spark.range(10).filter(_ % 2 == 0).filter(_ == 0)
scala> dataset.queryExecution.optimizedPlan
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.CombineTypedFilters ==
  TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], newInstance(class java.lang.Long)      TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], newInstance(class java.lang.Long)
!+- TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], newInstance(class java.lang.Long)      +- Range (0, 10, step=1, splits=Some(8))
!   +- Range (0, 10, step=1, splits=Some(8))

TRACE SparkOptimizer: Fixed point reached for batch Typed Filter Optimization after 2 iterations.
DEBUG SparkOptimizer:
== Result of Batch Typed Filter Optimization ==
  TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], newInstance(class java.lang.Long)      TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], newInstance(class java.lang.Long)
!+- TypedFilter <function1>, class java.lang.Long, [StructField(value,LongType,true)], newInstance(class java.lang.Long)      +- Range (0, 10, step=1, splits=Some(8))
!   +- Range (0, 10, step=1, splits=Some(8))
...
```

ConstantFolding Logical Plan Optimization

`ConstantFolding` is a operator optimization rule in [Catalyst](#) that replaces expressions that can be statically evaluated with their equivalent literal values.

`ConstantFolding` object is a logical plan optimization rule in [Operator Optimizations](#) batch in the base [Optimizer](#).

```
scala> spark.range(1).select(lit(3) > 2).explain(true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.ConstantFolding ==
!Project [(3 > 2) AS (3 > 2)#3]          Project [true AS (3 > 2)#3]
+- Range (0, 1, step=1, splits=Some(8))   +- Range (0, 1, step=1, splits=Some(8))
```

```
scala> spark.range(1).select('id + 'id > 0).explain(true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.ConstantFolding ==
!Project [((id#7L + id#7L) > cast(0 as bigint)) AS ((id + id) > 0)#10]   Project [((id
#7L + id#7L) > 0) AS ((id + id) > 0)#10]
+- Range (0, 1, step=1, splits=Some(8))                                     +- Range (0,
1, step=1, splits=Some(8))
```

CostBasedJoinReorder

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

DecimalAggregates Logical Plan Optimization

`DecimalAggregates` is a [logical optimization](#) (aka `Rule[LogicalPlan]`) in [Optimizer](#) that transforms `Sum` and `Average` aggregate functions on fixed-precision `DecimalType` values to use `UnscaledValue` (unscaled Long) values in [WindowExpression](#) and [AggregateExpression](#) expressions.

Note

`DecimalAggregates` is the only optimization in the fixed-point [Decimal Optimizations](#) batch of rules in `optimizer`.

Tip Import `DecimalAggregates` and apply the rule directly on your structured queries to see the rule works.

```
import org.apache.spark.sql.catalyst.optimizer.DecimalAggregates
val da = DecimalAggregates(spark.sessionState.conf)

// Build analyzed logical plan
// with sum aggregate function and Decimal field
import org.apache.spark.sql.types.DecimalType
val query = spark.range(5).select(sum($"id" cast DecimalType(1,0)) as "sum")
scala> val plan = query.execution.analyzed
plan: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Aggregate [sum(cast(id#91L as decimal(1,0))) AS sum#95]
+- Range (0, 5, step=1, splits=Some(8))

// Apply DecimalAggregates rule
// Note MakeDecimal and UnscaledValue operators
scala> da.apply(plan)
res27: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Aggregate [MakeDecimal(sum(UnscaledValue(cast(id#91L as decimal(1,0)))),11,0) AS
+- Range (0, 5, step=1, splits=Some(8))
```

Example: sum Aggregate Function on Decimal with Precision Smaller Than 9

```
// sum aggregate with Decimal field with precision <= 8
val q = "SELECT sum(cast(id AS DECIMAL(5,0))) FROM range(1)"

scala> sql(q).explain(true)
== Parsed Logical Plan ==
'Project [unresolvedalias('sum(cast('id as decimal(5,0))), None)]
+- 'UnresolvedTableValuedFunction range, [1]

== Analyzed Logical Plan ==
sum(CAST(id AS DECIMAL(5,0))): decimal(15,0)
Aggregate [sum(cast(id#104L as decimal(5,0))) AS sum(CAST(id AS DECIMAL(5,0)))#106]
+- Range (0, 1, step=1, splits=None)

== Optimized Logical Plan ==
Aggregate [MakeDecimal(sum(UnscaledValue(cast(id#104L as decimal(5,0)))),15,0) AS sum(
CAST(id AS DECIMAL(5,0)))#106]
+- Range (0, 1, step=1, splits=None)

== Physical Plan ==
*HashAggregate(keys=[], functions=[sum(UnscaledValue(cast(id#104L as decimal(5,0))))],
output=[sum(CAST(id AS DECIMAL(5,0)))#106])
+- Exchange SinglePartition
  +- *HashAggregate(keys=[], functions=[partial_sum(UnscaledValue(cast(id#104L as decimal(5,0))))], output=[sum#108L])
    +- *Range (0, 1, step=1, splits=None)
```

Example: avg Aggregate Function on Decimal with Precision Smaller Than 12

```
// avg aggregate with Decimal field with precision <= 11
val q = "SELECT avg(cast(id AS DECIMAL(10,0))) FROM range(1)"

scala> val q = "SELECT avg(cast(id AS DECIMAL(10,0))) FROM range(1)"
q: String = SELECT avg(cast(id AS DECIMAL(10,0))) FROM range(1)

scala> sql(q).explain(true)
== Parsed Logical Plan ==
'Project [unresolvedalias('avg(cast('id as decimal(10,0))), None)]
+- 'UnresolvedTableValuedFunction range, [1]

== Analyzed Logical Plan ==
avg(CAST(id AS DECIMAL(10,0))): decimal(14,4)
Aggregate [avg(cast(id#115L as decimal(10,0))) AS avg(CAST(id AS DECIMAL(10,0)))#117]
+- Range (0, 1, step=1, splits=None)

== Optimized Logical Plan ==
Aggregate [cast((avg(UnscaledValue(cast(id#115L as decimal(10,0)))) / 1.0) as decimal(
14,4)) AS avg(CAST(id AS DECIMAL(10,0)))#117]
+- Range (0, 1, step=1, splits=None)

== Physical Plan ==
*HashAggregate(keys=[], functions=[avg(UnscaledValue(cast(id#115L as decimal(10,0))))])
, output=[avg(CAST(id AS DECIMAL(10,0)))#117])
+- Exchange SinglePartition
    +- *HashAggregate(keys=[], functions=[partial_avg(UnscaledValue(cast(id#115L as decimal(10,0))))], output=[sum#120, count#121L])
        +- *Range (0, 1, step=1, splits=None)
```

EliminateSerialization Logical Plan Optimization

`EliminateSerialization` is a optimization rule for a logical plan in `SparkOptimizer`.

`EliminateSerialization` optimizes logical plans with `DeserializeToObject` (after `SerializeFromObject` or `TypedFilter`), `AppendColumns` (after `SerializeFromObject`), `TypedFilter` (after `SerializeFromObject`) logical operators.

Examples include:

1. `map` followed by `filter` Logical Plan
2. `map` followed by another `map` Logical Plan
3. `groupByKey` followed by `agg` Logical Plan

Example — `map` followed by `filter` Logical Plan

```

scala> spark.range(4).map(n => n * 2).filter(n => n < 3).explain(extended = true)
== Parsed Logical Plan ==
'TypedFilter <function1>, long, [StructField(value,LongType,false)], unresolveddeserializer(upcast(getcolumnbyordinal(0, LongType), LongType, - root class: "scala.Long"))
+- SerializeFromObject [input[0, bigint, true] AS value#185L]
  +- MapElements <function1>, class java.lang.Long, [StructField(value,LongType,true)]
], obj#184: bigint
  +- DeserializeToObject newInstance(class java.lang.Long), obj#183: java.lang.Long
  +- Range (0, 4, step=1, splits=Some(8))

== Analyzed Logical Plan ==
value: bigint
TypedFilter <function1>, long, [StructField(value,LongType,false)], cast(value#185L as
bigint)
+- SerializeFromObject [input[0, bigint, true] AS value#185L]
  +- MapElements <function1>, class java.lang.Long, [StructField(value,LongType,true)]
], obj#184: bigint
  +- DeserializeToObject newInstance(class java.lang.Long), obj#183: java.lang.Long
  +- Range (0, 4, step=1, splits=Some(8))

== Optimized Logical Plan ==
SerializeFromObject [input[0, bigint, true] AS value#185L]
+- Filter <function1>.apply
  +- MapElements <function1>, class java.lang.Long, [StructField(value,LongType,true)]
], obj#184: bigint
  +- DeserializeToObject newInstance(class java.lang.Long), obj#183: java.lang.Long
  +- Range (0, 4, step=1, splits=Some(8))

== Physical Plan ==
*SerializeFromObject [input[0, bigint, true] AS value#185L]
+- *Filter <function1>.apply
  +- *MapElements <function1>, obj#184: bigint
    +- *DeserializeToObject newInstance(class java.lang.Long), obj#183: java.lang.Long
    +- *Range (0, 4, step=1, splits=Some(8))

```

Example — map followed by another map Logical Plan

```

// Notice unnecessary mapping between String and Int types
val query = spark.range(3).map(_.toString).map(_.toInt)

scala> query.explain(extended = true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.EliminateSerialization ===
SerializeFromObject [input[0, int, true] AS value#91]

```

S

```

    serializeFromObject [input[0, int, true] AS value#91]
      +- MapElements <function1>, class java.lang.String, [StructField(value, StringType, true)], obj#90: int
      +-
    - MapElements <function1>, class java.lang.String, [StructField(value, StringType, true)], obj#90: int
    !   +- DeserializeToObject value#86.toString, obj#89: java.lang.String

      +- Project [obj#85 AS obj#89]
    !     +- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.String, true], true) AS value#86]
          +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#85: java.lang.String
    !     +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#85: java.lang.String
          +- DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Long
    !           +- DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Long
          +- Range (0, 3, step=1, splits=Some(8))
    !
          +- Range (0, 3, step=1, splits=Some(8))
...
== Parsed Logical Plan ==
'SerializeFromObject [input[0, int, true] AS value#91]
+- 'MapElements <function1>, class java.lang.String, [StructField(value, StringType, true)], obj#90: int
  +- 'DeserializeToObject unresolveddeserializer(upcast(getcolumnbyordinal(0, StringType), StringType, - root class: "java.lang.String").toString), obj#89: java.lang.String
    +- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.String, true], true) AS value#86]
      +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#85: java.lang.String
        +- DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Long
          +- Range (0, 3, step=1, splits=Some(8))

== Analyzed Logical Plan ==
value: int
SerializeFromObject [input[0, int, true] AS value#91]
+- MapElements <function1>, class java.lang.String, [StructField(value, StringType, true)], obj#90: int
  +- DeserializeToObject cast(value#86 as string).toString, obj#89: java.lang.String
    +- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.String, true], true) AS value#86]
      +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#85: java.lang.String
        +- DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Long
          +- Range (0, 3, step=1, splits=Some(8))

== Optimized Logical Plan ==
SerializeFromObject [input[0, int, true] AS value#91]
+- MapElements <function1>, class java.lang.String, [StructField(value, StringType, true)

```

```

)], obj#90: int
+- MapElements <function1>, class java.lang.Long, [StructField(value,LongType,true)
], obj#85: java.lang.String
    +- DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Long
        +- Range (0, 3, step=1, splits=Some(8))

== Physical Plan ==
*SerializeFromObject [input[0, int, true] AS value#91]
+- *MapElements <function1>, obj#90: int
    +- *MapElements <function1>, obj#85: java.lang.String
        +- *DeserializeToObject newInstance(class java.lang.Long), obj#84: java.lang.Lon
g
    +- *Range (0, 3, step=1, splits=Some(8))

```

Example — groupByKey followed by agg Logical Plan

```

scala> spark.range(4).map(n => (n, n % 2)).groupByKey(_._2).agg(typed.sum(_._2)).expla
in(true)
== Parsed Logical Plan ==
'Aggregate [value#454L], [value#454L, unresolvedalias(typedsumdouble(org.apache.spark.
sql.execution.aggregate.TypedSumDouble@4fcb0de4, Some(unresolveddeserializer(newInstan
ce(class scala.Tuple2), _1#450L, _2#451L)), Some(class scala.Tuple2), Some(StructType(
StructField(_1,LongType,true), StructField(_2,LongType,false))), input[0, double, true]
] AS value#457, unresolveddeserializer(upcast(getcolumnbyordinal(0, DoubleType), Doub
leType, - root class: "scala.Double"), value#457), input[0, double, true] AS value#456,
DoubleType, DoubleType, false), Some(<function1>))]
+- AppendColumns <function1>, class scala.Tuple2, [StructField(_1,LongType,true), Stru
ctField(_2,LongType,false)], newInstance(class scala.Tuple2), [input[0, bigint, true]
AS value#454L]
    +- SerializeFromObject [assertnonnull(input[0, scala.Tuple2, true], top level non-f
lat input object)._1.longValue AS _1#450L, assertnonnull(input[0, scala.Tuple2, true],
top level non-flat input object)._2 AS _2#451L]
        +- MapElements <function1>, class java.lang.Long, [StructField(value,LongType,tr
ue)], obj#449: scala.Tuple2
            +- DeserializeToObject newInstance(class java.lang.Long), obj#448: java.lang.
Long
                +- Range (0, 4, step=1, splits=Some(8))

== Analyzed Logical Plan ==
value: bigint, TypedSumDouble(scala.Tuple2): double
Aggregate [value#454L], [value#454L, typedsumdouble(org.apache.spark.sql.execution.agg
regate.TypedSumDouble@4fcb0de4, Some(newInstance(class scala.Tuple2)), Some(class scal
a.Tuple2), Some(StructType(StructField(_1,LongType,true), StructField(_2,LongType,fals
e))), input[0, double, true] AS value#457, cast(value#457 as double), input[0, double,
true] AS value#456, DoubleType, DoubleType, false) AS TypedSumDouble(scala.Tuple2)#46
2]
+- AppendColumns <function1>, class scala.Tuple2, [StructField(_1,LongType,true), Stru
ctField(_2,LongType,false)], newInstance(class scala.Tuple2), [input[0, bigint, true]
AS value#454L]
    +- SerializeFromObject [assertnonnull(input[0, scala.Tuple2, true], top level non-f
lat input object)._1.longValue AS _1#450L, assertnonnull(input[0, scala.Tuple2, true],

```

```

top level non-flat input object)._2 AS _2#451L]
    +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#449: scala.Tuple2
        +- DeserializeToObject newInstance(class java.lang.Long), obj#448: java.lang.Long
Long
    +- Range (0, 4, step=1, splits=Some(8))

== Optimized Logical Plan ==
Aggregate [value#454L], [value#454L, typedsumdouble(org.apache.spark.sql.execution.aggregate.TypedSumDouble@4fc0de4, Some(newInstance(class scala.Tuple2)), Some(class scala.Tuple2), Some(StructType(StructField(_1, LongType, true), StructField(_2, LongType, false))), input[0, double, true] AS value#457, value#457, input[0, double, true] AS value#456, DoubleType, DoubleType, false) AS TypedSumDouble(scala.Tuple2)#462]
+- AppendColumnsWithObject <function1>, [assertnonnull(input[0, scala.Tuple2, true], top level non-flat input object)._1.longValue AS _1#450L, assertnonnull(input[0, scala.Tuple2, true], top level non-flat input object)._2 AS _2#451L], [input[0, bigint, true] AS value#454L]
    +- MapElements <function1>, class java.lang.Long, [StructField(value, LongType, true)], obj#449: scala.Tuple2
        +- DeserializeToObject newInstance(class java.lang.Long), obj#448: java.lang.Long
    +- Range (0, 4, step=1, splits=Some(8))

== Physical Plan ==
*HashAggregate(keys=[value#454L], functions=[typedsumdouble(org.apache.spark.sql.execution.aggregate.TypedSumDouble@4fc0de4, Some(newInstance(class scala.Tuple2)), Some(class scala.Tuple2), Some(StructType(StructField(_1, LongType, true), StructField(_2, LongType, false))), input[0, double, true] AS value#457, value#457, input[0, double, true] AS value#456, DoubleType, DoubleType, false)], output=[value#454L, TypedSumDouble(scala.Tuple2)#462])
+- Exchange hashpartitioning(value#454L, 200)
    +- *HashAggregate(keys=[value#454L], functions=[partial_typedsumdouble(org.apache.spark.sql.execution.aggregate.TypedSumDouble@4fc0de4, Some(newInstance(class scala.Tuple2)), Some(class scala.Tuple2), Some(StructType(StructField(_1, LongType, true), StructField(_2, LongType, false))), input[0, double, true] AS value#457, value#457, input[0, double, true] AS value#456, DoubleType, DoubleType, false)], output=[value#454L, value#463])
        +- AppendColumnsWithObject <function1>, [assertnonnull(input[0, scala.Tuple2, true], top level non-flat input object)._1.longValue AS _1#450L, assertnonnull(input[0, scala.Tuple2, true], top level non-flat input object)._2 AS _2#451L], [input[0, bigint, true] AS value#454L]
            +- MapElements <function1>, obj#449: scala.Tuple2
                +- DeserializeToObject newInstance(class java.lang.Long), obj#448: java.lang.Long
                +- *Range (0, 4, step=1, splits=Some(8))

```

GetCurrentDatabase and ComputeCurrentTime Logical Plan Optimizations

`GetCurrentDatabase` and `ComputeCurrentTime` optimization rules are part of **Finish Analysis** batch in the base [Optimizer](#).

GetCurrentDatabase Optimization Rule

`GetCurrentDatabase` optimization rule returns the current database for `current_database` SQL function.

```
scala> sql("SELECT current_database() AS database").show
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.GetCurrentDatabase ===
GlobalLimit 21                               GlobalLimit 21
+- LocalLimit 21                             +- LocalLimit 21
!  +- Project [currentdatabase() AS database#20]   +- Project [default AS database
#20]
      +- OneRowRelation$                         +- OneRowRelation$
...
+-----+
|database|
+-----+
| default|
+-----+
```

Note

`GetCurrentDatabase` corresponds to SQL's `current_database()` function.

You can access the current database in Scala using

```
scala> val database = spark.catalog.currentDatabase
database: String = default
```

ComputeCurrentTime Optimization Rule

`ComputeCurrentTime` logical plan optimization rule computes the current date and timestamp.

```
scala> spark.range(1).select(current_date()).explain
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.ComputeCurrentTime ==
!Project [current_date() AS current_date()#29]   Project [17055 AS current_date()#29]
+- Range (0, 1, step=1, splits=Some(8))           +- Range (0, 1, step=1, splits=Some(8))
))
```

```
scala> spark.range(1).select(current_timestamp()).explain
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.ComputeCurrentTime ==
!Project [current_timestamp() AS current_timestamp()#36]   Project [1473599927969000 A
S current_timestamp()#36]
+- Range (0, 1, step=1, splits=Some(8))           +- Range (0, 1, step=1, spl
its=Some(8))
```

LimitPushDown Logical Plan Optimization

`LimitPushDown` is a [LogicalPlan](#) optimization rule that [transforms](#) the following logical plans:

- `LocalLimit` with `Union`
- `LocalLimit` with `Join`

`LimitPushDown` is a part of [Operator Optimizations](#) batch in the base [Optimizer](#).

```
// test datasets
scala> val ds1 = spark.range(4)
ds1: org.apache.spark.sql.Dataset[Long] = [value: bigint]

scala> val ds2 = spark.range(2)
ds2: org.apache.spark.sql.Dataset[Long] = [value: bigint]

// Case 1. Rather than `LocalLimit` of `Union` do `Union` of `LocalLimit`
scala> ds1.union(ds2).limit(2).explain(true)
== Parsed Logical Plan ==
GlobalLimit 2
+- LocalLimit 2
  +- Union
    :- Range (0, 4, step=1, splits=Some(8))
    +- Range (0, 2, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
GlobalLimit 2
+- LocalLimit 2
  +- Union
    :- Range (0, 4, step=1, splits=Some(8))
    +- Range (0, 2, step=1, splits=Some(8))

== Optimized Logical Plan ==
GlobalLimit 2
+- LocalLimit 2
  +- Union
    :- LocalLimit 2
      :- Range (0, 4, step=1, splits=Some(8))
    +- LocalLimit 2
      :- Range (0, 2, step=1, splits=Some(8))

== Physical Plan ==
CollectLimit 2
+- Union
  :- *LocalLimit 2
    :- *Range (0, 4, step=1, splits=Some(8))
  +- *LocalLimit 2
    :- *Range (0, 2, step=1, splits=Some(8))
```

apply Method

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

Creating LimitPushDown Instance

`LimitPushDown` takes the following when created:

- CatalystConf

LimitPushDown initializes the internal registries and counters.

Note	LimitPushDown is created when
------	-------------------------------

NullPropagation — Nullability (NULL Value) Propagation Logical Plan Optimization

`NullPropagation` is a [logical optimization](#) (aka `Rule[LogicalPlan]`) in [Optimizer](#).

Note

`NullPropagation` is one of the optimizations in the fixed-point [Operator Optimizations](#) optimization rule batch in `Optimizer`.

Example: Count Aggregate Operator with Nullable Expressions Only

`NullPropagation` optimization rewrites `Count` aggregate expressions that include expressions that are all nullable to `cast(Literal(0L))`.

```
val table = (0 to 9).toDF("num").as[Int]

// NullPropagation applied
scala> table.select(countDistinct($"num" === null)).explain(true)
== Parsed Logical Plan ==
'Project [count(distinct ('num = null)) AS count(DISTINCT (num = NULL))#45]
+- Project [value#1 AS num#3]
  +- LocalRelation [value#1]

== Analyzed Logical Plan ==
count(DISTINCT (num = NULL)): bigint
Aggregate [count(distinct (num#3 = cast(null as int))) AS count(DISTINCT (num = NULL))#45L]
+- Project [value#1 AS num#3]
  +- LocalRelation [value#1]

== Optimized Logical Plan ==
Aggregate [0 AS count(DISTINCT (num = NULL))#45L] // <-- HERE
+- LocalRelation

== Physical Plan ==
*HashAggregate(keys=[], functions[], output=[count(DISTINCT (num = NULL))#45L])
+- Exchange SinglePartition
  +- *HashAggregate(keys=[], functions[], output[])
    +- LocalTableScan
```

Example: Count Aggregate Operator with Non-Nullable Non-Distinct Expressions

NullPropagation optimization rewrites any non- nullable non-distinct Count aggregate expressions to Literal(1) .

```
val table = (0 to 9).toDF("num").as[Int]

// NullPropagation applied
// current_timestamp() is a non-nullable expression (see the note below)
val query = table.select(count(current_timestamp()) as "count")

scala> println(query.queryExecution.optimizedPlan)
Aggregate [count(1) AS count#64L]
+- LocalRelation

// NullPropagation skipped
val tokens = Seq((0, null), (1, "hello")).toDF("id", "word")
val query = tokens.select(count("word") as "count")

scala> println(query.queryExecution.optimizedPlan)
Aggregate [count(word#55) AS count#71L]
+- LocalRelation [word#55]
```

Count aggregate expression represents count function internally.

Note

```
import org.apache.spark.sql.catalyst.expressions.aggregate.Count
import org.apache.spark.sql.functions.count

scala> count("*").expr.children(0).asInstanceOf[Count]
res0: org.apache.spark.sql.catalyst.expressions.aggregate.Count = count(1)
```

current_timestamp() function is non- nullable expression.

Note

```
import org.apache.spark.sql.catalyst.expressions.CurrentTimestamp
import org.apache.spark.sql.functions.current_timestamp

scala> current_timestamp().expr.asInstanceOf[CurrentTimestamp].nullable
res38: Boolean = false
```

Example

```
val table = (0 to 9).toDF("num").as[Int]
val query = table.where('num === null)

scala> query.explain(extended = true)
== Parsed Logical Plan ==
'Filter ('num = null)
+- Project [value#1 AS num#3]
  +- LocalRelation [value#1]

== Analyzed Logical Plan ==
num: int
Filter (num#3 = cast(null as int))
+- Project [value#1 AS num#3]
  +- LocalRelation [value#1]

== Optimized Logical Plan ==
LocalRelation <empty>, [num#3]

== Physical Plan ==
LocalTableScan <empty>, [num#3]
```

PropagateEmptyRelation Logical Plan Optimization

`PropagateEmptyRelation` is a [LogicalPlan](#) optimization rule that collapses plans with empty [LocalRelation](#) logical query plans, e.g. `explode` or `join`.

`PropagateEmptyRelation` is a part of [LocalRelation](#) batch in the base [Optimizer](#).

Explode

```
scala> val emp = spark.emptyDataset[Seq[String]]
emp: org.apache.spark.sql.Dataset[Seq[String]] = [value: array<string>]

scala> emp.select(explode($"value")).show
+---+
|col|
+---+
+---+

scala> emp.select(explode($"value")).explain(true)
== Parsed Logical Plan ==
'Project [explode('value) AS List()]
+- LocalRelation <empty>, [value#77]

== Analyzed Logical Plan ==
col: string
Project [col#89]
+- Generate explode(value#77), false, false, [col#89]
  +- LocalRelation <empty>, [value#77]

== Optimized Logical Plan ==
LocalRelation <empty>, [col#89]

== Physical Plan ==
LocalTableScan <empty>, [col#89]
```

Join

```
scala> spark.emptyDataset[Int].join(spark.range(1)).explain(extended = true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.PropagateEmptyRelation ==
!Join Inner                               LocalRelation <empty>, [value#40, id#42L]
!:- LocalRelation <empty>, [value#40]
!+- Range (0, 1, step=1, splits=Some(8))

TRACE SparkOptimizer: Fixed point reached for batch LocalRelation after 2 iterations.
DEBUG SparkOptimizer:
== Result of Batch LocalRelation ==
!Join Inner                               LocalRelation <empty>, [value#40, id#42L]
!:- LocalRelation <empty>, [value#40]
!+- Range (0, 1, step=1, splits=Some(8))
...
== Parsed Logical Plan ==
Join Inner
:- LocalRelation <empty>, [value#40]
+- Range (0, 1, step=1, splits=Some(8))

== Analyzed Logical Plan ==
value: int, id: bigint
Join Inner
:- LocalRelation <empty>, [value#40]
+- Range (0, 1, step=1, splits=Some(8))

== Optimized Logical Plan ==
LocalRelation <empty>, [value#40, id#42L]

== Physical Plan ==
LocalTableScan <empty>, [value#40, id#42L]
```

PushDownPredicate — Predicate Pushdown / Filter Pushdown Logical Plan Optimization

`PushDownPredicate` is a [rule](#) for [logical plan](#) optimization (i.e. `Rule[LogicalPlan]`) that the base [Optimizer](#) includes in [Operator Optimizations](#) batch of rules.

When you execute [where](#) or [filter](#) operators right after [loading a dataset](#), Spark SQL will try to push the where/filter predicate down to the data source using a corresponding SQL query with `WHERE` clause (or whatever the proper language for the data source is).

This optimization is called **filter pushdown** or **predicate pushdown** and aims at pushing down the filtering to the "bare metal", i.e. a data source engine. That is to increase the performance of queries since the filtering is performed at the very low level rather than dealing with the entire dataset after it has been loaded to Spark's memory and perhaps causing memory issues.

`PushDownPredicate` is also applied to structured queries with [filters after projections](#) or [filtering on window partitions](#).

Pushing Filter Operator Down Using Projection

```

val dataset = spark.range(2)

scala> dataset.select('id as "_id").filter('_id === 0).explain(extended = true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.PushDownPredicate ==
!Filter (_id#14L = cast(0 as bigint))          Project [id#11L AS _id#14L]
+- Project [id#11L AS _id#14L]                +- Filter (id#11L = cast(0 as bigint))
   +- Range (0, 2, step=1, splits=Some(8))     +- Range (0, 2, step=1, splits=Some(8))
())
...
== Parsed Logical Plan ==
'Filter ('_id = 0)
+- Project [id#11L AS _id#14L]
  +- Range (0, 2, step=1, splits=Some(8))

== Analyzed Logical Plan ==
_id: bigint
Filter (_id#14L = cast(0 as bigint))
+- Project [id#11L AS _id#14L]
  +- Range (0, 2, step=1, splits=Some(8))

== Optimized Logical Plan ==
Project [id#11L AS _id#14L]
+- Filter (id#11L = 0)
  +- Range (0, 2, step=1, splits=Some(8))

== Physical Plan ==
*Project [id#11L AS _id#14L]
+- *Filter (id#11L = 0)
  +- *Range (0, 2, step=1, splits=Some(8))

```

Optimizing Window Aggregate Operators

```

val dataset = spark.range(5).withColumn("group", 'id % 3)
scala> dataset.show
+---+----+
| id|group|
+---+----+
|  0|    0|
|  1|    1|
|  2|    2|
|  3|    0|
|  4|    1|
+---+----+

import org.apache.spark.sql.expressions.Window
val groupW = Window.partitionBy('group).orderBy('id)

// Filter out group 2 after window

```

```

// No need to compute rank for group 2
// Push the filter down
val ranked = dataset.withColumn("rank", rank over groupW).filter('group != 2)

scala> ranked.queryExecution.optimizedPlan
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.PushDownPredicate ==
!Filter NOT (group#35L = cast(2 as bigint))
                                         Proj
ct [id#32L, group#35L, rank#203]
!+- Project [id#32L, group#35L, rank#203]
                                         +- Pr
object [id#32L, group#35L, rank#203, rank#203]
!   +- Project [id#32L, group#35L, rank#203, rank#203]
                                         +- Pr
Window [rank(id#32L) windowspecdefinition(group#35L, id#32L ASC, ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW) AS rank#203], [group#35L], [id#32L ASC]
!     +- Window [rank(id#32L) windowspecdefinition(group#35L, id#32L ASC, ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW) AS rank#203], [group#35L], [id#32L ASC]
+- Project [id#32L, group#35L]
!         +- Project [id#32L, group#35L]
                                         +- Pr
        +- Project [id#32L, (id#32L % cast(3 as bigint)) AS group#35L]
!             +- Project [id#32L, (id#32L % cast(3 as bigint)) AS group#35L]
                                         +- Pr
        +- Filter NOT ((id#32L % cast(3 as bigint)) = cast(2 as bigint))
            +- Range (0, 5, step=1, splits=Some(8))
                                         +- Pr
        +- Range (0, 5, step=1, splits=Some(8))
...
res1: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Window [rank(id#32L) windowspecdefinition(group#35L, id#32L ASC, ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW) AS rank#203], [group#35L], [id#32L ASC]
+- Project [id#32L, (id#32L % 3) AS group#35L]
    +- Filter NOT ((id#32L % 3) = 2)
        +- Range (0, 5, step=1, splits=Some(8))

```

JDBC Data Source

Tip

Follow the instructions on how to set up PostgreSQL in [Creating DataFrames from Tables using JDBC and PostgreSQL](#).

Given the following code:

```
// Start with the PostgreSQL driver on CLASSPATH

case class Project(id: Long, name: String, website: String)

// No optimizations for typed queries
// LOG: execute <unnamed>: SELECT "id", "name", "website" FROM projects
val df = spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:sparkdb")
  .option("dbtable", "projects")
  .load()
  .as[Project]
  .filter(_.name.contains("Spark"))

// Only the following would end up with the pushdown
val df = spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:sparkdb")
  .option("dbtable", "projects")
  .load()
  .where("""name like "%Spark%"""")
```

`PushDownPredicate` translates the above query to the following SQL query:

```
LOG: execute <unnamed>: SELECT "id", "name", "website" FROM projects WHERE (name LIKE '%Spark%')
```

Tip

Enable `all` logs in PostgreSQL to see the above SELECT and other query statements.

```
log_statement = 'all'
```

Add `log_statement = 'all'` to `/usr/local/var/postgres/postgresql.conf` on Mac OS X with PostgreSQL installed using `brew`.

Parquet Data Source

```

val spark: SparkSession = ...
import spark.implicits._

// paste it to REPL individually to make the following line work
case class City(id: Long, name: String)

import org.apache.spark.sql.SaveMode.Overwrite
Seq(
  City(0, "Warsaw"),
  City(1, "Toronto"),
  City(2, "London"),
  City(3, "Redmond"),
  City(4, "Boston")).toDF.write.mode(Overwrite).parquet("cities.parquet")

val cities = spark.read.parquet("cities.parquet").as[City]

// Using DataFrame's Column-based query
scala> cities.where('name === "Warsaw").queryExecution.executedPlan
res21: org.apache.spark.sql.execution.SparkPlan =
*Project [id#128L, name#129]
+- *Filter (isnotnull(name#129) && (name#129 = Warsaw))
   +- *FileScan parquet [id#128L,name#129] Batched: true, Format: ParquetFormat, Input
Paths: file:/Users/jacek/dev/oss/spark/cities.parquet, PartitionFilters: [], PushedFil
ters: [IsNotNull(name), EqualTo(name,Warsaw)], ReadSchema: struct<id:bigint,name:string
g>

// Using SQL query
scala> cities.where("""name = "Warsaw""").queryExecution.executedPlan
res23: org.apache.spark.sql.execution.SparkPlan =
*Project [id#128L, name#129]
+- *Filter (isnotnull(name#129) && (name#129 = Warsaw))
   +- *FileScan parquet [id#128L,name#129] Batched: true, Format: ParquetFormat, Input
Paths: file:/Users/jacek/dev/oss/spark/cities.parquet, PartitionFilters: [], PushedFil
ters: [IsNotNull(name), EqualTo(name,Warsaw)], ReadSchema: struct<id:bigint,name:string
g>

// Using Dataset's strongly type-safe filter
// Why does the following not push the filter down?
scala> cities.filter(_.name == "Warsaw").queryExecution.executedPlan
res24: org.apache.spark.sql.execution.SparkPlan =
*Filter <function1>.apply
+- *FileScan parquet [id#128L,name#129] Batched: true, Format: ParquetFormat, InputPat
hs: file:/Users/jacek/dev/oss/spark/cities.parquet, PartitionFilters: [], PushedFilters
: [], ReadSchema: struct<id:bigint,name:string>

```

Hive Data Source

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

ReorderJoin Logical Plan Optimization

`ReorderJoin` is a logical plan optimization in [Operator Optimizations](#) batch of rules in [Optimizer](#) that transforms `Filter` (with CROSS and INNER joins) and `Join` logical plans with 3 or more joins and non-empty join conditions.

Note

Operator Optimizations is a [fixed-point batch of rules](#).

Import `ReorderJoin` and apply the rule directly on your structured queries to learn how it works.

```
import org.apache.spark.sql.catalyst.optimizer.ReorderJoin
val rj = ReorderJoin(spark.sessionState.conf)

// Build analyzed logical plan with at least 3 joins and zero or more filters
val t1 = spark.range(4)
val t2 = spark.range(4)
val t3 = spark.range(4)

val query = t1.join(t2)
  .where(t1("id") === t2("id"))
  .join(t3)
  .where(t3("id") === t1("id"))
  .filter(t1("id") % 2 === 0)

scala> val plan = query.queryExecution.analyzed
plan: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Filter ((id#6L % cast(2 as bigint)) = cast(0 as bigint))
+- Filter (id#12L = id#6L)
  +- Join Inner
    :- Filter (id#6L = id#9L)
    :  +- Join Inner
      :    :- Range (0, 4, step=1, splits=Some(8))
      :    +- Range (0, 4, step=1, splits=Some(8))
    +- Range (0, 4, step=1, splits=Some(8))
```

Tip

```
// Apply ReorderJoin rule
scala> val optimized = rj.apply(plan)
optimized: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
Filter ((id#6L % cast(2 as bigint)) = cast(0 as bigint))
+- Join Inner, (id#12L = id#6L)
  :- Join Inner, (id#6L = id#9L)
    :  :- Range (0, 4, step=1, splits=Some(8))
    :  +- Range (0, 4, step=1, splits=Some(8))
  +- Range (0, 4, step=1, splits=Some(8))

scala> plan.stats(spark.sessionState.conf)
res5: org.apache.spark.sql.catalyst.plans.logical.Statistics = Statistics(sizeInBytes=0, rowCount=0, numPartitions=1, totalRows=0, totalMemoryUsed=0)

// CBO disabled
scala> optimized.stats(spark.sessionState.conf)
res6: org.apache.spark.sql.catalyst.plans.logical.Statistics = Statistics(sizeInBytes=0, rowCount=0, numPartitions=1, totalRows=0, totalMemoryUsed=0)

// ReorderJoin works differently when the following holds:
// * starSchemaDetection is enabled
// * cboEnabled is disabled
import org.apache.spark.sql.internal.SQLConf.STARSCHEMA_DETECTION
spark.sessionState.conf.setConf(STARSCHEMA_DETECTION, true)

spark.sessionState.conf.starSchemaDetection
spark.sessionState.conf.cboEnabled
```

Transforming Logical Plan — `apply` Method

`apply` transforms `Filter` (with CROSS and INNER join types) and `Join` logical plans.

Note

`apply` uses `ExtractFiltersAndInnerJoins` Scala extractor object (using `unapply` method) to "destructure" a logical plan to its logical operators.

createOrderedJoin Recursive Method

Caution

FIXME

Extracting Filter and Join Operators from Logical Plan — `unapply` Method (of `ExtractFiltersAndInnerJoins`)

```
unapply(plan: LogicalPlan): Option[(Seq[(LogicalPlan, InnerLike)], Seq[Expression])]
```

`unapply` takes `Filter` (with CROSS and INNER joins) and any `Join` logical operators out of the input logical `plan` and flattens the joins.

Flattening Join — `flattenJoin` Method (of `ExtractFiltersAndInnerJoins`)

```
flattenJoin(plan: LogicalPlan, parentJoinType: InnerLike = Inner)
  : (Seq[(LogicalPlan, InnerLike)], Seq[Expression])
```

`flattenJoin` takes CROSS and INNER join types... [FIXME](#)

SimplifyCasts Logical Plan Optimization

`SimplifyCasts` is a [LogicalPlan](#) optimization rule that eliminates redundant casts in the following cases:

1. The input is already the type to cast to.
2. The input is of `ArrayType` or `MapType` type and contains no `null` elements.

`SimplifyCasts` is a part of [Operator Optimizations](#) batch in the base [Optimizer](#).

```
// Case 1. The input is already the type to cast to
scala> val ds = spark.range(1)
ds: org.apache.spark.sql.Dataset[Long] = [id: bigint]

scala> ds.printSchema
root
| -- id: long (nullable = false)

scala> ds.selectExpr("CAST (id AS long)").explain(true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.SimplifyCasts ==
!Project [cast(id#0L as bigint) AS id#7L]   Project [id#0L AS id#7L]
+- Range (0, 1, step=1, splits=Some(8))     +- Range (0, 1, step=1, splits=Some(8))

TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.RemoveAliasOnlyProject ==
!Project [id#0L AS id#7L]                     Range (0, 1, step=1, splits=Some(8))
+- Range (0, 1, step=1, splits=Some(8))

TRACE SparkOptimizer: Fixed point reached for batch Operator Optimizations after 2 iterations.
DEBUG SparkOptimizer:
== Result of Batch Operator Optimizations ==
!Project [cast(id#0L as bigint) AS id#7L]   Range (0, 1, step=1, splits=Some(8))
+- Range (0, 1, step=1, splits=Some(8))
...
== Parsed Logical Plan ==
'Project [unresolvedalias(cast('id as bigint), None)]
+- Range (0, 1, step=1, splits=Some(8))

== Analyzed Logical Plan ==
id: bigint
Project [cast(id#0L as bigint) AS id#7L]
+- Range (0, 1, step=1, splits=Some(8))

== Optimized Logical Plan ==
Range (0, 1, step=1, splits=Some(8))
```

```

== Physical Plan ==
*Range (0, 1, step=1, splits=Some(8))

// Case 2A. The input is of `ArrayType` type and contains no `null` elements.
scala> val intArray = Seq(Array(1)).toDS
intArray: org.apache.spark.sql.Dataset[Array[Int]] = [value: array<int>]

scala> intArray.printSchema
root
|-- value: array (nullable = true)
|   |-- element: integer (containsNull = false)

scala> intArray.map(arr => arr.sum).explain(true)
...
TRACE SparkOptimizer:
== Applying Rule org.apache.spark.sql.catalyst.optimizer.SimplifyCasts ==
SerializeFromObject [input[0, int, true] AS value#36]
    SerializeFromObject [input[0, int, true] AS value#36]
+- MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType, false),true)], obj#35: int +- MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType, false),true)], obj#35: int
!   +- DeserializeToObject cast(value#15 as array<int>).toIntArray, obj#34: [I
        +- DeserializeToObject value#15.toIntArray, obj#34: [I
        +- LocalRelation [value#15]
            +- LocalRelation [value#15]

TRACE SparkOptimizer: Fixed point reached for batch Operator Optimizations after 2 iterations.
DEBUG SparkOptimizer:
== Result of Batch Operator Optimizations ==
SerializeFromObject [input[0, int, true] AS value#36]
    SerializeFromObject [input[0, int, true] AS value#36]
+- MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType, false),true)], obj#35: int +- MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType, false),true)], obj#35: int
!   +- DeserializeToObject cast(value#15 as array<int>).toIntArray, obj#34: [I
        +- DeserializeToObject value#15.toIntArray, obj#34: [I
        +- LocalRelation [value#15]
            +- LocalRelation [value#15]
...
== Parsed Logical Plan ==
'SerializeFromObject [input[0, int, true] AS value#36]
+- 'MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType, false),true)], obj#35: int
    +- 'DeserializeToObject unresolveddeserializer(upcast(getcolumnbyordinal(0, ArrayType(IntegerType, false)), ArrayType(IntegerType, false), - root class: "scala.Array").toIntArray), obj#34: [I
        +- LocalRelation [value#15]

== Analyzed Logical Plan ==
value: int
SerializeFromObject [input[0, int, true] AS value#36]

```

```

++ MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType,false),
true)], obj#35: int
+- DeserializeToObject cast(value#15 as array<int>).toIntArray, obj#34: [I
  +- LocalRelation [value#15]

== Optimized Logical Plan ==
SerializeFromObject [input[0, int, true] AS value#36]
++ MapElements <function1>, class [I, [StructField(value,ArrayType(IntegerType,false),
true)], obj#35: int
+- DeserializeToObject value#15.toIntArray, obj#34: [I
  +- LocalRelation [value#15]

== Physical Plan ==
*SerializeFromObject [input[0, int, true] AS value#36]
+- *MapElements <function1>, obj#35: int
  +- *DeserializeToObject value#15.toIntArray, obj#34: [I
    +- LocalTableScan [value#15]

// Case 2B. The input is of `MapType` type and contains no `null` elements.
scala> val mapDF = Seq(("one", 1), ("two", 2)).toDF("k", "v").withColumn("m", map(col("k"),
"v")))
mapDF: org.apache.spark.sql.DataFrame = [k: string, v: int ... 1 more field]

scala> mapDF.printSchema
root
|-- k: string (nullable = true)
|-- v: integer (nullable = false)
|-- m: map (nullable = false)
|   |-- key: string
|   |-- value: integer (valueContainsNull = false)

scala> mapDF.selectExpr("""CAST (m AS map<string, int>)""").explain(true)
...
TRACE SparkOptimizer:
==== Applying Rule org.apache.spark.sql.catalyst.optimizer.SimplifyCasts ====
!Project [cast(map(_1#250, _2#251) as map<string,int>) AS m#272]   Project [map(_1#250
, _2#251) AS m#272]
  +- LocalRelation [_1#250, _2#251]                                     +- LocalRelation [_
1#250, _2#251]
...
== Parsed Logical Plan ==
'Project [unresolvedalias(cast('m as map<string,int>), None)]
+- Project [k#253, v#254, map(k#253, v#254) AS m#258]
  +- Project [_1#250 AS k#253, _2#251 AS v#254]
    +- LocalRelation [_1#250, _2#251]

== Analyzed Logical Plan ==
m: map<string,int>
Project [cast(m#258 as map<string,int>) AS m#272]
+- Project [k#253, v#254, map(k#253, v#254) AS m#258]
  +- Project [_1#250 AS k#253, _2#251 AS v#254]
    +- LocalRelation [_1#250, _2#251]

```

```
== Optimized Logical Plan ==
LocalRelation [m#272]

== Physical Plan ==
LocalTableScan [m#272]
```

SparkPlan — Physical Query Plan / Physical Operator

`SparkPlan` is the base [Catalyst query plan](#) for **physical operators** that used (*composed*) together build a **physical query plan** (aka *query execution plan*).

	A physical operator is a Catalyst node that may have zero or more children . Spark SQL uses Catalyst (tree manipulation framework) to compose nodes to build a tree that, in this context, translates to composing physical plan nodes to build a physical plan tree.
--	--

When [executed](#), a physical operator produces an RDD of rows (in the [internal binary row format](#)).

Note	<code>execute</code> is called when <code>QueryExecution</code> is requested for an <code>RDD</code> which happens exactly when your query is executed.
------	---

	Use explain operator to see the execution plan of a structured query. <pre>val q = // your query here q.explain</pre>
Tip	You may also access the execution plan of a <code>dataset</code> using its <code>queryExecution</code> property. <pre>val q = // your query here q.queryExecution.sparkPlan</pre>

The [SparkPlan contract](#) assumes that concrete physical operators define `doExecute` method (with optional [hooks](#) like `doPrepare`) which are executed when the physical operator is [executed](#).

Caution	FIXME A picture with methods/hooks called.
Caution	FIXME <code>SparkPlan</code> is <code>Serializable</code> . Why?

Table 1. SparkPlan's Atributes

Name	Description
metadata	
metrics	
outputOrdering	

SparkPlan has the following `final` methods that prepare execution environment and pass calls on to corresponding methods (that constitute [SparkPlan Contract](#)).

Table 2. SparkPlan's Final Methods

Name	Description
	<p>Executes a physical operator and generates a <code>RDD</code> of internal binary rows.</p> <pre>final def execute(): RDD[InternalRow]</pre>
execute	<p>Used <i>most importantly</i> when <code>QueryExecution</code> is requested for a <code>RDD</code> (that in turn triggers execution of any children the physical operator may have).</p> <p>Internally, <code>execute</code> first prepares the query for execution, waits for any subqueries (the query may have) and eventually calls <code>doExecute</code>.</p>
prepare	<p>Prepares a query for execution.</p> <p>Internally, <code>prepare</code> calls <code>doPrepare</code> of its children first followed by <code>prepareSubqueries</code> and <code>doPrepare</code>.</p>
executeBroadcast	Calls <code>doExecuteBroadcast</code>

Table 3. Physical Query Operators / Specialized SparkPlans

Name	Description
BinaryExecNode	Binary physical operator with two child <code>left</code> and <code>right</code> physical operators
LeafExecNode	Leaf physical operator with no children By default, the set of all attributes that are produced is exactly the set of attributes that are output .
UnaryExecNode	Unary physical operator with one <code>child</code> physical operator

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> or <code>LocalTableScanExec</code> that is however removed when the operator is displayed, e.g. in web UI .
------	---

prepareSubqueries Method

Caution

FIXME

getByteArrayRdd Internal Method

`getByteArrayRdd(n: Int = -1): RDD[Array[Byte]]`

Caution

FIXME

waitForSubqueries Method

Caution

FIXME

executeCollect Method

Caution

FIXME

Note `executeCollect` does not convert data to JVM types.

executeToIterator Method

Caution

FIXME

SparkPlan Contract

`SparkPlan` contract requires that concrete physical operators define their own custom `doExecute`.`doExecute(): RDD[InternalRow]``doExecute` produces the result of a structured query as an `RDD` of `InternalRow` objects.

Table 4. SparkPlan's Extension Hooks (in alphabetical order)

Name	Description
doExecuteBroadcast	<p>By default reports a <code>UnsupportedOperationException</code>.</p> <p><code>[nodeName]</code> does not implement <code>doExecuteBroadcast</code></p> <p>Executed exclusively as part of <code>executeBroadcast</code> to return the result of a structured query as a broadcast variable.</p>
doPrepare	<p>Prepares a physical operator for execution.</p> <p>Executed exclusively as part of <code>prepare</code> and is supposed to set some state up before executing a query (e.g. <code>BroadcastExchangeExec</code> to broadcast asynchronously).</p>
outputPartitioning	Specifies how data is partitioned across different nodes in the cluster
requiredChildDistribution	<p>Specifies any partition requirements on the input data</p> <p>Used exclusively when <code>EnsureRequirements</code> enforces partition requirements of a physical operator.</p>

Executing Query in Scope (after Preparations)

— `executeQuery` Final Method

```
executeQuery[T](query: => T): T
```

`executeQuery` executes `query` in a scope (i.e. so that all RDDs created will have the same scope for visualization like web UI).

Internally, `executeQuery` calls `prepare` and `waitForSubqueries` followed by executing `query`.

Note

`executeQuery` is executed as part of `execute`, `executeBroadcast` and when `CodegenSupport`-enabled physical operator produces a Java source code.

Broadcasting Result of Structured Query

— `executeBroadcast` Final Method

```
executeBroadcast[T](): broadcast.Broadcast[T]
```

`executeBroadcast` returns the result of a structured query as a broadcast variable.

Internally, `executeBroadcast` calls `doExecuteBroadcast` inside `executeQuery`.

Note

`executeBroadcast` is called in `BroadcastHashJoinExec`, `BroadcastNestedLoopJoinExec` and `ReusedExchangeExec` physical operators.

metrics Internal Registry

```
metrics: Map[String, SQLMetric] = Map.empty
```

`metrics` is a registry of supported `SQLMetrics` by their names.

BroadcastExchangeExec Unary Operator for Broadcasting Joins

`BroadcastExchangeExec` is a [physical operator](#) (with one [child](#) physical operator) to broadcast rows (of a relation) to worker nodes.

`BroadcastExchangeExec` is [created](#) exclusively when `EnsureRequirements` physical query plan optimization [ensures BroadcastDistribution](#) of the input data of a physical operator (that seemingly can be either `BroadcastHashJoinExec` or `BroadcastNestedLoopJoinExec` operators).

```
val t1 = spark.range(5)
val t2 = spark.range(5)
val q = t1.join(t2).where(t1("id") === t2("id"))

scala> q.explain
== Physical Plan ==
*BroadcastHashJoin [id#19L, [id#22L], Inner, BuildRight
:- *Range (0, 5, step=1, splits=Some(8))
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
   +- *Range (0, 5, step=1, splits=Some(8))
```

Table 1. BroadcastExchangeExec SQLMetrics (in alphabetical order)

Name	Description
broadcastTime	time to broadcast (ms)
buildTime	time to build (ms)
collectTime	time to collect (ms)
dataSize	data size (bytes)

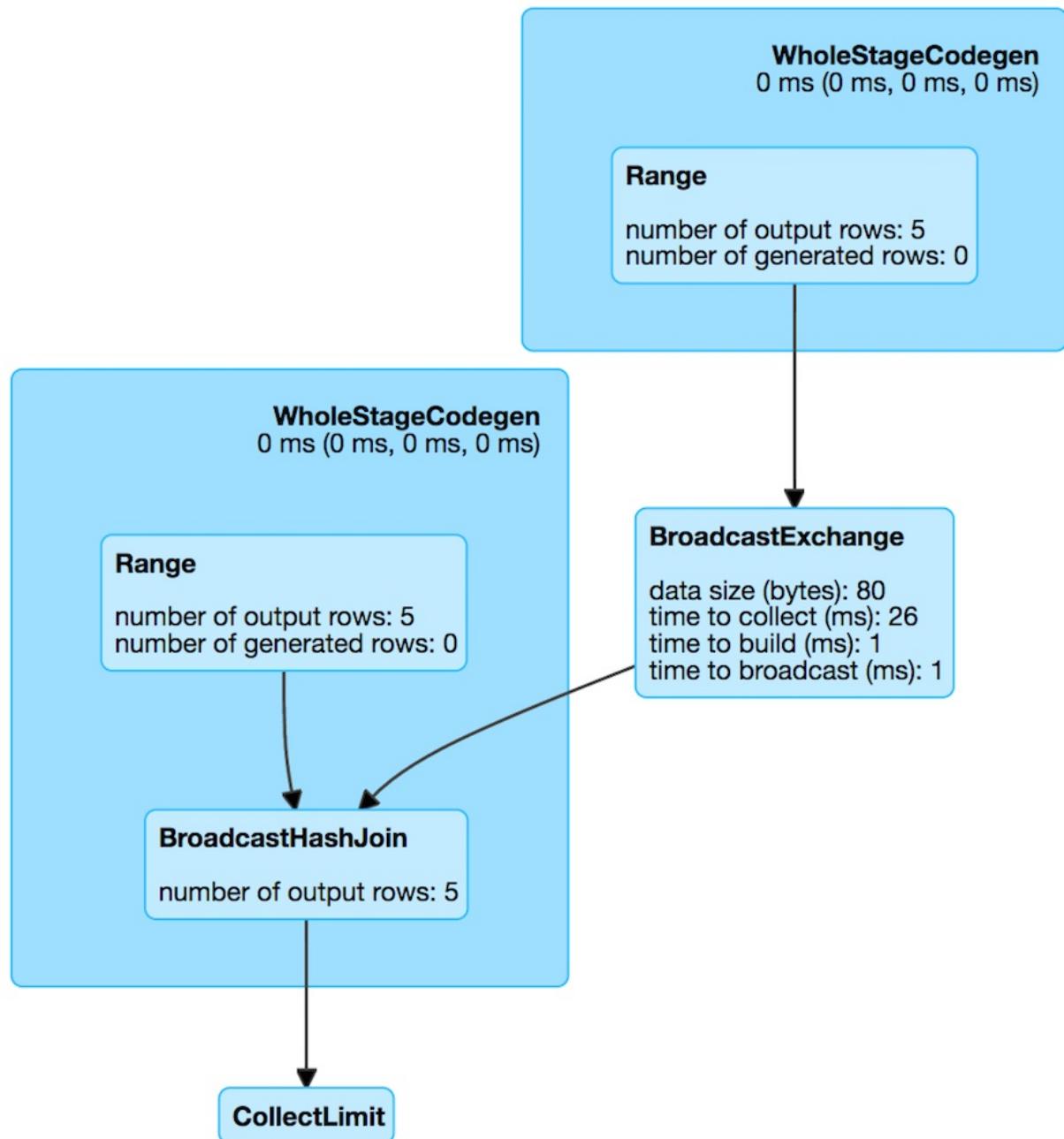


Figure 1. BroadcastExchangeExec in web UI (Details for Query)

`BroadcastExchangeExec` uses `BroadcastPartitioning` partitioning scheme (with the input `BroadcastMode`).

Creating BroadcastExchangeExec Instance

`BroadcastExchangeExec` takes the following when created:

- `BroadcastMode`
- Child [logical plan](#)

Preparing Asynchronous Broadcast (with Rows)

— doPrepare Method

```
doPrepare(): Unit
```

`doPrepare` "materializes" the internal lazily-once-initialized [asynchronous broadcast](#).

Note

`doPrepare` is a part of [SparkPlan Contract](#) to prepare a physical operator for execution.

Broadcasting Rows — doExecuteBroadcast Method

```
def doExecuteBroadcast[T](): broadcast.Broadcast[T]
```

`doExecuteBroadcast` waits until the [rows](#) are [broadcast](#).

Note

`doExecuteBroadcast` waits [spark.sql.broadcastTimeout](#) (i.e. 5 minutes).

Note

`doExecuteBroadcast` is a part of [SparkPlan Contract](#) to return the result of a structured query as a broadcast variable.

Lazily-Once-Initialized Asynchronously-Broadcast relationFuture Internal Attribute

```
relationFuture: Future[broadcast.Broadcast[Any]]
```

When "materialized" (aka [executed](#)), `relationFuture` finds the current [execution id](#) and sets it to the `Future` thread.

`relationFuture` requests [child physical operator](#) to [executeCollect](#).

`relationFuture` records the time for `executeCollect` in [collectTime](#) metrics and the size of the data in [dataSize](#) metrics.

Note

`relationFuture` accepts a relation with up to 512 millions rows and 8GB in size, and reports a `SparkException` if the conditions are violated.

`relationFuture` requests the input [BroadcastMode](#) to [transform](#) the internal rows and records the time in [buildTime](#) metrics.

`relationFuture` requests the current `SparkContext` to `broadcast` the transformed internal rows and records the time in [broadcastTime](#) metrics.

In the end, `relationFuture` posts `SparkListenerDriverAccumUpdates` (with the execution id and the metrics) and returns the broadcast internal rows.

In case of `OutOfMemoryError`, `relationFuture` reports another `OutOfMemoryError` with the following message:

Not enough memory to build and broadcast the table to all worker nodes. As a workaround, you can either disable broadcast by setting `spark.sql.autoBroadcastJoinThreshold` to `-1` or increase the spark driver memory by setting `spark.driver.memory` to a higher value

Note	<code>relationFuture</code> is executed on a separate thread from a custom <code>scala.concurrent.ExecutionContext</code> (built from a cached <code>java.util.concurrent.ThreadPoolExecutor</code> with the prefix broadcast-exchange and 128 threads).
------	---

BroadcastHashJoinExec Binary Physical Operator

`BroadcastHashJoinExec` is a [binary physical operator](#) (with two child `left` and `right` physical operators) that [supports code generation](#) (aka `codegen`).

`BroadcastHashJoinExec` is [created](#) after applying [JoinSelection](#) execution planning strategy to [ExtractEquiJoinKeys](#)-destructurable logical query plans (i.e. [INNER](#), [CROSS](#), [LEFT OUTER](#), [LEFT SEMI](#), [LEFT ANTI](#)) of which the `right` physical operator [can be broadcast](#).

`BroadcastHashJoinExec` [requires that partition requirements](#) for the two children physical operators match `BroadcastDistribution` (with `HashedRelationBroadcastMode`) and `UnspecifiedDistribution` (for `left` and `right` sides of a join or vice versa).

```

val tokens = Seq(
  (0, "playing"),
  (1, "with"),
  (2, "BroadcastHashJoinExec")
).toDF("id", "token")

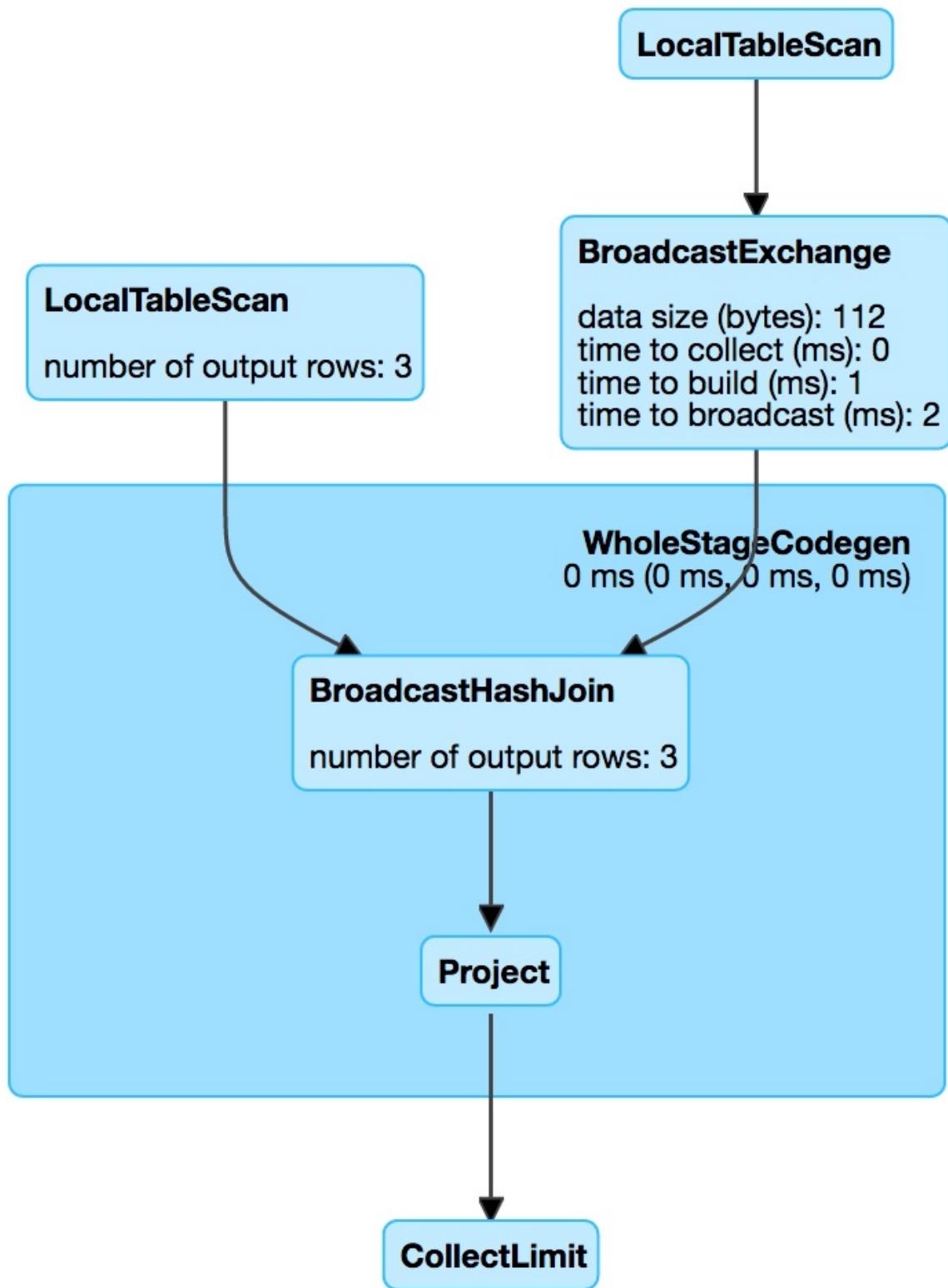
scala> spark.conf.get("spark.sql.autoBroadcastJoinThreshold")
res0: String = 10485760

val q = tokens.join(tokens, Seq("id"), "inner")
scala> q.explain
== Physical Plan ==
*Project [id#15, token#16, token#21]
+- *BroadcastHashJoin [id#15], [id#20], Inner, BuildRight
  :- LocalTableScan [id#15, token#16]
  +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as
bigint)))
    +- LocalTableScan [id#20, token#21]

```

Table 1. BroadcastHashJoinExec SQLMetrics (in alphabetical order)

Name	Description
<code>numOutputRows</code>	Number of output rows

Figure 1. `BroadcastHashJoinExec` in web UI (Details for Query)

Note

The prefix for variable names for `BroadcastHashJoinExec` operators in [CodegenSupport](#)-generated code is **bhj**.

```

scala> q.queryExecution.debug.codegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Project [id#15, token#16, token#21]
+- *BroadcastHashJoin [id#15], [id#20], Inner, BuildRight
  :- LocalTableScan [id#15, token#16]
  +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as
bigint)))
    +- LocalTableScan [id#20, token#21]

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;
/* 007 */   private scala.collection.Iterator[] inputs;
/* 008 */   private scala.collection.Iterator inputadapter_input;
/* 009 */   private org.apache.spark.broadcast.TorrentBroadcast bhj_broadcast;
/* 010 */   private org.apache.spark.sql.execution.joins.LongHashedRelation bhj_relation;
/* 011 */   private org.apache.spark.sql.execution.metric.SQLMetric bhj_numOutputRows;
/* 012 */   private UnsafeRow bhj_result;
/* 013 */   private org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder bhj_holder;
/* 014 */   private org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter bhj_rowWriter;
...

```

Creating BroadcastHashJoinExec Instance

`BroadcastHashJoinExec` takes the following when created:

- Left join key [expressions](#)
- Right join key [expressions](#)
- [Join type](#)
- [BuildSide](#)
- Optional join condition [expression](#)
- Left [physical operator](#)
- Right [physical operator](#)

requiredChildDistribution Method

```
requiredChildDistribution: Seq[Distribution]
```

`requiredChildDistribution` gives a two-element collection for two-child partition requirements with the following:

- `BroadcastDistribution` and `UnspecifiedDistribution` for [BuildLeft build side](#)
- `UnspecifiedDistribution` and `BroadcastDistribution` for [BuildRight build side](#)

Note	<code>BroadcastDistribution</code> uses <code>HashedRelationBroadcastMode</code> broadcast mode.
------	--

Note	<code>requiredChildDistribution</code> is a part of SparkPlan Contract to specify partition requirements on the input data.
------	---

BroadcastNestedLoopJoinExec Binary Physical Operator

`BroadcastNestedLoopJoinExec` is a [binary physical operator](#) (with two child `left` and `right` physical operators) that is [created](#) (and converted to) when [JoinSelection](#) physical plan strategy finds a [Join](#) logical operator that meets either case:

1. `canBuildRight` join type and `right` physical operator [broadcastable](#)
2. `canBuildLeft` join type and `left` [broadcastable](#)
3. non- `InnerLike` join type

Note	<code>BroadcastNestedLoopJoinExec</code> is the default physical operator when no other operators have matched selection requirements .
------	---

Note	<code>canBuildRight</code> join types are: <ul style="list-style-type: none"> • CROSS, INNER, LEFT ANTI, LEFT OUTER, LEFT SEMI or Existence <code>canBuildLeft</code> join types are: <ul style="list-style-type: none"> • CROSS, INNER, RIGHT OUTER
------	--

```

val nums = spark.range(2)
val letters = ('a' to 'c').map(_.toString).toDF("letter")
val q = nums.crossJoin(letters)

scala> q.explain
== Physical Plan ==
BroadcastNestedLoopJoin BuildRight, Cross
:- *Range (0, 2, step=1, splits=Some(8))
+- BroadcastExchange IdentityBroadcastMode
  +- LocalTableScan [letter#69]

```

Table 1. BroadcastNestedLoopJoinExec's SQLMetrics

Name	Description
numOutputRows	Number of output rows

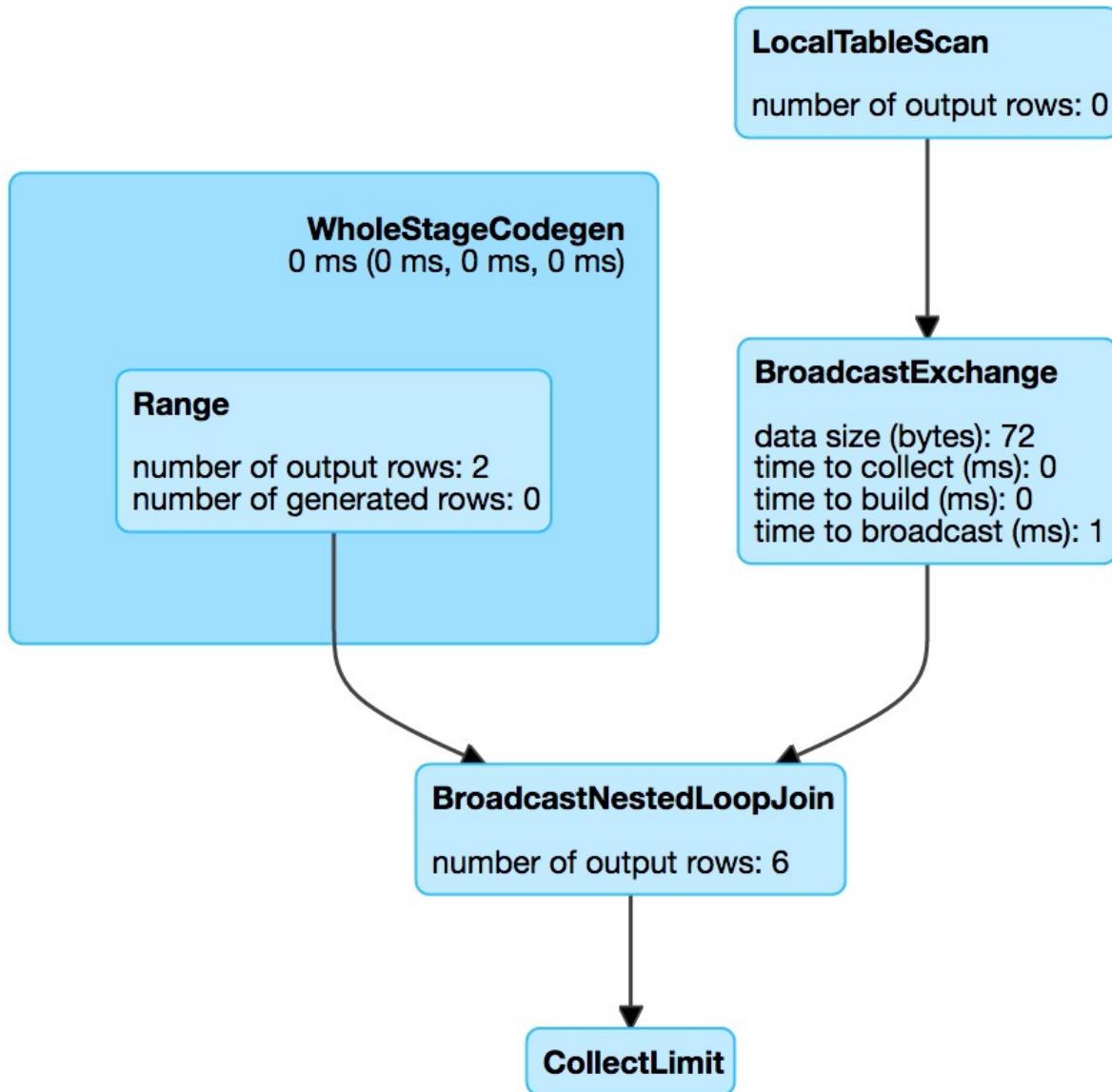


Figure 1. BroadcastNestedLoopJoinExec in web UI (Details for Query)

`BroadcastNestedLoopJoinExec` requires that partition requirements for the two child physical operators match `BroadcastDistribution` (with `IdentityBroadcastMode`) and `UnspecifiedDistribution` (for `left` and `right` sides of a join or vice versa).

Creating BroadcastNestedLoopJoinExec Instance

`BroadcastNestedLoopJoinExec` takes the following when created:

- Left `physical operator`
- Right `physical operator`
- `BuildSide`
- `Join type`

- Optional join condition [expressions](#)

requiredChildDistribution Method

```
requiredChildDistribution: Seq[Distribution]
```

`requiredChildDistribution` gives a two-element collection for two-child partition requirements with the following:

- `BroadcastDistribution` and `UnspecifiedDistribution` for [BuildLeft build side](#)
- `UnspecifiedDistribution` and `BroadcastDistribution` for [BuildRight build side](#)

Note	<code>BroadcastDistribution</code> uses <code>IdentityBroadcastMode</code> broadcast mode.
------	--

Note	<code>requiredChildDistribution</code> is a part of SparkPlan Contract to specify partition requirements on the input data.
------	---

CoalesceExec Unary Physical Operator

`CoalesceExec` is a [unary physical operator](#) with `numPartitions` number of partitions and a child `spark plan`.

`CoalesceExec` represents [Repartition](#) logical operator at execution (when shuffle was disabled — see [BasicOperators](#) execution planning strategy). When executed, it executes the input `child` and calls [coalesce](#) on the result RDD (with `shuffle` disabled).

Please note that since physical operators present themselves without the suffix *Exec*, `CoalesceExec` is the `Coalesce` in the Physical Plan section in the following example:

```
scala> df.rdd.getNumPartitions
res6: Int = 8

scala> df.coalesce(1).rdd.getNumPartitions
res7: Int = 1

scala> df.coalesce(1).explain(extended = true)
== Parsed Logical Plan ==
Repartition 1, false
+- LocalRelation [value#1]

== Analyzed Logical Plan ==
value: int
Repartition 1, false
+- LocalRelation [value#1]

== Optimized Logical Plan ==
Repartition 1, false
+- LocalRelation [value#1]

== Physical Plan ==
Coalesce 1
+- LocalTableScan [value#1]
```

`output collection of Attribute` matches the `child`'s (since `CoalesceExec` is about changing the number of partitions not the internal representation).

`outputPartitioning` returns a `SinglePartition` when the input `numPartitions` is `1` while a `UnknownPartitioning` for the other cases.

DataSourceScanExec — Contract for Leaf Physical Operators with Code Generation

`DataSourceScanExec` is a [contract](#) for leaf physical operators with [support for code generation](#) that...[FIXME](#)

Note	The prefix for variable names for <code>DataSourceScanExec</code> operators in CodegenSupport -generated code is scan .
------	--

Table 1. `DataSourceScanExec`'s Known Implementations (in alphabetical order)

DataSourceScanExec	Description
FileSourceScanExec	
RowDataSourceScanExec	

DataSourceScanExec Contract

```
package org.apache.spark.sql.execution

trait DataSourceScanExec extends LeafExecNode with CodegenSupport {
    // only required vals and methods that have no implementation
    val metastoreTableIdentifier: Option[TableIdentifier]
    val relation: BaseRelation
}
```

Table 2. (Subset of) `DataSourceScanExec` Contract (in alphabetical order)

Method	Description
<code>metastoreTableIdentifier</code>	<code>TableIdentifier</code> that... FIXME
<code>relation</code>	<code>BaseRelation</code> that... FIXME

FileSourceScanExec Physical Operator

RowDataSourceScanExec Physical Operator

RowDataSourceScanExec is a [DataSourceScanExec](#) for scanning data from a relation.

RowDataSourceScanExec is [created](#) for [LogicalRelation](#) with different kinds of relations (in [DataSourceStrategy](#) execution planning strategy).

Creating RowDataSourceScanExec Instance

RowDataSourceScanExec takes the following when created:

- Output attributes
- RDD of [internal binary rows](#)
- [BaseRelation](#)
- Output partitioning
- Metadata (as a collection of pairs)
- Optional [TableIdentifier](#)

ExecutedCommandExec Physical Operator for Command Execution

`ExecutedCommandExec` is a [SparkPlan](#) for executing [logical commands with side effects](#).

`ExecutedCommandExec` runs a command and caches the result in `sideEffectResult` internal attribute.

Table 1. ExecutedCommandExec's Methods (in alphabetical order)

Method	Description
<code>doExecute</code>	
<code>executeCollect</code>	
<code>executeTake</code>	
<code>executeToIterator</code>	

Executing Logical RunnableCommand and Caching Result As InternalRows — `sideEffectResult` Internal Lazy Attribute

```
sideEffectResult: Seq[InternalRow]
```

`sideEffectResult` runs the `RunnableCommand` (that produces a `Seq[Row]`) and converts the result to a `Seq[InternalRow]` using a Catalyst converter function for a given [schema](#).

Caution	FIXME <code>CatalystTypeConverters.createToCatalystConverter</code> ?
---------	---

Note	<code>sideEffectResult</code> is used when <code>ExecutedCommandExec</code> is requested for executeCollect , executeToIterator , executeTake , doExecute .
------	---

HashAggregateExec Aggregate Physical Operator

`HashAggregateExec` is a [unary physical operator](#) that [supports code generation](#) (aka `codegen`).

`HashAggregateExec` is [created](#) (indirectly through `AggUtils.createAggregate`) when:

- [Aggregation](#) execution planning strategy creates [physical plans](#) for an [Aggregate](#) unary logical operator (with or no distinct)
- Structured Streaming's `StatefulAggregationStrategy` strategy creates plan for streaming `EventTimeWatermark` or [Aggregate](#) unary logical operators

```
// groupBy aggregate logical operator mapped to HashAggregateExec
val q = spark.range(10).groupBy('id % 2 as "group").sum("id")
scala> q.explain
== Physical Plan ==
*HashAggregate(keys=[(id#109L % 2)#121L], functions=[sum(id#109L)])
+- Exchange hashpartitioning((id#109L % 2)#121L, 200)
  +- *HashAggregate(keys=[(id#109L % 2) AS (id#109L % 2)#121L], functions=[partial_sum(id#109L)])
    +- *Range (0, 10, step=1, splits=Some(8))

scala> q.queryExecution.sparkPlan
res11: org.apache.spark.sql.execution.SparkPlan =
HashAggregate(keys=[(id#109L % 2)#124L], functions=[sum(id#109L)], output=[group#112L,
  sum(id)#116L])
+- HashAggregate(keys=[(id#109L % 2) AS (id#109L % 2)#124L], functions=[partial_sum(id#
  109L)], output=[(id#109L % 2)#124L, sum#123L])
  +- Range (0, 10, step=1, splits=Some(8))

import org.apache.spark.sql.execution.aggregate.HashAggregateExec
scala> val agg = q.queryExecution.sparkPlan.children(0).asInstanceOf[HashAggregateExec]
]
agg: org.apache.spark.sql.execution.aggregate.HashAggregateExec =
HashAggregate(keys=[(id#109L % 2) AS (id#109L % 2)#124L], functions=[partial_sum(id#109
L)], output=[(id#109L % 2)#124L, sum#123L])
+- Range (0, 10, step=1, splits=Some(8))
```

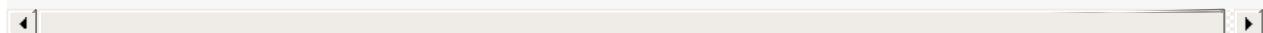


Table 1. HashAggregateExec SQLMetrics (in alphabetical order)

Name	Description
aggTime	aggregate time
numOutputRows	number of output rows
peakMemory	peak memory
spillSize	spill size

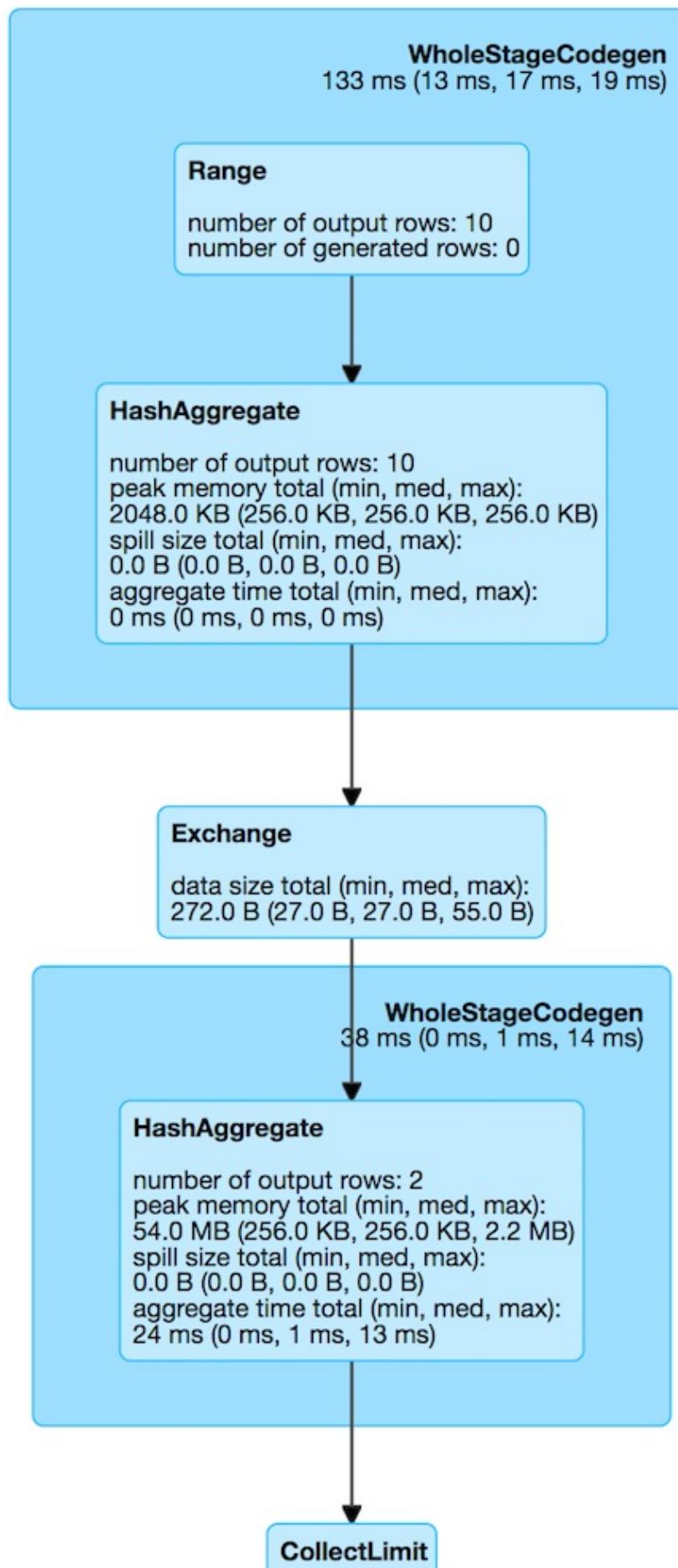


Figure 1. HashAggregateExec in web UI (Details for Query)

Table 2. HashAggregateExec's Properties (in alphabetical order)

Name	Description
aggregateBufferAttributes	Collection of <code>AttributeReference</code> references of the aggregate functions of the input <code>AggregateExpressions</code>
output	Output <code>attributes</code> for the input <code>NamedExpressions</code>
Note	The prefix for variable names for <code>HashAggregateExec</code> operators in <code>CodegenSupport</code> -generated code is <code>agg</code> .

Creating HashAggregateExec Instance

`HashAggregateExec` takes the following when created:

- Optional `expressions`
- Grouping `named expressions`
- Collection of `AggregateExpressions`
- Collection of `Attributes`
- Initial offset
- Output `named expressions`
- Child `physical plan`

Generating InternalRows (using TungstenAggregationIterator) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

`doExecute` executes the input `child SparkPlan` (to produce `InternalRow` objects) and applies calculation over partitions (using `RDD.mapPartitions`).

In the `mapPartitions` block, `doExecute` creates one of the following:

- an empty iterator for no-record partitions with at least one grouping expression
- `TungstenAggregationIterator`

Note	<code>doExecute</code> is a part of <code>SparkPlan Contract</code> to produce the result of a structured query as an <code>RDD</code> of <code>InternalRow</code> objects.
------	---

doProduce Method

```
doProduce(ctx: CodegenContext): String
```

`doProduce` executes `doProduceWithoutKeys` when no `groupingExpressions` were specified for the `HashAggregateExec` or `doProduceWithKeys` otherwise.

Note

`doProduce` is a part of `CodegenSupport Contract`.

InMemoryTableScanExec Physical Operator

`InMemoryTableScanExec` is a [leaf physical operator](#) that...[FIXME](#)

`InMemoryTableScanExec` is [created](#) exclusively when `InMemoryScans` execution planning strategy finds `InMemoryRelation` logical operators.

```
// Sample DataFrames
val tokens = Seq(
  (0, "playing"),
  (1, "with"),
  (2, "InMemoryTableScanExec")
).toDF("id", "token")
val ids = spark.range(10)

// Cache DataFrames
tokens.cache
ids.cache

val q = tokens.join(ids, Seq("id"), "outer")
scala> q.explain
== Physical Plan ==
*Project [coalesce(cast(id#5 as bigint), id#10L) AS id#33L, token#6]
+- SortMergeJoin [cast(id#5 as bigint)], [id#10L], FullOuter
  :- *Sort [cast(id#5 as bigint) ASC NULLS FIRST], false, 0
  :  +- Exchange hashpartitioning(cast(id#5 as bigint), 200)
  :    +- InMemoryTableScan [id#5, token#6]
  :      +- InMemoryRelation [id#5, token#6], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
  :         +- LocalTableScan [id#5, token#6]
  +- *Sort [id#10L ASC NULLS FIRST], false, 0
    +- Exchange hashpartitioning(id#10L, 200)
      +- InMemoryTableScan [id#10L]
        +- InMemoryRelation [id#10L], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
          +- *Range (0, 10, step=1, splits=8)
```

Table 1. InMemoryTableScanExec SQLMetrics (in alphabetical order)

Name	Description
<code>numOutputRows</code>	Number of output rows

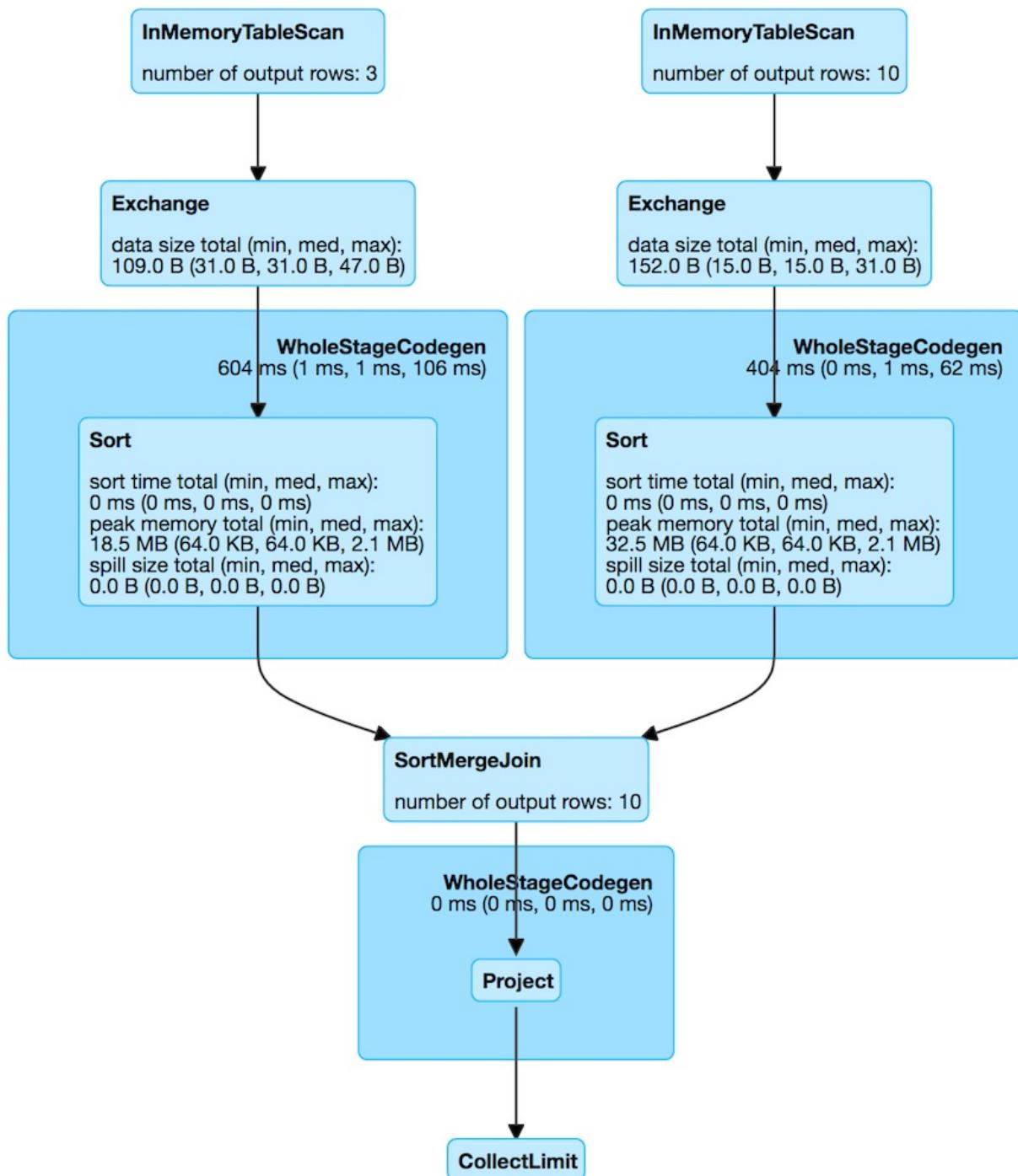


Figure 1. InMemoryTableScanExec in web UI (Details for Query)

InMemoryTableScanExec uses `spark.sql.inMemoryTableScanStatistics.enable` flag (default: disabled) to enable accumulators (that appears exclusively for testing purposes).

Creating InMemoryTableScanExec Instance

InMemoryTableScanExec takes the following when created:

- Attribute expressions
- Predicate expressions

- [InMemoryRelation](#) logical operator

LocalTableScanExec Physical Operator

`LocalTableScanExec` is a [leaf physical operator](#) with no `children` and `producedAttributes` being `outputSet`.

`LocalTableScanExec` is a result of applying [BasicOperators](#) execution planning strategy to [LocalRelation](#) and [MemoryPlan](#) logical query plans.

```
scala> Seq(1).toDS.explain(extended = true)
== Parsed Logical Plan ==
LocalRelation [value#1]

== Analyzed Logical Plan ==
value: int
LocalRelation [value#1]

== Optimized Logical Plan ==
LocalRelation [value#1]

== Physical Plan ==
LocalTableScan [value#1]
```

Table 1. LocalTableScanExec's Metrics

name	description
numOutputRows	the number of output rows

When executed (as `doExecute`), `LocalTableScanExec` creates an `RDD` of `InternalRow`s.

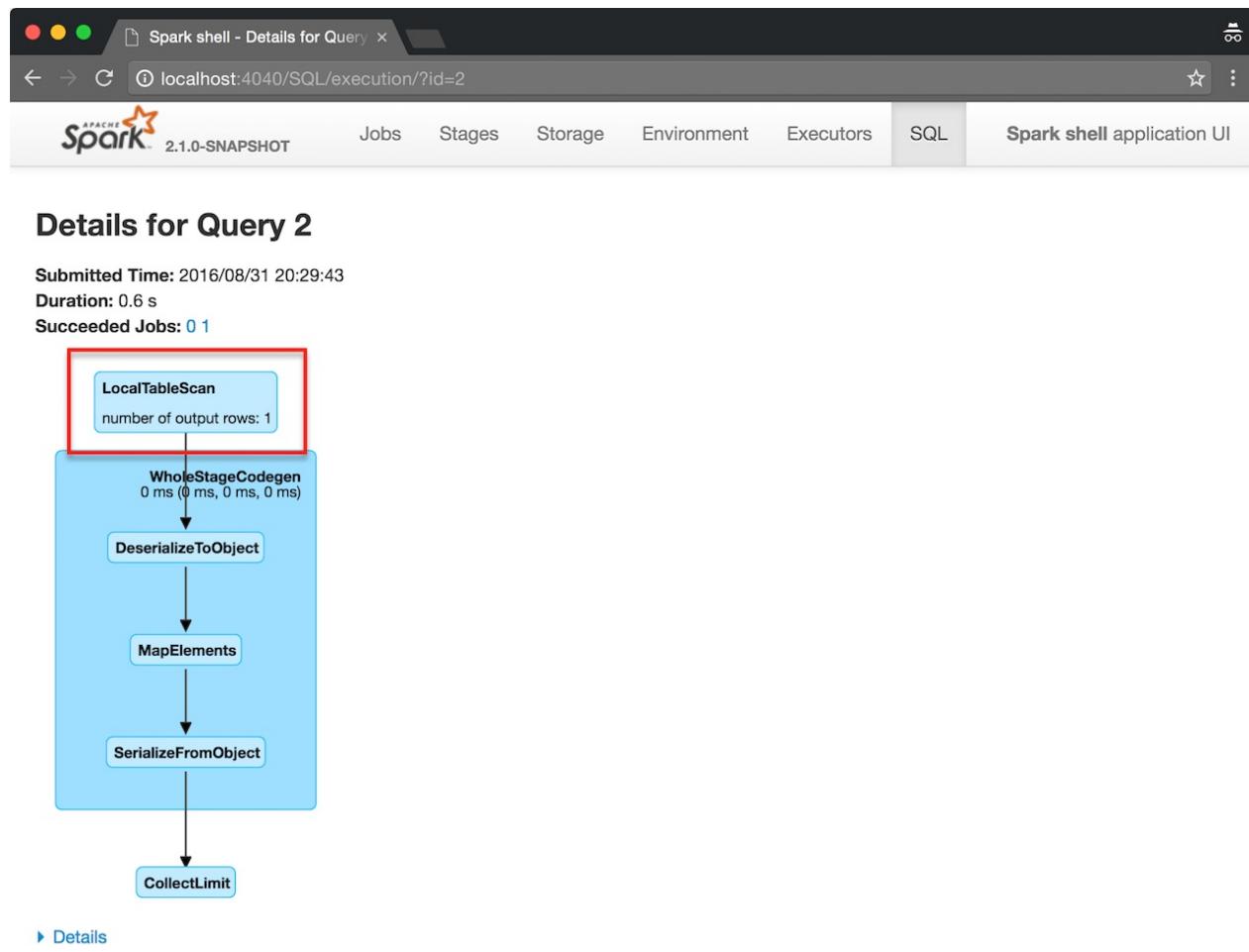


Figure 1. LocalTableScanExec in SQL tab in web UI

ShuffleExchange Unary Physical Operator

`ShuffleExchange` is a [physical operator](#) (with one `child` physical operator) to perform a shuffle.

`ShuffleExchange` corresponds to `Repartition` (with shuffle enabled) and `RepartitionByExpression` logical operators (as translated in [BasicOperators](#) execution planning strategy).

Note

`ShuffleExchange` shows as **Exchange** in physical plans.

```
// Uses Repartition logical operator
// ShuffleExchange with RoundRobinPartitioning
val q1 = spark.range(6).repartition(2)
scala> q1.explain
== Physical Plan ==
Exchange RoundRobinPartitioning(2)
+- *Range (0, 6, step=1, splits=Some(8))

// Uses RepartitionByExpression logical operator
// ShuffleExchange with HashPartitioning
val q2 = spark.range(6).repartition(2, 'id % 2)
scala> q2.explain
== Physical Plan ==
Exchange hashpartitioning((id#38L % 2), 2)
+- *Range (0, 6, step=1, splits=Some(8))
```

When created, `ShuffleExchange` takes a `Partitioning`, a single `child` [physical operator](#) and an optional [ExchangeCoordinator](#).

Table 1. ShuffleExchange SQLMetrics (in alphabetical order)

Name	Description
<code>dataSize</code>	data size

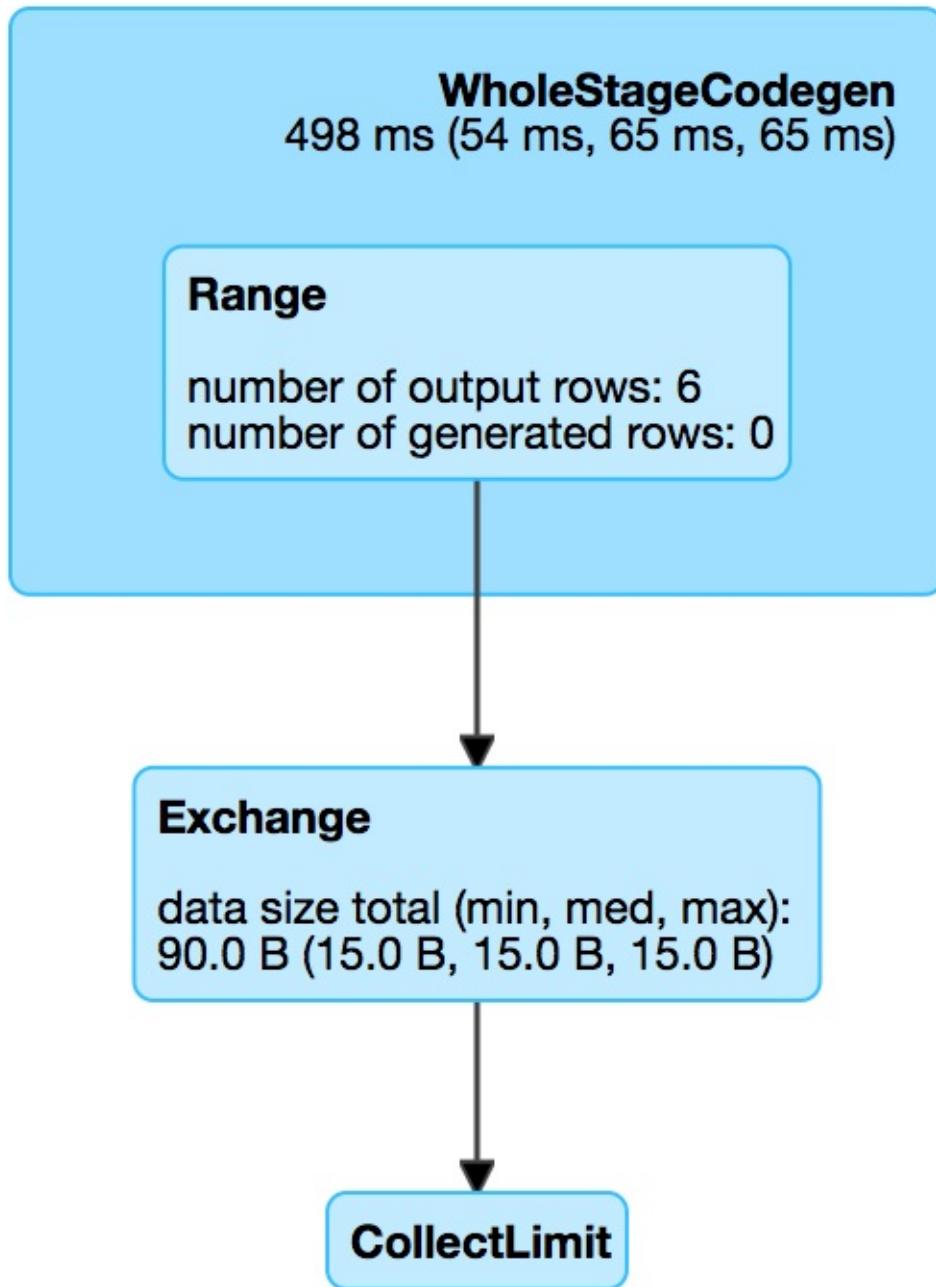


Figure 1. ShuffleExchange in web UI (Details for Query)

`nodeName` is computed based on the optional `ExchangeCoordinator` with `Exchange` prefix and possibly `(coordinator id: [coordinator-hash-code])`.

`outputPartitioning` is the input `Partitioning`.

While `preparing execution` (using `doPrepare`), `shuffleExchange` registers itself with the `ExchangeCoordinator` if available.

When `doExecute`, `shuffleExchange` computes a `ShuffledRowRDD` and caches it (to reuse avoiding possibly expensive executions).

Table 2. ShuffleExchange's Internal Registries and Counters (in alphabetical order)

Name	Description
cachedShuffleRDD	ShuffledRowRDD that is cached after <code>ShuffleExchange</code> has been <code>executed</code> .

Creating ShuffledRowRDD (with InternalRow) Using Optional ExchangeCoordinator — `doExecute` Method

`doExecute(): RDD[InternalRow]`

`doExecute` creates a new `ShuffledRowRDD` or takes cached one.

`doExecute` branches off per optional `ExchangeCoordinator`.

If `ExchangeCoordinator` was specified, `doExecute` requests `ExchangeCoordinator` for a `ShuffledRowRDD`.

Otherwise (with no `ExchangeCoordinator` specified), `doExecute` `prepareShuffleDependency` and `preparePostShuffleRDD`.

In the end, `doExecute` saves the result `shuffledRowRDD` for later use.

Note	<code>doExecute</code> is a part of <code>SparkPlan Contract</code> .
------	---

preparePostShuffleRDD Method

Caution	<code>FIXME</code>
---------	--------------------

prepareShuffleDependency Internal Method

`prepareShuffleDependency(): ShuffleDependency[Int, InternalRow, InternalRow]`

Caution	<code>FIXME</code>
---------	--------------------

prepareShuffleDependency Helper Method

```
prepareShuffleDependency(  
    rdd: RDD[InternalRow],  
    outputAttributes: Seq[Attribute],  
    newPartitioning: Partitioning,  
    serializer: Serializer): ShuffleDependency[Int, InternalRow, InternalRow]
```

`prepareShuffleDependency` creates a [ShuffleDependency](#) dependency.

Note

`prepareShuffleDependency` is used when `shuffleExchange` prepares a [ShuffleDependency](#) (as part of...[FIXME](#)), `collectLimitExec` and `TakeOrderedAndProjectExec` physical operators are executed.

ShuffledHashJoinExec Binary Physical Operator

`ShuffledHashJoinExec` is a [binary physical operator](#) for hash-based joins.

`ShuffledHashJoinExec` is [created](#) for joins with joining keys and one of the following holds:

- `spark.sql.join.preferSortMergeJoin` is disabled, `canBuildRight`, `canBuildLocalHashMap` for right join side and finally right join side is [much smaller](#) than left side
- `spark.sql.join.preferSortMergeJoin` is disabled, `canBuildLeft`, `canBuildLocalHashMap` for left join side and finally left join side is [much smaller](#) than right
- Left join keys are **not** [orderable](#)

```
*****
Start spark-shell with ShuffledHashJoinExec's selection requirements

./bin/spark-shell \
  -c spark.sql.join.preferSortMergeJoin=False \
  -c spark.sql.autoBroadcastJoinThreshold=1
*****

scala> spark.conf.get("spark.sql.join.preferSortMergeJoin")
res0: String = false

scala> spark.conf.get("spark.sql.autoBroadcastJoinThreshold")
res1: String = 1

scala> spark.conf.get("spark.sql.shuffle.partitions")
res2: String = 200

val dataset = Seq(
  (0, "playing"),
  (1, "with"),
  (2, "ShuffledHashJoinExec")
).toDF("id", "token")
val q = dataset.join(dataset, Seq("id"), "leftsemi")

scala> q.queryExecution.optimizedPlan.stats(spark.sessionState.conf).sizeInBytes
res3: BigInt = 72

scala> q.explain
== Physical Plan ==
ShuffledHashJoin [id#15], [id#20], LeftSemi, BuildRight
:- Exchange hashpartitioning(id#15, 200)
: +- LocalTableScan [id#15, token#16]
+- Exchange hashpartitioning(id#20, 200)
  +- LocalTableScan [id#20]
```

Note

`ShuffledHashJoinExec` operator is chosen in [JoinSelection](#) execution planning strategy.

Table 1. ShuffledHashJoinExec's SQL Metrics

Name	Description
<code>buildDataSize</code>	data size of build side
<code>buildTime</code>	time to build hash map
<code>numOutputRows</code>	number of output rows

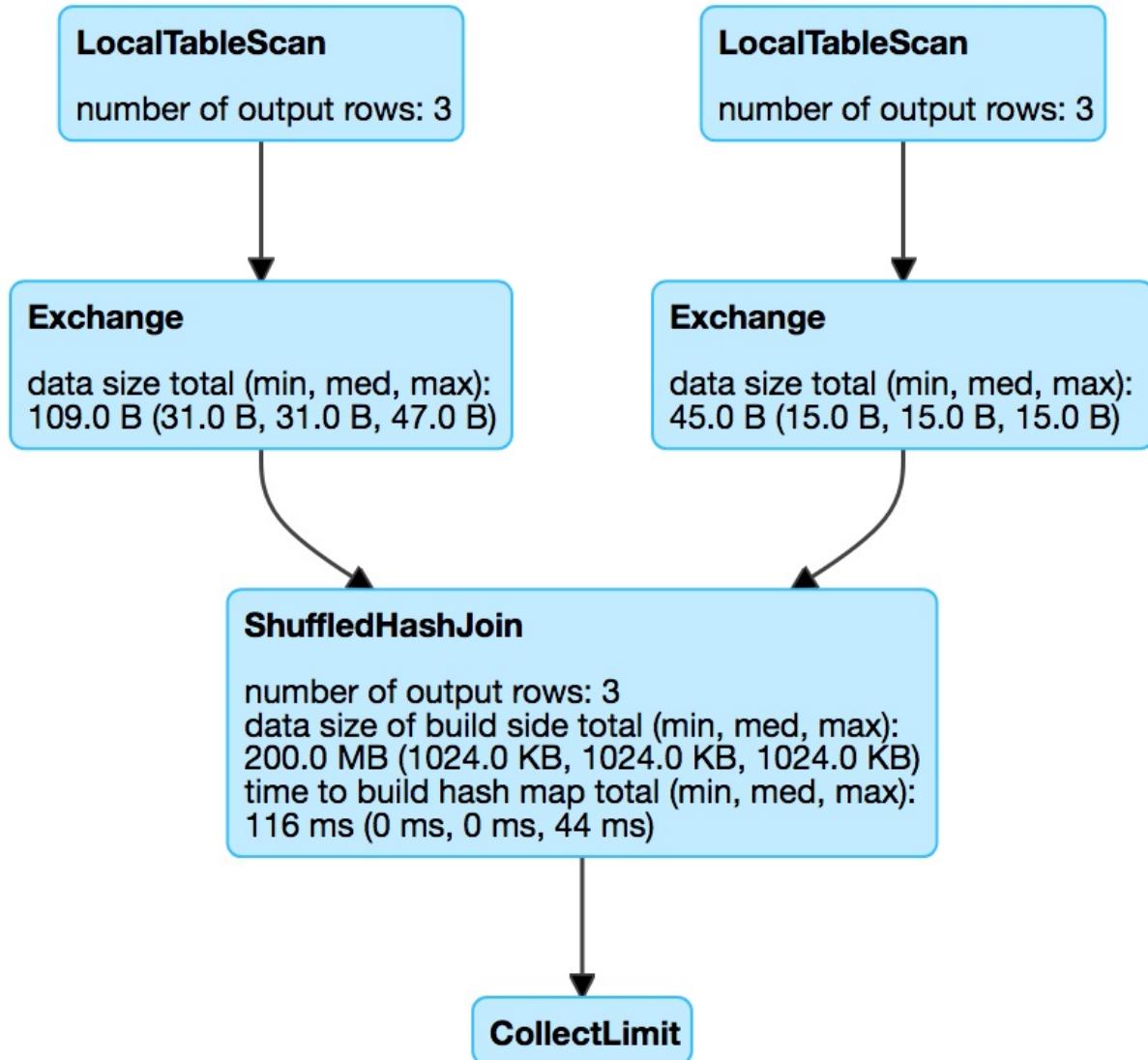


Figure 1. ShuffledHashJoinExec in web UI (Details for Query)

`ShuffledHashJoinExec` requires that the partition requirement for the two child physical operators is `ClusteredDistribution`.

doExecute Method

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

buildHashedRelation Internal Method

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

Creating ShuffledHashJoinExec Instance

`ShuffledHashJoinExec` takes the following when created:

- Left join key [expressions](#)
- Right join key [expressions](#)
- [Join type](#)
- [BuildSide](#)
- Optional join condition [expression](#)
- Left [physical operator](#)
- Right [physical operator](#)

SortMergeJoinExec Binary Physical Operator

`SortMergeJoinExec` is a [binary physical operator](#) that supports code generation (aka *whole-stage codegen*).

`SortMergeJoinExec` is [created](#) exclusively for joins with [left join keys](#) orderable, i.e. that can be ordered (sorted).

	<p>A join key is orderable when is of one of the following data types:</p> <ul style="list-style-type: none">• <code>NullType</code>• AtomicType (that represents all the available types except <code>NullType</code>, <code>StructType</code>, <code>ArrayType</code>, <code>UserDefinedType</code>, <code>MapType</code>, and <code>ObjectType</code>)• <code>StructType</code> with orderable fields• <code>ArrayType</code> of orderable type• <code>UserDefinedType</code> of orderable type <p>Therefore, a join key is not orderable when is of the following data type:</p> <ul style="list-style-type: none">• <code>MapType</code>• <code>ObjectType</code>
--	---

```

// Start spark-shell with broadcast hash join disabled, i.e. spark.sql.autoBroadcastJoinThreshold=-1
// ./bin/spark-shell -c spark.sql.autoBroadcastJoinThreshold=-1
// Mind the data types so ShuffledHashJoinExec is not selected
val dataset = Seq(
  (0, "playing"),
  (1, "with"),
  (2, "SortMergeJoinExec")
).toDF("id", "token")

// all data types are orderable
scala> dataset.printSchema
root
|-- id: integer (nullable = false)
|-- token: string (nullable = true)

scala> spark.conf.get("spark.sql.autoBroadcastJoinThreshold")
res0: String = -1

val q = dataset.join(tokens, Seq("id"), "inner")
scala> q.explain
== Physical Plan ==
*Project [id#27, token#28, token#6]
+- *SortMergeJoin [id#27], [id#5], Inner
  :- *Sort [id#27 ASC NULLS FIRST], false, 0
  :  +- Exchange hashpartitioning(id#27, 200)
  :    +- LocalTableScan [id#27, token#28]
  +- *Sort [id#5 ASC NULLS FIRST], false, 0
     +- ReusedExchange [id#5, token#6], Exchange hashpartitioning(id#27, 200)

```

Table 1. SortMergeJoinExec's SQLMetrics

Name	Description
numOutputRows	number of output rows

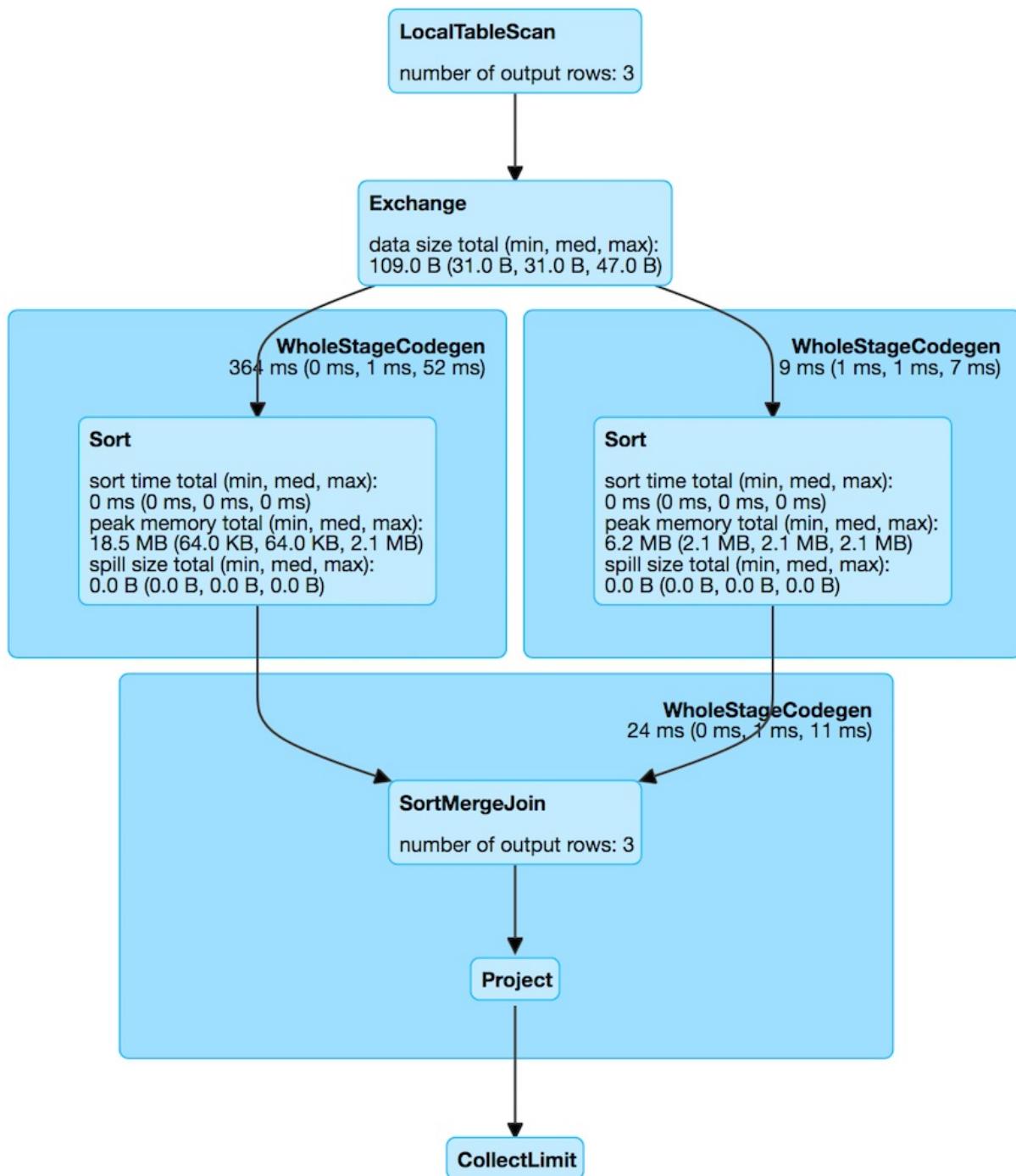


Figure 1. SortMergeJoinExec in web UI (Details for Query)

Note

The prefix for variable names for `SortMergeJoinExec` operators in `CodegenSupport`-generated code is **smj**.

```

scala> q.queryExecution.debug.codegen
Found 3 WholeStageCodegen subtrees.
== Subtree 1 / 3 ==
*Project [id#5, token#6, token#11]
+- *SortMergeJoin [id#5], [id#10], Inner
  :- *Sort [id#5 ASC NULLS FIRST], false, 0
    : +- Exchange hashpartitioning(id#5, 200)
      :   +- LocalTableScan [id#5, token#6]
    +- *Sort [id#10 ASC NULLS FIRST], false, 0
      +- ReusedExchange [id#10, token#11], Exchange hashpartitioning(id#5, 200)

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;
/* 007 */   private scala.collection.Iterator[] inputs;
/* 008 */   private scala.collection.Iterator smj_leftInput;
/* 009 */   private scala.collection.Iterator smj_rightInput;
/* 010 */   private InternalRow smj_leftRow;
/* 011 */   private InternalRow smj_rightRow;
/* 012 */   private int smj_value2;
/* 013 */   private org.apache.spark.sql.execution.ExternalAppendOnlyUnsafeRowArray smj_matches;
/* 014 */   private int smj_value3;
/* 015 */   private int smj_value4;
/* 016 */   private UTF8String smj_value5;
/* 017 */   private boolean smj_isNull2;
/* 018 */   private org.apache.spark.sql.execution.metric.SQLMetric smj_numOutputRows;
/* 019 */   private UnsafeRow smj_result;
/* 020 */   private org.apache.spark.sql.catalyst.expressions.codegen.BufferHolder smj_holder;
/* 021 */   private org.apache.spark.sql.catalyst.expressions.codegen.UnsafeRowWriter smj_rowWriter;
...

```

Note

`SortMergeJoinExec` operator is chosen in [JoinSelection](#) execution planning strategy (after [BroadcastHashJoinExec](#) and [ShuffledHashJoinExec](#) physical join operators have not met the requirements).

Creating SortMergeJoinExec Instance

`SortMergeJoinExec` takes the following when created:

- Left join key [expressions](#)
- Right join key [expressions](#)

- Join type
- Optional join condition [expression](#)
- Left [physical operator](#)
- Right [physical operator](#)

InputAdapter Unary Physical Operator

`InputAdapter` is a [unary physical operator](#) (with [CodegenSupport](#)) that is an adapter to generate code for the single child operator that does not support [whole-stage code generation](#) but participates in such code generation for a structured query.

`InputAdapter` is created exclusively when `collapseCodegenStages` inserts a [WholeStageCodegenExec operator](#) into a physical plan.

`InputAdapter` removes the star from a stringified tree representation of a physical plan (that [WholeStageCodegenExec](#) adds), e.g. for [explain](#) operator.

```
// explode (that uses Generate operator) does not support codegen
val ids = Seq(Seq(0,1,2,3)).toDF("ids").select(explode($"ids") as "id")
val query = spark.range(1).join(ids, "id")
scala> query.explain
== Physical Plan ==
*Project [id#150L]
+- *BroadcastHashJoin [id#150L], [cast(id#147 as bigint)], Inner, BuildRight
  :- *Range (0, 1, step=1, splits=8)
  +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, false] as
  bigint)))
    +- Generate explode(ids#143), false, false, [id#147]
      +- LocalTableScan [ids#143]
```

doProduce Method

```
doProduce(ctx: CodegenContext): String
```

`doProduce` generates a Java source code that consumes [internal row](#) of a single input `RDD` one at a time (in a `while` loop).

Note	<code>doProduce</code> supports one input RDD only (that the single child generates when executed).
------	--

Internally, `doProduce` generates two terms `input` and `row` and uses the code from [consume](#) code generator.

Note	<code>doProduce</code> is a part of CodegenSupport Contract to generate a Java source code.
------	---

WindowExec Physical Operator

`WindowExec` is a [unary physical operator](#) with a collection of [named expressions](#) (for windows), a collection of [expressions](#) (for partitions), a collection of `sortOrder` (for sorting) and a `child` [physical operator](#).

The `output` of `WindowExec` are the `output` of `child` physical plan and windows.

```
import org.apache.spark.sql.expressions.Window
val orderId = Window.orderBy('id)

val dataset = spark.range(5).withColumn("group", 'id % 3)

scala> dataset.select('*', rank over orderId as "rank").show
+---+-----+----+
| id|group|rank|
+---+-----+----+
|  0|    0|    1|
|  1|    1|    2|
|  2|    2|    3|
|  3|    0|    4|
|  4|    1|    5|
+---+-----+----+
```

When executed (i.e. `show`) with no partitions, `WindowExec` prints out the following `WARN` message to the logs:

```
WARN WindowExec: No Partition Defined for Window operation! Moving all data to a single partition, this can cause serious performance degradation.
```

Tip Enable `WARN` logging level for `org.apache.spark.sql.execution.WindowExec` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.sql.execution.WindowExec=WARN
```

Refer to [Logging](#).

Caution

[FIXME](#) Describe `clusteredDistribution`

When the number of rows exceeds `4096`, `WindowExec` creates `UnsafeExternalSorter`.

Caution	FIXME What's UnsafeExternalSorter ?
---------	---

WholeStageCodegenExec Unary Operator with Java Code Generation

`WholeStageCodegenExec` is a [unary physical operator](#) that [supports code generation](#) for a **codegened pipeline** of a single physical operator.

`WholeStageCodegenExec` is created when [CollapseCodegenStages](#) physical preparation rule transforms a [physical plan](#) and `spark.sql.codegen.wholeStage` is enabled.

Note	<code>spark.sql.codegen.wholeStage</code> property is enabled by default.
------	---

`WholeStageCodegenExec` is marked with `*` prefix in the tree output of a physical plan.

Note	Use executedPlan phase of a query execution to see <code>WholeStageCodegenExec</code> in the plan.
------	--

```
val q = spark.range(9)
val plan = q.queryExecution.executedPlan
scala> println(plan.numberedTreeString)
00 *Range (0, 9, step=1, splits=8)
```

Table 1. WholeStageCodegenExec SQLMetrics (in alphabetical order)

Name	Description
<code>pipelineTime</code>	duration

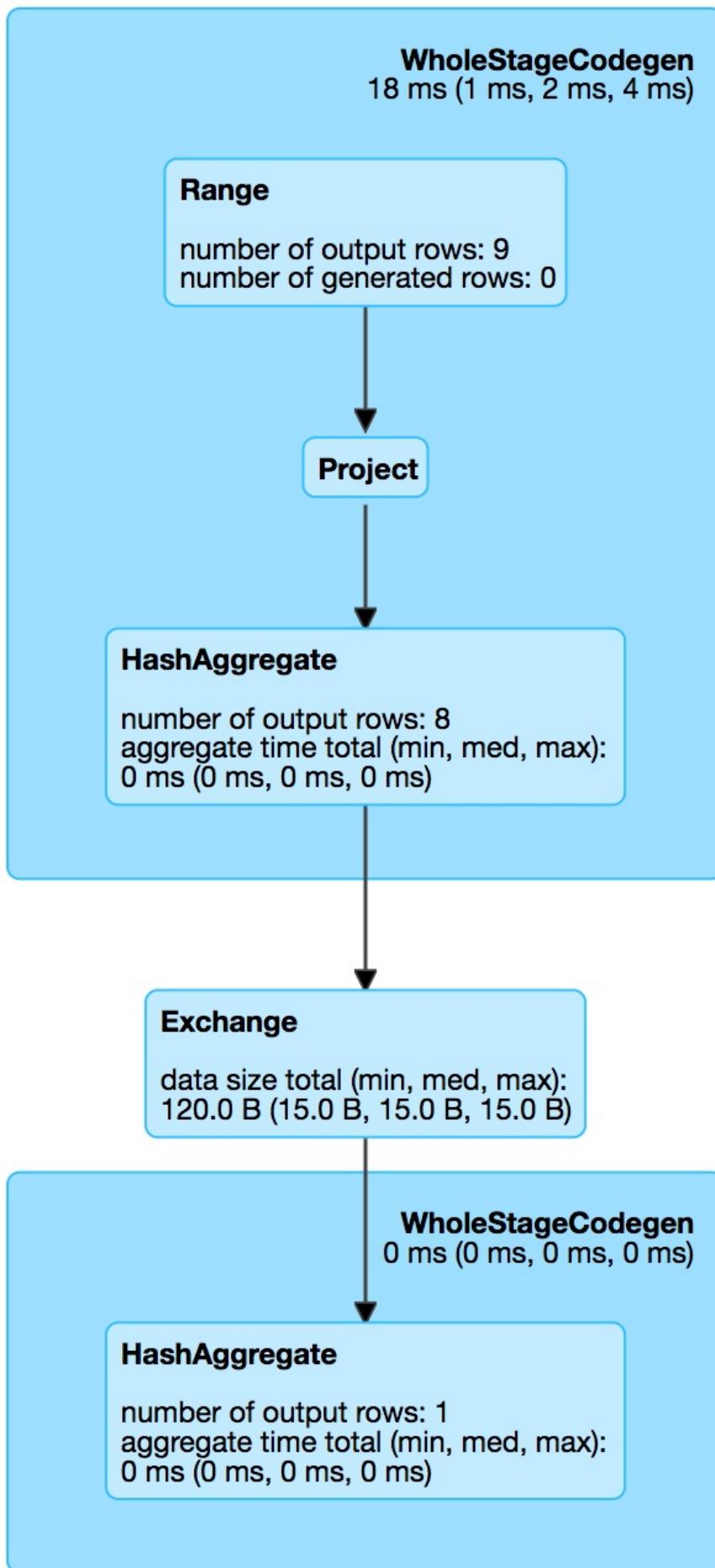


Figure 1. WholeStageCodegenExec in web UI (Details for Query)

Tip

Use `explain` operator to know the physical plan of a query and find out whether or not `WholeStageCodegen` is in use.

```
val q = spark.range(10).where('id === 4)
// Note the stars in the output that are for codegened operators
scala> q.explain
== Physical Plan ==
*Filter (id#0L = 4)
+- *Range (0, 10, step=1, splits=8)
```

Tip

Consider using [Debugging Query Execution facility](#) to deep dive into whole stage codegen.

```
scala> q.queryExecution.debug.codegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Filter (id#5L = 4)
+- *Range (0, 10, step=1, splits=8)
```

Note

[Physical plans](#) that support code generation extend [CodeGenSupport](#).

Enable `DEBUG` logging level for `org.apache.spark.sql.execution.WholeStageCodegenExec` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.sql.execution.WholeStageCodegenExec=DEBUG
```

Refer to [Logging](#).

doConsume Method

Caution

FIXME

doExecute Method

```
doExecute(): RDD[InternalRow]
```

`doExecute` generates the Java code that is [compiled](#) right afterwards.

If compilation fails and `spark.sql.codegen.fallback` is enabled, you should see the following WARN message in the logs and `doExecute` returns the result of executing the child physical operator.

```
WARN WholeStageCodegenExec: Whole-stage codegen disabled for this plan:  
[tree]
```

If however code generation and compilation went well, `doExecute` branches off per the number of input RDDs.

Note	<code>doExecute</code> only supports up to two input RDDs.
------	--

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

Note	<code>doExecute</code> is a part of SparkPlan Contract .
------	--

Generating Java Code for Child Subtree — `doCodeGen` Method

```
doCodeGen(): (CodegenContext, CodeAndComment)
```

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

You should see the following DEBUG message in the logs:

```
DEBUG WholeStageCodegenExec:  
[cleanedSource]
```

Note	<code>doCodeGen</code> is used when <code>WholeStageCodegenExec</code> <code>doExecute</code> (and for <code>debugCodegen</code>).
------	---

SparkPlanner — Query Planner with no Hive Support

`SparkPlanner` is a concrete Catalyst query planner that converts a logical plan to one or more physical plans using execution planning strategies with support for extra strategies (by means of `ExperimentalMethods`) and `extraPlanningStrategies`.

Note

`SparkPlanner` is expected to plan (aka `generate`) at least one physical plan for a logical plan.

`SparkPlanner` is available as planner of a `SessionState`.

```
val spark: SparkSession = ...
spark.sessionState.planner
```

Table 1. SparkPlanner's Execution Planning Strategies (in execution order)

SparkStrategy	Description
<code>ExperimentalMethods</code> 's <code>extraStrategies</code>	
<code>extraPlanningStrategies</code>	Extension point for extra planning strategies
<code>FileSourceStrategy</code>	
<code>DataSourceStrategy</code>	
<code>SpecialLimits</code>	
<code>Aggregation</code>	
<code>JoinSelection</code>	
<code>InMemoryScans</code>	
<code>BasicOperators</code>	

Note

`SparkPlanner` extends `SparkStrategies` abstract class.

Creating SparkPlanner Instance

`SparkPlanner` takes the following when created:

- `SparkContext`
- `SQLConf`
- `ExperimentalMethods`

Note

`SparkPlanner` is created in:

- `BaseSessionStateBuilder`
- `HiveSessionStateBuilder`
- Structured Streaming's `IncrementalExecution`

Extension Point for Extra Planning Strategies

— `extraPlanningStrategies` Method

```
extraPlanningStrategies: Seq[Strategy] = Nil
```

`extraPlanningStrategies` is an extension point to register extra [planning strategies](#) with the query planner.

Note

`extraPlanningStrategies` are executed after [extraStrategies](#).

Note

`extraPlanningStrategies` is used when `SparkPlanner` is requested for planning [strategies](#).

`extraPlanningStrategies` is overriden in the `SessionState` builders — [BaseSessionStateBuilder](#) and [HiveSessionStateBuilder](#).

Collecting PlanLater Physical Operators

— `collectPlaceholders` Method

```
collectPlaceholders(plan: SparkPlan): Seq[(SparkPlan, LogicalPlan)]
```

`collectPlaceholders` collects all [PlanLater](#) physical operators in the `plan` [physical plan](#).

Note

`collectPlaceholders` is a part of [QueryPlanner Contract](#).

Pruning "Bad" Physical Plans — `prunePlans` Method

```
prunePlans(plans: Iterator[SparkPlan]): Iterator[SparkPlan]
```

`prunePlans` gives the input `plans` [physical plans](#) back (i.e. with no changes).

Note

`prunePlans` is a part of [QueryPlanner Contract](#) to remove somehow "bad" plans.

pruneFilterProject Method

Caution

[FIXME](#)

Note

`pruneFilterProject` is a helper method used exclusively in [InMemoryScans](#) and [HiveTableScans](#) execution planning strategies.

SparkStrategy — Base for Execution Planning Strategies

`SparkStrategy` is a Catalyst `GenericStrategy` that converts a `logical plan` into zero or more `physical plans`.

`SparkStrategy` marks `logical plans` (i.e. `LogicalPlan`) to be planned later (by some other `SparkStrategy` or after other `SparkStrategy` strategies have finished) using `PlanLater` physical operator.

```
planLater(plan: LogicalPlan): SparkPlan = PlanLater(plan)
```

	<p><code>SparkStrategy</code> is used as <code>Strategy</code> type alias (aka <i>type synonym</i>) in Spark's code base that is defined in <code>org.apache.spark.sql</code> package object, i.e.</p>
Note	<pre>type Strategy = SparkStrategy</pre>

PlanLater Physical Operator

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

SparkStrategies — Container of Execution Planning Strategies

`SparkStrategies` is an abstract Catalyst [query planner](#) that *merely* serves as a "container" (or a namespace) of the concrete [execution planning strategies](#) (for [SparkPlanner](#)):

- [Aggregation](#)
- [BasicOperators](#)
- `FlatMapGroupsWithStateStrategy`
- [InMemoryScans](#)
- [JoinSelection](#)
- `SpecialLimits`
- `StatefulAggregationStrategy`
- `StreamingDeduplicationStrategy`
- `StreamingRelationStrategy`

`SparkStrategies` has a single lazily-instantiated `singleRowRdd` value that is an `RDD` of [InternalRow](#) that [BasicOperators](#) execution planning strategy uses when [converting](#) `OneRowRelation` to `RDDScanExec` physical operator.

Note	<code>OneRowRelation</code> logical operator represents SQL's SELECT clause without FROM clause or EXPLAIN DESCRIBE TABLE .
------	---

Aggregation Execution Planning Strategy

Aggregation is an [execution planning strategy](#) (of [SparkPlanner](#)) that...[FIXME](#)

```
val spark: SparkSession = ...
// structured query with count aggregate function
val q = spark.range(5).
  groupBy($"id" % 2 as "group").
  agg(count("id") as "count")

import q.queryExecution.optimizedPlan
scala> println(optimizedPlan.numberedTreeString)
00 Aggregate [(id#38L % 2)], [(id#38L % 2) AS group#41L, count(1) AS count#46L]
01 +- Range (0, 5, step=1, splits=Some(8))

import spark.sessionState.planner.Aggregation
scala> println(Aggregation.apply(optimizedPlan).head.numberedTreeString)
00 HashAggregate(keys=[(id#38L % 2)#66L], functions=[count(1)], output=[group#41L, count#46L])
01 +- HashAggregate(keys=[(id#38L % 2) AS (id#38L % 2)#66L], functions=[partial_count(1)], output=[(id#38L % 2)#66L, count#59L])
02   +- PlanLater Range (0, 5, step=1, splits=Some(8))
```

Executing Planning Strategy — `apply` Method

```
apply(plan: LogicalPlan): Seq[SparkPlan]
```

`apply` ...[FIXME](#)

Note	<code>apply</code> is a part of GenericStrategy Contract to execute a planning strategy.
------	--

Destructuring Logical Plan — `PhysicalAggregation.unapply` Method

```
unapply(a: Any): Option[ReturnType]
```

`unapply` destructures a [logical plan](#), i.e. breaks a [LogicalPlan](#) into pieces.

Caution	FIXME What pieces?
---------	------------------------------------

Note

`ReturnType` is a type alias (aka *type synonym*) for a 4-element tuple with grouping aggregate and result [expressions](#), and child [logical operator](#).

```
type ReturnType = (Seq[NamedExpression], Seq[AggregateExpression], Seq[NamedExpression], LogicalOp)
```

Note

`PhysicalAggregation` is a Scala [extractor object](#) with a single [unapply](#) method.

AggUtils.planAggregateWithOneDistinct Method

Creating Aggregate Physical Operator Per Aggregate Expressions — AggUtils.createAggregate Internal Method

```
createAggregate(  
    requiredChildDistributionExpressions: Option[Seq[Expression]] = None,  
    groupingExpressions: Seq[NamedExpression] = Nil,  
    aggregateExpressions: Seq[AggregateExpression] = Nil,  
    aggregateAttributes: Seq[Attribute] = Nil,  
    initialInputBufferOffset: Int = 0,  
    resultExpressions: Seq[NamedExpression] = Nil,  
    child: SparkPlan): SparkPlan
```

`createAggregate` creates a [physical operator](#) given the input `aggregateExpressions`.

Table 1. createAggregate and Selecting Physical Operator (in execution order)

Operator	Conditions (for selection)
HashAggregateExec	<p>Builds a schema from <code>aggBufferAttributes</code> (of every <code>AggregateFunction</code>) of every <code>AggregateExpression</code> (in the input <code>aggregateExpressions</code>) and checks for any non-mutable fields (that <i>cannot</i> be updated in place in <code>UnsafeRows</code>). When not found, <code>HashAggregateExec</code> is selected.</p> <p>Mutable field types include:</p> <ul style="list-style-type: none"> • <code>NullType</code> • <code>BooleanType</code> • <code>ByteType</code> • <code>ShortType</code> • <code>IntegerType</code> • <code>LongType</code> • <code>FloatType</code> • <code>DoubleType</code> • <code>DateType</code> • <code>TimestampType</code> • <code>DecimalType</code>
ObjectHashAggregateExec	When <code>spark.sql.execution.useObjectHashAggregateExec</code> internal flag enabled (it is by default) and <code>aggregateExpressions</code> contains at least one <code>TypedImperativeAggregate</code> .
SortAggregateExec	When all the above conditions fail.

Note

`createAggregate` is used in `AggUtils.planAggregateWithoutDistinct`, `AggUtils.planAggregateWithOneDistinct` and `planStreamingAggregation` (for Structured Streaming's `statefulAggregationStrategy`).

AggUtils.planAggregateWithoutDistinct Method

```
planAggregateWithoutDistinct(
  groupingExpressions: Seq[NamedExpression],
  aggregateExpressions: Seq[AggregateExpression],
  resultExpressions: Seq[NamedExpression],
  child: SparkPlan): Seq[SparkPlan]
```

`planAggregateWithoutDistinct` is a two-step physical operator generator.

`planAggregateWithoutDistinct` first creates an aggregate physical operator with `aggregateExpressions` in `Partial` mode.

In the end, `planAggregateWithoutDistinct` creates another aggregate physical operator with `aggregateExpressions` in `Final` mode and the first aggregate operator as the child.

Note

`planAggregateWithoutDistinct` is used exclusively when Aggregation execution planning strategy is executed (with no `AggregateExpressions` being distinct).

BasicOperators Execution Planning Strategy

`BasicOperators` is an [execution planning strategy](#) (of `SparkPlanner`) that in general does simple [conversions](#) from [logical operators](#) to their [physical counterparts](#).

Table 1. BasicOperators' Logical to Physical Operator Conversions

Logical Operator	Physical Operator
<code>RunnableCommand</code>	<code>ExecutedCommandExec</code>
<code>MemoryPlan</code>	<code>LocalTableScanExec</code>
<code>DeserializeToObject</code>	<code>DeserializeToObjectExec</code>
<code>SerializeFromObject</code>	<code>SerializeFromObjectExec</code>
<code>MapPartitions</code>	<code>MapPartitionsExec</code>
<code>MapElements</code>	<code>MapElementsExec</code>
<code>AppendColumns</code>	<code>AppendColumnsExec</code>
<code>AppendColumnsWithObject</code>	<code>AppendColumnsWithObjectExec</code>
<code>MapGroups</code>	<code>MapGroupsExec</code>
<code>CoGroup</code>	<code>CoGroupExec</code>
<code>Repartition</code> (with shuffle enabled)	<code>ShuffleExchange</code>
<code>Repartition</code>	<code>CoalesceExec</code>
<code>SortPartitions</code>	<code>SortExec</code>
<code>Sort</code>	<code>SortExec</code>
<code>Project</code>	<code>ProjectExec</code>
<code>Filter</code>	<code>FilterExec</code>
<code>TypedFilter</code>	<code>FilterExec</code>
<code>Expand</code>	<code>ExpandExec</code>
<code>Window</code>	<code>WindowExec</code>

Sample	SampleExec
LocalRelation	LocalTableScanExec
LocalLimit	LocalLimitExec
GlobalLimit	GlobalLimitExec
Union	UnionExec
Generate	GenerateExec
OneRowRelation	RDDScanExec
Range	RangeExec
RepartitionByExpression	ShuffleExchange
ExternalRDD	ExternalRDDScanExec
LogicalRDD	RDDScanExec
BroadcastHint	PlanLater

Tip	Confirm the operator mapping in the source code of BasicOperators .
-----	---

Note	BasicOperators expects that Distinct, Intersect, and Except logical operators are not used in a logical plan and throws a IllegalStateException if not.
------	---

DataSourceStrategy Execution Planning Strategy

`DataSourceStrategy` is an [execution planning strategy](#) (of [SparkPlanner](#)) that converts `LogicalRelation` logical operator to `RowDataSourceScanExec` physical operator.

Table 1. `DataSourceStrategy`'s Selection Requirements (in execution order)

Logical Operator	Selection Requirements
<code>LogicalRelation</code> with <code>CatalystScan</code> relation	Uses pruneFilterProjectRaw <code>CatalystScan</code> does not seem to be used in Spark SQL.
<code>LogicalRelation</code> with <code>PrunedFilteredScan</code> relation	Uses pruneFilterProjectRaw Matches <code>JDBCRelation</code> exclusively (as it is <code>PrunedFilteredScan</code>)
<code>LogicalRelation</code> with <code>PrunedScan</code> relation	Uses pruneFilterProjectRaw <code>PrunedScan</code> does not seem to be used in Spark SQL.
<code>LogicalRelation</code> with <code>TableScan</code> relation	Matches <code>KafkaRelation</code> exclusively (as it is <code>TableScan</code>)

Note	<code>DataSourceStrategy</code> uses PhysicalOperation to destructure a logical plan .
------	--

Creating `RowDataSourceScanExec` (under `FilterExec` and `ProjectExec`) — `pruneFilterProjectRaw` Internal Method

```
pruneFilterProjectRaw(
  relation: LogicalRelation,
  projects: Seq[NamedExpression],
  filterPredicates: Seq[Expression],
  scanBuilder: (Seq[Attribute], Seq[Expression], Seq[Filter]) => RDD[InternalRow]): SparkPlan
```

`pruneFilterProjectRaw` creates a `RowDataSourceScanExec` (possibly as a child of `FilterExec` that in turn could be a child of `ProjectExec`).

Note	<code>pruneFilterProjectRaw</code> is used when <code>DataSourceStrategy</code> executes (and selects <code>RowDataSourceScanExec</code> per <code>LogicalRelation</code>).
------	--

FileSourceStrategy Execution Planning Strategy

`FileSourceStrategy` is an [execution planning strategy](#) (of [SparkPlanner](#)) that destructures and then optimizes a [LogicalPlan](#).

Tip

Enable `INFO` logging level for `org.apache.spark.sql.execution.datasources.FileSourceStrategy` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.sql.execution.datasources.FileSourceStrategy=INFO
```

Refer to [Logging](#).

Caution**FIXME**

PhysicalOperation

`PhysicalOperation` is a pattern used to destructure a [LogicalPlan](#) object into a tuple.

```
(Seq[NamedExpression], Seq[Expression], LogicalPlan)
```

The following idiom is often used in `Strategy` implementations (e.g. [HiveTableScans](#), [InMemoryScans](#), [DataSourceStrategy](#), [FileSourceStrategy](#)):

```
def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
  case PhysicalOperation(projections, predicates, plan) =>
    // do something
  case _ => Nil
}
```

Whenever used to pattern match to a `LogicalPlan`, `PhysicalOperation`'s `unapply` is called.

```
unapply(plan: LogicalPlan): Option[ReturnType]
```

`unapply` uses [collectProjectsAndFilters](#) method that recursively destructures the input `LogicalPlan`.

Note

`unapply` is almost `collectProjectsAndFilters` method itself (with some manipulations of the return value).

collectProjectsAndFilters Method

```
collectProjectsAndFilters(plan: LogicalPlan):  
  (Option[Seq[NamedExpression]], Seq[Expression], LogicalPlan, Map[Attribute, Expression])
```

`collectProjectsAndFilters` is a pattern used to destructure a `LogicalPlan` that can be `Project`, `Filter` or `BroadcastHint`. Any other `LogicalPlan` give an *all-empty* response.

InMemoryScans Execution Planning Strategy

InMemoryScans is an execution planning strategy (of SparkPlanner) that translates InMemoryRelation logical operator for cached query plans to a pruned physical plan with InMemoryTableScanExec physical operator.

```
val spark: SparkSession = ...
// query uses InMemoryRelation logical operator
val q = spark.range(5).cache
val plan = q.queryExecution.optimizedPlan
scala> println(plan.numberedTreeString)
00 InMemoryRelation [id#208L], true, 10000, StorageLevel(disk, memory, deserialized, 1
replicas)
01   +- *Range (0, 5, step=1, splits=8)

// InMemoryScans is an internal class of SparkStrategies
import spark.sessionState.planner.InMemoryScans
val physicalPlan = InMemoryScans.apply(plan).head
scala> println(physicalPlan.numberedTreeString)
00 InMemoryTableScan [id#208L]
01   +- InMemoryRelation [id#208L], true, 10000, StorageLevel(disk, memory, deseriali
zed, 1 replicas)
02     +- *Range (0, 5, step=1, splits=8)
```

JoinSelection Execution Planning Strategy

`JoinSelection` is an execution planning strategy (of `SparkPlanner`) that translates `Join` logical operator to one of the available join physical operators per `join physical operator selection requirements`.

Table 1. Join Physical Operator Selection Requirements (in execution order)

Physical Join Operator	Selection Requirements
<code>BroadcastHashJoinExec</code>	<p>There are joining keys and one of the following holds:</p> <ul style="list-style-type: none"> • <code>canBuildRight</code> and right join side <code>can be broadcast</code> • <code>canBuildLeft</code> and left join side <code>can be broadcast</code>
<code>ShuffledHashJoinExec</code>	<p>There are joining keys and one of the following holds:</p> <ul style="list-style-type: none"> • <code>spark.sql.join.preferSortMergeJoin</code> is disabled, <code>canBuildRight</code>, <code>canBuildLocalHashMap</code> for right join side and finally right join side is <code>much smaller</code> than left side • <code>spark.sql.join.preferSortMergeJoin</code> is disabled, <code>canBuildLeft</code>, <code>canBuildLocalHashMap</code> for left join side and finally left join side is <code>much smaller</code> than right • Left join keys are <code>not orderable</code>
<code>SortMergeJoinExec</code>	Left join keys <code>orderable</code>
<code>BroadcastNestedLoopJoinExec</code>	<p>There are no joining keys and one of the following holds:</p> <ul style="list-style-type: none"> • <code>canBuildRight</code> and right join side <code>can be broadcast</code> • <code>canBuildLeft</code> and left join side <code>can be broadcast</code>
<code>cartesianProductExec</code>	There are no joining keys and <code>join type</code> is <code>INNER</code> or <code>CROSS</code>
<code>BroadcastNestedLoopJoinExec</code>	Default when no other have matched

Note

`JoinSelection` uses `ExtractEquiJoinKeys` to destructure a `Join` logical plan.

ExtractEquiJoinKeys

`ExtractEquiJoinKeys` is a pattern used to destructure a `Join` logical operator into a tuple for [join physical operator selection](#).

```
(JoinType, Seq[Expression], Seq[Expression], Option[Expression], LogicalPlan, LogicalPlan)
```

Is Left-Side Plan At Least 3 Times Smaller Than Right-Side Plan?— `muchSmaller` Internal Condition

```
muchSmaller(a: LogicalPlan, b: LogicalPlan): Boolean
```

`muchSmaller` condition holds when plan `a` is at least 3 times smaller than plan `b`.

Internally, `muchSmaller` [calculates the estimated statistics for the input logical plans](#) and compares their physical size in bytes (`sizeInBytes`).

Note	<code>muchSmaller</code> is used exclusively when <code>JoinSelection</code> checks join selection requirements for <code>ShuffledHashJoinExec</code> physical operator.
------	--

canBuildLocalHashMap Internal Condition

```
canBuildLocalHashMap(plan: LogicalPlan): Boolean
```

`canBuildLocalHashMap` condition holds for the logical `plan` whose single partition is small enough to build a hash table (i.e. `spark.sql.autoBroadcastJoinThreshold` multiplied by `spark.sql.shuffle.partitions`).

Internally, `canBuildLocalHashMap` [calculates the estimated statistics for the input logical plans](#) and takes the size in bytes (`sizeInBytes`).

Note	<code>canBuildLocalHashMap</code> is used when <code>JoinSelection</code> checks join selection requirements for <code>ShuffledHashJoinExec</code> physical operator.
------	---

canBuildLeft Internal Condition

```
canBuildLeft(joinType: JoinType): Boolean
```

`canBuildLeft` condition holds for CROSS, INNER and RIGHT OUTER join types.

Otherwise, `canBuildLeft` is `false`.

Note

`canBuildLeft` is used when `JoinSelection` checks join selection requirements for `BroadcastHashJoinExec`, `ShuffledHashJoinExec` or `BroadcastNestedLoopJoinExec` physical operators.

canBuildRight Internal Condition

```
canBuildRight(joinType: JoinType): Boolean
```

`canBuildRight` condition holds for [joins](#) that are:

- CROSS, INNER, LEFT ANTI, LEFT OUTER, LEFT SEMI or Existence

Otherwise, `canBuildRight` is `false`.

Note

`canBuildRight` is used when `JoinSelection` checks join selection requirements for `BroadcastHashJoinExec`, `ShuffledHashJoinExec` or `BroadcastNestedLoopJoinExec` physical operators.

Can Logical Plan Be Broadcast? — canBroadcast Internal Condition

```
canBroadcast(plan: LogicalPlan): Boolean
```

`canBroadcast` condition holds for [logical operators](#) with statistics that can be broadcast and of non-negative size up to [spark.sql.autoBroadcastJoinThreshold](#).

Physical Plan Preparations Rules

Note

For the time being, this page **Physical Plan Preparations Rules** serves mainly as a placeholder for the menu layout so the physical plan preparation rules show up nicely in the menu.
The page is *merely* a compilation of what you may have found on [QueryExecution](#) page.

[QueryExecution](#) has multiple phases of query execution in a so-called **Structured Query Execution Pipeline**.

Among the phases is the [executedPlan](#) phase that is one of the last phases in a query execution which is the result of [executing physical preparation rules](#) on a physical plan of a structured query.

Physical preparation rules are [rules](#) that transform a [physical plan](#) and produce a physical plan (i.e. `Rule[SparkPlan]`).

[QueryExecution](#) defines [preparations](#) batch of rules that are executed on a physical plan sequentially and include the following:

1. `ExtractPythonUDFs`
2. `PlanSubqueries`
3. `EnsureRequirements`
4. `CollapseCodegenStages`
5. `ReuseExchange`
6. `ReuseSubquery`

CollapseCodegenStages Physical Preparation Rule — Collapsing Physical Operators for Whole-Stage CodeGen

`CollapseCodegenStages` is a physical preparation rule that collapses physical operators for Java code generation (as part of Whole-Stage CodeGen).

Note

You can disable `collapseCodegenStages` (and so whole-stage codegen) by turning `spark.sql.codegen.wholeStage` internal property off.

`spark.sql.codegen.wholeStage` property is enabled by default.

```
import org.apache.spark.sql.internal.SQLConf.WHOLESTAGE_CODEGEN_ENABLED
scala> spark.conf.get(WHOLESTAGE_CODEGEN_ENABLED)
res0: String = true
```

Use `SQLConf.wholeStageEnabled` method to access the current value.

```
scala> spark.sessionState.conf.wholeStageEnabled
res1: Boolean = true
```

`CollapseCodegenStages` acts only on physical operators with `CodegenSupport` for which Java code can really be generated.

`CollapseCodegenStages` takes a `SQLConf` when created.

Tip

Import `CollapseCodegenStages` and apply the rule directly to a physical plan to learn how the rule works.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...
val query = spark.range(2).join(spark.range(2), "id")

// the final result (after CollapseCodegenStages among other rules)
scala> query.explain
== Physical Plan ==
*Project [id#9L]
+- *BroadcastHashJoin [id#9L], [id#12L], Inner, BuildRight
  :- *Range (0, 2, step=1, splits=8)
  +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
    +- *Range (0, 2, step=1, splits=8)

val plan = query.queryExecution.sparkPlan

// wholeStageEnabled is enabled
```

```
scala> println(spark.sessionState.conf.wholeStageEnabled)
true

import org.apache.spark.sql.execution.CollapseCodegenStages
val ccs = CollapseCodegenStages(conf = spark.sessionState.conf)

scala> ccs.ruleName
res0: String = org.apache.spark.sql.execution.CollapseCodegenStages

// Before CollapseCodegenStages
scala> println(plan.numberedTreeString)
00 Project [id#9L]
01 +- BroadcastHashJoin [id#9L], [id#12L], Inner, BuildRight
02   :- Range (0, 2, step=1, splits=8)
03   +- Range (0, 2, step=1, splits=8)

// After CollapseCodegenStages
// Note the star
val executedPlan = ccs.apply(plan)
scala> println(executedPlan.numberedTreeString)
00 *Project [id#9L]
01 +- *BroadcastHashJoin [id#9L], [id#12L], Inner, BuildRight
02   :- *Range (0, 2, step=1, splits=8)
03   +- *Range (0, 2, step=1, splits=8)

import org.apache.spark.sql.execution.WholeStageCodegenExec
val wsc = executedPlan(0).asInstanceOf[WholeStageCodegenExec]
scala> println(wsc.numberedTreeString)
00 *Project [id#9L]
01 +- *BroadcastHashJoin [id#9L], [id#12L], Inner, BuildRight
02   :- *Range (0, 2, step=1, splits=8)
03   +- *Range (0, 2, step=1, splits=8)

scala> println(wsc.child.numberedTreeString)
00 Project [id#9L]
01 +- BroadcastHashJoin [id#9L], [id#12L], Inner, BuildRight
02   :- Range (0, 2, step=1, splits=8)
03   +- Range (0, 2, step=1, splits=8)

// Let's disable wholeStage codegen
// CollapseCodegenStages becomes a noop

val newSpark = spark.newSession()
import org.apache.spark.sql.internal.SQLConf.WHOLESTAGE_CODEGEN_ENABLED
newSpark.sessionState.conf.setConf(WHOLESTAGE_CODEGEN_ENABLED, false)

scala> println(newSpark.sessionState.conf.wholeStageEnabled)
false

val ccsWholeStageDisabled = CollapseCodegenStages(conf = newSpark.sessionState.conf)
scala> println(ccsWholeStageDisabled.apply(plan).numberedTreeString)
00 Project [id#9L]
01 +- BroadcastHashJoin [id#9L], [id#12L], Inner, BuildRight
```

```
02    :- Range (0, 2, step=1, splits=8)
03    +- Range (0, 2, step=1, splits=8)
```

Inserting WholeStageCodegenExec to Physical Plan for Operators with CodeGen Support — apply Method

```
apply(plan: SparkPlan): SparkPlan
```

`apply` starts inserting `WholeStageCodegenExec` (with `InputAdapter`) in the input `plan` physical plan only when `spark.sqlcodegen.wholeStage` internal property is enabled. Otherwise, it does nothing at all (i.e. passes the input physical plan through unchanged).

Note

Input Adapters show themselves with no star in `explain`.

```
scala> spark.range(1).groupBy("id").count.explain
== Physical Plan ==
*HashAggregate(keys=[id#31L], functions=[count(1)])
+- Exchange hashpartitioning(id#31L, 200) // <- no star here
   +- *HashAggregate(keys=[id#31L], functions=[partial_count(1)])
      +- *Range (0, 1, step=1, splits=8)
```

Note

`spark.sqlcodegen.wholeStage` property is enabled by default.

```
import org.apache.spark.sql.internal.SQLConf.WHOLESTAGE_CODEGEN_ENABLED
scala> spark.conf.get(WHOLESTAGE_CODEGEN_ENABLED)
res0: String = true
```

Use `SQLConf.wholeStageEnabled` method to access the current value.

```
scala> spark.sessionState.conf.wholeStageEnabled
res1: Boolean = true
```

Inserting WholeStageCodegenExec (with InputAdapter) for Physical Operators with Codegen Support — insertWholeStageCodegen Internal Method

```
insertWholeStageCodegen(plan: SparkPlan): SparkPlan
```

`insertWholeStageCodegen` is the main recursive method of `CollapseCodegenStages` that (walks down the `plan` tree and) finds physical operators with optional Java code generation for which Java code can really be generated and inserts `WholeStageCodegenExec` operator

(with [InputAdapter](#)) for them.

Note	<code>insertWholeStageCodegen</code> skips physical operators with <code>output</code> with just a single <code>objectType</code> value (regardless of their support for codegen).
------	--

Note	<code>insertWholeStageCodegen</code> is used recursively by itself and insertInputAdapter , but more importantly when <code>CollapseCodegenStages</code> runs.
------	--

Inserting InputAdapter Unary Operator — `insertInputAdapter` Internal Method

```
insertInputAdapter(plan: SparkPlan): SparkPlan
```

`insertInputAdapter` inserts an [InputAdapter](#) unary operator in a physical plan.

- For `SortMergeJoinExec` (with inner and outer joins) [inserts an InputAdapter operator](#) for both children physical operators individually
- For [codegen-unsupported](#) operators [inserts an InputAdapter operator](#)
- For other operators (except `SortMergeJoinExec` operator above or for which [Java code cannot be generated](#)) [inserts an InputAdapter operator](#) for every child operator

Caution	FIXME Examples for every case + screenshots from web UI
---------	---

Note	<code>insertInputAdapter</code> is used in <code>insertWholeStageCodegen</code> and recursively.
------	--

Physical Operators with Codegen Support — `supportCodegen` Internal Predicate

```
supportCodegen(plan: SparkPlan): Boolean
```

`supportCodegen` finds [physical operators](#) with `CodeGenSupport` and `supportCodegen` flag enabled.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...
// both where and select support codegen
val query = spark.range(2).where('id === 0).select('id)
scala> query.explain
== Physical Plan ==
*Filter (id#88L = 0)
+- *Range (0, 2, step=1, splits=8)
```

`supportCodegen` is positive when all of the following hold:

- Catalyst expressions of the physical operator all support codegen
- Number of nested fields of the schema of the physical operator is up to `spark.sqlcodegen.maxFields` internal property (100 by default)
- Number of the nested fields in the schema of the children is up to `spark.sqlcodegen.maxFields` (same as above)

Otherwise, `supportCodegen` is negative/disabled.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...
// both where and select support codegen
// let's break the requirement of having up to spark.sqlcodegen.maxFields
val newSpark = spark.newSession()
import org.apache.spark.sql.internal.SQLConf.WHOLESTAGE_MAX_NUM_FIELDS
newSpark.sessionState.conf.setConf(WHOLESTAGE_MAX_NUM_FIELDS, 2)

scala> println(newSpark.sessionState.conf.wholeStageMaxNumFields)
2

import newSpark.implicits._
val query = Seq((1,2,3)).toDF("id", "c0", "c1").where('id === 0)
scala> query.explain
== Physical Plan ==
Project [_1#452 AS id#456, _2#453 AS c0#457, _3#454 AS c1#458]
+- Filter (_1#452 = 0)
   +- LocalTableScan [_1#452, _2#453, _3#454]
```

Expressions with Codegen Support — `supportCodegen` Internal Predicate

```
supportCodegen(e: Expression): Boolean
```

`supportCodegen` is positive when the Catalyst expression `e` is (in the order of verification):

1. LeafExpression
2. non-`CodegenFallback` expression

Otherwise, `supportCodegen` is negative.

Note

`supportCodegen` (for expressions) is used when `supportCodegen` (for physical plans) finds operators that support codegen.

EnsureRequirements Physical Preparation Rule

`EnsureRequirements` is a [physical preparation rule](#) that transforms [physical operators](#) (up the plan tree):

1. Removes two adjacent [ShuffleExchange](#) physical operators if the child partitioning scheme guarantees the parent's partitioning
2. For any other physical operator, [ensures partition distribution and ordering](#).

`EnsureRequirements` is a part of [preparations](#) batch of physical plan rules and is executed in [executedPlan](#) phase of a [query execution](#).

`EnsureRequirements` takes a [SQLConf](#) when created.

Transforming Physical Plan — `apply` Method

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

Ensuring Partition Requirements of Physical Operator — `ensureDistributionAndOrdering` Internal Method

<code>ensureDistributionAndOrdering(operator: SparkPlan): SparkPlan</code>	
--	--

`createPartitioning` Internal Method

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

`defaultNumPreShufflePartitions` Internal Method

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

Adding ExchangeCoordinator (When Adaptive Query Execution Enabled) — `withExchangeCoordinator` Internal Method

```
withExchangeCoordinator(
    children: Seq[SparkPlan],
    requiredChildDistributions: Seq[Distribution]): Seq[SparkPlan]
```

`withExchangeCoordinator` adds `ExchangeCoordinator` to `ShuffleExchange` operators if adaptive query execution is enabled and partitioning scheme of the `shuffleExchanges` support `ExchangeCoordinator`.

Internally, `withExchangeCoordinator` checks if the input `children` operators support `ExchangeCoordinator` which is that either holds:

- If there is at least one `ShuffleExchange` operator, all children are either `ShuffleExchange` with `HashPartitioning` or their output partitioning is `HashPartitioning` (even inside `PartitioningCollection`)
- There are at least two `children` operators and the input `requiredChildDistributions` are all `ClusteredDistribution`

With adaptive query execution enabled (i.e. when `spark.sql.adaptive.enabled` flag is `true`) and the operator supports `ExchangeCoordinator`, `withExchangeCoordinator` creates a `ExchangeCoordinator` and:

- For every `ShuffleExchange`, registers the `ExchangeCoordinator`
- Creates `HashPartitioning` partitioning scheme with the default number of partitions to use when shuffling data for joins or aggregations (as `spark.sql.shuffle.partitions` which is `200` by default) and adds `ShuffleExchange` to the final result (for the current physical operator)

Otherwise (when adaptive query execution is disabled or `children` do not support `ExchangeCoordinator`), `withExchangeCoordinator` returns the input `children` unchanged.

Note

`withExchangeCoordinator` is used exclusively for enforcing partition requirements of a physical operator.

SQL Parsing Framework

SQL Parser Framework in Spark SQL uses ANTLR to translate a SQL text to a [data type](#), [Expression](#), [TableIdentifier](#) or [LogicalPlan](#).

The contract of the SQL Parser Framework is described by [ParserInterface](#) contract. The contract is then abstracted in [AbstractSqlParser](#) class so subclasses have to provide custom [AstBuilder](#) only.

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.

SparkSqlParser — Default SQL Parser

`SparkSqlParser` is the default parser of the SQL statements supported in Spark SQL with the `astBuilder` as `SparkSqlAstBuilder` and support for variable substitution.

Note

Spark SQL supports SQL statements as described in [SqlBase.g4](#) ANTLR grammar.

`SparkSqlParser` is available as `sqlParser` of a `SessionState`.

```
val spark: SparkSession = ...
spark.sessionState.sqlParser
```

`SparkSqlParser` is used to translate an expression to its corresponding `Column` in the following:

- `expr` function
- `selectExpr` method (of `Dataset`)
- `filter` method (of `Dataset`)
- `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:

- `table` methods in `DataFrameReader` and `SparkSession`
- `insertInto` and `saveAsTable` methods of `DataFrameWriter`
- `createExternalTable` and `refreshTable` methods of `Catalog` (and `SessionState`)

`SparkSqlParser` is used to translate a SQL text to its corresponding `LogicalPlan` in `sql` method in `SparkSession`.

	<p>Enable <code>INFO</code> logging level for <code>org.apache.spark.sql.execution.SparkSqlParser</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.sql.execution.SparkSqlParser=INFO</pre> <p>Refer to Logging.</p>
--	---

Variable Substitution

Caution	<p>FIXME See <code>SparkSqlParser</code> and <code>substitutor</code>.</p>
---------	--

SparkSqlAstBuilder

Table 1. SparkSqlAstBuilder's Visit Callback Methods (in alphabetical order)

Callback Method	ANTLR rule / labeled alternative	Spark SQL Entity
visitCacheTable	#cacheTable	<ul style="list-style-type: none"> • CacheTableCommand logical command for CACHE LAZY? TABLE [table] (AS? [query])?
visitCreateTable	#createTable	<ul style="list-style-type: none"> • CreateTable logical operator for CREATE TABLE ... AS ... • CreateTempViewUsing logical operators for CREATE TEMPORARY VIEW ... USING ...

Table 2. SparkSqlAstBuilder's Parsing Handlers (in alphabetical order)

Parsing Handler	LogicalPlan Added
withRepartitionByExpression	

CatalystSqlParser — DataTypes and StructTypes Parser

`CatalystSqlParser` is a `AbstractSqlParser` with `AstBuilder` as the required `astBuilder`.

`CatalystSqlParser` is used to translate `DataTypes` from their canonical string representation (e.g. when [adding fields to a schema](#) or [casting column to a different data type](#)) or `StructTypes`.

```
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](#).

AstBuilder—ANTLR-based SQL Parser

`AstBuilder` converts a SQL string into Spark SQL's corresponding entity (i.e. `DataType`, `Expression`, `LogicalPlan` or `TableIdentifier`) using [visit callback methods](#).

`AstBuilder` is the [AST builder](#) of `AbstractSqlParser` (i.e. the base SQL parsing infrastructure in Spark SQL).

Spark SQL supports SQL queries as described in [SqlBase.g4](#). Using the file can tell SQL supports at any given time.

"Almost" being that although the grammar accepts a SQL statement it can be reported by `AstBuilder`, e.g.

```
scala> sql("EXPLAIN FORMATTED SELECT * FROM myTable").show
org.apache.spark.sql.catalyst.parser.ParseException:
Operation not allowed: EXPLAIN FORMATTED(line 1, pos 0)

== SQL ==
EXPLAIN FORMATTED SELECT * FROM myTable
^^^

Tip   at org.apache.spark.sql.catalyst.parser.ParserUtils$.operationNotAllowed(Parse
      at org.apache.spark.sql.execution.SparkSqlAstBuilder$$anonfun$visitExplain$1.a
      at org.apache.spark.sql.execution.SparkSqlAstBuilder$$anonfun$visitExplain$1.a
      at org.apache.spark.sql.catalyst.parser.ParserUtils$.withOrigin(ParserUtils.sc
      at org.apache.spark.sql.execution.SparkSqlAstBuilder.visitExplain(SparkSqlPars
      at org.apache.spark.sql.execution.SparkSqlAstBuilder.visitExplain(SparkSqlPars
      at org.apache.spark.sql.catalyst.parser.SqlBaseParser$ExplainContext.accept(Sq
      at org.antlr.v4.runtime.tree.AbstractParseTreeVisitor.visit(AbstractParseTreeV
      at org.apache.spark.sql.catalyst.parser.AstBuilder$$anonfun$visitSingleStateme
      at org.apache.spark.sql.catalyst.parser.AstBuilder$$anonfun$visitSingleStateme
      at org.apache.spark.sql.catalyst.parser.ParserUtils$.withOrigin(ParserUtils.sc
      at org.apache.spark.sql.catalyst.parser.AstBuilder.visitSingleStatement(AstBui
      at org.apache.spark.sql.catalyst.parser.AbstractSqlParser$$anonfun$parsePlan$1
      at org.apache.spark.sql.catalyst.parser.AbstractSqlParser$$anonfun$parsePlan$1
      at org.apache.spark.sql.catalyst.parser.AbstractSqlParser.parse(ParseDriver.sc
      at org.apache.spark.sql.execution.SparkSqlParser.parse(SparkSqlParser.scala:46
      at org.apache.spark.sql.catalyst.parser.AbstractSqlParser.parsePlan(ParseDrive
      at org.apache.spark.sql.SparkSession.sql(SparkSession.scala:617)
... 48 elided
```

Technically, `AstBuilder` is a ANTLR `AbstractParseTreeVisitor` (as `SqlBaseBaseVisitor`) that is generated from [SqlBase.g4](#) ANTLR grammar for Spark SQL.

`SqlBaseBaseVisitor` is a ANTLR-specific base class that is auto-generated at build time from a ANTLR grammar in `SqlBase.g4`.

`SqlBaseBaseVisitor` is an ANTLR `AbstractParseTreeVisitor`.

Table 1. AstBuilder's Visit Callback Methods (in a table)

Callback Method	ANTLR rule / labeled alternative
-----------------	----------------------------------

visitDescribeTable	describeTable	DescribeTableCommand logic
visitExplain	explain	ExplainCommand Can be a OneRowRelation or DescribeTableCommand for TABLE. Note <pre>val q = sql("EXPLAIN SELECT * FROM test") scala> println(q.q) scala> println(q.q) 00 ExplainCommand</pre>
visitFromClause	fromClause	LogicalPlan Supports multiple comma-separated relations (e.g. INNER JOIN) with optional LAT . A relation can be one of the following: <ul style="list-style-type: none">• Table identifier• Inline table using <code>VALUES</code> expression• Table-valued function (current catalog)
visitFunctionCall	functionCall labeled alternative	<ul style="list-style-type: none">• UnresolvedFunction for a function reference• UnresolvedWindowExpression for a WindowSpecReference• WindowExpression for a function call Tip See the function call section.
visitNamedExpression	namedExpression	<ul style="list-style-type: none">• Alias (for a single alias)• MultiAlias (for a parenthesized list of aliases)• a bare Expression
visitRelation	relation	LogicalPlan for a <code>FROM</code> clause.
visitSingleDataType	singleDataType	DataType
visitSingleExpression	singleExpression	Expression Takes the named expression and converts it to a LogicalPlan.
visitSingleStatement	singleStatement	LogicalPlan from a single statement.

visitSingleStatement	singleStatement	Note	A single statement
visitSingleTableIdentifier	singleTableIdentifier	TableIdentifier	
visitWindowDef	windowDef labeled alternative	WindowSpecDefinition	<pre>'(' CLUSTER BY partition+=expr '(' ((PARTITION DISTRIBUTE ((ORDER SORT) BY sortIdentifier windowFrame? ')')</pre>
visitQuerySpecification	querySpecification	LogicalPlan	<p>Can be a OneRowRelation</p> <p>Note</p> <pre>val q = sql("select * from t").as[OneRowRelation]</pre>

Table 2. AstBuilder's Parsing Handlers (in alphabetical order)

Parsing Handler	LogicalPlan Added		
withAggregation	<ul style="list-style-type: none"> GroupingSets for GROUP BY ... GROUPING SETS (...) Aggregate for GROUP BY ... (WITH CUBE WITH ROLLUP)? 		
withGenerate	Generate with UnresolvedGenerator and join enabled		
withHints	<p>Hint for /*+ hint */ in SELECT .</p> <table border="1"> <tr> <td>Tip</td> <td>Note + (plus) between /* and */</td> </tr> </table> <p>hint is of the format name OR name (params) with name as BROADCAST , BROADCASTJOIN OR MAPJOIN .</p> <pre>/*+ BROADCAST (table) */</pre>	Tip	Note + (plus) between /* and */
Tip	Note + (plus) between /* and */		
	<p>Join for a FROM clause and relation alone.</p> <p>The following join types are supported:</p> <ul style="list-style-type: none"> INNER (default) CROSS 		

	<ul style="list-style-type: none"> • LEFT (with optional OUTER) • LEFT SEMI • RIGHT (with optional OUTER) • FULL (with optional OUTER) • ANTI (optionally prefixed with LEFT) <p>The following join criteria are supported:</p> <ul style="list-style-type: none"> • ON booleanExpression • USING '(' identifier (',' identifier)* ')' <p>Joins can be NATURAL (with no join criteria).</p>
withQueryResultClauses	
withQuerySpecification	
withWindows	<p>WithWindowDefinition for window aggregates (given WINDOW definitions).</p> <p>Used for withQueryResultClauses and withQuerySpecification with windows definition.</p> <pre>WINDOW identifier AS windowSpec (',' identifier AS windowSpec)*</pre> <p>Tip Consult windows , namedWindow , windowSpec , windowFrame , and frameBound (with windowRef and windowDef) ANTLR parsing rules for Spark SQL in SqlBase.g4.</p>

Note

AstBuilder belongs to org.apache.spark.sql.catalyst.parser package.

Function Examples

The examples are handled by [visitFunctionCall](#).

```
import spark.sessionState.sqlParser

scala> sqlParser.parseExpression("foo()")
res0: org.apache.spark.sql.catalyst.expressions.Expression = 'foo()

scala> sqlParser.parseExpression("foo() OVER windowSpecRef")
res1: org.apache.spark.sql.catalyst.expressions.Expression = unresolvedwindowexpression('foo(), WindowSpecReference(windowSpecRef))

scala> sqlParser.parseExpression("foo() OVER (CLUSTER BY field)")
res2: org.apache.spark.sql.catalyst.expressions.Expression = 'foo() windowspecdefinition('field, UnspecifiedFrame)
```

AbstractSqlParser — Base SQL Parsing Infrastructure

`AbstractSqlParser` is the one and only [ParserInterface](#) in Spark SQL that acts as the foundation of the SQL parsing infrastructure with [two concrete implementations available](#) (that are *merely* required to define their custom [AstBuilder](#) for the final transformation of SQL textual representation to their Spark SQL equivalent entities, i.e. [DataType](#), [Expression](#), [LogicalPlan](#) and [TableIdentifier](#)).

`AbstractSqlParser` first [sets up `SqlBaseLexer`](#) and [`sqlBaseParser`](#) for parsing (and pass the latter on to a parsing function) and use [AstBuilder](#) for the actual parsing.

Table 1. `AbstractSqlParser`'s Implementations (in alphabetical order)

Name	Description
<code>SparkSqlParser</code>	The default SQL parser available as <code>sqlParser</code> in SessionState . <pre>val spark: SparkSession = ... spark.sessionState.sqlParser</pre>
<code>CatalystSqlParser</code>	Parses DataType or StructType (schema) from their canonical string representation.

`AbstractSqlParser` simply relays all the SQL parsing to translate a SQL string to that specialized [AstBuilder](#).

AbstractSqlParser Contract

```
abstract class AbstractSqlParser extends ParserInterface {
  def astBuilder: AstBuilder
  def parse[T](command: String)(toResult: SqlBaseParser => T): T
  def parseDataType(sqlText: String): DataType
  def parsePlan(sqlText: String): LogicalPlan
  def parseExpression(sqlText: String): Expression
  def parseTableIdentifier(sqlText: String): TableIdentifier
  def parseTableSchema(sqlText: String): StructType
}
```

Table 2. AbstractSqlParser Contract (in alphabetical order)

Method	Description
astBuilder	<p>Gives AstBuilder for the actual SQL parsing.</p> <p>Used in all the <code>parse</code> methods, i.e. parseDataType, parseExpression, parsePlan, parseTableIdentifier, and parseTableSchema.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>Note Both implementations, i.e. SparkSqlParser and CatalystSqlParser, come with their own <code>astBuilder</code> method.</p> </div>
parse	<p>Sets up <code>SqlBaseLexer</code> and <code>SqlBaseParser</code> for parsing and passes the latter on to the input <code>toResult</code> function where the parsing finally happens.</p> <p>Used in all the <code>parse</code> methods, i.e. parseDataType, parseExpression, parsePlan, parseTableIdentifier, and parseTableSchema.</p>
parseDataType	Used when...
parseExpression	Used when...
parsePlan	<p>Creates a LogicalPlan for a given SQL textual statement.</p> <p><code>parsePlan</code> builds a <code>SqlBaseParser</code> and requests AstBuilder to parse a single SQL statement.</p> <p>When a SQL statement could not be parsed, <code>parsePlan</code> reports a <code>ParseException</code>:</p> <div style="border: 1px solid #ccc; padding: 5px; background-color: #f0f0f0; margin-top: 10px;"> <p>Unsupported SQL statement</p> </div>
parseTableIdentifier	Used when...
parseTableSchema	Used when...

Setting Up SqlBaseLexer and SqlBaseParser for Parsing — `parse` Method

```
parse[T](command: String)(toResult: SqlBaseParser => T): T
```

`parse` sets up a proper ANTLR parsing infrastructure with `SqlBaseLexer` and `SqlBaseParser` (which are the ANTLR-specific classes of Spark SQL that are auto-generated at build time from the `SqlBase.g4` grammar).

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/v3.1.2/sql/catalyst/src/main/antlr4/org/apache/spark/sql/catalyst/parser/SqlBase.g4).

Internally, `parse` first prints out the following INFO message to the logs:

```
INFO SparkSqlParser: Parsing command: [command]
```

Tip

Enable `INFO` logging level for the custom `AbstractSqlParser`, i.e. `SparkSqlParser` or `CatalystSqlParser`, to see the above INFO message.

`parse` then creates and sets up a `SqlBaseLexer` and `SqlBaseParser` that in turn passes the latter on to the input `toResult` function where the parsing finally happens.

Note

`parse` uses `SLL` prediction mode for parsing first before falling back to `LL` mode.

In case of parsing errors, `parse` reports a `ParseException`.

ParserInterface — SQL Parser Contract

`ParserInterface` is the [parser contract](#) for creating `Expression` (to create `Columns` from), `LogicalPlan`, `TableIdentifier`, and `StructType` for a given SQL textual representation.

Note	The one and only <code>ParserInterface</code> in Spark SQL is AbstractSqlParser .
------	---

`ParserInterface` is available as `sqlParser` in [SessionState](#).

```
val spark: org.apache.spark.sql.SparkSession = ...
spark.sessionState.sqlParser
```

ParserInterface Contract

```
package org.apache.spark.sql.catalyst.parser

trait ParserInterface {
  def parseExpression(sqlText: String): Expression
  def parsePlan(sqlText: String): LogicalPlan
  def parseTableIdentifier(sqlText: String): TableIdentifier
  def parseTableSchema(sqlText: String): StructType
}
```

Table 1. ParserInterface Contract (in alphabetical order)

Method	Description
<code>parseExpression</code>	Used when...
<code>parsePlan</code>	Used mainly when <code>SparkSession</code> is requested to execute a SQL query using <code>sql</code> method. <pre>scala> :type spark org.apache.spark.sql.SparkSession scala> spark.sql("show databases").show +-----+ databaseName +-----+ default +-----+</pre>
<code>parseTableIdentifier</code>	Used when...
<code>parseTableSchema</code>	Used when...

It has the only single abstract subclass [AbstractSqlParser](#).

SQLMetric — Physical Operator Metric

`SQLMetric` is a [accumulator](#) metric used to monitor performance of [physical operators](#).

Note	Use Details for Query page in SQL tab in web UI to see the metrics of a structured query.
------	---

`SQLMetric` takes a metric type and an initial value when created.

Table 1. Metric Types and Corresponding Create Methods (in alphabetical order)

Metric Type	Create Method	Failed Values Counted?	Description
size	<code>createSizeMetric</code>	no	Used when...
sum	<code>createMetric</code>	no	Used when...
timing	<code>createTimingMetric</code>	no	Used when...

Posting Driver-Side Updates to SQLMetrics — `postDriverMetricUpdates` Method

```
postDriverMetricUpdates(
  sc: SparkContext,
  executionId: String,
  metrics: Seq[SQLMetric]): Unit
```

`postDriverMetricUpdates` posts a [SparkListenerDriverAccumUpdates](#) to a [SparkListenerBus](#) when `executionId` is specified.

Note	<code>postDriverMetricUpdates</code> used in BroadcastExchangeExec , FileSourceScanExec and SubqueryExec physical operators.
------	--

Catalyst—Tree Manipulation Framework

Catalyst is an execution-agnostic framework to represent and manipulate a **dataflow graph**, i.e. trees of [relational operators](#) and [expressions](#).

Note

The Catalyst framework were first introduced in [SPARK-1251 Support for optimizing and executing structured queries](#) and became part of Apache Spark on 20/Mar/14 19:12.

The main abstraction in Catalyst is [TreeNode](#) that is then used to build trees of [Expressions](#) or [QueryPlans](#).

Spark 2.0 uses the Catalyst tree manipulation framework to build an extensible **query plan optimizer** with a number of query optimizations.

Catalyst supports both rule-based and cost-based optimization.

TreeNode — Node in Catalyst Tree

`TreeNode` is a node in [Catalyst](#) tree with zero or more [children](#) (and can build [expression](#) or [structured query plan trees](#)).

`TreeNode` offers not only functions that you may have used from Scala Collection API, e.g. `map`, `flatMap`, `collect`, `collectFirst`, `foreach`, but also `mapChildren`, `transform`, `transformDown`, `transformUp`, `foreachUp`, `numberedTreeString`, `p`, `asCode`, `prettyJson`, etc. that are particularly useful for tree manipulation or debugging.

Note

Scala-specific, `TreeNode` is an abstract class that is the base class of [Expression](#) and Catalyst's [QueryPlan](#) abstract classes.

`TreeNode` abstract type is a quite advanced Scala type definition (at least comparing to the other Scala types in Spark) so understanding its behaviour even outside Spark might be worthwhile by itself.

Tip

```
abstract class TreeNode[BaseType <: TreeNode[BaseType]] extends Product {
  self: BaseType =>
  // ...
}
```

TreeNode Contract

```
package org.apache.spark.sql.catalyst.trees

abstract class TreeNode[BaseType <: TreeNode[BaseType]] extends Product {
  self: BaseType =>

  // only required methods that have no implementation
  def children: Seq[BaseType]
  def verboseString: String
}
```

Table 1. (Subset of) TreeNode Contract (in alphabetical order)

Method	Description
<code>children</code>	
<code>verboseString</code>	

QueryPlan — Structured Query Plan

`QueryPlan` is a part of [Catalyst](#) to model a [tree of relational operators](#), i.e. a structured query.

Scala-specific, `QueryPlan` is an abstract class that is the base class of [LogicalPlan](#) and [SparkPlan](#) (for logical and physical plans, respectively).

A `QueryPlan` has an [output](#) attributes (that serves as the base for the schema), a collection of [expressions](#) and a [schema](#).

`QueryPlan` has [statePrefix](#) that is used when displaying a plan with `!` to indicate an invalid plan, and `'` to indicate an unresolved plan.

A `QueryPlan` is **invalid** if there are [missing input attributes](#) and `children` subnodes are non-empty.

A `QueryPlan` is **unresolved** if the column names have not been verified and column types have not been looked up in the [Catalog](#).

QueryPlan Contract

```
abstract class QueryPlan[T] extends TreeNode[T] {
    def output: Seq[Attribute]
    def validConstraints: Set[Expression]
    // FIXME
}
```

Table 1. QueryPlan Contract (in alphabetical order)

Method	Description
<code>validConstraints</code>	
<code>output</code>	Attribute expressions

outputSet Property

Caution

FIXME

producedAttributes Property

Caution

FIXME

Missing Input Attributes — `missingInput` Property

```
def missingInput: AttributeSet
```

`missingInput` are [attributes](#) that are referenced in expressions but not provided by this node's children (as `inputSet`) and are not produced by this node (as `producedAttributes`).

Query Output Schema — `schema` Property

You can request the schema of a `QueryPlan` using `schema` that builds [StructType](#) from the [output attributes](#).

```
// the query
val dataset = spark.range(3)

scala> dataset.queryExecution.analyzed.schema
res6: org.apache.spark.sql.types.StructType = StructType(StructField(id,LongType, false
))
```

Output Schema — `output` Property

```
output: Seq[Attribute]
```

`output` property are [attributes](#) that represent the result of a projection in a query that is later used to build a [schema](#).

Note

`output` property is also called **output schema** or **result schema**.

You can access the `output` schema through a [LogicalPlan](#).

```
// the query
val dataset = spark.range(3)

scala> dataset.queryExecution.analyzed.output
res0: Seq[org.apache.spark.sql.catalyst.expressions.Attribute] = List(id#0L)

scala> dataset.queryExecution.withCachedData.output
res1: Seq[org.apache.spark.sql.catalyst.expressions.Attribute] = List(id#0L)

scala> dataset.queryExecution.optimizedPlan.output
res2: Seq[org.apache.spark.sql.catalyst.expressions.Attribute] = List(id#0L)

scala> dataset.queryExecution.sparkPlan.output
res3: Seq[org.apache.spark.sql.catalyst.expressions.Attribute] = List(id#0L)

scala> dataset.queryExecution.executedPlan.output
res4: Seq[org.apache.spark.sql.catalyst.expressions.Attribute] = List(id#0L)
```

You can build a `StructType` from `output` collection of attributes using `toStructType` method (that is available through the implicit class `AttributeSeq`).

```
scala> dataset.queryExecution.analyzed.output.toStructType
res5: org.apache.spark.sql.types.StructType = StructType(StructField(id,LongType,false
))
```

statePrefix method

```
statePrefix: String
```

`statePrefix` method is used when printing a plan with `!` to indicate an invalid plan and `'` to indicate an unresolved plan.

RuleExecutor — Tree Transformation Rule Executor

RuleExecutor executes a collection of rules (as batches) to transform a TreeNode.

Note	Available <code>TreeNodes</code> are either logical or physical operators.
------	--

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 HiveSessionStateBuilder$$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 HiveSessionStateBuilder$$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 HiveSessionStateBuilder$$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 HiveSessionStateBuilder$$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 HiveSessionStateBuilder$$anon$1: Batch [batchName] has no effect.
```

Batch — Collection of Rules

`Batch` in Catalyst is a named collection of [optimization 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	<code>Once</code> strategy is a <code>FixedPoint</code> strategy with one iteration.
------	--

Rule

A `rule` in Catalyst is a named transformation that can be applied to a plan tree.

`Rule` abstract class defines `ruleName` attribute and a single method `apply`:

```
apply(plan: TreeType): TreeType
```

Note	<code>TreeType</code> is the type of a (plan) tree that a <code>Rule</code> works with, e.g. LogicalPlan , SparkPlan or Expression .
------	--

GenericStrategy

Executing Planning Strategy — apply Method

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

QueryPlanner — Converting Logical Plan to Physical Trees

`QueryPlanner` plans a logical plan for execution, i.e. converts a logical plan to one or more physical plans using strategies.

Note

`QueryPlanner` generates at least one physical plan.

`QueryPlanner`'s main method is `plan` that defines the extension points, i.e. `strategies`, `collectPlaceholders` and `prunePlans`.

`QueryPlanner` is a part of Catalyst Framework.

QueryPlanner Contract

```
abstract class QueryPlanner[PhysicalPlan <: TreeNode[PhysicalPlan]] {
  def collectPlaceholders(plan: PhysicalPlan): Seq[(PhysicalPlan, LogicalPlan)]
  def prunePlans(plans: Iterator[PhysicalPlan]): Iterator[PhysicalPlan]
  def strategies: Seq[GenericStrategy[PhysicalPlan]]
}
```

Table 1. QueryPlanner Contract (in alphabetical order)

Method	Description
<code>strategies</code>	Collection of <code>GenericStrategy</code> planning strategies. Used exclusively as an extension point in <code>plan</code> .
<code>collectPlaceholders</code>	Collection of "placeholder" physical plans and the corresponding <code>logical plans</code> . Used exclusively as an extension point in <code>plan</code> . Overridden in <code>SparkPlanner</code>
<code>prunePlans</code>	Prunes physical plans (e.g. bad or somehow incorrect plans). Used exclusively as an extension point in <code>plan</code> .

Planning Logical Plan — `plan` Method

```
plan(plan: LogicalPlan): Iterator[PhysicalPlan]
```

`plan` converts the input `plan` [logical plan](#) to zero or more `PhysicalPlan` [plans](#).

Internally, `plan` applies [planning strategies](#) to the input `plan` (one by one collecting all as the plan candidates).

`plan` then walks over the plan candidates to [collect placeholders](#).

If a plan does not contain a placeholder, the plan is returned as is. Otherwise, `plan` walks over placeholders (as pairs of `PhysicalPlan` and unplanned [logical plan](#)) and (recursively) [plans](#) the child logical plan. `plan` then replaces the placeholders with the planned child logical plan.

In the end, `plan` [prunes "bad" physical plans](#).

Note	<code>plan</code> is used exclusively (through the concrete SparkPlanner) when a <code>QueryExecution</code> is requested for a physical plan .
------	--

`plan` is used exclusively (through the concrete [SparkPlanner](#)) when a `QueryExecution` [is requested for a physical plan](#).

Catalyst DSL — Implicit Conversions for Catalyst Data Structures

`package object dsl` is a [collection of implicit conversions](#) that create a DSL for constructing Catalyst data structures, i.e. [expressions](#) and [logical plans](#).

Note

Most implicit conversions from `package object dsl` interfere with the implicits conv automatically in `spark-shell` as `spark.implicits._`.

```
scala> 'hello.decimal
<console>:30: error: type mismatch;
           found   : Symbol
           required: ?{def decimal: ?}
Note that implicit conversions are not applicable because they are ambiguous:
  both method symbolToColumn in class SQLImplicits of type (s: Symbol)org.apache.spark.sql.catalyst.expressions.Expression
  and method DslSymbol in trait ExpressionConversions of type (sym: Symbol)org.apache.spark.sql.catalyst.dsl.expressions.Expression
are possible conversion functions from Symbol to ?{def decimal: ?}
           'hello.decimal
           ^
<console>:30: error: value decimal is not a member of Symbol
           'hello.decimal
           ^
```

```
import org.apache.spark.sql.catalyst.dsl.expressions._
import org.apache.spark.sql.catalyst.dsl.plans._

// ExpressionConversions

import org.apache.spark.sql.catalyst.expressions.Literal
scala> val trueLit: Literal = true
trueLit: org.apache.spark.sql.catalyst.expressions.Literal = true

import org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute
scala> val name: UnresolvedAttribute = 'name
name: org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute = 'name

// NOTE: This conversion may not work, e.g. in spark-shell
// There is another implicit conversion StringToColumn in SQLImplicits
// It is automatically imported in spark-shell
// See :imports
val id: UnresolvedAttribute = $"id"

import org.apache.spark.sql.catalyst.expressions.Expression
scala> val expr: Expression = sum('id)
expr: org.apache.spark.sql.catalyst.expressions.Expression = sum('id)

// implicit class DslSymbol
scala> 'hello.s
res2: String = hello
```

```
scala> 'hello.attr
res4: org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute = 'hello

// implicit class DslString
scala> "hello".expr
res0: org.apache.spark.sql.catalyst.expressions.Expression = hello

scala> "hello".attr
res1: org.apache.spark.sql.catalyst.analysis.UnresolvedAttribute = 'hello

// plans

scala> val t1 = table("t1")
t1: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
'UnresolvedRelation `t1`

scala> val p = t1.select('*).serialize[String].where('id % 2 == 0)
p: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =
'Filter false
+- 'SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String,
StringType, fromString, input[0, java.lang.String, true], true) AS value#1]
  +- 'Project ['*]
    +- 'UnresolvedRelation `t1

// FIXME Does not work because SimpleAnalyzer's catalog is empty
// the p plan references a t1 table
import org.apache.spark.sql.catalyst.analysis.SimpleAnalyzer
scala> p.analyze
```

Table 1. Catalyst DSL's Implicit Conversions (in alphabetical order)

Name	Description
ExpressionConversions	<ul style="list-style-type: none"> • Adds <code>ImplicitOperators</code> operators to <code>Catalyst expressions</code> • Converts Scala native types (e.g. <code>Boolean</code>, <code>Long</code>, <code>String</code>, <code>Date</code>, <code>Timestamp</code>) and Spark SQL types (i.e. <code>Decimal</code>) to <code>Literal expressions</code> • Converts Scala's <code>Symbol</code> to <code>UnresolvedAttribute</code> and <code>AttributeReference</code> expressions • Converts <code>\$\$col name</code> to an <code>UnresolvedAttribute</code> expression • Adds aggregate and non-aggregate functions to <code>Catalyst expressions</code> (e.g. <code>sum</code>, <code>count</code>, <code>upper</code>, <code>star</code>, <code>callFunction</code>, <code>windowSpec</code>, <code>windowExpr</code>) • Creates <code>UnresolvedFunction</code> (<code>function operator</code>) and <code>BoundReference</code> (<code>at operator</code>) expressions
ImplicitOperators	Operators for <code>expressions</code>
plans	<ul style="list-style-type: none"> • <code>table</code> to create a <code>UnresolvedRelation</code> logical plan • Logical operators (e.g. <code>select</code>, <code>where</code>, <code>filter</code>, <code>serialize</code>, <code>join</code>, <code>groupBy</code>, <code>window</code>, <code>generate</code>)

ExchangeCoordinator and Adaptive Query Execution

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

postShuffleRDD Method

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

ShuffledRowRDD

`ShuffledRowRDD` is a specialized `RDD` of `InternalRow`s.

Note

`ShuffledRowRDD` looks like `ShuffledRDD`, and the difference is in the type of the values to process, i.e. `InternalRow` and `(K, C)` key-value pairs, respectively.

`ShuffledRowRDD` takes a `ShuffleDependency` (of integer keys and `InternalRow` values).

Note

The `dependency` property is mutable and is of type `ShuffleDependency[Int, InternalRow, InternalRow]`.

`ShuffledRowRDD` takes an optional `specifiedPartitionStartIndices` collection of integers that is the number of post-shuffle partitions. When not specified, the number of post-shuffle partitions is managed by the `Partitioner` of the input `shuffleDependency`.

Note

Post-shuffle partition is...[FIXME](#)

Table 1. ShuffledRowRDD and RDD Contract

Name	Description
<code>getDependencies</code>	A single-element collection with <code>ShuffleDependency[Int, InternalRow, InternalRow]</code> .
<code>partitioner</code>	<code>CoalescedPartitioner</code> (with the <code>Partitioner</code> of the dependency)
<code>getPreferredLocations</code>	
<code>compute</code>	

numPreShufflePartitions Property

Caution

[FIXME](#)

Computing Partition (in TaskContext) — compute Method

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

Note

`compute` is a part of [RDD contract](#) to compute a given partition in a [TaskContext](#).

Internally, `compute` makes sure that the input `split` is a [ShuffledRowRDDPartition](#). It then requests `ShuffleManager` for a `ShuffleReader` to read `InternalRow`s for the `split`.

Note

`compute` uses `SparkEnv` to access the current `shuffleManager`.

Note

`compute` uses `ShuffleHandle` (of [ShuffleDependency](#) dependency) and the pre-shuffle start and end partition offsets.

Getting Placement Preferences of Partition — `getPreferredLocations` Method

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

Note

`getPreferredLocations` is a part of [RDD contract](#) to specify placement preferences (aka *preferred task locations*), i.e. where tasks should be executed to be as close to the data as possible.

Internally, `getPreferredLocations` requests `MapOutputTrackerMaster` for the preferred locations of the input `partition` (for the single [ShuffleDependency](#)).

Note

`getPreferredLocations` uses `SparkEnv` to access the current `MapOutputTrackerMaster` (which runs on the driver).

CoalescedPartitioner

Caution**FIXME**

ShuffledRowRDDPartition

Caution**FIXME**

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](#) or [debugCodegen](#) methods.

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: {}
```

"Debugging" Codegen — debugCodegen Method

You use `debugCodegen` method to review the [CodegenSupport](#)-generated code.

```

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;
...

```

`debugCodegen` is equivalent to using `debug` interface of the [QueryExecution](#).

Note

```

val q = spark.range(1, 1000).select('id+1+2+3, 'id+4+5+6)
scala> q.queryExecution.debug.codegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Project [(id#3L + 6) AS (((id + 1) + 2) + 3)#6L, (id#3L + 15) AS (((id + 4) +
+- *Range (1, 1000, step=1, 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...

```

Datasets vs DataFrames 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.

SQLConf

`SQLConf` is an internal key-value configuration store for [parameters and hints](#) used in Spark SQL.

`SQLConf` is publicly available through the user-facing interface `RuntimeConfig` that access using [SparkSession](#).

Note

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

scala> spark.conf
res0: org.apache.spark.sql.RuntimeConfig = org.apache.spark.sql.RuntimeConfig@5e
388a77
```

`SQLConf` offers methods to [get](#), [set](#), [unset](#) or [clear](#) their values, but has also the [accessor methods](#) to read the current value of a parameter or hint.

You can access a session-specific `SQLConf` using [SessionState](#):

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

import org.apache.spark.sql.internal.SQLConf
scala> val conf: SQLConf = spark.sessionState.conf
conf: org.apache.spark.sql.internal.SQLConf = org.apache.spark.sql.internal.SQLConf@5e
388a77
```

Table 1. SQLConf's Accessor Methods (in alphabetical order)

Name	Parameter / Hint	
adaptiveExecutionEnabled	spark.sql.adaptive.enabled	Used Enabled Execution Adaptive
autoBroadcastJoinThreshold	spark.sql.autoBroadcastJoinThreshold	Used Join Broadcast Placeholder
broadcastTimeout	spark.sql.broadcastTimeout	Used Broadcast Expiration

<code>dataFramePivotMaxValues</code>	spark.sql.pivotMaxValues	Used Op
<code>dataFrameRetainGroupColumns</code>	spark.sql.retainGroupColumns	Used Rewritten when Data columns modified
<code>defaultSizeInBytes</code>	spark.sql.defaultSizeInBytes	Used
<code>numShufflePartitions</code>	spark.sql.shuffle.partitions	Used Optimizer plan so has
<code>joinReorderEnabled</code>	spark.sql.cbo.joinReorder.enabled	Used Optimizer logic
<code>preferSortMergeJoin</code>	spark.sql.join.preferSortMergeJoin	Used Join plan so has
<code>starSchemaDetection</code>	spark.sql.cbo.starSchemaDetection	Used Rewritten operator statement
		Used

<code>wholeStageEnabled</code>	<code>spark.sqlcodegen.wholeStage</code>	Used when whole stage codegen is enabled.
<code>wholeStageFallback</code>	<code>spark.sqlcodegen.fallback</code>	Used when whole stage codegen is disabled.
<code>wholeStageMaxNumFields</code>	<code>spark.sqlcodegen.maxFields</code>	Used when whole stage codegen is enabled.
<code>useObjectHashAggregation</code>	<code>spark.sql.execution.useObjectHashAggregateExec</code>	Used when object hash aggregation is planned or selected.

Table 2. Parameters and Hints (in alphabetical order)

Name	Default Value	Description
<code>spark.sql.adaptive.enabled</code>	<code>false</code>	Enables adaptive query execution.
<code>spark.sql.autoBroadcastJoinThreshold</code>	<code>10L * 1024 * 1024</code>	<p>Maximum size (in bytes) of a broadcasted DataFrame that will be broadcasted across all nodes when performing a broadcast join.</p> <p>If the size of the DataFrame in the logical plan of a broadcast join exceeds this setting, the DataFrames will be broadcasted as multiple partitions for join.</p> <p>Negative values indicate that broadcasting is disabled.</p> <p>Use <code>autoBroadcastJoinThreshold</code> method to access this parameter.</p>
<code>spark.sql.broadcastTimeout</code>	<code>5 * 60</code>	<p>Timeout in seconds for broadcast wait time in broadcast joins.</p> <p>When negative, it is treated as infinity (i.e. <code>Duration.Infinity()</code>).</p> <p>Used through <code>SQLConf.broadcastTimeout</code>.</p>

<code>spark.sql.cbo.enabled</code>	false	<p>Enables cost-based optimizer (CBO) for estimation when enabled (i.e., when <code>spark.sql.cbo.enabled</code> is set to <code>true</code>).</p> <p>Used (through <code>StarSchemaDetection.reorderStar</code>) in:</p> <ul style="list-style-type: none"> • ReorderJoin optimization • StarSchemaDetection.reorderStar • CostBasedJoin plan optimization • For statistics, Project, Filter, Aggregate logical plan optimization.
<code>spark.sql.cbo.joinReorder.enabled</code>	false	<p>Enables join reorder optimization (CBO).</p> <p>Use joinReorder access the current setting.</p>
<code>spark.sql.cbo.starSchemaDetection</code>	false	<p>Enables join reorder for star schema detection optimization (CBO).</p> <p>Use starSchemaDetection access the current setting.</p>
<code>spark.sqlcodegen.fallback</code>	true	<p>(internal) Whether codegen could be used for the part of a codegen stage that did not (<code>false</code>).</p> <p>Use wholeStageCodegen access the current setting.</p>
<code>spark.sqlcodegen.maxFields</code>	100	<p>(internal) Maximum number of fields (including regular and nullable) for whole-stage codegen above which the codegen will be split into multiple stages.</p> <p>Use wholeStageCodegen method to access the current setting.</p>

<code>spark.sqlcodegen.wholeStage</code>	<code>true</code>	<p>(internal) Whether (of multiple physical plans) to be compiled into a single stage (<code>true</code>) or not (<code>false</code>).</p> <p>Use wholeStage method to access the current value.</p>
<code>spark.sql.defaultSizeInBytes</code>	<code>Java's Long.MaxValue</code>	<p>(internal) Table size limit for planning.</p> <p>It is by default set to <code>Long.MaxValue</code>. We can set it lower to be more conservative. It is say by default the optimizer choose to broadcast a table if it knows for sure its size is enough.</p> <p>Use useObjectHashAggregateExec method to access the current value.</p>
<code>spark.sql.execution.useObjectHashAggregateExec</code>	<code>true</code>	<p>Decides if we use ObjectHashAggregateExec or Aggregation exec strategy.</p> <p>Use useObjectHashAggregateExec method to access the current value.</p>
<code>spark.sql.join.preferSortMergeJoin</code>	<code>true</code>	<p>(internal) Controls whether execution planner prefers sort merge join over broadcast hash join.</p> <p>Use preferSortMergeJoin method to access the current value.</p>
<code>spark.sql.optimizer.maxIterations</code>	<code>100</code>	<p>Maximum number of iterations for Analyzer and Optimizer.</p>
<code>spark.sql.pivotMaxValues</code>	<code>10000</code>	<p>Maximum number of distinct values that will be collected (when doing a pivot) for the values for the pivot columns.</p> <p>Use dataFramePivot method to access the current value.</p>
<code>spark.sql.retainGroupColumns</code>	<code>true</code>	<p>Controls whether group columns are used for aggregation in RelationalGroupedDataset.</p>

		Use dataFrameR method to access.
spark.sql.selfJoinAutoResolveAmbiguity	true	Control whether to resolve ambiguity in join conditions automatically.
spark.sql.shuffle.partitions	200	Default number of partitions when shuffling data for aggregations. Corresponds to mapred.reduce.tasks . Spark considers
spark.sql.streaming.fileSink.log.deletion	true	Controls whether to delete log files in file storage .
spark.sql.streaming.fileSink.log.cleanupDelay	FIXME	FIXME
spark.sql.streaming.schemaInference	FIXME	FIXME
spark.sql.streaming.fileSink.log.compactInterval	FIXME	FIXME

Note SQLConf is a `private[sql] serializable` class in `org.apache.spark.sql.internal` 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`.

CatalystConf

`CatalystConf` is...[FIXME](#)

Note

The default `catalystconf` is [SQLConf](#) that is...[FIXME](#)

Table 1. CatalystConf's Internal Properties (in alphabetical order)

Name	Initial Value	Description
<code>caseSensitiveAnalysis</code>		
<code>cboEnabled</code>		<p>Enables cost-based optimizations (CBO) for estimation of plan statistics when enabled.</p> <p>Used in CostBasedJoinReorder logical plan optimization and Project , Filter , Join and Aggregate logical operators.</p>
<code>optimizerMaxIterations</code>	<code>spark.sql.optimizer.maxIterations</code>	Maximum number of iterations for Analyzer and Optimizer .
<code>sessionLocalTimeZone</code>		

resolver Method

`resolver` gives case-sensitive or case-insensitive `Resolvers` per `caseSensitiveAnalysis` setting.

Note

`Resolver` is a mere function of two `String` parameters that returns `true` if both refer to the same entity (i.e. for case insensitive equality).

Catalog

`Catalog` is the [interface to work with a metastore](#), i.e. a data catalog of database(s), local and external tables, functions, table columns, and temporary views in Spark SQL.

You can access the current catalog using [SparkSession.catalog](#) attribute.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

scala> spark.catalog
    lazy val catalog: org.apache.spark.sql.catalog.Catalog

scala> spark.catalog
res0: 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|
+-----+-----+-----+-----+

scala> spark.catalog.clearCache
```

Note

The one and only `Catalog` in Spark SQL 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
    def functionExists(functionName: String): Boolean
    def functionExists(dbName: String, functionName: String): Boolean
}

```

Table 1. Catalog Contract (in alphabetical order)

Method	Description
cacheTable	Caches the specified table in memory Used for SQL's CACHE TABLE and <code>AlterTableRenameCommand</code> command.
functionExists	

CatalogImpl

`CatalogImpl` is the one and only `Catalog` that... [FIXME](#)

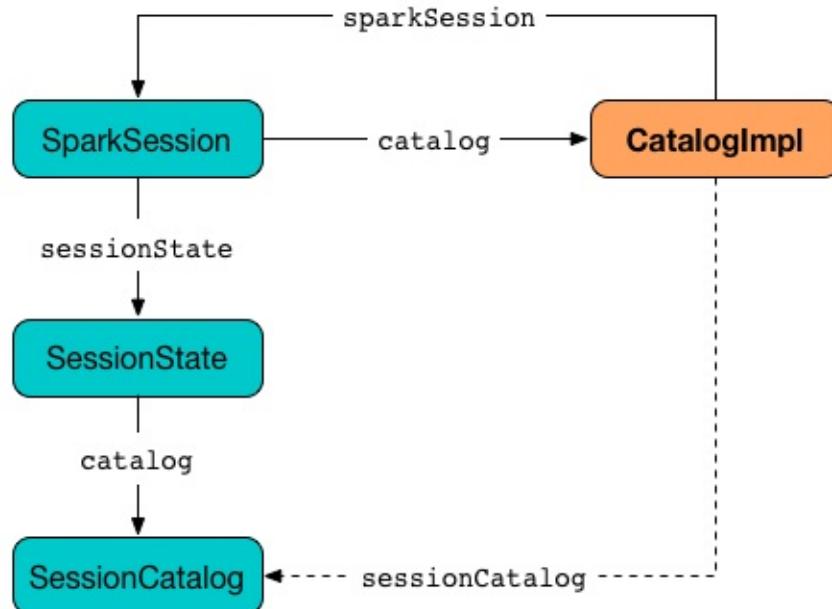


Figure 1. `CatalogImpl` uses `SessionCatalog` (through `SparkSession`)

Note

`CatalogImpl` is in `org.apache.spark.sql.internal` package.

functionExists Method

Caution

[FIXME](#)

refreshTable Method

Caution

[FIXME](#)

Caching Table or View In-Memory — cacheTable Method

```
cacheTable(tableName: String): Unit
```

Internally, `cacheTable` first creates a `DataFrame` for the table followed by requesting `CacheManager` to [cache it](#).

Note

`cacheTable` uses the `session-scoped SharedState` to access the `CacheManager`.

Note	cacheTable is a part of Catalog contract.
------	---

Removing All Cached Tables From In-Memory Cache — clearCache Method

```
clearCache(): Unit
```

clearCache requests CacheManager to remove all cached tables from in-memory cache.

Note	clearCache is a part of Catalog contract.
------	---

Creating External Table From Path — createExternalTable Method

```
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
```

createExternalTable creates an external table `tableName` from the given `path` and returns the corresponding [DataFrame](#).

```

import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

val readmeTable = spark.catalog.createExternalTable("readme", "README.md", "text")
readmeTable: org.apache.spark.sql.DataFrame = [value: string]

scala> spark.catalog.listTables.filter(_.name == "readme").show
+-----+-----+-----+-----+
| name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|readme| default|      null| EXTERNAL|     false|
+-----+-----+-----+-----+

scala> sql("select count(*) as count from readme").show(false)
+---+
|count|
+---+
|99   |
+---+

```

The `source` input parameter is the name of the data source provider for the table, e.g. parquet, json, text. If not specified, `createExternalTable` uses `spark.sql.sources.default` setting to know the data source format.

Note	<code>source</code> input parameter must not be <code>hive</code> as it leads to a <code>AnalysisException</code> .
-------------	---

`createExternalTable` sets the mandatory `path` option when specified explicitly in the input parameter list.

`createExternalTable` parses `tableName` into `TableIdentifier` (using `SparkSqlParser`). It creates a `CatalogTable` and then `executes` (by `toRDD`) a `CreateTable` logical plan. The result `DataFrame` is a `Dataset[Row]` with the `QueryExecution` after executing `SubqueryAlias` logical plan and `RowEncoder`.

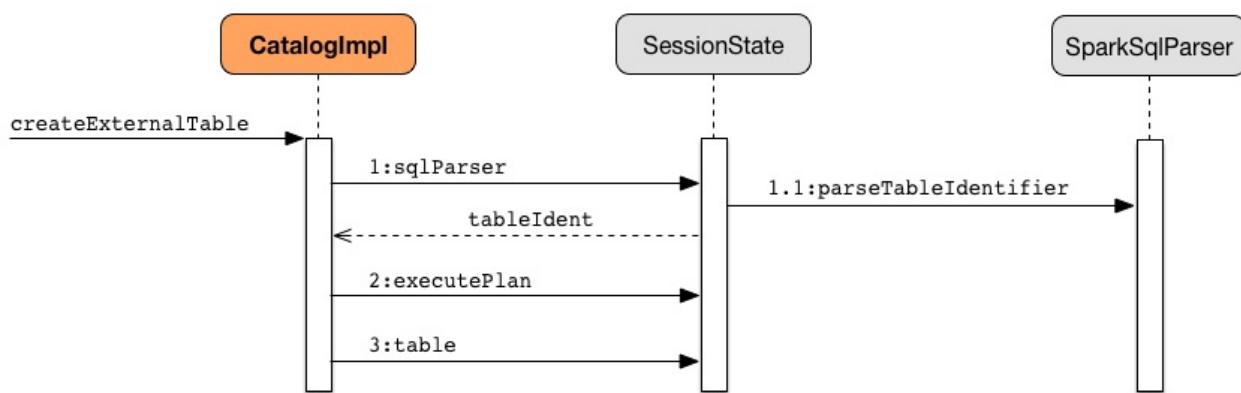


Figure 2. CatalogImpl.createExternalTable

Note	<code>createExternalTable</code> is a part of Catalog contract .
-------------	--

ExternalCatalog — System Catalog of Permanent Entities

`ExternalCatalog` is the [contract for system catalog](#) of permanent entities, i.e. databases, tables, partitions, and functions.

There are currently two implementations of `ExternalCatalog`.

Table 1. ExternalCatalog Implementations

Catalog Alias	Catalog Class	Description
in-memory	<code>org.apache.spark.sql.catalyst.catalog.InMemoryCatalog</code>	An in-memory (ephemeral) system catalog
hive	<code>org.apache.spark.sql.hive.HiveExternalCatalog</code>	

[spark.sql.catalogImplementation](#) property sets the current `ExternalCatalog` implementation (with `in-memory` being the default).

ExternalCatalog Contract

`ExternalCatalog` contract assumes that implementations offer the following features:

Table 2. ExternalCatalog Features per Entity

Feature	Function	Partitions	Tables	Databases
Create	X	X	X	X
Drop	X	X	X	X
Rename	X	X	X	
Get	X	X	X	
Check Existence	X		X	X
List	X	X	X	
Alter		X	X	X
Load		X	X	X
Set				X

SessionState

`SessionState` is the [state separation layer](#) between Spark SQL sessions, including SQL configuration, tables, functions, UDFs, SQL parser, and everything else that depends on a [SQLConf](#).

You can access `SessionState` of a `SparkSession` through [sessionState](#) property.

```
val spark: SparkSession = ...
spark.sessionState
```

`SessionState` is created when...[FIXME](#)

Table 1. SessionState's (Lazily-Initialized) Attributes (in alphabetical order)

Name	Type	Description
<code>analyzer</code>	Analyzer	FIXME Used when... FIXME
<code>catalog</code>	SessionCatalog	Manages tables and databases. Used when... FIXME
<code>conf</code>	SQLConf	FIXME Used when... FIXME
<code>experimentalMethods</code>	ExperimentalMethods	FIXME Used when... FIXME
<code>functionRegistry</code>	FunctionRegistry	FIXME Used when... FIXME
<code>functionResourceLoader</code>	FunctionResourceLoader	FIXME Used when... FIXME
<code>listenerManager</code>	ExecutionListenerManager	FIXME Used when... FIXME
<code>optimizer</code>	Optimizer	Logical query plan optimizer

optimizer	Optimizer	Used exclusively when QueryExecution creates an optimized logical plan.
planner	SparkPlanner	FIXME Used when...FIXME
resourceLoader	SessionResourceLoader	FIXME Used when...FIXME
sqlParser	ParserInterface	FIXME Used when...FIXME
streamingQueryManager	StreamingQueryManager	FIXME Used when...FIXME
udfRegistration	UDFRegistration	Interface to register user-defined functions. Used when...FIXME

Note

SessionState is a `private[sql]` class and, given the package `org.apache.spark.sql.internal`, SessionState should be considered *internal*.

Creating SessionState Instance

SessionState takes the following when created:

- SharedState
- SQLConf
- ExperimentalMethods
- FunctionRegistry
- UDFRegistration
- SessionCatalog
- ParserInterface
- Analyzer
- Optimizer

- [SparkPlanner](#)
- [StreamingQueryManager](#)
- [ExecutionListenerManager](#)
- [SessionResourceLoader](#)
- Function to create [QueryExecution](#) for a given [logical plan](#)
- Function to clone the current `SessionState` for a given pair of [SparkSession](#) and `SessionState`

`SessionState` initializes the [attributes](#).

apply Factory Methods

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

```
apply(sparkSession: SparkSession): SessionState (1)
apply(sparkSession: SparkSession, sqlConf: SQLConf): SessionState
```

1. Passes `sparkSession` to the other `apply` with a new `SQLConf`

Note	<code>apply</code> is used when <code>SparkSession</code> is requested for <code>SessionState</code> .
------	--

clone Method

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

Note	<code>clone</code> is used when...
------	------------------------------------

createAnalyzer Internal Method

```
createAnalyzer(
  sparkSession: SparkSession,
  catalog: SessionCatalog,
  sqlConf: SQLConf): Analyzer
```

`createAnalyzer` creates a logical query plan [Analyzer](#) with rules specific to a non-Hive `SessionState` .

Table 2. Analyzer's Evaluation Rules for non-Hive SessionState (in the order of execution)

Method	Rules	Description
extendedResolutionRules	FindDataSourceTable	Replaces <code>InsertIntoTable</code> (with <code>CatalogRelation</code>) and <code>CatalogRelation</code> logical plans with <code>LogicalRelation</code> .
	ResolveSQLOnFile	
postHocResolutionRules	PreprocessTableCreation	
	PreprocessTableInsertion	
	DataSourceAnalysis	
extendedCheckRules	PreWriteCheck	
	HiveOnlyCheck	

Note`createAnalyzer` is used when `SessionState` is [created](#) or [cloned](#).

Executing Logical Plan — `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...

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 `ScriptTransformation`, `ParquetRelation`, `StateStoreRDD`, and `SessionState` itself, and few other places.

Caution

FIXME What is `ScriptTransformation`? `StateStoreRDD`?

BaseSessionStateBuilder — Base for Builders of SessionState

Note

`BaseSessionStateBuilder` is an experimental and unstable API. You've been warned!.

`BaseSessionStateBuilder` is created when `SparkSession` is requested for a `SessionState` (and also when `newBuilder` is called).

```
val spark: SparkSession = ...
scala> spark.sessionState
res0: org.apache.spark.sql.internal.SessionState = org.apache.spark.sql.internal.SessionState@5feb8e9a
```

Note

`SessionStateBuilder` and `HiveSessionStateBuilder` are concrete `BaseSessionStateBuilder`.

Table 1. BaseSessionStateBuilder's Properties (in alphabetical order)

Name	Description
analyzer	
catalog	
conf	
createClone	
createQueryExecution	
experimentalMethods	
functionRegistry	FunctionRegistry CAUTION: FIXME Where is this used?
listenerManager	
optimizer	
planner	
resourceLoader	
sqlParser	
streamingQueryManager	
udfRegistration	

Note	BaseSessionStateBuilder defines a type alias NewBuilder for a function to create a BaseSessionStateBuilder .
	<pre>type NewBuilder = (SparkSession, Option[SessionState]) => BaseSessionStateBuilder</pre>

BaseSessionStateBuilder Contract

```
abstract class BaseSessionStateBuilder {
  def newBuilder: NewBuilder
}
```

Table 2. BaseSessionStateBuilder Contract (in alphabetical order)

Method	Description
newBuilder	Function to create a BaseSessionStateBuilder

Creating BaseSessionStateBuilder Instance

BaseSessionStateBuilder takes the following when created:

- [SparkSession](#)
- Optional [SessionState](#)

Building SessionState — `build` Method

```
build(): SessionState
```

`build` creates a [SessionState](#) (based on the [SharedState](#) of the input [SparkSession](#) and [properties](#)).

SessionCatalog — Metastore of Session-Specific Relational Entities

`SessionCatalog` is a catalog (aka *registry* or *metastore*) of session-specific relational entities.

You can access a session-specific `SessionCatalog` through [SessionState](#).

```
val spark: SparkSession = ...
spark.sessionState.catalog
```

`SessionCatalog` is [created](#) when `SessionState` [sets](#) `catalog` (lazily).

Table 1. SessionCatalog's Internal Registries and Counters (in alphabetical order)

Name	Description
<code>tempTables</code>	FIXME Used when... FIXME
<code>currentDb</code>	FIXME Used when... FIXME
<code>functionResourceLoader</code>	FIXME Used when... FIXME
<code>tableRelationCache</code>	A cache of fully-qualified table names to table relation plans (i.e. <code>LogicalPlan</code>). Used when <code>SessionCatalog</code> refreshes a table

functionExists Method

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

	<code>functionExists</code> is used in:
Note	<ul style="list-style-type: none"> • <code>LookupFunctions</code> logical evaluation rule (to make sure that <code>UnresolvedFunction</code> can be resolved, i.e. is registered with <code>SessionCatalog</code>)
	<ul style="list-style-type: none"> • <code>CatalogImpl</code> to check if a function exists in a database
	• ...

listFunctions Method

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

refreshTable Method

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

createTempFunction Method

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

loadFunctionResources Method

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

alterTempViewDefinition Method

```
alterTempViewDefinition(name: TableIdentifier, viewDefinition: LogicalPlan): Boolean
```

`alterTempViewDefinition` alters the temporary view by [updating an in-memory temporary table](#) (when a database is not specified and the table has already been registered) or a global temporary table (when a database is specified and it is for global temporary tables).

Note	"Temporary table" and "temporary view" are synonyms.
------	--

`alterTempViewDefinition` returns `true` when an update could be executed and finished successfully.

createTempView Method

Caution

FIXME

createGlobalTempView Method

Caution

FIXME

createTable Method

Caution

FIXME

alterTable Method

Caution

FIXME

Creating SessionCatalog Instance

`SessionCatalog` takes the following when created:

- `ExternalCatalog`
- `GlobalTempViewManager`
- `FunctionResourceLoader`
- `FunctionRegistry`
- `CatalystConf`
- Hadoop's `Configuration`
- `ParserInterface`

`SessionCatalog` initializes the `internal registries and counters`.

Finding Function by Name (Using FunctionRegistry) — `lookupFunction` Method

```
lookupFunction(  
    name: FunctionIdentifier,  
    children: Seq[Expression]): Expression
```

`lookupFunction` finds a function by `name`.

For a function with no database defined that exists in [FunctionRegistry](#), `lookupFunction` requests `FunctionRegistry` to [find the function](#) (by its unqualified name, i.e. with no database).

If the `name` function has the database defined or does not exist in `FunctionRegistry`, `lookupFunction` uses the fully-qualified function `name` to check if the function exists in [FunctionRegistry](#) (by its fully-qualified name, i.e. with a database).

For other cases, `lookupFunction` requests [ExternalCatalog](#) to find the function and [loads its resources](#). It then [creates a corresponding temporary function](#) and [looks up the function again](#).

Note

`lookupFunction` is used exclusively when `Analyzer` [resolves functions](#).

UDFRegistration

`UDFRegistration` is an interface to the session-scoped `FunctionRegistry` to register user-defined functions (UDFs) and [user-defined aggregate functions](#) (UDAFs).

`UDFRegistration` is available using [SparkSession](#).

```
spark.udf
```

`UDFRegistration` is [created](#) exclusively for [SessionState](#).

Creating UDFRegistration Instance

`UDFRegistration` takes the following when created:

- `FunctionRegistry`

Registering UserDefinedAggregateFunction (with FunctionRegistry) — `register` Method

```
register(  
    name: String,  
    udaf: UserDefinedAggregateFunction): UserDefinedAggregateFunction
```

`register` [registers](#) a `UserDefinedAggregateFunction` under `name` with `FunctionRegistry`.

`register` [creates](#) a `ScalaUDAF` internally to register a UDAF.

Note

`register` gives `udaf` back after the function has been registered with `FunctionRegistry`.

FunctionRegistry

`FunctionRegistry` is a base registry (aka *catalog*) of native and user-defined functions.

Note

The one and only `FunctionRegistry` available in Spark SQL is [SimpleFunctionRegistry](#).

You can access a session-specific `FunctionRegistry` through [SessionState](#).

```
val spark: SparkSession = ...
spark.sessionState.functionRegistry
```

Note

You can register a new user-defined function using [UDFRegistration](#).

Table 1. FunctionRegistry's Attributes (in alphabetical order)

Name	Description
<code>builtin</code>	SimpleFunctionRegistry with the built-in functions registered.
<code>expressions</code>	Collection of expressions that represent built-in/native functions.

lookupFunction Method

Caution

[FIXME](#)

registerFunction Methods

```
registerFunction(name: String, builder: FunctionBuilder): Unit (1)
registerFunction(name: String, info: ExpressionInfo, builder: FunctionBuilder): Unit
```

1. Relays calls to the other `registerFunction`

Note

`registerFunction` is used when...[FIXME](#)

SimpleFunctionRegistry

`SimpleFunctionRegistry` is the default `FunctionRegistry` that is backed by a hash map (with optional case sensitivity).

ExperimentalMethods

`ExperimentalMethods` holds extra [strategies](#) and [optimizations](#) (as `Rule[LogicalPlan]`) that are used in [SparkPlanner](#) and [SparkOptimizer](#), respectively.

Table 1. ExperimentalMethods' Attributes (in alphabetical order)

Name	Description
<code>extraStrategies</code>	Collection of <code>Strategy</code> objects that are used when: <ul style="list-style-type: none">• <code>SessionState</code> is requested for <code>sparkPlanner</code>
<code>extraOptimizations</code>	Collection of rules to optimize LogicalPlans (i.e. <code>Rule[LogicalPlan]</code> objects) that are used when: <ul style="list-style-type: none">• <code>SparkOptimizer</code> is requested for the batches (with "User Provided Optimizers" batch for the extra optimizations)

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 Method

```
withExecutionId[T](
  sc: SparkContext,
  executionId: String)(body: => T): T  (1)

withNewExecutionId[T](
  sparkSession: SparkSession,
  queryExecution: QueryExecution)(body: => T): T  (2)
```

1. With explicit `executionId` execution identifier
2. `QueryExecution` -variant with an auto-generated execution identifier

`withNewExecutionId` executes `body` query action with the **execution id** local property set (as `executionId` or auto-generated).

The execution id is set as `spark.sql.execution.id` local property.

The use case is to track Spark jobs (e.g. when running in separate threads) that belong to a single SQL query execution, e.g. to report them as one single Spark SQL query in web UI.

Note

`withNewExecutionId` 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 `QueryExecution` variant.

CatalystSerde

`CatalystSerde` is a Scala object that consists of three utility methods:

1. `deserialize` to create a new logical plan with the input logical plan wrapped inside `DeserializeToObject` logical operator.
2. `serialize`
3. `generateObjAttr`

`CatalystSerde` and belongs to `org.apache.spark.sql.catalyst.plans.logical` package.

Creating Logical Plan with DeserializeToObject Logical Operator for Logical Plan — `deserialize` Method

```
deserialize[T : Encoder](child: LogicalPlan): DeserializeToObject
```

`deserialize` creates a `DeserializeToObject` logical operator for the input `child` logical plan.

Internally, `deserialize` creates a `UnresolvedDeserializer` for the deserializer for the type `T` first and passes it on to a `DeserializeToObject` with a `AttributeReference` (being the result of `generateObjAttr`).

serialize Method

```
serialize[T : Encoder](child: LogicalPlan): SerializeFromObject
```

generateObjAttr Method

```
generateObjAttr[T : Encoder]: Attribute
```

Tungsten Execution Backend (aka Project Tungsten)

The goal of **Project Tungsten** is to improve Spark execution by optimizing Spark jobs for **CPU and memory efficiency** (as opposed to network and disk I/O which are considered fast enough). Tungsten focuses on the hardware architecture of the platform Spark runs on, including but not limited to JVM, LLVM, GPU, NVRAM, etc. It does so by offering the following optimization features:

1. [Off-Heap Memory Management](#) using binary in-memory data representation aka **Tungsten row format** and managing memory explicitly,
2. [Cache Locality](#) which is about cache-aware computations with cache-aware layout for high cache hit rates,
3. [Whole-Stage Code Generation](#) (aka *CodeGen*).

Important	Project Tungsten uses <code>sun.misc.unsafe</code> API for direct memory access to bypass the JVM in order to avoid garbage collection.
-----------	---

```
// million integers
val intsMM = 1 to math.pow(10, 6).toInt

// that gives ca 3.8 MB in memory
scala> sc.parallelize(intsMM).cache.count
res0: Long = 1000000

// that gives ca 998.4 KB in memory
scala> intsMM.toDF.cache.count
res1: Long = 1000000
```

The screenshot shows the Apache Spark 2.1.0-SNAPSHOT Storage UI. The top navigation bar includes links for Jobs, Stages, Storage, Environment, Executors, SQL, and Spark shell application UI. The Storage tab is active. Below the navigation, there's a table titled "RDDs" with the following data:

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
ParallelCollectionRDD	Memory Deserialized 1x Replicated	8	100%	3.8 MB	0.0 B
LocalTableScan [value#1]	Memory Deserialized 1x Replicated	8	100%	998.4 KB	0.0 B

Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
ParallelCollectionRDD	Memory Deserialized 1x Replicated	8	100%	3.8 MB	0.0 B
LocalTableScan [value#1]	Memory Deserialized 1x Replicated	8	100%	998.4 KB	0.0 B

Figure 1. RDD vs DataFrame Size in Memory in web UI—Thank you, Tungsten!

Off-Heap Memory Management

Project Tungsten aims at substantially reducing the usage of JVM objects (and therefore JVM garbage collection) by introducing its own off-heap binary memory management. Instead of working with Java objects, Tungsten uses `sun.misc.Unsafe` to manipulate raw memory.

Tungsten uses the compact storage format called [UnsafeRow](#) for data representation that further reduces memory footprint.

Since [Datasets](#) have known [schema](#), Tungsten properly and in a more compact and efficient way lays out the objects on its own. That brings benefits similar to using extensions written in low-level and hardware-aware languages like C or assembler.

It 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).

Cache Locality

Tungsten uses algorithms and **cache-aware data structures** that exploit the physical machine caches at different levels - L1, L2, L3.

BytesToBytesMap

1. Low space overhead,
2. Good memory locality, esp. for scans.

Whole-Stage Code Generation

Tungsten does code generation at compile time and generates JVM bytecode to access Tungsten-managed memory structures that gives a very fast access. It uses the [Janino compiler](#)—a super-small, super-fast Java compiler.

Note	The code generation was tracked under SPARK-8159 Improve expression function coverage (Spark 1.5) .
------	---

Tip	Read Whole-Stage Code Generation .
-----	--

Further reading or watching

1. [Project Tungsten: Bringing Spark Closer to Bare Metal](#)

2. (video) [From DataFrames to Tungsten: A Peek into Spark's Future](#) by Reynold Xin (Databricks)
3. (video) [Deep Dive into Project Tungsten: Bringing Spark Closer to Bare Metal](#) by Josh Rosen (Databricks)

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 *Whole-Stage CodeGen*) fuses multiple operators (as a subtree of plans that [support code generation](#)) 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.

Note

Whole-Stage Code Generation is enabled by default (using [spark.sql.codegen.wholeStage](#) property).

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 \(PDF\)](#).

Note

Janino is used to compile a Java source code into a Java class.

Before a query is executed, [CollapseCodegenStages](#) physical preparation rule 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.

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)
```

CodegenSupport — Physical Operators with Optional Java Code Generation

`CodegenSupport` is an extension of [physical operators](#) that support **Java code generation** (aka **codegen**).

`CodegenSupport` allows physical operators to [disable codegen](#).

Tip

Use [debugCodegen](#) or [QueryExecution.debug.codegen](#) methods to review a `CodegenSupport`-generated Java source code.

```
val q = spark.range(1)

import org.apache.spark.sql.execution.debug._
scala> q.debugCodegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Range (0, 1, step=1, splits=8)

Generated code:
...
// The above is equivalent to the following method chain
scala> q.queryExecution.debug.codegen
Found 1 WholeStageCodegen subtrees.
== Subtree 1 / 1 ==
*Range (0, 1, step=1, splits=8)

Generated code:
...
```

CodegenSupport Contract

```
package org.apache.spark.sql.execution

trait CodegenSupport extends SparkPlan {
    // only required methods that have no implementation
    def doProduce(ctx: CodegenContext): String
    def inputRDDs(): Seq[RDD[InternalRow]]
}
```

Table 1. (Subset of) CodegenSupport Contract (in alphabetical order)

Method	Description
doProduce	Used exclusively in the final <code>produce</code> method to generate a Java source code for processing the <code>internal binary rows</code> from <code>input RDDs</code> .
inputRDDs	

Generating Java Source Code For...~~FIXME~~ — `consume` Final Method

Caution

`FIXME`

`supportCodegen` Flag

```
supportCodegen: Boolean = true
```

Note

`supportCodegen` is used exclusively when `CollapseCodegenStages` checks if a physical operator supports codegen.

Note

`supportCodegen` is disabled for the following physical operators:

- `GenerateExec`
- `HashAggregateExec` with `ImperativeAggregate`
- `SortMergeJoinExec` for all `join types` except `INNER` and `CROSS`

Producing Java Source Code — `produce` Method

```
produce(ctx: CodegenContext, parent: CodegenSupport): String
```

`produce` creates a Java source code for processing the `internal binary rows` from input RDD.

Internally, `produce` executes a "query" that creates a Java source code with the result of `doProduce`.

Note

Executing a "query" is about preparing the query for execution followed by `waitForSubqueries`.

You can see the blocks of Java source code generated by `produce` that are marked with `PRODUCE: comment`.

Tip

Enable `spark.sql.codegen.comments` property to have the comments in the generated Java source code.

```
// ./bin/spark-shell -c spark.sql.codegen.comments=true
import org.apache.spark.sql.execution.debug._
val query = Seq((0 to 4).toList).toDF.
  select(explode('value) as "id").
  join(spark.range(1), "id")

scala> query.debugCodegen
Found 2 WholeStageCodegen subtrees.
== Subtree 1 / 2 ==
*Project [id#6]
+- *BroadcastHashJoin [cast(id#6 as bigint)], [id#9L], Inner, BuildRight
  :- Generate explode(value#1), false, false, [id#6]
  : +- LocalTableScan [value#1]
  +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
    +- *Range (0, 1, step=1, splits=8)
...
/* 043 */ protected void processNext() throws java.io.IOException {
/* 044 */   // PRODUCE: Project [id#6]
/* 045 */   // PRODUCE: BroadcastHashJoin [cast(id#6 as bigint)], [id#9L], Inner, Bu
ildRight
/* 046 */   // PRODUCE: InputAdapter
/* 047 */   while (inputadapter_input.hasNext() && !stopEarly()) {
...
== Subtree 2 / 2 ==
*Range (0, 1, step=1, splits=8)
...
/* 082 */ protected void processNext() throws java.io.IOException {
/* 083 */   // PRODUCE: Range (0, 1, step=1, splits=8)
/* 084 */   // initialize Range
```

Note

`produce` is used mainly when `WholeStageCodegenExec` is requested to generate the Java source code for a physical plan.

InternalRow — Abstract Binary Row Format

Note	<code>InternalRow</code> is also called a Spark SQL row .
------	--

Note	<code>UnsafeRow</code> is a concrete <code>InternalRow</code> .
------	---

```
// 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)]
```

getString Method

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

UnsafeRow — Mutable Raw-Memory Unsafe Binary Row Format

`UnsafeRow` is a concrete [InternalRow](#) that represents a mutable internal raw-memory (and hence unsafe) binary row format.

```
// Use ExpressionEncoder for simplicity
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
val stringEncoder = ExpressionEncoder[String]
val row = stringEncoder.toRow("hello world")

import org.apache.spark.sql.catalyst.expressions.UnsafeRow
val unsafeRow = row match { case ur: UnsafeRow => ur }

scala> println(unsafeRow.getSizeInBytes)
32

scala> unsafeRow.getBytes
res0: Array[Byte] = Array(0, 0, 0, 0, 0, 0, 0, 0, 11, 0, 0, 0, 16, 0, 0, 0, 104, 101,
108, 108, 111, 32, 119, 111, 114, 108, 100, 0, 0, 0, 0, 0, 0)

scala> unsafeRow.getUTF8String(0)
res1: org.apache.spark.unsafe.types.UTF8String = hello world
```

`UnsafeRow` supports Java's [Externalizable](#) and Kryo's [KryoSerializable](#) serialization/deserialization protocols.

The fields of a data row are placed using **field offsets**.

UnsafeRow's mutable field [data types](#) (in alphabetical order):

- BooleanType
- ByteType
- DateType
- DoubleType
- FloatType
- IntegerType
- LongType
- NullType

- ShortType
- TimestampTyp

UnsafeRow is composed of three regions:

1. Null Bit Set Bitmap Region (1 bit/field) for tracking null values
2. Fixed-Length 8-Byte Values Region
3. Variable-Length Data Section

That gives the property of rows being always 8-byte word aligned and so their size is always a multiple of 8 bytes.

Equality comparision and hashing of rows can be performed on raw bytes since if two rows are identical so should be their bit-wise representation. No type-specific interpretation is required.

Kryo's KryoSerializable SerDe Protocol

Tip	Read up on KryoSerializable .
-----	---

Serializing JVM Object — KryoSerializable's write Method

```
void write(Kryo kryo, Output out)
```

Deserializing Kryo-Managed Object — KryoSerializable's read Method

```
void read(Kryo kryo, Input in)
```

Java's Externalizable SerDe Protocol

Tip	Read up on java.io.Externalizable .
-----	---

Serializing JVM Object — Externalizable's writeExternal Method

```
void writeExternal(ObjectOutput out)
throws IOException
```

Deserializing Java-Externalized Object — Externalizable's `readExternal` Method

```
void readExternal(ObjectInput in)
throws IOException, ClassNotFoundException
```

UnsafeProjection

Creating UnsafeProjection — `create` Factory Method

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

CodeGenerator

`CodeGenerator` is a base class for generators of JVM bytecode for expression evaluation.

Table 1. CodeGenerator's Internal Properties (in alphabetical order)

Name	Description
<code>cache</code>	Guava's LoadingCache with at most 100 pairs of <code>CodeAndComment</code> and <code>GeneratedClass</code> .
<code>genericMutableRowType</code>	

Tip	Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator</code> logger to see what happens inside. Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator=DEBUG</code> Refer to Logging .

CodeGenerator Contract

```
package org.apache.spark.sql.catalyst.expressions.codegen

abstract class CodeGenerator[InType, OutType] {
  def create(in: InType): OutType
  def canonicalize(in: InType): InType
  def bind(in: InType, inputSchema: Seq[Attribute]): InType
  def generate(expressions: InType, inputSchema: Seq[Attribute]): OutType
  def generate(expressions: InType): OutType
}
```

Table 2. CodeGenerator Contract (in alphabetical order)

Method	Description
<code>generate</code>	Generates an evaluator for expression(s) that may (optionally) have expression(s) bound to a schema (i.e. a collection of Attribute). Used in: <ul style="list-style-type: none">• <code>ExpressionEncoder</code> for UnsafeProjection (for serialization)

Compiling Java Source Code using Janino — `doCompileInternal Method`

Caution

[FIXME](#)

Finding or Compiling Java Source Code — `compile Method`

Caution

[FIXME](#)

Creating CodegenContext — `newCodeGenContext Method`

Caution

[FIXME](#)

`create` Method

```
create(references: Seq[Expression]): UnsafeProjection
```

Caution

[FIXME](#)

Note

`create` is used when:

- `CodeGenerator` generates an expression evaluator
- `GenerateOrdering` creates a code gen ordering for `SortOrder` expressions

GenerateUnsafeProjection

`GenerateUnsafeProjection` is a [CodeGenerator](#) for converting [Catalyst expressions](#) to [UnsafeProjection](#).

```
GenerateUnsafeProjection: Seq[Expression] => UnsafeProjection
```

Creating ExprCode for Catalyst Expressions — `createCode` Method

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

generate Method

```
generate(  
  expressions: Seq[Expression],  
  subexpressionEliminationEnabled: Boolean): UnsafeProjection
```

`generate` [creates](#) a [UnsafeProjection](#) with `expressions` [canonicalized](#).

Note	<code>generate</code> is used when <code>unsafeProjection</code> factory object creates a UnsafeProjection .
------	--

canonicalize Method

```
canonicalize(in: Seq[Expression]): Seq[Expression]
```

`canonicalize` [removes unnecessary](#) `Alias` [expressions](#).

Internally, `canonicalize` uses `ExpressionCanonicalizer` rule executor (that in turn uses just one `CleanExpressions` expression rule).

create Method

```
create(  
  expressions: Seq[Expression],  
  subexpressionEliminationEnabled: Boolean): UnsafeProjection  
create(references: Seq[Expression]): UnsafeProjection (1)
```

1. Calls the former `create` with `subexpressionEliminationEnabled` disabled

`create` first creates a [CodegenContext](#) and creates a [ExprCode](#) for the input `expressions` that is converted to a Java source code (as `codeAndComment`).

You should see the following DEBUG message in the logs:

```
DEBUG GenerateUnsafeProjection: code for [expressions]:  
[code]
```

Note	<p>Enable <code>DEBUG</code> logging level for <code>org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator</code> logger to see the message above.</p> <p>See CodeGenerator.</p>
------	---

`create` requests `CodeGenerator` to compile the Java source code into a [GeneratedClass](#).

You should see the following INFO message in the logs:

```
INFO CodeGenerator: Code generated in [time] ms
```

Note	<p>Enable <code>INFO</code> logging level for <code>org.apache.spark.sql.catalyst.expressions.codegen.CodeGenerator</code> logger to see the message above.</p> <p>See CodeGenerator.</p>
------	--

`create` passes references into the `GeneratedClass` that eventually becomes the final [UnsafeProjection](#).

Note	(Single-argument) <code>create</code> is a part of CodeGenerator Contract .
------	---

JdbcDialect

isCascadingTruncateTable Method

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

getTableExistsQuery Method

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

KafkaWriter — Writing Dataset to Kafka

`KafkaWriter` is used to [write](#) the result of a batch or structured streaming query to Apache Kafka (with a new execution id attached so you can see the execution in web UI's SQL tab).

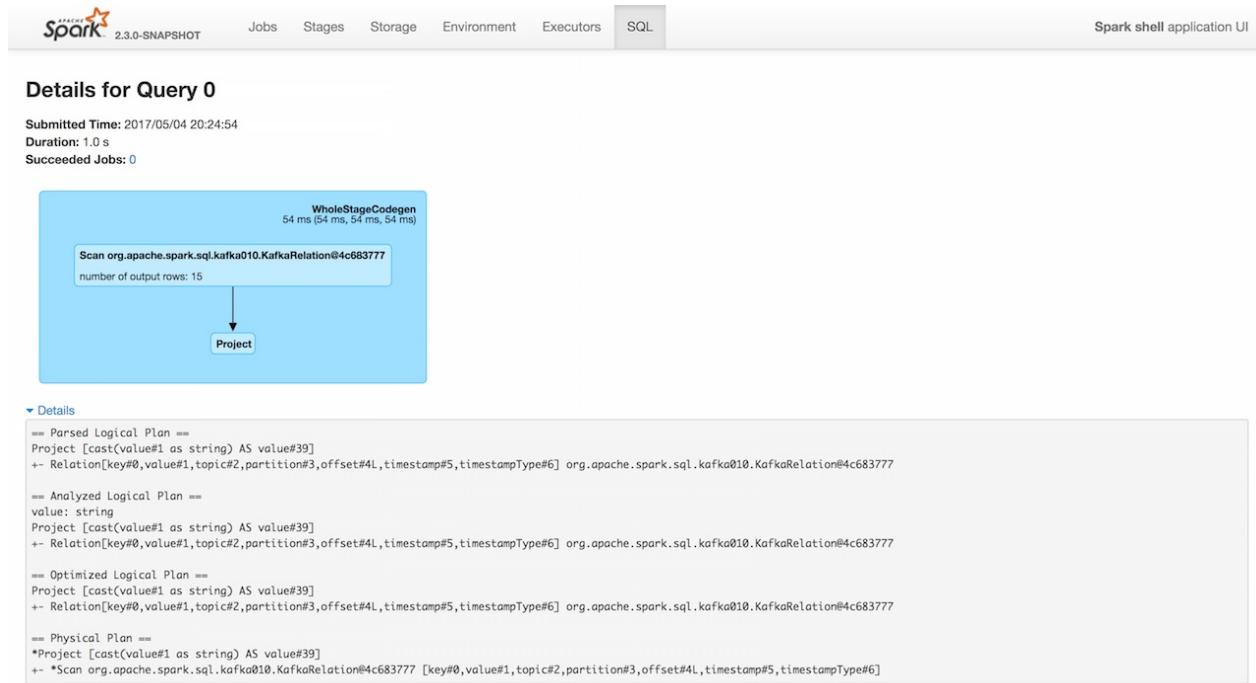


Figure 1. KafkaWriter (write) in web UI

`KafkaWriter` makes sure that the schema of the `Dataset` to write records of contains:

1. Required **topic** as a field of type `StringType` or specified explicitly
2. Required **value** as a field of type `StringType` or `BinaryType`
3. Optional **key** as a field of type `StringType` or `BinaryType`

```
// KafkaWriter is a private `kafka010` package object
// and so the code to use it should also be in the same package
// BEGIN: Use `:paste -raw` in spark-shell
package org.apache.spark.sql.kafka010

object PublicKafkaWriter {
    import org.apache.spark.sql.execution.QueryExecution
    def validateQuery(
        queryExecution: QueryExecution,
        kafkaParameters: Map[String, Object],
        topic: Option[String] = None): Unit = {
        import scala.collection.JavaConversions.mapAsJavaMap
        KafkaWriter.validateQuery(queryExecution, kafkaParameters, topic)
    }
}
// END

import org.apache.spark.sql.kafka010.{PublicKafkaWriter => PKW}

val spark: SparkSession = ...
val q = spark.range(1).select('id)
scala> PKW.validateQuery(
    queryExecution = q.queryExecution,
    kafkaParameters = Map.empty[String, Object])
org.apache.spark.sql.AnalysisException: topic option required when no 'topic' attribute is present. Use the topic option for setting a topic.;
  at org.apache.spark.sql.kafka010.KafkaWriter$$anonfun$2.apply(KafkaWriter.scala:53)
  at org.apache.spark.sql.kafka010.KafkaWriter$$anonfun$2.apply(KafkaWriter.scala:52)
  at scala.Option.getOrElse(Option.scala:121)
  at org.apache.spark.sql.kafka010.KafkaWriter$.validateQuery(KafkaWriter.scala:51)
  at org.apache.spark.sql.kafka010.PublicKafkaWriter$.validateQuery(<pastie>:10)
  ... 50 elided
```

Writing Query Results to Kafka — `write` Method

```
write(
  sparkSession: SparkSession,
  queryExecution: QueryExecution,
  kafkaParameters: ju.Map[String, Object],
  topic: Option[String] = None): Unit
```

`write` creates and executes a [KafkaWriteTask](#) per partition of the `QueryExecution`'s `RDD` ([with a new execution id](#) attached so you can see the execution in web UI's [SQL tab](#)).

Note

`write` is used when:

- `KafkaSourceProvider` creates a `BaseRelation` (after writing the result of a structure query)
- Structured Streaming's `KafkaSink` commits a batch

Validating QueryExecution — `validateQuery` Method

```
validateQuery(  
    queryExecution: QueryExecution,  
    kafkaParameters: java.util.Map[String, Object],  
    topic: Option[String] = None): Unit
```

`validateQuery` validates the schema of the input `analyzed` `QueryExecution`, i.e.

1. Whether the required **topic** is available as a field of type `StringType` in the schema or as the input `topic`
2. Whether the optional **key** is available as a field of type `stringType` or `BinaryType` in the schema
3. Whether the required **value** is available as a field of type `StringType` or `BinaryType` in the schema

Note

`validateQuery` is used exclusively when `KafkaWriter` writes the result of a query to Kafka.

KafkaSourceProvider

`KafkaSourceProvider` is an [interface to register](#) Apache Kafka as a data source.

`KafkaSourceProvider` is a [CreatableRelationProvider](#) and [RelationProvider](#).

`KafkaSourceProvider` is registered under `kafka` alias.

```
// start Spark application like spark-shell with the following package
// --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0-SNAPSHOT
scala> val fromKafkaTopic1 = spark.
  read.
  format("kafka").
  option("subscribe", "topic1"). // subscribe, subscribepattern, or assign
  option("kafka.bootstrap.servers", "localhost:9092").
  load("gauge_one")
```

`KafkaSourceProvider` uses a fixed schema (and makes sure that a user did not set a custom one).

```
import org.apache.spark.sql.types.StructType
val schema = new StructType().add($"id".int)
scala> spark
  .read
  .format("kafka")
  .option("subscribe", "topic1")
  .option("kafka.bootstrap.servers", "localhost:9092")
  .schema(schema) // <-- defining a custom schema is not supported
  .load
org.apache.spark.sql.AnalysisException: kafka does not allow user-specified schemas.;
  at org.apache.spark.sql.execution.datasources.DataSource.resolveRelation(DataSource.
scala:307)
  at org.apache.spark.sql.DataFrameReader.load(DataFrameReader.scala:178)
  at org.apache.spark.sql.DataFrameReader.load(DataFrameReader.scala:146)
... 48 elided
```

createRelation Method (from RelationProvider)

```
createRelation(
  sqlContext: SQLContext,
  parameters: Map[String, String]): BaseRelation
```

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

Note	<code>createRelation</code> is a part of RelationProvider Contract .
------	--

createRelation Method (from CreatableRelationProvider)

```
createRelation(  
    sqlContext: SQLContext,  
    mode: SaveMode,  
    parameters: Map[String, String],  
    df: DataFrame): BaseRelation
```

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

Note	<code>createRelation</code> is a part of CreatableRelationProvider Contract .
------	---

createSource Method

```
createSource(  
    sqlContext: SQLContext,  
    metadataPath: String,  
    schema: Option[StructType],  
    providerName: String,  
    parameters: Map[String, String]): Source
```

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

Note	<code>createSource</code> is a part of Structured Streaming's streamSourceProvider Contract .
------	---

sourceSchema Method

```
sourceSchema(  
    sqlContext: SQLContext,  
    schema: Option[StructType],  
    providerName: String,  
    parameters: Map[String, String]): (String, StructType)
```

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

```
val fromKafka = spark.read.format("kafka")...
scala> fromKafka.printSchema
root
|-- key: binary (nullable = true)
|-- value: binary (nullable = true)
|-- topic: string (nullable = true)
|-- partition: integer (nullable = true)
|-- offset: long (nullable = true)
|-- timestamp: timestamp (nullable = true)
|-- timestampType: integer (nullable = true)
```

Note

`sourceSchema` is a part of Structured Streaming's `StreamSourceProvider` Contract.

KafkaWriteTask

`KafkaWriteTask` is used to [write rows](#) (from a structured query) to Apache Kafka.

`KafkaWriteTask` is used exclusively when `KafkaWriter` is requested to [write query results to Kafka](#) (and creates one per partition).

`KafkaWriteTask` [writes](#) keys and values in their binary format (as JVM's bytes) and so uses the [raw-memory unsafe row format](#) only (i.e. `UnsafeRow`). That is supposed to save time for reconstructing the rows to very tiny JVM objects (i.e. byte arrays).

Table 1. KafkaWriteTask's Internal Properties (in alphabetical order)

Name	Description
<code>projection</code>	UnsafeProjection Created once when <code>KafkaWriteTask</code> is created.

Sending Rows to Kafka Asynchronously — `execute` Method

```
execute(iterator: Iterator[InternalRow]): Unit
```

`execute` uses Apache Kafka's Producer API to create a `KafkaProducer` and `ProducerRecord` for every row in `iterator`, and sends the rows to Kafka in batches asynchronously.

Internally, `execute` creates a `KafkaProducer` using `Array[Byte]` for the keys and values, and `producerConfiguration` for the producer's configuration.

Note	<code>execute</code> creates a single <code>KafkaProducer</code> for all rows.
------	--

For every row in the `iterator`, `execute` uses the internal [UnsafeProjection](#) to *project* (aka *convert*) [binary internal row format](#) to a `UnsafeRow` object and take 0th, 1st and 2nd fields for a topic, key and value, respectively.

`execute` then creates a `ProducerRecord` and sends it to Kafka (using the `KafkaProducer`). `execute` registers a asynchronous `callback` to monitor the writing.

Note

From [KafkaProducer's documentation](#):

The `send()` method is asynchronous. When called it adds the record to a buffer of pending record sends and immediately returns. This allows the producer to batch together individual records for efficiency.

Creating UnsafeProjection — `createProjection` Internal Method

```
createProjection: UnsafeProjection
```

`createProjection` creates a [UnsafeProjection](#) with `topic`, `key` and `value` [expressions](#) and the `inputSchema`.

`createProjection` makes sure that the following holds (and reports an `IllegalStateException` otherwise):

- `topic` was defined (either as the input `topic` or in `inputSchema`) and is of type `StringType`
- Optional `key` is of type `StringType` or `BinaryType` if defined
- `value` was defined (in `inputSchema`) and is of type `StringType` or `BinaryType`

`createProjection` casts `key` and `value` [expressions](#) to `BinaryType` in [UnsafeProjection](#).

Note

`createProjection` is used exclusively when `KafkaWriteTask` is created (as [projection](#)).

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
```

Experimental: Metastore Tables with non-Hive SerDe

Caution

FIXME Review the uses of `convertMetastoreParquet`,
`convertMetastoreParquetWithSchemaMerging`, `convertMetastoreOrc`,
`convertCTAS`.

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	FIXME Review <code>HiveContext</code> object.
---------	---

Spark SQL CLI - spark-sql

Caution

FIXME

Tip

Read about Spark SQL CLI in Spark's official documentation in [Running the Spark SQL CLI](#).

```
spark-sql> describe function `<>`;
Function: <>
Usage: a <> b - Returns TRUE if a is not equal to b
```

Tip

Functions are registered in [FunctionRegistry](#).

```
spark-sql> show functions;
```

```
spark-sql> explain extended show tables;
```

DataSinks

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

CacheManager—In-Memory Cache for Tables and Views

`CacheManager` is an in-memory cache for tables and views (as [logical plans](#)). It uses the internal `cachedData` collection of [CachedData](#) to track logical plans and their cached [InMemoryRelation](#) representation.

`CacheManager` is shared across `SparkSessions` through [SharedState](#).

```
sparkSession.sharedState.cacheManager
```

Note	A Spark developer can use <code>CacheManager</code> to cache <code>Dataset</code> s using cache or persist operators.
------	---

Cached Queries—`cachedData` Internal Registry

`cachedData` is a collection of [CachedData](#) with [logical plans](#) and their cached [InMemoryRelation](#) representation.

A new [CachedData](#) is added when a [Dataset](#) is [cached](#) and removed when a [Dataset](#) is [uncached](#) or when [invalidating cache data with a resource path](#).

`cachedData` is [cleared](#) when...[FIXME](#)

`invalidateCachedPath` Method

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

`invalidateCache` Method

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

`lookupCachedData` Method

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

`uncacheQuery` Method

Caution

FIXME

isEmpty Method

Caution

FIXME

Caching Dataset (by Registering Logical Plan as InMemoryRelation)-- cacheQuery Method

```
cacheQuery(
    query: Dataset[_],
    tableName: Option[String] = None,
    storageLevel: StorageLevel = MEMORY_AND_DISK): Unit
```

Internally, `cacheQuery` registers [logical plan](#) of the input `query` in `cachedData` internal registry of cached queries.

While registering, `cacheQuery` creates a [InMemoryRelation](#) with the following properties:

- `spark.sql.inMemoryColumnarStorage.compressed` (enabled by default)
- `spark.sql.inMemoryColumnarStorage.batchSize` (default: `10000`)
- `MEMORY_AND_DISK` [storage level](#)
- Logical query plan ready for execution (aka [executedPlan](#))
- Input `tableName`

If however the input `query` has already been cached, `cacheQuery` simply prints the following `WARN` message to the logs:

```
WARN CacheManager: Asked to cache already cached data.
```

Note

`cacheQuery` is used when:

- `Dataset`'s [persist](#) operator executes
- `CatalogImpl` is requested to [cache a table or view in-memory](#) or [refreshTable](#)

Removing All Cached Tables From In-Memory Cache — clearCache Method

```
clearCache(): Unit
```

`clearCache` acquires a write lock and unpersists `RDD[CachedBatch]`s of the queries in `cachedData` before removing them altogether.

Note

`clearCache` is executed when the [CatalogImpl](#) is requested to `clearCache`.

CachedData

Caution

[FIXME](#)

Thrift JDBC/ODBC Server — Spark Thrift Server (STS)

Thrift JDBC/ODBC Server (aka *Spark Thrift Server* or *STS*) is Spark SQL's port of [Apache Hive's HiveServer2](#) that allows JDBC/ODBC clients to execute SQL queries over JDBC and ODBC protocols on Apache Spark.

With Spark Thrift Server, business users can work with their shiny Business Intelligence (BI) tools, e.g. [Tableau](#) or Microsoft Excel, and connect to Apache Spark using the ODBC interface. That brings the in-memory distributed capabilities of Spark SQL's query engine (with all the [Catalyst query optimizations](#) you surely like very much) to environments that were initially "disconnected".

Beside, SQL queries in Spark Thrift Server share the same [SparkContext](#) that helps further improve performance of SQL queries using the same data sources.

Spark Thrift Server is a Spark standalone application that you start using [start-thriftserver.sh](#) and stop using [stop-thriftserver.sh](#) shell scripts.

Spark Thrift Server has its own tab in web UI — [JDBC/ODBC Server](#) available at `/sqlserver` URL.

The screenshot shows the Apache Spark 2.1.0-SNAPSHOT web interface. The top navigation bar includes links for Jobs, Stages, Storage, Environment, Executors, SQL, and JDBC/ODBC Server. The JDBC/ODBC Server tab is currently active. Below the tabs, the title 'JDBC/ODBC Server' is displayed. A status message indicates it was started at 2016/09/19 17:16:57 and has been running for 4 hours 8 minutes 18 seconds. A summary states '1 session(s) are online, running 0 SQL statement(s)'. Under 'Session Statistics', there is a table with one row for user 'jacek' with IP '127.0.0.1'. The 'SQL Statistics' section shows a single completed statement for user 'jacek' with ID '43523612-4564-45ad-a9f4-d5970433cc14', starting at 17:43:05 and finishing at 17:43:08, with a duration of 3 seconds 439 ms, labeled as 'FINISHED'.

User	IP	Session ID	Start Time	Finish Time	Duration	Total Execute
jacek	127.0.0.1	b8446967-2a00-49ec-b9bb-518b2dd8cc14	2016/09/19 17:18:41		4 hours 6 minutes 35 seconds	1

User	JobID	GroupID	Start Time	Finish Time	Duration	Statement	State	Detail
jacek	43523612-4564-45ad-a9f4-d5970433cc14		2016/09/19 17:43:05	2016/09/19 17:43:08	3 seconds 439 ms	show tables	FINISHED	== Parsed Logical Plan == + details

Figure 1. Spark Thrift Server's web UI

Spark Thrift Server can work in [HTTP or binary transport modes](#).

Use [beeline command-line tool](#) or [SQlireL SQL Client](#) or Spark SQL's [DataSource API](#) to connect to Spark Thrift Server through the JDBC interface.

Spark Thrift Server extends [spark-submit](#)'s command-line options with `--hiveconf [prop=value]`.

	You have to enable <code>hive-thriftserver</code> build profile to include Spark Thrift Server build.
Important	<pre>./build/mvn -Phadoop-2.7,yarn,mesos,hive,hive-thriftserver -DskipTests clean package</pre> <p>Refer to Building Apache Spark from Sources.</p>
Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging levels for <code>org.apache.spark.sql.hive.thriftserver</code> and <code>org.apache.hive.service.server</code> loggers 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.hive.thriftserver=DEBUG log4j.logger.org.apache.hive.service.server=INFO</pre> <p>Refer to Logging.</p>

Starting Thrift JDBC/ODBC Server — `start-thriftserver.sh`

You can start Thrift JDBC/ODBC Server using `./sbin/start-thriftserver.sh` shell script.

With `INFO` logging level enabled, when you execute the script you should see the following INFO messages in the logs:

```
INFO HiveThriftServer2: Started daemon with process name: 16633@japila.local
INFO HiveThriftServer2: Starting SparkContext
...
INFO HiveThriftServer2: HiveThriftServer2 started
```

Internally, `start-thriftserver.sh` script submits

`org.apache.spark.sql.hive.thriftserver.HiveThriftServer2` standalone application for execution (using [spark-submit](#)).

```
$ ./bin/spark-submit --class org.apache.spark.sql.hive.thriftserver.HiveThriftServer2
```

Tip	Using the more explicit approach with <code>spark-submit</code> to start Spark Thrift Server could be easier to trace execution by seeing the logs printed out to the standard output and hence terminal directly.
-----	--

Using Beeline JDBC Client to Connect to Spark Thrift Server

`beeline` is a command-line tool that allows you to access Spark Thrift Server using the JDBC interface on command line. It is included in the Spark distribution in `bin` directory.

```
$ ./bin/beeline  
Beeline version 1.2.1.spark2 by Apache Hive  
beeline>
```

You can connect to Spark Thrift Server using `connect` command as follows:

```
beeline> !connect jdbc:hive2://localhost:10000
```

When connecting in non-secure mode, simply enter the username on your machine and a blank password.

```
beeline> !connect jdbc:hive2://localhost:10000  
Connecting to jdbc:hive2://localhost:10000  
Enter username for jdbc:hive2://localhost:10000: jacek  
Enter password for jdbc:hive2://localhost:10000: [press ENTER]  
Connected to: Spark SQL (version 2.1.0-SNAPSHOT)  
Driver: Hive JDBC (version 1.2.1.spark2)  
Transaction isolation: TRANSACTION_REPEATABLE_READ  
0: jdbc:hive2://localhost:10000>
```

Once connected, you can send SQL queries (as if Spark SQL were a JDBC-compliant database).

```
0: jdbc:hive2://localhost:10000> show databases;  
+-----+  
| databaseName |  
+-----+  
| default     |  
+-----+  
1 row selected (0.074 seconds)
```

Connecting to Spark Thrift Server using SQuirreL SQL Client 3.7.1

Spark Thrift Server allows for remote access to Spark SQL using JDBC protocol.

Note	This section was tested with SQuirreL SQL Client 3.7.1 (<code>squirrelsql-3.7.1-standard.zip</code>) on Mac OS X.
------	---

SQuirreL SQL Client is a Java SQL client for JDBC-compliant databases.

Run the client using `java -jar squirrel-sql.jar`.

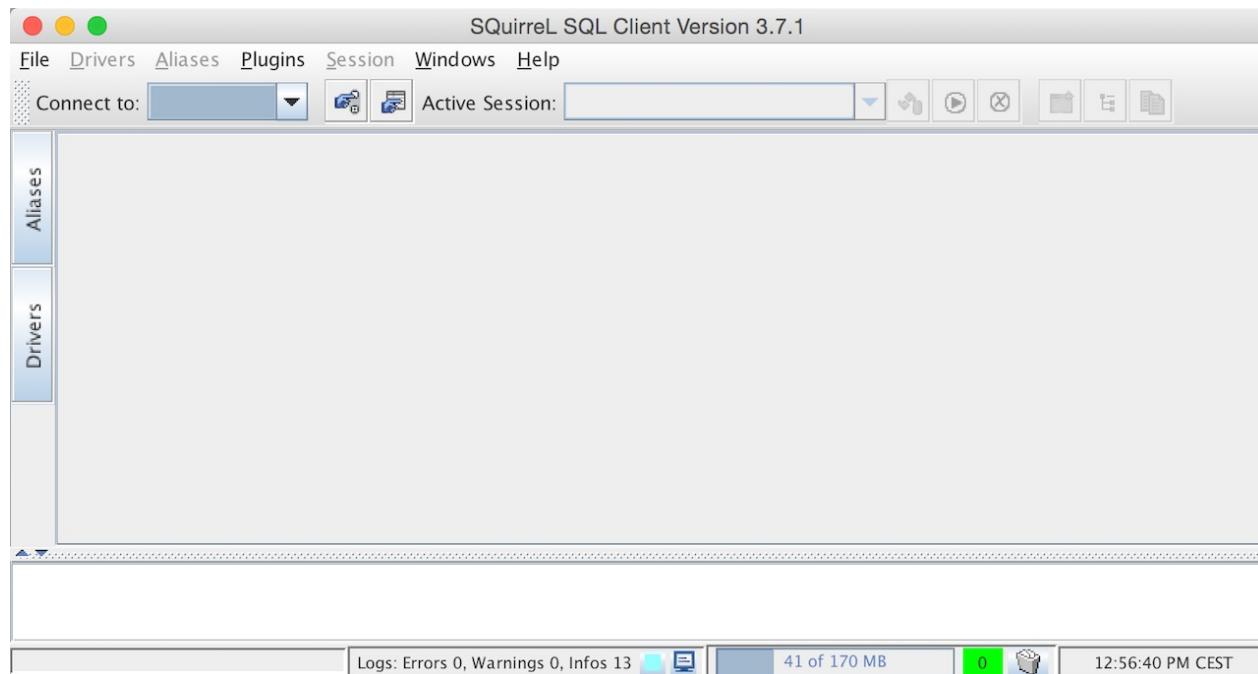


Figure 2. SQuirreL SQL Client

You first have to configure a JDBC driver for Spark Thrift Server. Spark Thrift Server uses `org.spark-project.hive:hive-jdbc:1.2.1.spark2` dependency that is the JDBC driver (that also downloads transitive dependencies).

Tip

The Hive JDBC Driver, i.e. `hive-jdbc-1.2.1.spark2.jar` and other jar files are in `jars` directory of the Apache Spark distribution (or `assembly/target/scala-2.11/jars` for local builds).

Table 1. SQuirreL SQL Client's Connection Parameters

Parameter	Description
Name	Spark Thrift Server
Example URL	<code>jdbc:hive2://localhost:10000</code>
Extra Class Path	All the jar files of your Spark distribution
Class Name	<code>org.apache.hive.jdbc.HiveDriver</code>

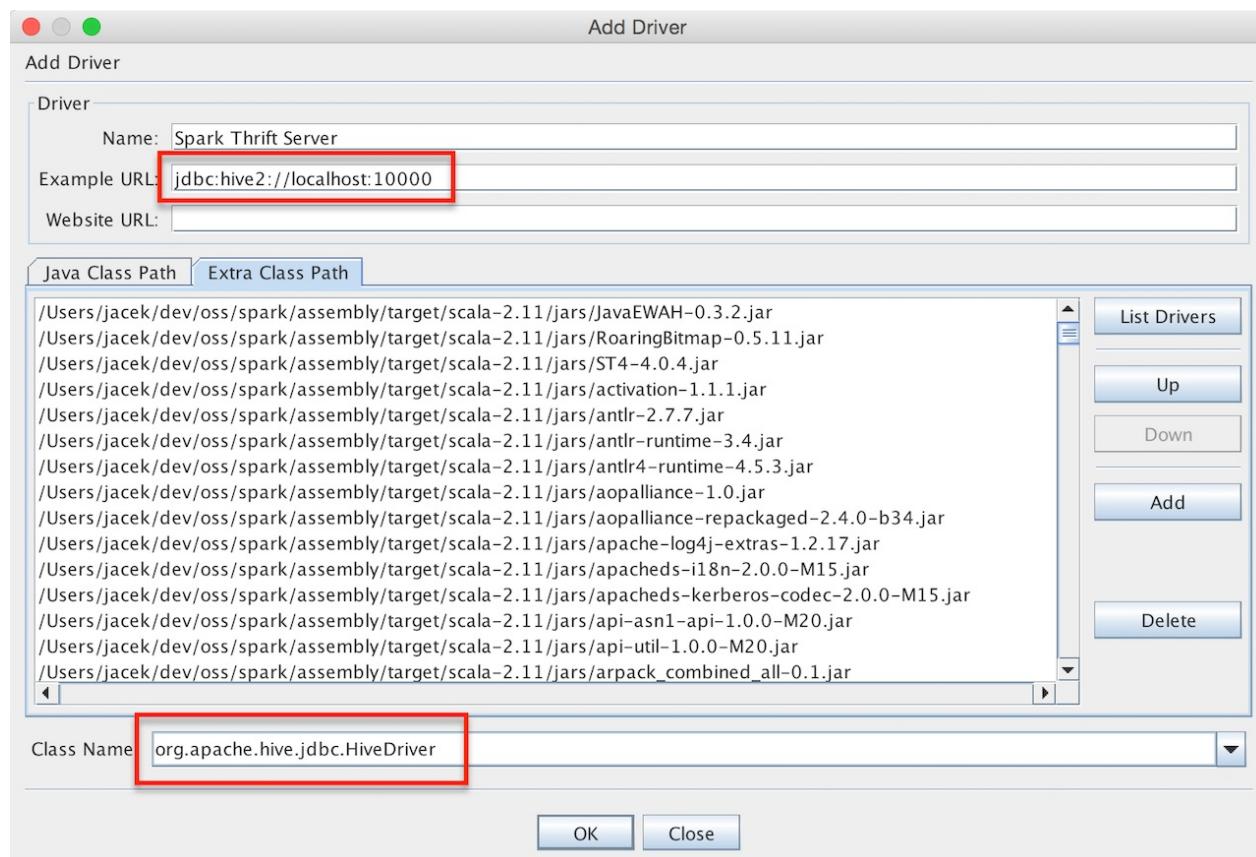


Figure 3. Adding Hive JDBC Driver in SQuirreL SQL Client

With the Hive JDBC Driver defined, you can connect to Spark SQL Thrift Server.

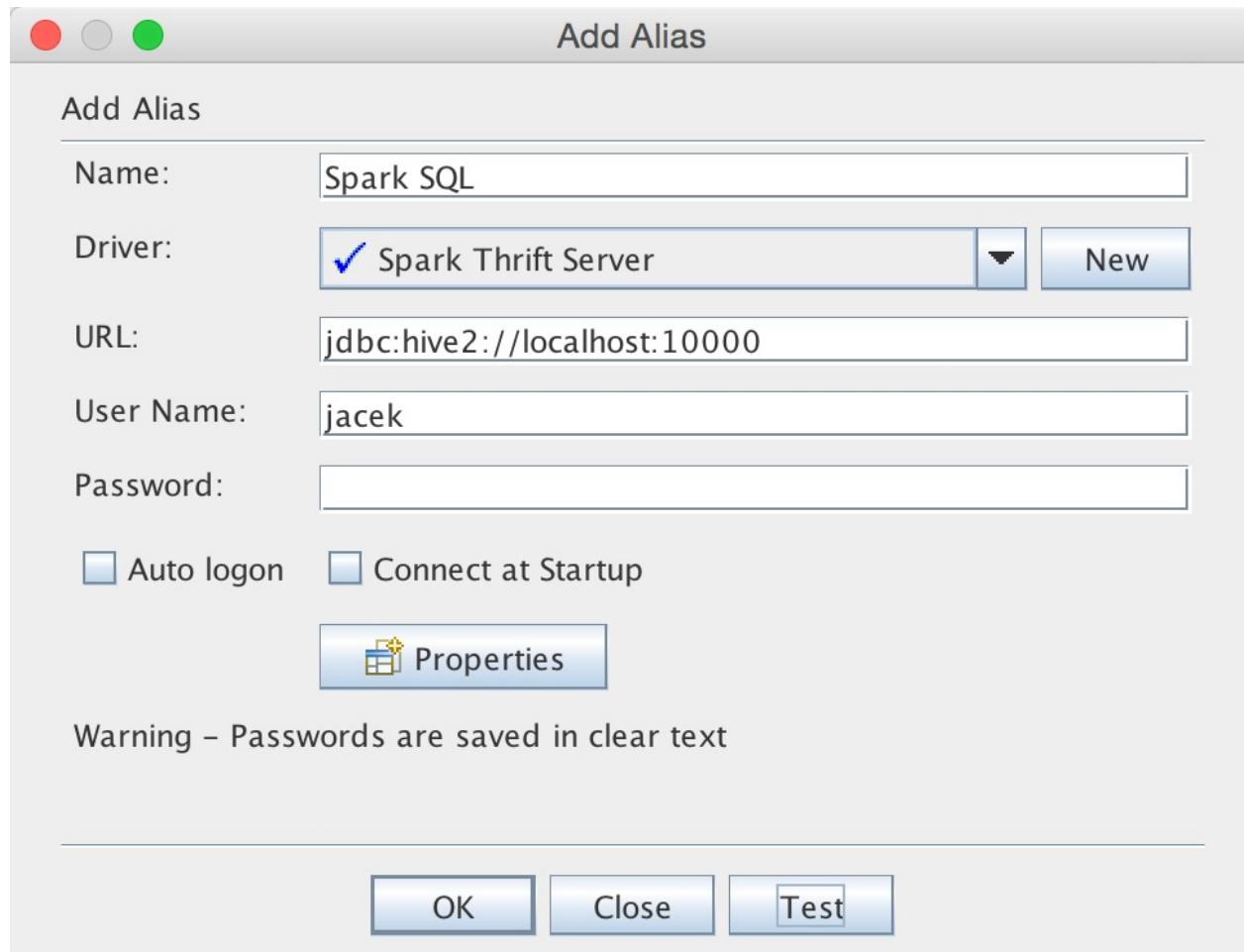


Figure 4. Adding Hive JDBC Driver in SQuirreL SQL Client

Since you did not specify the database to use, Spark SQL's `default` is used.

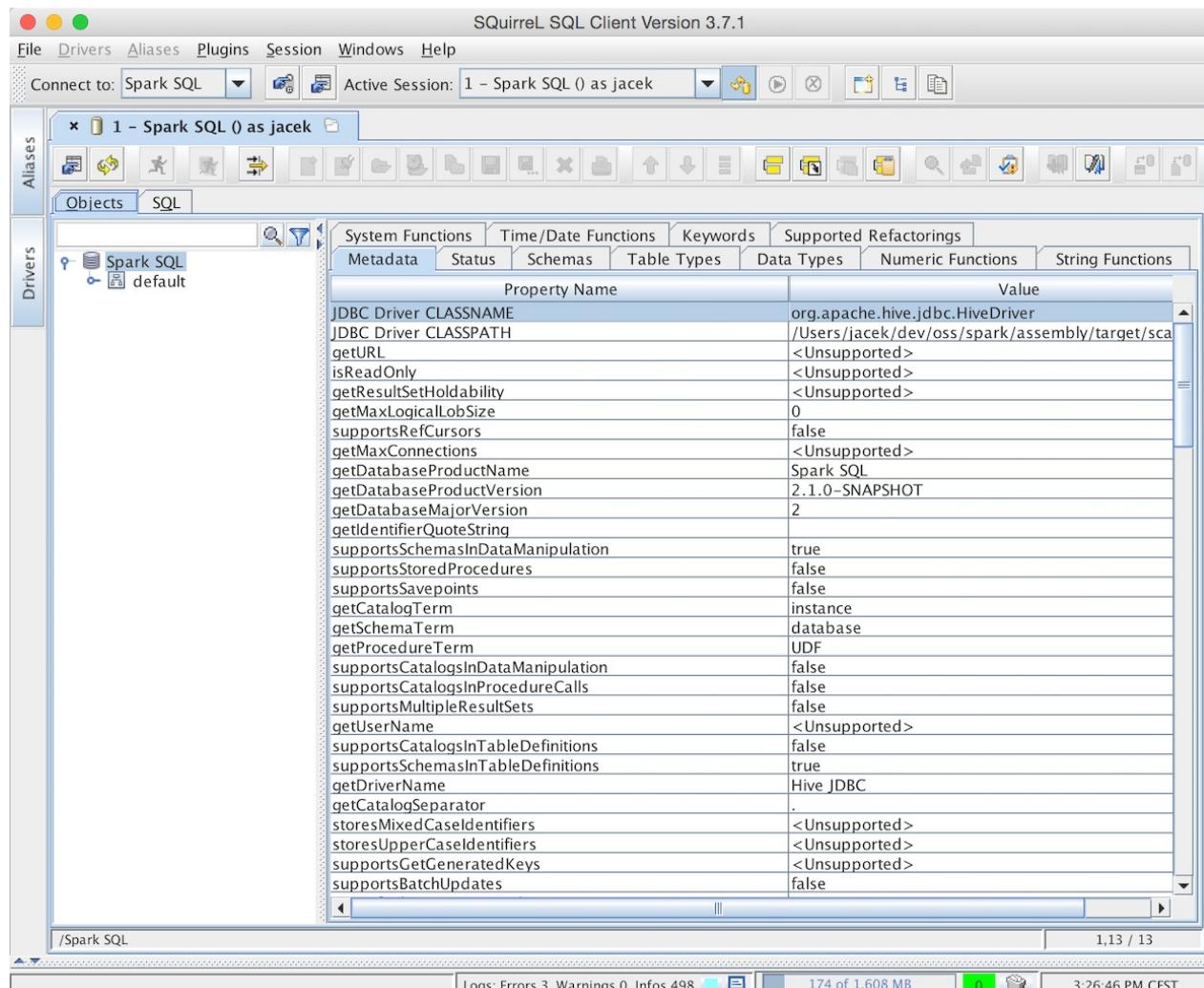
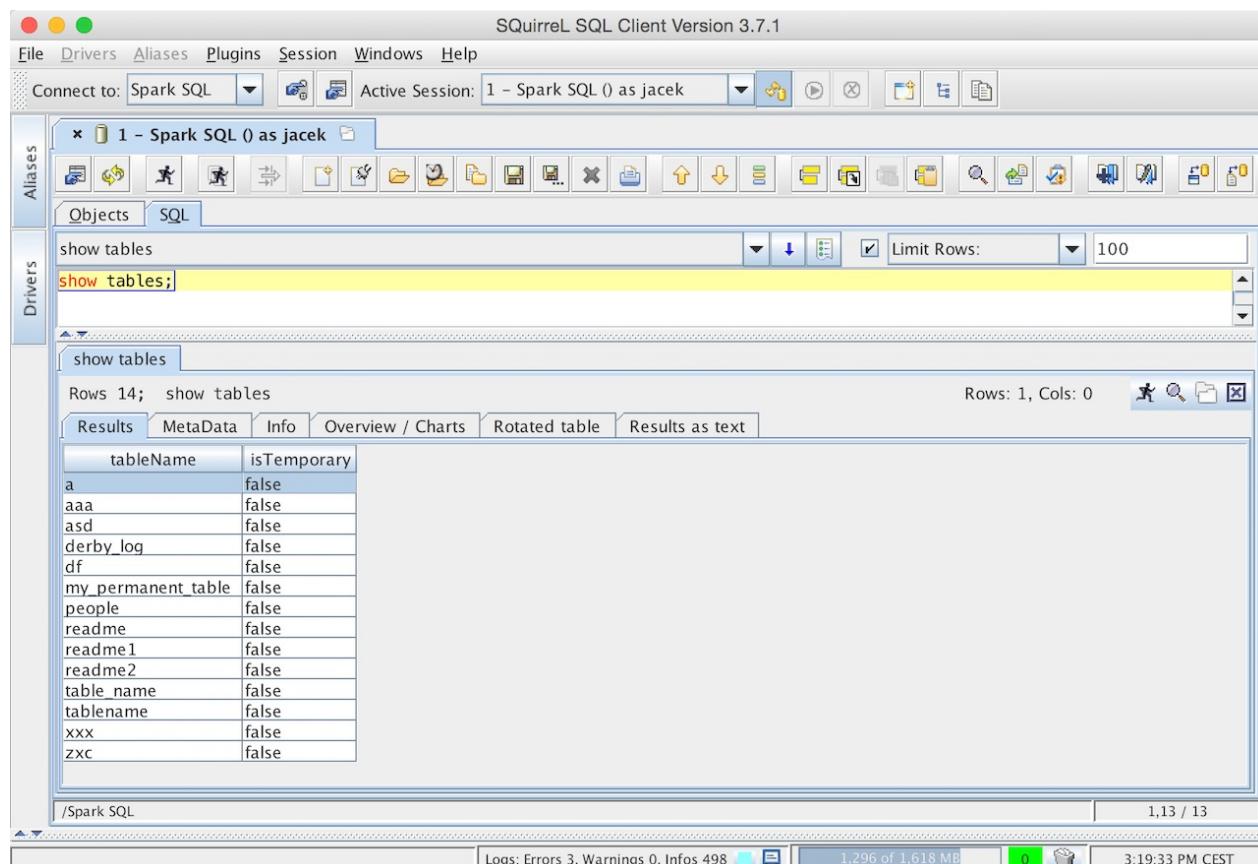


Figure 5. SQuirreL SQL Client Connected to Spark Thrift Server (Metadata Tab)

Below is `show tables` SQL query in SQuirreL SQL Client executed in Spark SQL through Spark Thrift Server.

Figure 6. `show tables` SQL Query in SQuirreL SQL Client using Spark Thrift Server

Using Spark SQL's DataSource API to Connect to Spark Thrift Server

What might seem a quite artificial setup at first is accessing Spark Thrift Server using Spark SQL's [DataSource API](#), i.e. `DataFrameReader`'s [jdbc method](#).

When executed in `local` mode, Spark Thrift Server and `spark-shell` will try to access the same Hive Warehouse's directory that will inevitably lead to an error.

Use `spark.sql.warehouse.dir` to point to another directory for `spark-shell`.

Tip

```
./bin/spark-shell --conf spark.sql.warehouse.dir=/tmp/spark-warehouse
```

You should also not share the same home directory between them since `metastore_db` becomes an issue.

```
// Inside spark-shell
// Paste in :paste mode
val df = spark
  .read
  .option("url", "jdbc:hive2://localhost:10000") (1)
  .option("dbtable", "people") (2)
  .format("jdbc")
  .load
```

1. Connect to Spark Thrift Server at localhost on port 10000
2. Use `people` table. It assumes that `people` table is available.

ThriftServerTab — web UI's Tab for Spark Thrift Server

ThriftServerTab is... [FIXME](#)

Caution	FIXME Elaborate
---------	---------------------------------

Stopping Thrift JDBC/ODBC Server— `stop-thriftserver.sh`

You can stop a running instance of Thrift JDBC/ODBC Server using `./sbin/stop-thriftserver.sh` shell script.

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

```
ERROR HiveThriftServer2: RECEIVED SIGNAL TERM
DEBUG SparkSQLEnv: Shutting down Spark SQL Environment
INFO HiveServer2: Shutting down HiveServer2
INFO BlockManager: BlockManager stopped
INFO SparkContext: Successfully stopped SparkContext
```

Tip	You can also send <code>SIGTERM</code> signal to the process of Thrift JDBC/ODBC Server, i.e. <code>kill [PID]</code> that triggers the same sequence of shutdown steps as <code>stop-thriftserver.sh</code> .
-----	--

Transport Mode

Spark Thrift Server can be configured to listen in two modes (aka *transport modes*):

1. **Binary mode**—clients should send thrift requests in binary
2. **HTTP mode**—clients send thrift requests over HTTP.

You can control the transport modes using `HIVE_SERVER2_TRANSPORT_MODE=http` or `hive.server2.transport.mode` (default: `binary`). It can be `binary` (default) or `http`.

main method

Thrift JDBC/ODBC Server is a Spark standalone application that you...

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

HiveThriftServer2Listener

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

SparkSQLEnv

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

SQLContext

Caution

As of Spark **2.0.0** `SQLContext` is only for backward compatibility and is a *mere wrapper of [SparkSession](#)*.

In the pre-Spark 2.0's ear, **SQLContext** was the entry point for Spark SQL. Whatever you did in Spark SQL it had 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	FIXME 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	FIXME 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

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

<code>streams: StreamingQueryManager</code>	
---	--

The `streams` method returns a [StreamingQueryManager](#) that is used to...TK

Caution

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

Managing Active SQLContext for JVM

<code>SQLContext.getOrCreate(sparkContext: SparkContext): SQLContext</code>	
---	--

`SQLContext.getOrCreate` method returns an active `SQLContext` object for the JVM or creates a new one using a given `sparkContext` .

Note

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.

<code>setActive(spark: SQLContext): Unit</code>	
<code>clearActive(): Unit</code>	

Executing SQL Queries

<code>sql(sqlText: String): DataFrame</code>	
--	--

`sql` executes the `sqlText` SQL query.

Note

Note	It supports Hive statements through HiveContext .
-------------	---

```

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> println(s"This is Spark \${sc.version}") This is Spark 2.0.0-SNAPSHOT scala> :imports 1) import spark.implicits._ (52 terms, 31 are implicit) 2) import spark.sql (1 terms) </pre>

Tip	You may also use spark-sql shell script to interact with Hive.
-----	--

Internally, it uses `SessionState.sqlParser.parsePlan(sql)` method to create a [LogicalPlan](#).

Caution

FIXME 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 [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

FIXME Why would I need that?

Settings

The following list are the settings used to configure Spark SQL applications.

You can set them in a [SparkSession](#) upon instantiation using [config](#) method.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = SparkSession.builder
  .master("local[*]")
  .appName("My Spark Application")
  .config("spark.sql.warehouse.dir", "c:/Temp") (1)
  .getOrCreate
```

1. Sets [spark.sql.warehouse.dir](#) for the Spark SQL session

Table 1. Spark SQL Properties (in alphabetical order)

Name	Default	Description
spark.sql.catalogImplementation	in-memory	<p>(internal) Selects the active catalog implementation from:</p> <ul style="list-style-type: none"> • <code>in-memory</code> • <code>hive</code> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> Tip Read ExternalCatalog — System Catalog of Permanent Entities. </div> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> Tip You can enable Hive support in a <code>SparkSession</code> using <code>enableHiveSupport</code> builder method. </div>
spark.sql.sources.default	parquet	<p>Defines the default data source to use for DataFrameReader.</p> <p>Used when:</p> <ul style="list-style-type: none"> • Reading (DataFrameWriter) or writing (DataFrameReader) datasets • Creating external table from a path (in <code>Catalog.createExternalTable</code>) • Reading (<code>DataStreamReader</code>) or writing (<code>DataStreamWriter</code>) in Structured Streaming

spark.sql.warehouse.dir

`spark.sql.warehouse.dir` (default: `${system:user.dir}/spark-warehouse`) is the default location of Hive warehouse directory (using Derby) with managed databases and tables.

See also the official [Hive Metastore Administration](#) document.

spark.sql.parquet.filterPushdown

`spark.sql.parquet.filterPushdown` (default: `true`) is a flag to control the [filter predicate push-down optimization](#) for data sources using parquet file format.

spark.sql.allowMultipleContexts

`spark.sql.allowMultipleContexts` (default: `true`) controls whether creating multiple SQLContexts/HiveContexts is allowed.

spark.sql.columnNameOfCorruptRecord

`spark.sql.columnNameOfCorruptRecord` ...[FIXME](#)

spark.sql.dialect

`spark.sql.dialect` - [FIXME](#)

spark.sql.streaming.checkpointLocation

`spark.sql.streaming.checkpointLocation` is the default location for storing checkpoint data for [continuously executing queries](#).

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 and PipelineStages (spark.ml)

ML Pipeline API (aka **Spark ML** or **spark.ml** due to the package the API lives in) lets Spark users quickly and easily assemble and configure practical distributed Machine Learning pipelines (aka workflows) by standardizing the APIs for different Machine Learning concepts.

Note

Both [scikit-learn](#) and [GraphLab](#) have the concept of **pipelines** built into their system.

The ML Pipeline API is a new [DataFrame](#)-based API developed under `org.apache.spark.ml` package and is the primary API for MLlib as of Spark 2.0.

Important

The previous RDD-based API under `org.apache.spark.mllib` package is in maintenance-only mode which means that it is still maintained with bug fixes but no new features are expected.

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).

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

While a *RDD computation* in Spark Core, a *Dataset manipulation* in Spark SQL, a *continuous DStream computation* in Spark Streaming are the main data abstractions a **ML Pipeline** is 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*)

Note	The Pipeline API lives under org.apache.spark.ml package.
------	---

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](#)
- (video) [Apache Spark MLlib 2.0 Preview: Data Science and Production](#) by Joseph K. Bradley (Databricks)

ML Pipeline Components — 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 UnaryTransformer 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 a string column 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.

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 [org.apache.spark.ml.feature.RegexTokenizer](#).

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`.

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 `setLowerCase` 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        world|[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|
+---+-----+
```

Tokenizer

`Tokenizer` is a [unary transformer](#) that converts the column of String values 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("id", "sentence")

val tokenized = tok.setInputCol("sentence").setOutputCol("tokens").transform(df)

scala> tokenized.show(truncate = false)
+---+-----+-----+
|id |sentence           |tokens          |
+---+-----+-----+
|1  |Hello world!       |[hello, world!] |
|2  |Here is yet another sentence.|[here, is, yet, another, sentence.]|
+---+-----+-----+
```

ML Pipeline Components — 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 Lloyd's algorithm 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

[FIXME](#)

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!"),
```


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|
+---+-----+-----+
```

ML Pipeline 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 Estimator) 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 marker no-method abstract class , it is more a PredictionModel .
------	--

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()

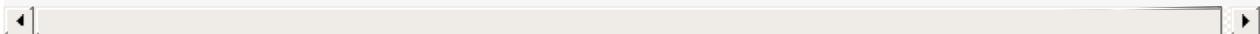
import org.apache.spark.ml.tuning.CrossValidator
val cv = new CrossValidator()
.setEstimator(lr) // just LogisticRegression not Pipeline
.setEvaluator(regEval)
.setEstimatorParamMaps(paramGrid)

// 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
```



Params (and ParamMaps)

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

ML Persistence — Saving and Loading Models and Pipelines

[MLWriter](#) and [MLReader](#) belong to `org.apache.spark.ml.util` package.

They allow you to save and load [models](#) despite the languages — Scala, Java, Python or R — they have been saved in and loaded later on.

MLWriter

`MLWriter` abstract class comes with `save(path: String)` method to save a ML 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.empty[PipelineStage])  
pipeline.write.overwrite.save("sample-pipeline")
```

The result of `save` for "unfitted" pipeline is a JSON file for metadata (as shown below).

```
$ cat sample-pipeline/metadata/part-00000 | jq
{
  "class": "org.apache.spark.ml.Pipeline",
  "timestamp": 1472747720477,
  "sparkVersion": "2.1.0-SNAPSHOT",
  "uid": "pipeline_181c90b15d65",
  "paramMap": {
    "stageUids": []
  }
}
```

The result of `save` for pipeline model is a JSON file for metadata while Parquet for model data, e.g. coefficients.

```
val model = pipeline.fit(training)
model.write.save("sample-model")
```

```
$ cat sample-model/metadata/part-00000 | jq
{
  "class": "org.apache.spark.ml.PipelineModel",
  "timestamp": 1472748168005,
  "sparkVersion": "2.1.0-SNAPSHOT",
  "uid": "pipeline_3ed598da1c4b",
  "paramMap": {
    "stageUids": [
      "regexTok_bf73e7c36e22",
      "hashingTF_ebece38da130",
      "logreg_819864aa7120"
    ]
  }
}

$ tree sample-model/stages/
sample-model/stages/
|-- 0_regexTok_bf73e7c36e22
|   '-- metadata
|       |-- _SUCCESS
|       '-- part-00000
|-- 1_hashingTF_ebece38da130
|   '-- metadata
|       |-- _SUCCESS
|       '-- part-00000
`-- 2_logreg_819864aa7120
    '-- data
        |-- _SUCCESS
        '-- part-r-00000-56423674-0208-4768-9d83-2e356ac6a8d2.snappy.parquet
    '-- metadata
        |-- _SUCCESS
        '-- part-00000

7 directories, 8 files
```

MLReader

`MLReader` abstract class comes with `load(path: String)` method to `load` a ML component from a given `path`.

```
import org.apache.spark.ml._  
val pipeline = Pipeline.read.load("sample-pipeline")  
  
scala> val stageCount = pipeline.getStages.size  
stageCount: Int = 0  
  
val pipelineModel = PipelineModel.read.load("sample-model")  
  
scala> pipelineModel.stages  
res1: Array[org.apache.spark.ml.Transformer] = Array(regexTok_bf73e7c36e22, hashingTF_  
ebece38da130, logreg_819864aa7120)
```

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	FIXME Review the available classes in the org.apache.spark.ml.tuning package .
---------	--

```

import org.apache.spark.ml.tuning.ParamGridBuilder
val paramGrid = new ParamGridBuilder()
  .addGrid(hashingTF.numFeatures, Array(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/sp>

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 <code>featuresCol</code> 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 Topic modeling with LDA: MLlib meets GraphX .
------	--

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	FIXME 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 \(HHMI Janelia Research Center\)](#)

GeneralizedLinearRegression (GLM)

`GeneralizedLinearRegression` is a regression algorithm. It supports the following error distribution families:

1. `gaussian`
2. `binomial`
3. `poisson`
4. `gamma`

`GeneralizedLinearRegression` supports the following relationship between the linear predictor and the mean of the distribution function links:

1. `identity`
2. `logit`
3. `log`
4. `inverse`
5. `probit`
6. `cloglog`
7. `sqrt`

`GeneralizedLinearRegression` supports 4096 features.

The label column has to be of `DoubleType` type.

Note

`GeneralizedLinearRegression` belongs to `org.apache.spark.ml.regression` package.

```
import org.apache.spark.ml.regression._
val glm = new GeneralizedLinearRegression()

import org.apache.spark.ml.linalg._
val features = Vectors.sparse(5, Seq((3,1.0)))
val trainDF = Seq((0, features, 1)).toDF("id", "features", "label")
val glmModel = glm.fit(trainDF)
```

`GeneralizedLinearRegression` is a [Regressor](#) with features of `Vector` type that can train a [GeneralizedLinearRegressionModel](#).

GeneralizedLinearRegressionModel

Regressor

Regressor is a custom Predictor.

Spark Structured Streaming — Streaming Datasets

Spark Structured Streaming is a new computation model introduced in Spark 2.0 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's engine) for continuous incremental execution of structured queries.

Tip

You can find more information about Spark Structured Streaming in my separate notebook titled [Spark Structured Streaming](#).

Spark Shell — spark-shell shell script

Spark shell is an interactive shell to learn how to make the most out of Apache Spark. This is a Spark application written in Scala to offer a command-line environment with auto-completion (under `TAB` key) where you can run ad-hoc queries and get familiar with the features of Spark (that help you in developing your own standalone Spark applications). It is a very convenient tool to explore the many things available in Spark with immediate feedback. It is one of the many reasons why [Spark is so helpful for tasks to process datasets of any size](#).

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

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

You can start Spark shell using `spark-shell` script.

```
$ ./bin/spark-shell  
scala>
```

`spark-shell` is an extension of Scala REPL with automatic instantiation of [SparkSession](#) as `spark` (and [SparkContext](#) as `sc`).

```
scala> :type spark  
org.apache.spark.sql.SparkSession  
  
// Learn the current version of Spark in use  
scala> spark.version  
res0: String = 2.1.0-SNAPSHOT
```

`spark-shell` also imports [Scala SQL's implicits](#) and `sql` method.

```
scala> :imports  
1) import spark.implicits._          (59 terms, 38 are implicit)  
2) import spark.sql                 (1 terms)
```

Note	<p>When you execute <code>spark-shell</code> you actually execute Spark submit as follows:</p>
	<pre>org.apache.spark.deploy.SparkSubmit --class org.apache.spark.repl.Main --name Spark shell spark- shell</pre>

Using Spark shell

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

```
$ ./bin/spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newL
evel).
WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using
  builtin-java classes where applicable
WARN ObjectStore: Failed to get database global_temp, returning NoSuchElementException
Spark context Web UI available at http://10.47.71.138:4040
Spark context available as 'sc' (master = local[*], app id = local-1477858597347).
Spark session available as 'spark'.
Welcome to

   __
  / _\|_ \  _ \  _ \ / /_ \
 _\ \V\ _ \V _ ` / _\ ' / \
/_\ / .__\_\_,/_\ / /_\ \  version 2.1.0-SNAPSHOT
  /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_112)
Type in expressions to have them evaluated.
Type :help for more information.
```

Spark shell creates an instance of `SparkSession` under the name `spark` for you (so you don't have to know the details how to do it yourself on day 1)

```
scala> :type spark
org.apache.spark.sql.SparkSession
```

Besides, there is also `sc` value created which is an instance of `SparkContext`.

```
scala> :type sc
org.apache.spark.SparkContext
```

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

```
scala> :q
```

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.repl.class.uri</code>	<code>null</code>	<p>Used in <code>spark-shell</code> to create REPL ClassLoader to load new classes defined in the Scala REPL as a user types code.</p> <p>Enable <code>INFO</code> logging level for <code>org.apache.spark.executor.Executor</code> logger to have the value printed out to the logs:</p> <pre>INFO Using REPL class URI: [classUri]</pre>

Web UI — Spark Application's Web Console

Web UI (aka **Application UI** or **webUI** or **Spark UI**) is the web interface of a running Spark application to monitor and inspect Spark job executions in a web browser.

The screenshot shows the Apache Spark 2.1.0-SNAPSHOT Web UI. At the top, there is a navigation bar with tabs: Jobs (which is selected), Stages, Storage, Environment, Executors, and SQL. To the right of the tabs, it says "Spark shell application UI".

Spark Jobs (?)

User: jacek
Total Uptime: 35 s
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 1
Failed Jobs: 1

▶ Event Timeline

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	show at <console>:24	2016/09/29 14:01:20	5 s	0/1	0/1

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	show at <console>:24	2016/09/29 14:01:07	0.3 s	1/1	1/1

Failed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	show at <console>:24	2016/09/29 14:01:14	87 ms	0/1 (1 failed)	0/1 (1 failed)

Figure 1. Welcome page - Jobs page

Every `SparkContext` launches its own instance of Web UI which is available at `http://[driver]:4040` by default (the port can be changed using `spark.ui.port` setting) and will increase if this port is already taken (until an open port is found).

web UI comes with the following tabs (which may not all be visible at once as they are lazily created on demand, e.g. `Streaming` tab):

1. [Jobs](#)
2. [Stages](#)
3. [Storage](#) with RDD size and memory use
4. [Environment](#)
5. [Executors](#)
6. [SQL](#)

Tip

You can use the web UI after the application has finished by persisting events using `EventLoggingListener` and using `Spark History Server`.

Note	All the information that is displayed in web UI is available thanks to JobProgressListener and other SparkListeners . One could say that web UI is a web layer to Spark listeners.
------	--

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.ui.enabled</code>	<code>true</code>	The flag to control whether the web UI is started (<code>true</code>) or not (<code>false</code>).
<code>spark.ui.port</code>	4040	The port web UI binds to. If multiple <code>SparkContext</code> s 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 <code>spark.ui.port</code> .
<code>spark.ui.killEnabled</code>	<code>true</code>	The flag to control whether you can kill stages in web UI (<code>true</code>) or not (<code>false</code>).
<code>spark.ui.retainedDeadExecutors</code>	100	The maximum number of entries in executorToTaskSummary (in <code>ExecutorsListener</code>) and deadExecutorStorageStatus (in <code>StorageStatusListener</code>) internal registries.

Jobs Tab

The **Jobs Tab** shows [status of all Spark jobs](#) in a Spark application (i.e. a [SparkContext](#)).



Spark Jobs (?)

User: jacek
 Total Uptime: 35 s
 Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 1
Failed Jobs: 1

[▶ Event Timeline](#)

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	show at <console>:24	2016/09/29 14:01:20	5 s	0/1	0/1

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	show at <console>:24	2016/09/29 14:01:07	0.3 s	1/1	1/1

Failed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	show at <console>:24	2016/09/29 14:01:14	87 ms	0/1 (1 failed)	0/1 (1 failed)

Figure 1. Jobs Tab

The Jobs tab is available under `/jobs` URL, i.e. <http://localhost:4040/jobs>.

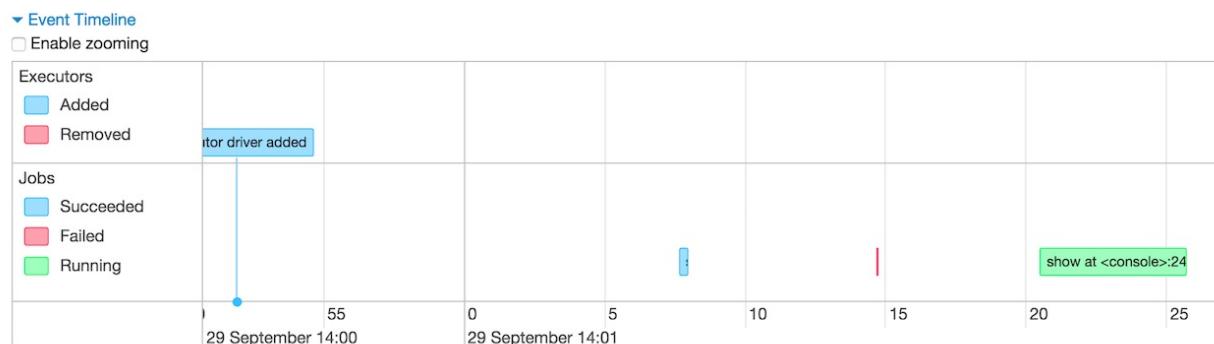


Figure 2. Event Timeline in Jobs Tab

The Jobs tab consists of two pages, i.e. [All Jobs](#) and [Details for Job](#) pages.

Internally, the Jobs Tab is represented by `JobsTab` class that is a custom `SparkUITab` with `jobs` prefix.

Note	The Jobs tab uses <code>JobProgressListener</code> to access statistics of job executions in a Spark application to display.
------	--

Showing All Jobs — [AllJobsPage](#) Page

AllJobsPage is a page (in [Jobs tab](#)) that renders a summary, an event timeline, and active, completed, and failed jobs of a Spark application.

Tip

Jobs (in any state) are displayed when their number is greater than 0.

AllJobsPage displays the **Summary** section with the [current Spark user](#), total uptime, scheduling mode, and the number of jobs per status.

Note

AllJobsPage uses [JobProgressListener](#) for Scheduling Mode.

Spark Jobs [\(?\)](#)

User: jacek

Total Uptime: 1.3 min

Scheduling Mode: FIFO

Active Jobs: 1

Completed Jobs: 1

Failed Jobs: 1

Figure 3. Summary Section in Jobs Tab

Under the summary section is the **Event Timeline** section.

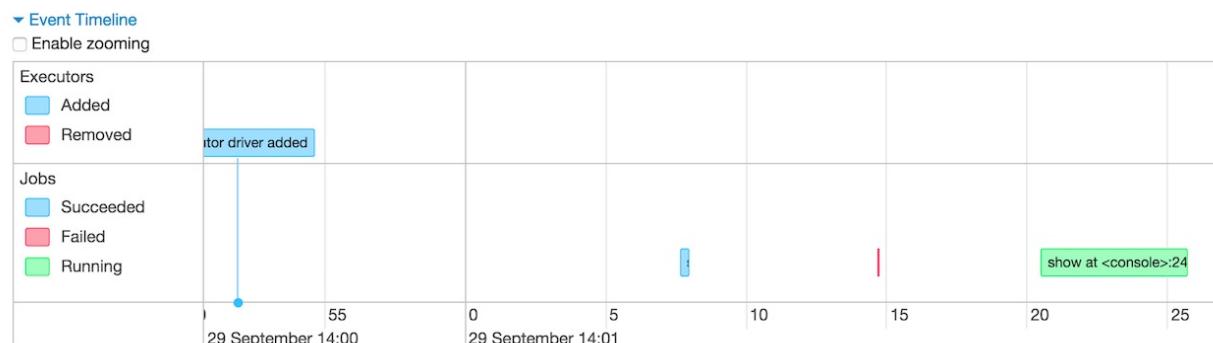


Figure 4. Event Timeline in Jobs Tab

Note

AllJobsPage uses [ExecutorsListener](#) to build the event timeline.

Active Jobs, Completed Jobs, and Failed Jobs sections follow.

Jobs Tab

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	show at <console>:24	2016/09/29 14:43:03	3 s	0/1	0/1

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	show at <console>:24	2016/09/29 14:42:09	0.4 s	1/1	1/1

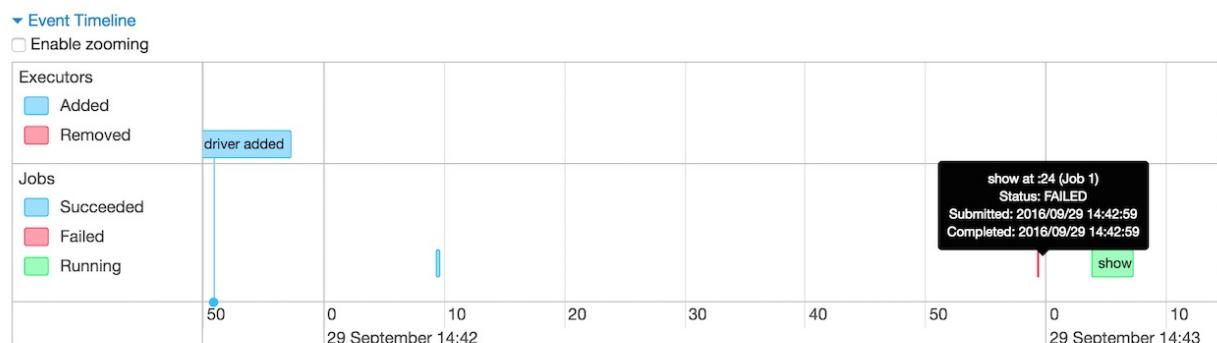
Failed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	show at <console>:24	2016/09/29 14:42:59	90 ms	0/1 (1 failed)	0/1 (1 failed)

Figure 5. Job Status Section in Jobs Tab

Jobs are clickable, i.e. you can click on a job to [see information about the stages of tasks inside it.](#)

When you hover over a job in Event Timeline not only you see the job legend but also the job is highlighted in the Summary section.



Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	show at <console>:24	2016/09/29 14:43:03	3 s	0/1	0/1

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	show at <console>:24	2016/09/29 14:42:09	0.4 s	1/1	1/1

Failed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	show at <console>:24	2016/09/29 14:42:59	90 ms	0/1 (1 failed)	0/1 (1 failed)

Figure 6. Hovering Over Job in Event Timeline Highlights The Job in Status Section
The Event Timeline section shows not only jobs but also executors.

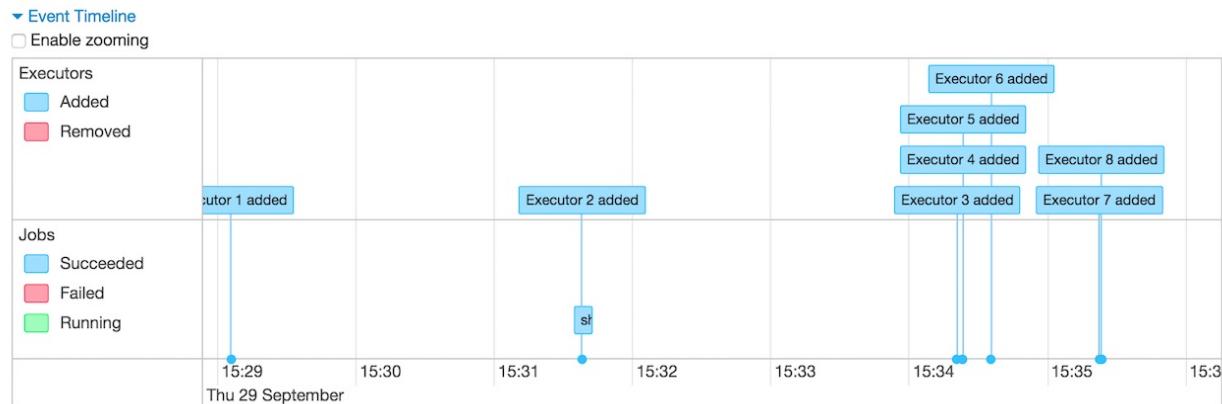


Figure 7. Executors in Event Timeline

Tip

Use [Programmable Dynamic Allocation](#) (using `SparkContext`) to manage executors for demo purposes.

Details for Job — JobPage Page

When you click a job in [AllJobsPage page](#), you see the **Details for Job** page.

The screenshot shows the Apache Spark 2.1.0-SNAPSHOT UI. The top navigation bar includes links for Jobs, Stages, Storage, Environment, Executors, SQL, and Spark shell application UI. The main content area is titled "Details for Job 0". It displays the status as "SUCCEEDED" and "Completed Stages: 1". Below this are links for "Event Timeline" and "DAG Visualization". The "DAG Visualization" section shows a Directed Acyclic Graph (DAG) for Stage 0. Stage 0 consists of two tasks: "WholeStageCodegen" and "mapPartitionsInternal", connected by a vertical arrow. The "Completed Stages (1)" table lists one stage:

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
0	show at <console>:24 +details	2016/09/29 17:24:15	0.2 s	1/1	3.7 KB			

Figure 8. Details for Job Page

`JobPage` is a custom `WebUIPage` that shows statistics and stage list for a given job.

Details for Job page is registered under `/job` URL, i.e. `http://localhost:4040/jobs/job/?id=0` and accepts one mandatory `id` request parameter as a job identifier.

When a job id is not found, you should see "No information to display for job ID" message.

Jobs Stages Storage Environment Executors SQL [Spark shell application UI](#)

Details for Job 2

No information to display for job 2

Figure 9. "No information to display for job" in Details for Job Page

JobPage displays the job's status, group (if available), and the stages per state: active, pending, completed, skipped, and failed.

Note

A job can be in a running, succeeded, failed or unknown state.

Details for Job 16

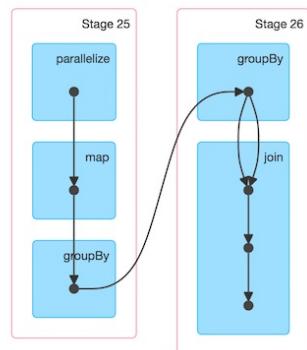
Status: RUNNING

Active Stages: 1

Pending Stages: 1

[Event Timeline](#)

[DAG Visualization](#)



Active Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
25	groupBy at <console>:24	+details (kill)	2016/09/29 17:54:04	4 s	2/8			

Pending Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
26	foreach at <console>:27	+details	Unknown	Unknown	0/8			

Figure 10. Details for Job Page with Active and Pending Stages

Details for Job 18

Status: SUCCEEDED

Completed Stages: 2

Skipped Stages: 2

- ▶ Event Timeline
- ▼ DAG Visualization

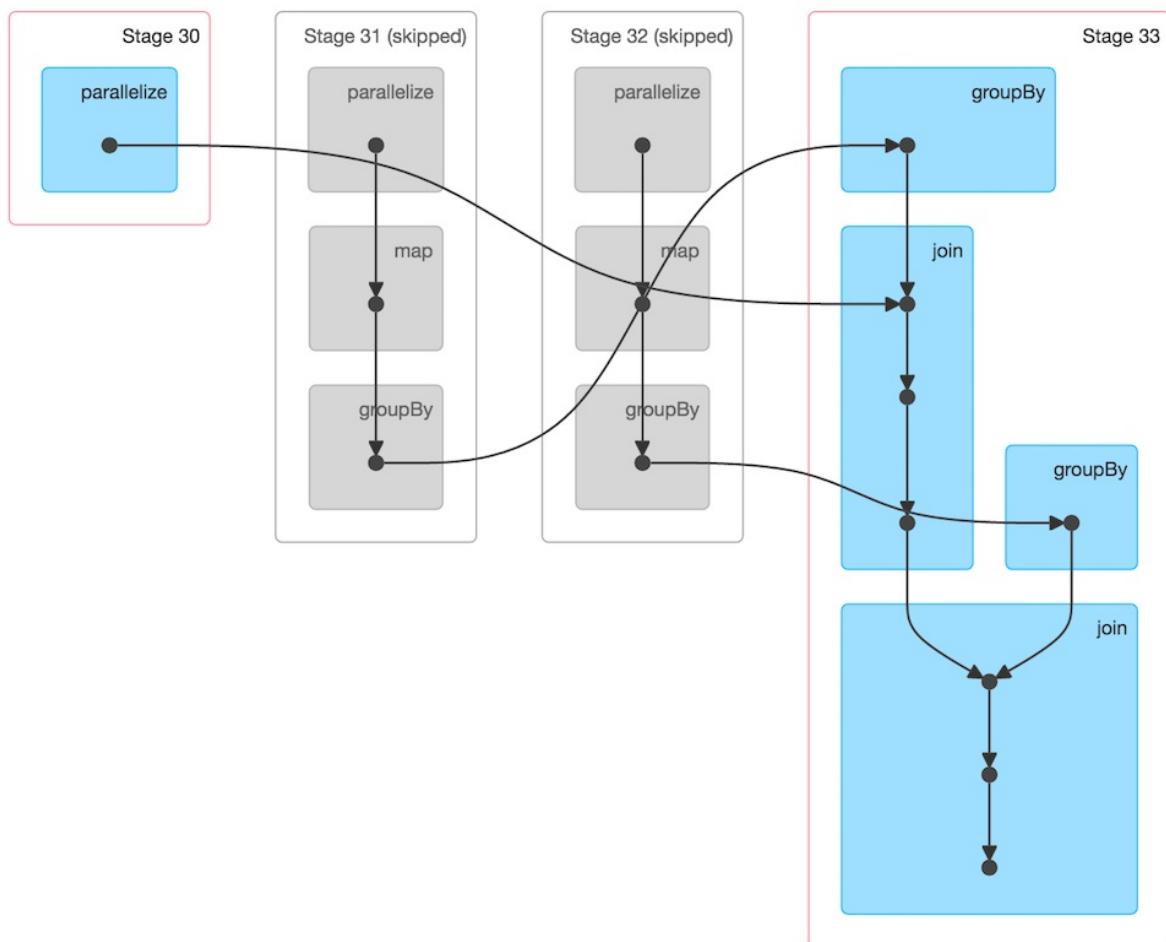


Figure 11. Details for Job Page with Four Stages

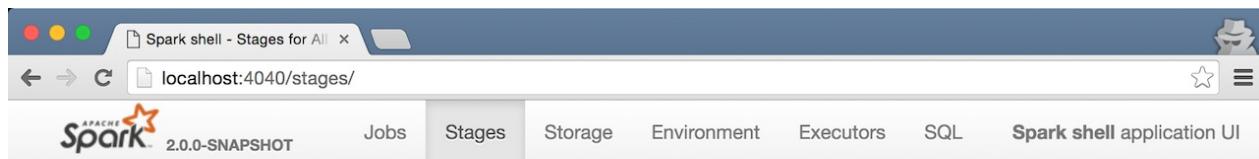
Stages Tab — Stages for All Jobs

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**.



Stages for All Jobs

[Completed Stages: 1](#)

[Completed Stages \(1\)](#)

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 [Stages for All Jobs](#).

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 to know whether [kill](#) is enabled or not.

Handling Kill Stage Request (from web UI) — `handleKillRequest` Method

Caution

[FIXME](#)

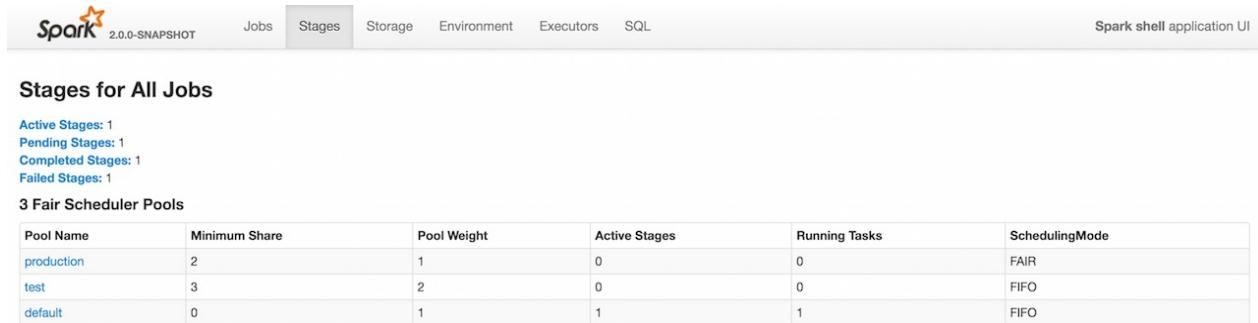
`killEnabled` flag

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 three pools: 'production' (2 minimum share, 1 pool weight, 0 active stages, 0 running tasks, FAIR mode), 'test' (3 minimum share, 2 pool weight, 0 active stages, 0 running tasks, FIFO mode), and 'default' (0 minimum share, 1 pool weight, 1 active stages, 1 running task, FIFO mode).

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 SparkContext.getAllPools .
------	--

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 StageInfo object.
------	---

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.

3 Fair Scheduler Pools

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

Active Stages (1)

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

Pending Stages (1)

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				

Completed Stages (1)

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			

Failed Stages (1)

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	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 26 (Attempt 0)

Total Time Across All Tasks: 94 ms
Locality Level Summary: Node local: 1; Process local: 74
Shuffle Read: 126.0 B / 2

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

Summary Metrics for 75 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0 ms	1 ms	1 ms	2 ms	3 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Blocked Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Size / Records	0.0 B / 0	0.0 B / 0	0.0 B / 0	0.0 B / 0	126.0 B / 2

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Shuffle Read Size / Records
0 stdout stderr	192.168.65.1:60723	0.3 s	75	0	0	75	126.0 B / 2

Tasks (75)

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Shuffle Read Blocked Time	Shuffle Read Size / Records ▾	Errors
66	333	0	SUCCESS	NODE_LOCAL	0 / 192.168.65.1 stdout stderr	2016/10/17 20:11:22	2 ms		0 ms	126.0 B / 2	
0	334	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 stdout stderr	2016/10/17 20:11:22	1 ms		0 ms	0.0 B / 0	
1	335	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 stdout stderr	2016/10/17 20:11:22	2 ms		0 ms	0.0 B / 0	

Figure 1. Details for Stage

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>.

StagePage is a part of Stages tab.

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

StagePage uses ExecutorsListener to display stdout and stderr logs of the executors in Tasks section.

Tasks Section

Stage Details

Tasks (75)

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Shuffle Read Blocked Time	Shuffle Read Size / Records ▾	Errors
66	333	0	SUCCESS	NODE_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	2 ms		0 ms	126.0 B / 2	
0	334	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	1 ms		0 ms	0.0 B / 0	
1	335	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	2 ms		0 ms	0.0 B / 0	
2	336	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	2 ms		0 ms	0.0 B / 0	
3	337	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	2 ms		0 ms	0.0 B / 0	
4	338	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	1 ms		0 ms	0.0 B / 0	
5	339	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	1 ms		0 ms	0.0 B / 0	
6	340	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	1 ms		0 ms	0.0 B / 0	
7	341	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	2 ms		0 ms	0.0 B / 0	
8	342	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	2 ms		0 ms	0.0 B / 0	
9	343	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	1 ms		0 ms	0.0 B / 0	
10	344	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	1 ms		0 ms	0.0 B / 0	
11	345	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	1 ms		0 ms	0.0 B / 0	
12	346	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.65.1 <code>stdout</code> <code>stderr</code>	2016/10/17 20:11:22	1 ms		0 ms	0.0 B / 0	

Figure 2. Tasks Section

Tasks paged table displays `StageUIData` that `JobProgressListener` collected for a stage and [stage attempt](#).

Note	The section uses <code>ExecutorsListener</code> to access <code>stdout</code> and <code>stderr</code> logs for <code>Executor ID / Host</code> column.
------	--

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
 (Deselect 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 3. Summary Metrics for Completed Tasks in Stage

Note	All the quantiles are doubles using <code>TaskUIData.metrics</code> (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

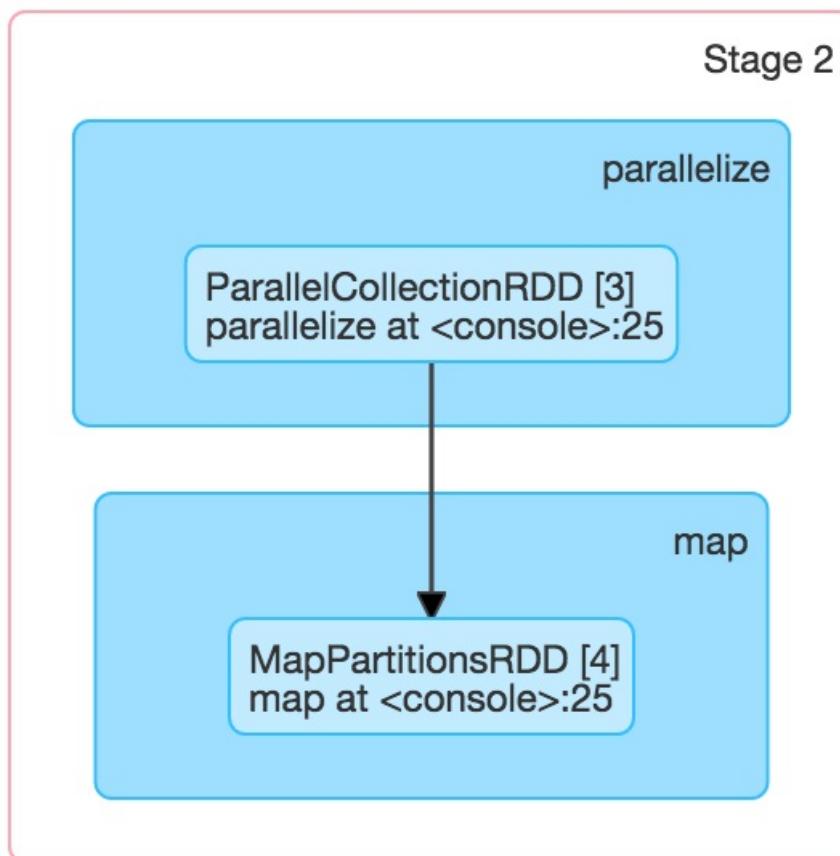


Figure 4. DAG Visualization

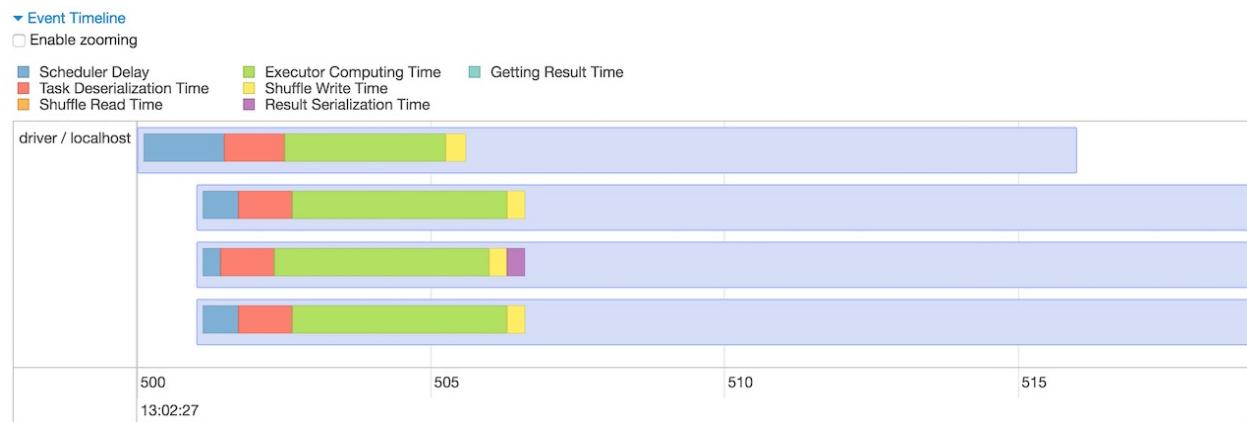


Figure 5. Event Timeline

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 6. 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 7. 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 8. Accumulators Section

Note

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

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.ui.timeline.tasks.maximum</code>	1000	
<code>spark.sql.unsafe.enabled</code>	true	

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 UI interface. At the top, there is a navigation bar with tabs: Jobs, Stages (which is selected and highlighted in grey), Storage, Environment, Executors, and SQL. To the right of the tabs, it says "Spark shell application UI". Below the navigation bar, the title "Fair Scheduler Pool: production" is displayed. Under this title, there is a section titled "Summary" containing a table with one row:

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

Below the summary table, there is a section titled "1 Active Stages" containing another 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).

`PoolPage` is a part of `Stages tab`.

`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

Storage tab in [web UI](#) shows ...

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

BlockStatusListener Spark Listener

`BlockStatusListener` is a [SparkListener](#) that tracks [BlockManagers](#) and the blocks for [Storage tab](#) in web UI.

Table 1. `BlockStatusListener` Registries

Registry	Description
<code>blockManagers</code>	The lookup table for a collection of <code>BlockId</code> and <code>BlockUIData</code> per <code>BlockManagerId</code> .

Caution

[FIXME](#) When are the events posted?

Table 2. `BlockStatusListener` Event Handlers

Event Handler	Description
<code>onBlockManagerAdded</code>	Registers a <code>BlockManager</code> in <code>blockManagers</code> internal registry (with no blocks).
<code>onBlockManagerRemoved</code>	Removes a <code>BlockManager</code> from <code>blockManagers</code> internal registry.
<code>onBlockUpdated</code>	<p>Puts an updated <code>BlockUIData</code> for <code>BlockId</code> for <code>BlockManagerId</code> in <code>blockManagers</code> internal registry.</p> <p>Ignores updates for unregistered <code>BlockManager</code>s or non-<code>StreamBlockId</code>s.</p> <p>For invalid <code>StorageLevels</code> (i.e. they do not use a memory or a disk or no replication) the block is removed.</p>

Environment Tab

The screenshot shows the Spark Web UI with the 'Environment' tab selected. The top navigation bar includes links for Jobs, Stages, Storage, Environment (which is highlighted), Executors, and SQL. To the right, it says 'Spark shell application UI'. The main content area is divided into two sections: 'Runtime Information' and 'Spark Properties', each represented by a table.

Runtime Information

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

Spark Properties

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.uri	http://192.168.1.4:62702
spark.scheduler.mode	FIFO
spark.submit.deployMode	client
spark.ui.showConsoleProgress	true

Figure 1. Environment tab in Web UI

EnvironmentListener Spark Listener

Caution

[FIXME](#)

Executors Tab

Executors tab in [web UI](#) shows ...

The screenshot shows the Spark web UI with the 'Executors' tab selected. The top navigation bar includes links for Jobs, Stages, Storage, Environment, Executors (which is highlighted), and SQL. The title bar indicates it's a 'Spark shell application UI'. The main content area has two sections: 'Summary' and 'Executors'.

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
Active(2)	10	40.4 KB / 1.9 GB	1.2 KB	2	0	0	4	4	1 s (48 ms)	0.0 B	0.0 B	0.0 B
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
Total(2)	10	40.4 KB / 1.9 GB	1.2 KB	2	0	0	4	4	1 s (48 ms)	0.0 B	0.0 B	0.0 B

Executors

Show: 20 entries											Search: <input type="text"/>					
Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	192.168.1.4:49478	Active	4	20.2 KB / 956.6 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	Thread Dump	
0	192.168.1.4:49484	Active	6	20.2 KB / 956.6 MB	1.2 KB	2	0	0	4	4	1 s (48 ms)	0.0 B	0.0 B	0.0 B	stdout stderr Thread Dump	

Showing 1 to 2 of 2 entries

Previous 1 Next

Figure 1. Executors Tab in web UI (local mode)

ExecutorsTab uses [ExecutorsListener](#) to collect information about executors in a Spark application.

The title of the tab is **Executors**.

You can access the Executors tab under `/executors` URL, e.g.

<http://localhost:4040/executors>.

What's interesting in how Storage Memory is displayed in the Executors tab is that in a way that is different from what the page displays (using the custom JavaScript

```
// local mode with spark.driver.memory 2g
// ./bin/spark-shell --conf spark.driver.memory=2g
// UnifiedMemoryManager reports 912MB
// You can see it after enabling INFO messages for BlockManagerMasterEndpoint

INFO BlockManagerMasterEndpoint: Registering block manager 192.168.1.8:54503 wj

Note // custom JavaScript `formatBytes` function (from utils.js) reports...956.6MB
// See https://github.com/apache/spark/blob/master/core/src/main/resources/org/
def formatBytes(bytes: Double) = {
  val k = 1000
  val i = math.floor(math.log(bytes) / math.log(k))
  val maxMemoryWebUI = bytes / math.pow(k, i)
  f"$maxMemoryWebUI%1.1f"
}
scala> println(formatBytes(maxMemory))
956.6
```

ExecutorsPage

ExecutorsPage is a WebUIPage .

Caution

[FIXME](#)

getExecInfo Method

```
getExecInfo(  
    listener: ExecutorsListener,  
    statusId: Int,  
    isActive: Boolean): ExecutorSummary
```

getExecInfo creates a ExecutorSummary .

Caution

[FIXME](#)

Note

getExecInfo is used when...[FIXME](#)

ExecutorThreadDumpPage

ExecutorThreadDumpPage is enabled or disabled using [spark.ui.threadDumpsEnabled](#) setting.

Settings

spark.ui.threadDumpsEnabled

`spark.ui.threadDumpsEnabled` (default: true) is to enable (true) or disable (false) [ExecutorThreadDumpPage](#).

ExecutorsListener Spark Listener

`ExecutorsListener` is a [SparkListener](#) that tracks [executors and their tasks](#) in a Spark application for [Stage Details](#) page, [Jobs](#) tab and `/allexecutors` REST endpoint.

Table 1. ExecutorsListener's SparkListener Callbacks (in alphabetical order)

Event Handler	Description
<code>onApplicationStart</code>	May create an entry for the driver in <code>executorToTaskSummary</code> registry
<code>onExecutorAdded</code>	May create an entry in <code>executorToTaskSummary</code> registry. It also makes sure that the number of entries for dead executors does not exceed <code>spark.ui.retainedDeadExecutors</code> and removes excess. Adds an entry to <code>executorEvents</code> registry and optionally removes the oldest if the number of entries exceeds <code>spark.ui.timeline.executors.maximum</code> .
<code>onExecutorBlacklisted</code>	<code>FIXME</code>
<code>onExecutorRemoved</code>	Marks an executor dead in <code>executorToTaskSummary</code> registry. Adds an entry to <code>executorEvents</code> registry and optionally removes the oldest if the number of entries exceeds <code>spark.ui.timeline.executors.maximum</code> .
<code>onExecutorUnblacklisted</code>	<code>FIXME</code>
<code>onNodeBlacklisted</code>	<code>FIXME</code>
<code>onNodeUnblacklisted</code>	<code>FIXME</code>
<code>onTaskStart</code>	May create an entry for an executor in <code>executorToTaskSummary</code> registry.
<code>onTaskEnd</code>	May create an entry for an executor in <code>executorToTaskSummary</code> registry.

`ExecutorsListener` requires a [StorageStatusListener](#) and [SparkConf](#).

Table 2. ExecutorsListener's Internal Registries and Counters

Registry	Description
executorToTaskSummary	The lookup table for <code>ExecutorTaskSummary</code> per executor id.
executorEvents	Used to build a <code>ExecutorSummary</code> for <code>/allexecutors</code> REST endpoint, to display stdout and stderr logs in Tasks and Aggregated Metrics by Executor sections in Stage Details page. A collection of SparkListenerEvents . Used to build the event timeline in All Jobs and Details for Job pages.

updateExecutorBlacklist Method

Caution

FIXME

Intercepting Executor Was Blacklisted Events

— onExecutorBlacklisted Callback

Caution

FIXME

Intercepting Executor Is No Longer Blacklisted Events

— onExecutorUnblacklisted Callback

Caution

FIXME

Intercepting Node Was Blacklisted Events

— onNodeBlacklisted Callback

Caution

FIXME

Intercepting Node Is No Longer Blacklisted Events

— onNodeUnblacklisted Callback

Caution

FIXME

Inactive/Dead BlockManagers

— `deadStorageStatusList` Method

```
deadStorageStatusList: Seq[StorageStatus]
```

`deadStorageStatusList` requests for the list of inactive/dead BlockManagers.

Note

`deadStorageStatusList` is used when:

- `ExecutorsPage` creates a `ExecutorSummary` to display in...FIXME
- `AllExecutorListResource` gives all executors in a Spark application (regardless of their status — active or inactive/dead).

Intercepting Application Started Events

— `onApplicationStart` Callback

```
onApplicationStart(applicationStart: SparkListenerApplicationStart): Unit
```

Note

`onApplicationStart` is a part of [SparkListener contract](#) to announce that a Spark application has been started.

`onApplicationStart` takes `driverLogs` property from the input `applicationStart` (if defined) and finds the driver's active `StorageStatus` (using the current `StorageStatusListener`). `onApplicationStart` then uses the driver's `StorageStatus` (if defined) to set `executorLogs`.

Table 3. ExecutorTaskSummary and ExecutorInfo Attributes

ExecutorTaskSummary Attribute	SparkListenerApplicationStart Attribute
<code>executorLogs</code>	<code>driverLogs</code> (if defined)

Intercepting Executor Added Events

— `onExecutorAdded` Callback

```
onExecutorAdded(executorAdded: SparkListenerExecutorAdded): Unit
```

Note

`onExecutorAdded` is a part of [SparkListener contract](#) to announce that a new executor has been registered with the Spark application.

`onExecutorAdded` finds the executor (using the input `executorAdded`) in the internal `executorToTaskSummary` registry and sets the attributes. If not found, `onExecutorAdded` creates a new entry.

Table 4. ExecutorTaskSummary and ExecutorInfo Attributes

ExecutorTaskSummary Attribute	ExecutorInfo Attribute
<code>executorLogs</code>	<code>logUrlMap</code>
<code>totalCores</code>	<code>totalCores</code>
<code>tasksMax</code>	<code>totalCores / spark.task.cpus</code>

`onExecutorAdded` adds the input `executorAdded` to `executorEvents` collection. If the number of elements in `executorEvents` collection is greater than `spark.ui.timeline.executors.maximum`, the first/oldest event is removed.

`onExecutorAdded` removes the oldest dead executor from `executorToTaskSummary` lookup table if their number is greater than `spark.ui.retainedDeadExecutors`.

Intercepting Executor Removed Events

— `onExecutorRemoved` Callback

```
onExecutorRemoved(executorRemoved: SparkListenerExecutorRemoved): Unit
```

Note

`onExecutorRemoved` is a part of [SparkListener contract](#) to announce that an executor has been unregistered with the Spark application.

`onExecutorRemoved` adds the input `executorRemoved` to `executorEvents` collection. It then removes the oldest event if the number of elements in `executorEvents` collection is greater than `spark.ui.timeline.executors.maximum`.

The executor is marked as removed/inactive in `executorToTaskSummary` lookup table.

Intercepting Task Started Events — `onTaskStart` Callback

```
onTaskStart(taskStart: SparkListenerTaskStart): Unit
```

Note

`onTaskStart` is a part of [SparkListener contract](#) to announce that a task has been started.

`onTaskStart` increments `tasksActive` for the executor (using the input `SparkListenerTaskStart`).

Table 5. ExecutorTaskSummary and SparkListenerTaskStart Attributes

ExecutorTaskSummary Attribute	Description
<code>tasksActive</code>	Uses <code>taskStart.taskInfo.executorId</code> .

Intercepting Task End Events — `onTaskEnd` Callback

`onTaskEnd(taskEnd: SparkListenerTaskEnd): Unit`

Note	<code>onTaskEnd</code> is a part of SparkListener contract to announce that a task has ended.
------	---

`onTaskEnd` takes [TaskInfo](#) from the input `taskEnd` (if available).

Depending on the reason for `SparkListenerTaskEnd` `onTaskEnd` does the following:

Table 6. `onTaskEnd` Behaviour per `SparkListenerTaskEnd` Reason

SparkListenerTaskEnd Reason	onTaskEnd Behaviour
Resubmitted	Does nothing
ExceptionFailure	Increment <code>tasksFailed</code>
anything	Increment <code>tasksComplete</code>

`tasksActive` is decremented but only when the number of active tasks for the executor is greater than `0` .

Table 7. ExecutorTaskSummary and `onTaskEnd` Behaviour

ExecutorTaskSummary Attribute	Description
<code>tasksActive</code>	Decremented if greater than 0.
<code>duration</code>	Uses <code>taskEnd.taskInfo.duration</code>

If the `TaskMetrics` (in the input `taskEnd`) is available, the metrics are added to the `taskSummary` for the task's executor.

Table 8. Task Metrics and Task Summary

Task Summary	Task Metric
inputBytes	inputMetrics.bytesRead
inputRecords	inputMetrics.recordsRead
outputBytes	outputMetrics.bytesWritten
outputRecords	outputMetrics.recordsWritten
shuffleRead	shuffleReadMetrics.remoteBytesRead
shuffleWrite	shuffleWriteMetrics.bytesWritten
jvmGCTime	metrics.jvmGCTime

Finding Active BlockManagers

— `activeStorageStatusList` Method

```
activeStorageStatusList: Seq[StorageStatus]
```

`activeStorageStatusList` requests [StorageStatusListener](#) for active BlockManagers (on executors).

Note	<p><code>activeStorageStatusList</code> is used when:</p> <ul style="list-style-type: none"> • <code>ExecutorsPage</code> does <code>getExecInfo</code> • <code>AllExecutorListResource</code> does <code>executorList</code> • <code>ExecutorListResource</code> does <code>executorList</code> • <code>ExecutorsListener</code> gets informed that the Spark application has started, onNodeBlacklisted, and onNodeUnblacklisted
-------------	--

Settings

Table 9. Spark Properties

Spark Property	Default Value	Description
<code>spark.ui.timeline.executors.maximum</code>	1000	The maximum number of entries in executorEvents registry.

SQL Tab

SQL tab in [web UI](#) shows [SQLMetrics](#) per [physical operator](#) in a structured query physical plan.

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 for the structured query execution.

AllExecutionsPage

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

ID	Description	Submitted	Duration	Running Jobs	Succeeded Jobs	Failed Jobs
2	foreach at <console>:24	+details 2016/06/29 22:30:45	2 s	1		

ID	Description	Submitted	Duration	Jobs
0	show at <console>:24	+details 2016/06/29 22:29:46	19 ms	

ID	Description	Submitted	Duration	Succeeded Jobs	Failed Jobs
1	foreach at <console>:24	+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 — Details for Query

`ExecutionPage` shows details for structured query execution by `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`).

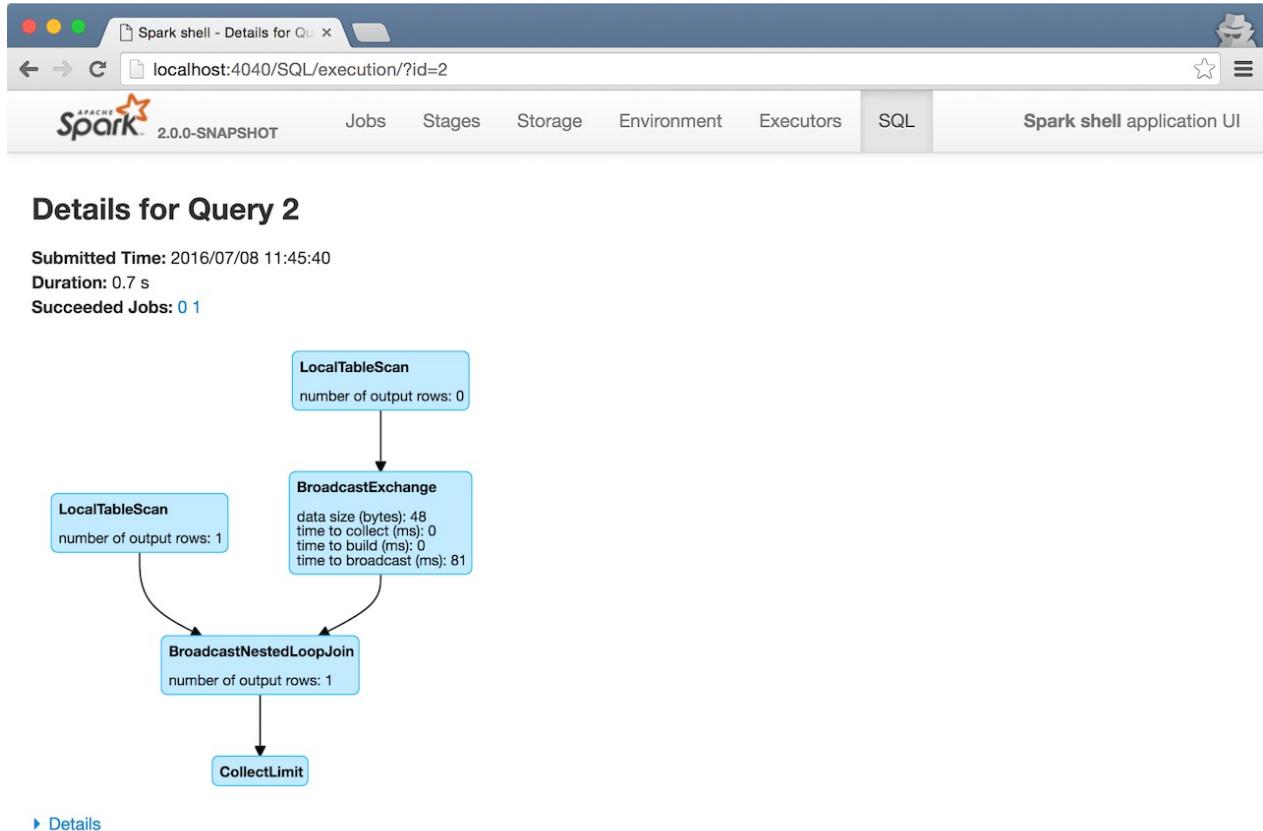


Figure 2. Details for Query in web UI

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 `SparkSessions`.

SQLListener Spark Listener

`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 SQLExecutionUIData 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 Spark Listener

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

`JobProgressListener` intercepts the following [Spark events](#).

Table 1. `JobProgressListener` Events

Handler	Purpose
<code>onJobStart</code>	Creates a <code>JobUIData</code> . It updates <code>jobGroupToJobIds</code> , <code>pendingStages</code> , <code>jobIdToData</code> , <code>activeJobs</code> , <code>stageIdToActiveJobIds</code> , <code>stageIdToInfo</code> and <code>stageIdToData</code> .
<code>onJobEnd</code>	Removes an entry in <code>activeJobs</code> . It also removes entries in <code>pendingStages</code> and <code>stageIdToActiveJobIds</code> . It updates <code>completedJobs</code> , <code>numCompletedJobs</code> , <code>failedJobs</code> , <code>numFailedJobs</code> and <code>skippedStages</code> .
<code>onStageCompleted</code>	Updates the <code>StageUIData</code> and <code>JobUIData</code> .
<code>onTaskStart</code>	Updates the task's <code>StageUIData</code> and <code>JobUIData</code> , and registers a new <code>TaskUIData</code> .
<code>onTaskEnd</code>	Updates the task's <code>StageUIData</code> (and <code>TaskUIData</code>), <code>ExecutorSummary</code> , and <code>JobUIData</code> .
<code>onExecutorMetricsUpdate</code>	
<code>onEnvironmentUpdate</code>	Sets <code>schedulingMode</code> property using the current <code>spark.scheduler.mode</code> (from Spark Properties environment details). Used in Jobs tab (for the Scheduling Mode), and to display pools in <code>JobsTab</code> and <code>StagesTab</code> . FIXME: Add the links/screenshots for pools.
<code>onBlockManagerAdded</code>	Records an executor and its block manager in the internal <code>executorIdToBlockManagerId</code> registry.
<code>onBlockManagerRemoved</code>	Removes the executor from the internal <code>executorIdToBlockManagerId</code> registry.
<code>onApplicationStart</code>	Records a Spark application's start time (in the internal <code>startTime</code>). Used in Jobs tab (for a total uptime and the event timeline) and Job page (for the event timeline).

<code>onApplicationEnd</code>	Records a Spark application's end time (in the internal <code>endTime</code>). Used in Jobs tab (for a total uptime).
<code>onTaskGettingResult</code>	Does nothing. FIXME: Why is this event intercepted at all?!

updateAggregateMetrics Method

Caution

[FIXME](#)

Registries and Counters

`JobProgressListener` uses registries to collect information about job executions.

Table 2. JobProgressListener Registries and Counters

Name	Description
numCompletedStages	
numFailedStages	
stageIdToData	Holds StageUIData per stage, i.e. the stage and stage attempt ids.
stageIdToInfo	
stageIdToActiveJobIds	
poolToActiveStages	
activeJobs	
completedJobs	
failedJobs	
jobIdToData	
jobGroupToJobIds	
pendingStages	
activeStages	
completedStages	
skippedStages	
failedStages	
executorIdToBlockManagerId	<p>The lookup table for BlockManagerId per executor id.</p> <p>Used to track block managers so the Stage page can display Address in Aggregated Metrics by Executor.</p> <p>FIXME: How does Executors page collect the very same information?</p>

onJobStart Method

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

`onJobStart` creates a `JobUIData`. It updates `jobGroupToJobIds`, `pendingStages`, `jobIdToData`, `activeJobs`, `stageIdToActiveJobIds`, `stageIdToInfo` and `stageIdToData`.

`onJobStart` reads the optional Spark Job group id as `spark.jobGroup.id` (from properties in the input `jobStart`).

`onJobStart` then creates a `JobUIData` using the input `jobStart` with `status` attribute set to `JobExecutionStatus.RUNNING` and records it in `jobIdToData` and `activeJobs` registries.

`onJobStart` looks the job ids for the group id (in `jobGroupToJobIds` registry) and adds the job id.

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

`onJobStart` records the stages of the job in `stageIdToActiveJobIds`.

`onJobStart` records `StageInfos` in `stageIdToInfo` and `stageIdToData`.

onJobEnd Method

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

`onJobEnd` removes an entry in `activeJobs`. It also removes entries in `pendingStages` and `stageIdToActiveJobIds`. It updates `completedJobs`, `numCompletedJobs`, `failedJobs`, `numFailedJobs` and `skippedStages`.

`onJobEnd` removes the job from `activeJobs` registry. It removes stages from `pendingStages` registry.

When completed successfully, the job is added to `completedJobs` registry with `status` attribute set to `JobExecutionStatus.SUCCEEDED`. `numCompletedJobs` gets incremented.

When failed, the job is added to `failedJobs` registry with `status` attribute set to `JobExecutionStatus.FAILED`. `numFailedJobs` gets incremented.

For every stage in the job, the stage is removed from the active jobs (in `stageIdToActiveJobIds`) that can remove the entire entry if no active jobs exist.

Every pending stage in `stageIdToInfo` gets added to `skippedStages`.

onExecutorMetricsUpdate Method

```
onExecutorMetricsUpdate(executorMetricsUpdate: SparkListenerExecutorMetricsUpdate): Unit
```

onTaskStart Method

```
onTaskStart(taskStart: SparkListenerTaskStart): Unit
```

`onTaskStart` updates `StageUIData` and `JobUIData`, and registers a new `TaskUIData`.

`onTaskStart` takes `TaskInfo` from the input `taskStart`.

`onTaskStart` looks the `StageUIData` for the stage and stage attempt ids up (in `stageIdToData` registry).

`onTaskStart` increments `numActiveTasks` and puts a `TaskUIData` for the task in `stageData.taskData`.

Ultimately, `onTaskStart` looks the stage in the internal `stageIdToActiveJobIds` and for each active job reads its `JobUIData` (from `jobIdToData`). It then increments `numActiveTasks`.

onTaskEnd Method

```
onTaskEnd(taskEnd: SparkListenerTaskEnd): Unit
```

`onTaskEnd` updates the `StageUIData` (and `TaskUIData`), `ExecutorSummary`, and `JobUIData`.

`onTaskEnd` takes `TaskInfo` from the input `taskEnd`.

Note

`onTaskEnd` does its processing when the `TaskInfo` is available and `stageAttemptId` is not `-1`.

`onTaskEnd` looks the `StageUIData` for the stage and stage attempt ids up (in `stageIdToData` registry).

`onTaskEnd` saves accumulables in the `StageUIData`.

`onTaskEnd` reads the `ExecutorSummary` for the executor (the task has finished on).

Depending on the task end's reason `onTaskEnd` increments `succeededTasks`, `killedTasks` or `failedTasks` counters.

`onTaskEnd` adds the task's duration to `taskTime`.

`onTaskEnd` decrements the number of active tasks (in the `StageUIData`).

Again, depending on the task end's reason `onTaskEnd` computes `errorMessage` and updates `StageUIData`.

Caution

FIXME Why is the same information in two different registries — `stageData` and `execSummary` ?!

If `taskMetrics` is available, `updateAggregateMetrics` is executed.

The task's `TaskUIData` is looked up in `stageData.taskData` and `updateTaskInfo` and `updateTaskMetrics` are executed. `errorMessage` is updated.

`onTaskEnd` makes sure that the number of tasks in `stageUIData` (`stageData.taskData`) is not above `spark.ui.retainedTasks` and drops the excess.

Ultimately, `onTaskEnd` looks the stage in the internal `stageIdToActiveJobIds` and for each active job reads its `JobUIData` (from `jobIdToData`). It then decrements `numActiveTasks` and increments `numCompletedTasks`, `numKilledTasks` or `numFailedTasks` depending on the task's end reason.

onStageSubmitted Method

```
onStageSubmitted(stageSubmitted: SparkListenerStageSubmitted): Unit
```

onStageCompleted Method

```
onStageCompleted(stageCompleted: SparkListenerStageCompleted): Unit
```

`onStageCompleted` updates the `StageUIData` and `JobUIData`.

`onStageCompleted` reads `stageInfo` from the input `stageCompleted` and records it in `stageIdToInfo` registry.

`onStageCompleted` looks the `StageUIData` for the stage and the stage attempt ids up in `stageIdToData` registry.

`onStageCompleted` records `accumulables` in `StageUIData`.

`onStageCompleted` removes the stage from `poolToActiveStages` and `activeStages` registries.

If the stage completed successfully (i.e. has no `failureReason`), `onStageCompleted` adds the stage to `completedStages` registry and increments `numCompletedStages` counter. It trims `completedStages`.

Otherwise, when the stage failed, `onStageCompleted` adds the stage to `failedStages` registry and increments `numFailedStages` counter. It trims `failedStages`.

Ultimately, `onStageCompleted` looks the stage in the internal `stageIdToActiveJobIds` and for each active job reads its `JobUIData` (from `jobIdToData`). It then decrements `numActiveStages`. When completed successfully, it adds the stage to `completedStageIndices`. With failure, `numFailedStages` gets incremented.

JobUIData

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

blockManagerIds method

<code>blockManagerIds: Seq[BlockManagerId]</code>	
---	--

StageUIData

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

Settings

Table 3. Spark Properties

Setting	Default Value	Description
<code>spark.ui.retainedJobs</code>	<code>1000</code>	The number of jobs to hold information about
<code>spark.ui.retainedStages</code>	<code>1000</code>	The number of stages to hold information about
<code>spark.ui.retainedTasks</code>	<code>100000</code>	The number of tasks to hold information about

StorageStatusListener — Spark Listener for Tracking BlockManagers

`StorageStatusListener` is a [SparkListener](#) that uses [SparkListener callbacks](#) to track status of every [BlockManager](#) in a Spark application.

`StorageStatusListener` is created and registered when `SparkUI` is created. It is later used to create [ExecutorsListener](#) and [StorageListener](#) Spark listeners.

Table 1. StorageStatusListener's SparkListener Callbacks (in alphabetical order)

Callback	Description
<code>onBlockManagerAdded</code>	<p>Adds an executor id with StorageStatus (with BlockManager and maximum memory on the executor) to executorIdToStorageStatus internal registry.</p> <p>Removes any other BlockManager that may have been registered for the executor earlier in deadExecutorStorageStatus internal registry.</p>
<code>onBlockManagerRemoved</code>	<p>Removes an executor from executorIdToStorageStatus internal registry and adds the removed StorageStatus to deadExecutorStorageStatus internal registry.</p> <p>Removes the oldest StorageStatus when the number of entries in deadExecutorStorageStatus is bigger than <code>spark.ui.retainedDeadExecutors</code>.</p>
<code>onBlockUpdated</code>	Updates StorageStatus for an executor in executorIdToStorageStatus internal registry, i.e. removes a block for <code>NONE</code> storage level and updates otherwise.
<code>onUnpersistRDD</code>	Removes the RDD blocks for an unpersisted RDD (on every BlockManager registered as StorageStatus in executorIdToStorageStatus internal registry).

Table 2. StorageStatusListener's Internal Registries and Counters

Name	Description
deadExecutorStorageStatus	<p>Collection of <code>StorageStatus</code> of removed/inactive <code>BlockManagers</code>.</p> <p>Accessible using <code>deadStorageStatusList</code> method.</p> <p>Adds an element when <code>StorageStatusListener handles a BlockManager being removed</code> (possibly removing one element from the head when the number of elements are above <code>spark.ui.retainedDeadExecutors</code> property).</p> <p>Removes an element when <code>StorageStatusListener handles a new BlockManager</code> (per executor) so the executor is not longer dead.</p>
executorIdToStorageStatus	<p>Lookup table of <code>StorageStatus</code> per executor (including the driver).</p> <p>Adds an entry when <code>storageStatusListener handles a new BlockManager</code>.</p> <p>Removes an entry when <code>storageStatusListener handles a BlockManager being removed</code>.</p> <p>Updates <code>storageStatus</code> of an executor when <code>StorageStatusListener handles StorageStatus updates</code>.</p>

Updating Storage Status For Executor — `updateStorageStatus` Method

Caution

FIXME

Active BlockManagers (on Executors) — `storageStatusList` Method

```
storageStatusList: Seq[StorageStatus]
```

`storageStatusList` gives a collection of `StorageStatus` (from `executorIdToStorageStatus` internal registry).

Note	<p><code>storageStatusList</code> is used when:</p> <ul style="list-style-type: none"> • <code>StorageStatusListener</code> removes the RDD blocks for an unpersisted RDD • <code>ExecutorsListener</code> does <code>activeStorageStatusList</code> • <code>StorageListener</code> does <code>activeStorageStatusList</code>
------	--

deadStorageStatusList Method

```
deadStorageStatusList: Seq[StorageStatus]
```

`deadStorageStatusList` gives `deadExecutorStorageStatus` internal registry.

Note	<p><code>deadStorageStatusList</code> is used when <code>ExecutorsListener</code> is requested for inactive/dead BlockManagers.</p>
------	---

Removing RDD Blocks for Unpersisted RDD — updateStorageStatus Internal Method

```
updateStorageStatus(unpersistedRDDId: Int)
```

`updateStorageStatus` takes active BlockManagers.

`updateStorageStatus` then finds RDD blocks for `unpersistedRDDId` RDD (for every BlockManager) and removes the blocks.

Note	<p><code>storageStatusList</code> is used exclusively when <code>StorageStatusListener</code> is notified that an RDD was unpersisted.</p>
------	--

StorageListener — Spark Listener for Tracking Persistence Status of RDD Blocks

`StorageListener` is a [BlockStatusListener](#) that uses [SparkListener callbacks](#) to track changes in the persistence status of RDD blocks in a Spark application.

Table 1. StorageListener's SparkListener Callbacks (in alphabetical order)

Callback	Description
<code>onBlockUpdated</code>	Updates <code>_rddInfoMap</code> with the update to a single block.
<code>onStageCompleted</code>	Removes <code>RDDInfos</code> from <code>_rddInfoMap</code> that participated in the completed stage as well as the ones that are no longer cached.
<code>onStageSubmitted</code>	Updates <code>_rddInfoMap</code> registry with the names of every <code>RDDInfo</code> in the submitted stage, possibly adding new <code>RDDInfos</code> if they were not registered yet.
<code>onUnpersistRDD</code>	Removes the <code>RDDInfo</code> from <code>_rddInfoMap</code> registry for the unpersisted RDD.

Table 2. StorageListener's Internal Registries and Counters

Name	Description
<code>_rddInfoMap</code>	Lookup table of <code>RDDInfo</code> per their ids Used when... FIXME

Creating StorageListener Instance

`StorageListener` takes the following when created:

- [StorageStatusListener](#)

`StorageListener` initializes the [internal registries and counters](#).

Note

`StorageListener` is created when `SparkUI` is created.

Finding Active BlockManagers — `activeStorageStatusList` Method

```
activeStorageStatusList: Seq[StorageStatus]
```

`activeStorageStatusList` requests [StorageStatusListener](#) for active BlockManagers (on executors).

Note

`activeStorageStatusList` is used when:

- `AllRDDResource` does `rddList` and `getRDDStorageInfo`
- `StorageListener` updates registered `RDDInfos` (with block updates from BlockManagers)

Intercepting Block Status Update Events

— `onBlockUpdated` Callback

```
onBlockUpdated(blockUpdated: SparkListenerBlockUpdated): Unit
```

`onBlockUpdated` creates a `BlockStatus` (from the input `SparkListenerBlockUpdated`) and updates registered `RDDInfos` (with block updates from BlockManagers) (passing in `BlockId` and `BlockStatus` as a single-element collection of updated blocks).

Note

`onBlockUpdated` is a part of [SparkListener contract](#) to announce that there was a change in a block status (on a `BlockManager` on an executor).

Intercepting Stage Completed Events

— `onStageCompleted` Callback

```
onStageCompleted(stageCompleted: SparkListenerStageCompleted): Unit
```

`onStageCompleted` finds the identifiers of the RDDs that have participated in the completed stage and removes them from `_rddInfoMap` registry as well as the RDDs that are no longer cached.

Note

`onStageCompleted` is a part of [SparkListener contract](#) to announce that a stage has finished.

Intercepting Stage Submitted Events

— `onStageSubmitted` Callback

```
onStageSubmitted(stageSubmitted: SparkListenerStageSubmitted): Unit
```

`onStageSubmitted` updates `_rddInfoMap` registry with the names of every `RDDInfo` in `stageSubmitted`, possibly adding new `RDDInfos` if they were not registered yet.

Note

`onStageSubmitted` is a part of [SparkListener contract](#) to announce that the missing tasks of a stage were submitted for execution.

Intercepting Unpersist RDD Events — `onUnpersistRDD` Callback

```
onUnpersistRDD(unpersistRDD: SparkListenerUnpersistRDD): Unit
```

`onUnpersistRDD` removes the `RDDInfo` from `_rddInfoMap` registry for the unpersisted RDD (from `unpersistRDD`).

Note

`onUnpersistRDD` is a part of [SparkListener contract](#) to announce that an RDD has been unpersisted.

Updating Registered RDDInfos (with Block Updates from BlockManagers) — `updateRDDInfo` Internal Method

```
updateRDDInfo(updatedBlocks: Seq[(BlockId, BlockStatus)]): Unit
```

`updateRDDInfo` finds the RDDs for the input `updatedBlocks` (for [BlockIds](#)).

Note

`updateRDDInfo` finds `BlockIds` that are [RDDBlockIds](#).

`updateRDDInfo` takes `RDDInfo` entries (in `_rddInfoMap` registry) for which there are blocks in the input `updatedBlocks` and [updates RDDInfos \(using StorageStatus\)](#) (from `activeStorageStatusList`).

Note

`updateRDDInfo` is used exclusively when `StorageListener` gets notified about a [change in a block status \(on a BlockManager or an executor\)](#).

Updating RDDInfos (using StorageStatus) — `StorageUtils.updateRddInfo` Method

```
updateRddInfo(rddInfos: Seq[RDDInfo], statuses: Seq[StorageStatus]): Unit
```

Caution**FIXME**

Note	<p><code>updateRddInfo</code> is used when:</p> <ul style="list-style-type: none">• <code>SparkContext</code> is requested for storage status of cached RDDs• <code>StorageListener</code> updates registered RDDInfos (with block updates from <code>BlockManagers</code>)
------	--

RDDOperationGraphListener Spark Listener

Caution

[FIXME](#)

SparkUI

`SparkUI` represents the web UI for a Spark application and [Spark History Server](#). It is created and bound when `SparkContext` is created (with `spark.ui.enabled` enabled).

Note

The only difference between `SparkUI` for a [Spark application](#) and [Spark History Server](#) is that... [FIXME](#)

When started, `SparkUI` binds to [appUIAddress](#) address that you can control using `SPARK_PUBLIC_DNS` environment variable or `spark.driver.host` Spark property.

Table 1. SparkUI's Internal Registries and Counters

Name	Description
<code>appId</code>	

Tip

Enable `INFO` logging level for `org.apache.spark.ui.SparkUI` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.ui.SparkUI=INFO
```

Refer to [Logging](#).

attachTab Method

Caution

[FIXME](#)

Creating SparkUI Instance

```

class SparkUI (
    val sc: Option[SparkContext],
    val conf: SparkConf,
    securityManager: SecurityManager,
    val environmentListener: EnvironmentListener,
    val storageStatusListener: StorageStatusListener,
    val executorsListener: ExecutorsListener,
    val jobProgressListener: JobProgressListener,
    val storageListener: StorageListener,
    val operationGraphListener: RDDOperationGraphListener,
    var appName: String,
    val basePath: String,
    val startTime: Long)
extends WebUI(securityManager,
    securityManager.getSSLOptions("ui"), SparkUI.getUIPort(conf),
    conf, basePath, "SparkUI")

```

When executed, `SparkUI` creates a `StagesTab` and initializes the tabs and handlers in web UI.

Note

`SparkUI` is created when `SparkContext` is created (with `spark.ui.enabled` enabled). `SparkUI` gets the references to the owning `SparkContext` and the other properties, i.e. `SparkConf`, `LiveListenerBus` Event Bus, `JobProgressListener`, `SecurityManager`, `appName`, and `startTime`.

Assigning Unique Identifier of Spark Application — `setappId` Method

```
setappId(id: String): Unit
```

`setappId` sets the internal `appId`.

Note

`setappId` is used exclusively when `SparkContext` is initialized.

Attaching Tabs and Context Handlers — `initialize` Method

```
initialize(): Unit
```

`initialize` attaches the following tabs:

1. `JobsTab`
2. `StagesTab`

3. StorageTab
4. EnvironmentTab
5. ExecutorsTab

`initialize` also attaches `servletContextHandler` handlers:

1. `/static` to serve static files from `org/apache/spark/ui/static` directory (on CLASSPATH).
2. Redirecting `/` to `/jobs/` (so [Jobs tab](#) is the first tab when you open web UI).
3. Serving `/api` context path (with `org.apache.spark.status.api.v1` provider package) using `ApiRootResource`.
4. Redirecting `/stages/stage/kill` to `/stages/`

Note

`initialize` is a part of the WebUI Contract and is executed when [SparkUI](#) is created.

Stopping SparkUI — stop Method

`stop(): Unit`

`stop` stops the HTTP server and prints the following INFO message to the logs:

```
INFO SparkUI: Stopped Spark web UI at [appUIAddress]
```

Note

`appUIAddress` in the above INFO message is the result of [appUIAddress](#) method.

appUIAddress Method

`appUIAddress: String`

`appUIAddress` returns the entire URL of a Spark application's web UI, including `http://` scheme.

Internally, `appUIAddress` uses [appUIHostPort](#).

getSparkUser Method

```
getSparkUser: String
```

`getSparkUser` returns the name of the user a Spark application runs as.

Internally, `getSparkUser` requests `user.name` System property from [EnvironmentListener](#) Spark listener.

Note

`getSparkUser` is only used to display the user name in [Spark Jobs page](#)

createLiveUI Method

```
createLiveUI(
  sc: SparkContext,
  conf: SparkConf,
  listenerBus: SparkListenerBus,
  jobProgressListener: JobProgressListener,
  securityManager: SecurityManager,
  appName: String,
  startTime: Long): SparkUI
```

`createLiveUI` creates a `SparkUI` for a live running Spark application.

Internally, `createLiveUI` simply forwards the call to [create](#).

Note

`createLiveUI` is called when `SparkContext` is created (and `spark.ui.enabled` is enabled).

createHistoryUI Method

Caution

[FIXME](#)

create Factory Method

```
create(
  sc: Option[SparkContext],
  conf: SparkConf,
  listenerBus: SparkListenerBus,
  securityManager: SecurityManager,
  appName: String,
  basePath: String = "",
  jobProgressListener: Option[JobProgressListener] = None,
  startTime: Long): SparkUI
```

`create` creates a `sparkUI` and is responsible for registering `SparkListeners` for `SparkUI`.

Note

`create` creates a web UI for a running Spark application and `Spark History Server`.

Internally, `create` registers the following `SparkListeners` with the input `listenerBus`.

- `EnvironmentListener`
- `StorageStatusListener`
- `ExecutorsListener`
- `StorageListener`
- `RDDOperationGraphListener`

`create` then creates a `SparkUI`.

appUIHostPort Method

`appUIHostPort: String`

`appUIHostPort` returns the Spark application's web UI which is the public hostname and port, excluding the scheme.

Note

`appUIAddress` uses `appUIHostPort` and adds `http://` scheme.

getAppName Method

`getAppName: String`

`getAppName` returns the name of the Spark application (of a `sparkUI` instance).

Note

`getAppName` is used when `SparkUITab` is requested the application's name.

SparkUITab — Custom WebUITab

`SparkUITab` is a `private[spark]` custom `WebUITab` that defines one method only, i.e. `appName`.

`appName: String`

`appName` returns the application's name.

Spark Submit — spark-submit shell script

`spark-submit` shell script allows you to manage your Spark applications.

You can [submit your Spark application](#) to a Spark deployment environment for execution, [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 first checks whether `SPARK_HOME` environment variable is set and sets it to the directory that contains `bin/spark-submit` shell script if not. It then executes [spark-class shell script](#) to run [SparkSubmit standalone application](#).

Caution	FIXME Add Cluster Manager and Deploy Mode to the table below (see options value)
---------	--

Table 1. Command-Line Options, Spark Properties and Environment Variables (from [Spark handle](#))

Command-Line Option	Spark Property	Environment Variable	
action			Defaults to
--archives			
--conf			
--deploy-mode	spark.submit.deployMode	DEPLOY_MODE	Deploy mode
--driver-class-path	spark.driver.extraClassPath		The driver's classpath
--driver-java-options	spark.driver.extraJavaOptions		The driver's Java options
--driver-library-path	spark.driver.extraLibraryPath		The driver's library path
--driver-memory	spark.driver.memory	SPARK_DRIVER_MEMORY	The driver's memory

--driver-cores	spark.driver.cores		
--exclude-packages	spark.jars.excludes		
--executor-cores	spark.executor.cores	SPARK_EXECUTOR_CORES	The number of cores per executor
--executor-memory	spark.executor.memory	SPARK_EXECUTOR_MEMORY	An executor's memory size
--files	spark.files		
ivyRepoPath	spark.jars.ivy		
--jars	spark.jars		
--keytab	spark.yarn.keytab		
--kill			submission to KILL
--master	spark.master	MASTER	Master URL
--class			
--name	spark.app.name	SPARK_YARN_APP_NAME (YARN only)	Uses main off primary ways set it
--num-executors	spark.executor.instances		
--packages	spark.jars.packages		
--principal	spark.yarn.principal		
--properties-file	spark.yarn.principal		
--proxy-user			
--py-files			
--queue			
--repositories			

--status		submission action <code>se</code>
--supervise		
--total-executor-cores	<code>spark.cores.max</code>	
--verbose		
--version		SparkSubmit
--help		<code>printUsage</code>
--usage-error		<code>printUsage</code>

Tip	<p>Set <code>SPARK_PRINT_LAUNCH_COMMAND</code> environment variable to have the complete Spark command printed out to the console, e.g.</p> <pre>\$ SPARK_PRINT_LAUNCH_COMMAND=1 ./bin/spark-shell Spark Command: /Library/Ja...</pre> <p>Refer to Print Launch Command of Spark Scripts (or org.apache.spark.launcher.Main Standalone Application where this environment variable is actually used).</p>
-----	---

Tip	<p>Avoid using <code>scala.App</code> trait for a Spark application's main class in Scala as reported in SPARK-4170 Closure problems when running Scala app that "extends App".</p> <p>Refer to Executing Main — <code>runMain</code> internal method in this document.</p>
-----	---

Preparing Submit Environment

— `prepareSubmitEnvironment` Internal Method

```
prepareSubmitEnvironment(args: SparkSubmitArguments)
  : (Seq[String], Seq[String], Map[String, String], String)
```

`prepareSubmitEnvironment` creates a 4-element tuple, i.e. `(childArgs, childClasspath, sysProps, childMainClass)`.

Table 2. `prepareSubmitEnvironment`'s Four-Element Return Tuple

Element	Description
<code>childArgs</code>	Arguments
<code>childClasspath</code>	Classpath elements
<code>sysProps</code>	Spark properties
<code>childMainClass</code>	Main class

`prepareSubmitEnvironment` uses options to...

Caution

FIXME

Note

`prepareSubmitEnvironment` is used in `SparkSubmit` object.

Tip

See the elements of the return tuple using `--verbose` command-line option.

Custom Spark Properties File—`--properties-file` command-line option

`--properties-file [FILE]`

`--properties-file` command-line option sets the path to a file `FILE` from which Spark loads extra [Spark properties](#).

Tip

Spark uses [conf/spark-defaults.conf](#) by default.

Driver Cores in Cluster Deploy Mode—`--driver-cores` command-line option

`--driver-cores NUM`

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

Note

`--driver-cores` switch is only available for cluster mode (for Standalone, Mesos, and YARN).

Note

It corresponds to [spark.driver.cores](#) setting.

Note	It is printed out to the standard error output in verbose mode .
------	--

Additional JAR Files to Distribute — `--jars` command-line option

```
--jars JARS
```

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

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

Additional Files to Distribute `--files` command-line option

```
--files FILES
```

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

Additional Archives to Distribute — `--archives` command-line option

```
--archives ARCHIVES
```

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

Specifying YARN Resource Queue — `--queue` command-line 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	FIXME 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 verbose mode .
-----	--

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](#)

Caution	FIXME Review why and when to use <code>proxyUser</code> .
---------	---

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

Executing Main — `runMain` internal 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.

Note	It is exclusively used when submitting applications for execution .
------	---

When `verbose` input flag is enabled (i.e. `true`) `runMain` prints out all the input parameters, i.e. `childMainClass`, `childArgs`, `sysProps`, and `childClasspath` (in that order).

```
Main class:  
[childMainClass]  
Arguments:  
[childArgs one per line]  
System properties:  
[sysProps one per line]  
Classpath elements:  
[childClasspath one per line]
```

Note Use `spark-submit`'s `--verbose` command-line option to enable `verbose` flag.

`runMain` builds the context classloader (as `loader`) depending on `spark.driver.userClassPathFirst` flag.

Caution **FIXME** Describe `spark.driver.userClassPathFirst`

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

Note `childclasspath` input parameter corresponds to `--jars` command-line option with the primary resource if specified in [client deploy mode](#).

It sets all the system properties specified in `sysProps` input parameter (using Java's [System.setProperty](#) method).

It creates an instance of `childMainClass` main class (as `mainclass`).

Note `childMainClass` is the main class `spark-submit` has been invoked with.

Tip Avoid using `scala.App` trait for a Spark application's main class 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 warning message in the logs:

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

Finally, `runMain` 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` internal 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	<code>addJarToClasspath</code> assumes <code>file</code> URI when <code>localJar</code> has no URI specified, e.g. <code>/path/to/local.jar</code> .
------	--

Caution	FIXME What is a URI fragment? How does this change re YARN distributed cache? See <code>utils#resolveURI</code> .
---------	---

Killing Applications — `--kill` command-line option

```
--kill
```

Requesting Application Status — `--status` command-line option

```
--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).
Your application's main class (for Java / Scala apps).
A name of your application.
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.
Comma-separated list of groupId:artifactId, to exclude while
resolving the dependencies provided in --packages to avoid
dependency conflicts.
Comma-separated list of additional remote repositories to
search for the maven coordinates given with --packages.
Comma-separated list of .zip, .egg, or .py files to place
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.

```

```

Spark standalone with cluster deploy mode only:
--driver-cores NUM          Cores for driver (Default: 1).

Spark standalone or Mesos with cluster deploy mode only:
--supervise                  If given, restarts the driver on failure.
--kill SUBMISSION_ID         If given, kills the driver specified.
--status SUBMISSION_ID       If given, requests the status of the driver specified.

Spark standalone and Mesos only:
--total-executor-cores NUM  Total cores for all executors.

Spark standalone and YARN only:
--executor-cores NUM        Number of cores per executor. (Default: 1 in YARN mode,
                            or all available cores on the worker in standalone mode)

YARN-only:
--driver-cores NUM           Number of cores used by the driver, only in cluster mode
                            (Default: 1).
--queue QUEUE_NAME           The YARN queue to submit to (Default: "default").
--num-executors NUM          Number of executors to launch (Default: 2).
--archives ARCHIVES          Comma separated list of archives to be extracted into th
e
--principal PRINCIPAL       working directory of each executor.
--keytab KEYTAB              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 t
he
--keytab KEYTAB              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 th
e
--keytab KEYTAB              delegation tokens periodically.

```

- `--class`
- `--conf` or `-c`
- `--deploy-mode` (see [Deploy Mode](#))
- `--driver-class-path` (see [--driver-class-path command-line option](#))
- `--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` ([see Custom Spark 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 \(--queue switch\)](#))

--driver-class-path command-line option

`--driver-class-path` command-line option sets the extra class path entries (e.g. jars and directories) that should be added to a driver's JVM.

Tip You should use `--driver-class-path` in client deploy mode (not [SparkConf](#)) to ensure that the CLASSPATH is set up with the entries. client deploy mode uses the same JVM for the driver as `spark-submit`'s.

--driver-class-path sets the internal `driverExtraClassPath` property (when `SparkSubmitArguments.handle` called).

It works for all cluster managers and deploy modes.

If `driverExtraClassPath` not set on command-line, the `spark.driver.extraClassPath` setting is used.

Note Command-line options (e.g. `--driver-class-path`) have higher precedence than their corresponding Spark settings in a Spark properties file (e.g. `spark.driver.extraClassPath`). You can therefore control the final settings by overriding Spark settings on command line using the command-line options.

Table 3. Spark Settings in Spark Properties File and on Command Line

Setting / System Property	Command-Line Option	Description
<code>spark.driver.extraClassPath</code>	<code>--driver-class-path</code>	Extra class path entries (e.g. jars and directories) to pass to a driver's JVM.

Version— `--version` command-line option

Verbose Mode — `--verbose` command-line option

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

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

```
FIXME
```

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

```
FIXME
```

Deploy Mode — `--deploy-mode` command-line option

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?

SparkSubmit Standalone Application — main method

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.

Tip	<code>spark-class</code> uses the class name — <code>org.apache.spark.deploy.SparkSubmit</code> — to parse command-line arguments appropriately. Refer to org.apache.spark.launcher.Main Standalone Application
-----	--

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: <code>SUBMIT</code> , <code>KILL</code> , or <code>REQUEST_STATUS</code> .
------	--

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.

SparkSubmitArguments — spark-submit's Command-Line Argument Parser

`SparkSubmitArguments` is a custom `sparkSubmitArgumentsParser` to handle the command-line arguments of `spark-submit` script that the actions (i.e. `submit`, `kill` and `status`) use for their execution (possibly with the explicit `env` environment).

Note

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

Calculating Spark Properties — `loadEnvironmentArguments` internal method

```
loadEnvironmentArguments(): Unit
```

`loadEnvironmentArguments` calculates the Spark properties for the current execution of `spark-submit`.

`loadEnvironmentArguments` reads command-line options first followed by Spark properties and System's environment variables.

Note

Spark config properties start with `spark.` prefix and can be set using `--conf [key=value]` command-line option.

handle Method

```
protected def handle(opt: String, value: String): Boolean
```

`handle` parses the input `opt` argument and returns `true` or throws an `IllegalArgumentException` when it finds an unknown `opt`.

`handle` sets the internal properties in the table [Command-Line Options, Spark Properties and Environment Variables](#).

mergeDefaultSparkProperties Internal Method

```
mergeDefaultSparkProperties(): Unit
```

`mergeDefaultSparkProperties` merges Spark properties from the [default Spark properties file](#), i.e. `spark-defaults.conf` with those specified through `--conf` command-line option.

SparkSubmitOptionParser — spark-submit's Command-Line Parser

`SparkSubmitOptionParser` is the parser of [spark-submit](#)'s command-line options.

Table 1. `spark-submit` Command-Line Options

Command-Line Option	Description
<code>--archives</code>	
<code>--class</code>	The main class to run (as <code>mainClass</code> internal attribute).
<code>--conf [prop=value]</code> or <code>-c [prop=value]</code>	All <code>=</code> -separated values end up in <code>conf</code> potentially overriding existing settings. Order on command-line matters.
<code>--deploy-mode</code>	<code>deployMode</code> internal property
<code>--driver-class-path</code>	<code>spark.driver.extraClassPath</code> in <code>conf</code> — the driver class path
<code>--driver-cores</code>	
<code>--driver-java-options</code>	<code>spark.driver.extraJavaOptions</code> in <code>conf</code> — the driver VM options
<code>--driver-library-path</code>	<code>spark.driver.extraLibraryPath</code> in <code>conf</code> — the driver native library path
<code>--driver-memory</code>	<code>spark.driver.memory</code> in <code>conf</code>
<code>--exclude-packages</code>	
<code>--executor-cores</code>	
<code>--executor-memory</code>	
<code>--files</code>	
<code>--help</code> or <code>-h</code>	The option is added to <code>sparkArgs</code>
<code>--jars</code>	
<code>--keytab</code>	
<code>--kill</code>	The option and a value are added to <code>sparkArgs</code>

--kill	The option and a value are added to <code>sparkArgs</code>
--master	<code>master</code> internal property
--name	
--num-executors	
--packages	
--principal	
--properties-file [FILE]	<code>propertiesFile</code> internal property. Refer to Custom Spark Properties File — <code>--properties-file</code> command-line option.
--proxy-user	
--py-files	
--queue	
--repositories	
--status	The option and a value are added to <code>sparkArgs</code>
--supervise	
--total-executor-cores	
--usage-error	The option is added to <code>sparkArgs</code>
--verbose OR -v	
--version	The option is added to <code>sparkArgs</code>

SparkSubmitOptionParser Callbacks

`SparkSubmitOptionParser` is supposed to be overriden for the following capabilities (as callbacks).

Table 2. Callbacks

Callback	Description
handle	Executed when an option with an argument is parsed.
handleUnknown	Executed when an unrecognized option is parsed.
handleExtraArgs	Executed for the command-line arguments that handle and handleUnknown callbacks have not processed.

SparkSubmitOptionParser belongs to org.apache.spark.launcher Scala package and spark-launcher Maven/sbt module.

Note

org.apache.spark.launcher.SparkSubmitArgumentsParser is a custom SparkSubmitOptionParser .

Parsing Command-Line Arguments — parse Method

```
final void parse(List<String> args)
```

parse parses a list of command-line arguments.

parse calls handle callback whenever it finds a known command-line option or a switch (a command-line option with no parameter). It calls handleUnknown callback for unrecognized command-line options.

parse keeps processing command-line arguments until handle or handleUnknown callback return false or all command-line arguments have been consumed.

Ultimately, parse calls handleExtraArgs callback.

SparkSubmitCommandBuilder Command Builder

`SparkSubmitCommandBuilder` is used to build a command that `spark-submit` and `SparkLauncher` use to launch a Spark application.

`SparkSubmitCommandBuilder` uses the first argument to distinguish between shells:

1. `pyspark-shell-main`
2. `sparkr-shell-main`
3. `run-example`

Caution	<code>FIXME</code> Describe <code>run-example</code>
---------	--

`SparkSubmitCommandBuilder` parses command-line arguments using `optionParser` (which is a `SparkSubmitOptionParser`). `OptionParser` comes with the following methods:

1. `handle` to handle the known options (see the table below). It sets up `master`, `deployMode`, `propertiesFile`, `conf`, `mainClass`, `sparkArgs` internal properties.
2. `handleUnknown` to handle unrecognized options that *usually* lead to `Unrecognized option` error message.
3. `handleExtraArgs` to handle extra arguments that are considered a Spark application's arguments.

Note	For <code>spark-shell</code> it assumes that the application arguments are after <code>spark-submit</code> 's arguments.
------	--

SparkSubmitCommandBuilder.buildCommand / buildSparkSubmitCommand

```
public List<String> buildCommand(Map<String, String> env)
```

Note	<code>buildCommand</code> is a part of the <code>AbstractCommandBuilder</code> public API.
------	--

`SparkSubmitCommandBuilder.buildCommand` simply passes calls on to `buildSparkSubmitCommand` private method (unless it was executed for `pyspark` or `sparkr` scripts which we are not interested in in this document).

buildSparkSubmitCommand Internal Method

```
private List<String> buildSparkSubmitCommand(Map<String, String> env)
```

`buildSparkSubmitCommand` starts by building so-called effective config. When in client mode, `buildSparkSubmitCommand` adds `spark.driver.extraClassPath` to the result Spark command.

Note	Use <code>spark-submit</code> to have <code>spark.driver.extraClassPath</code> in effect.
------	---

`buildSparkSubmitCommand` builds the first part of the Java command passing in the extra classpath (only for `client` deploy mode).

Caution	FIXME Add <code>isThriftServer</code> case.
---------	--

`buildSparkSubmitCommand` appends `SPARK_SUBMIT_OPTS` and `SPARK_JAVA_OPTS` environment variables.

(only for `client` deploy mode) ...

Caution	FIXME Elaborate on the client deploy mode case.
---------	--

`addPermGenSizeOpt` case...elaborate

Caution	FIXME Elaborate on <code>addPermGenSizeOpt</code>
---------	--

`buildSparkSubmitCommand` appends `org.apache.spark.deploy.SparkSubmit` and the command-line arguments (using `buildSparkSubmitArgs`).

buildSparkSubmitArgs method

```
List<String> buildSparkSubmitArgs()
```

`buildSparkSubmitArgs` builds a list of command-line arguments for `spark-submit`.

`buildSparkSubmitArgs` uses a `SparkSubmitOptionParser` to add the command-line arguments that `spark-submit` recognizes (when it is executed later on and uses the very same `SparkSubmitOptionParser` parser to parse command-line arguments).

Table 1. `SparkSubmitCommandBuilder` Properties and Corresponding `SparkSubmitOptionParser` Attributes

<code>SparkSubmitCommandBuilder</code> Property	<code>SparkSubmitOptionParser</code> Attribute
<code>verbose</code>	<code>VERBOSE</code>
<code>master</code>	<code>MASTER [master]</code>
<code>deployMode</code>	<code>DEPLOY_MODE [deployMode]</code>
<code>appName</code>	<code>NAME [appName]</code>
<code>conf</code>	<code>CONF [key=value]*</code>
<code>propertiesFile</code>	<code>PROPERTIES_FILE [propertiesFile]</code>
<code>jars</code>	<code>JARS [comma-separated jars]</code>
<code>files</code>	<code>FILES [comma-separated files]</code>
<code>pyFiles</code>	<code>PY_FILES [comma-separated pyFiles]</code>
<code>mainClass</code>	<code>CLASS [mainClass]</code>
<code>sparkArgs</code>	<code>sparkArgs (passed straight through)</code>
<code>appResource</code>	<code>appResource (passed straight through)</code>
<code>appArgs</code>	<code>appArgs (passed straight through)</code>

getEffectiveConfig Internal Method

```
Map<String, String> getEffectiveConfig()
```

`getEffectiveConfig` internal method builds `effectiveConfig` that is `conf` with the Spark properties file loaded (using `loadPropertiesFile` internal method) skipping keys that have already been loaded (it happened when the command-line options were parsed in `handle` method).

Note

Command-line options (e.g. `--driver-class-path`) have higher precedence than their corresponding Spark settings in a Spark properties file (e.g. `spark.driver.extraClassPath`). You can therefore control the final settings by overriding Spark settings on command line using the command-line options. `charset` and trims white spaces around values.

isClientMode Internal Method

```
private boolean isClientMode(Map<String, String> userProps)
```

`isClientMode` checks `master` first (from the command-line options) and then `spark.master` Spark property. Same with `deployMode` and `spark.submit.deployMode`.

Caution

[FIXME](#) Review `master` and `deployMode`. How are they set?

`isClientMode` responds positive when no explicit master and `client` deploy mode set explicitly.

OptionParser

`OptionParser` is a custom [SparkSubmitOptionParser](#) that [SparkSubmitCommandBuilder](#) uses to parse command-line arguments. It defines all the [SparkSubmitOptionParser callbacks](#), i.e. `handle`, `handleUnknown`, and `handleExtraArgs`, for command-line argument handling.

OptionParser's handle Callback

```
boolean handle(String opt, String value)
```

`OptionParser` comes with a custom `handle` callback (from the [SparkSubmitOptionParser callbacks](#)).

Table 2. handle Method

Command-Line Option	Property / Behaviour
--master	master
--deploy-mode	deployMode
--properties-file	propertiesFile
--driver-memory	Sets spark.driver.memory (in conf)
--driver-java-options	Sets spark.driver.extraJavaOptions (in conf)
--driver-library-path	Sets spark.driver.extraLibraryPath (in conf)
--driver-class-path	Sets spark.driver.extraClassPath (in conf)
--conf	Expects a key=value pair that it puts in conf
--class	Sets mainClass (in conf). It may also set allowsMixedArguments and appResource if the execution is for one of the special classes, i.e. spark-shell , SparkSQLCLIDriver , or HiveThriftServer2 .
--kill --status	Disables isAppResourceReq and adds itself with the value to sparkArgs .
--help --usage-error	Disables isAppResourceReq and adds itself to sparkArgs .
--version	Disables isAppResourceReq and adds itself to sparkArgs .
anything else	Adds an element to sparkArgs

OptionParser's handleUnknown Method

```
boolean handleUnknown(String opt)
```

If `allowsMixedArguments` is enabled, `handleUnknown` simply adds the input `opt` to `appArgs` and allows for further [parsing of the argument list](#).

Caution	FIXME Where's <code>allowsMixedArguments</code> enabled?
---------	--

If `isExample` is enabled, `handleUnknown` sets `mainClass` to be `org.apache.spark.examples.[opt]` (unless the input `opt` has already the package prefix) and stops further [parsing of the argument list](#).

Caution	FIXME Where's <code>isExample</code> enabled?
---------	---

Otherwise, `handleUnknown` sets `appResource` and stops further [parsing of the argument list](#).

OptionParser's `handleExtraArgs` Method

```
void handleExtraArgs(List<String> extra)
```

`handleExtraArgs` adds all the `extra` arguments to `appArgs`.

spark-class shell script

`spark-class` shell script is the Spark application command-line launcher that is responsible for setting up JVM environment and executing a Spark application.

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

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

When started, `spark-class` first loads `$SPARK_HOME/bin/load-spark-env.sh`, collects the Spark assembly jars, and executes [org.apache.spark.launcher.Main](#).

Depending on the Spark distribution (or rather lack thereof), i.e. whether `RELEASE` file exists or not, it sets `SPARK_JARS_DIR` environment variable to `[SPARK_HOME]/jars` or `[SPARK_HOME]/assembly/target/scala-[SPARK_SCALA_VERSION]/jars`, respectively (with the latter being a local build).

If `SPARK_JARS_DIR` does not exist, `spark-class` prints the following error message and exits with the code `1`.

```
Failed to find Spark jars directory ([SPARK_JARS_DIR]).  
You need to build Spark with the target "package" before running this program.
```

`spark-class` sets `LAUNCH_CLASSPATH` environment variable to include all the jars under `SPARK_JARS_DIR`.

If `SPARK_PREPEND_CLASSES` is enabled, `[SPARK_HOME]/launcher/target/scala-[SPARK_SCALA_VERSION]/classes` directory is added to `LAUNCH_CLASSPATH` as the first entry.

Note	Use <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 other Spark assembly jars. It is useful for development so your changes don't require rebuilding Spark again.
------	--

`SPARK_TESTING` and `SPARK_SQL_TESTING` environment variables enable **test special mode**.

Caution	FIXME What's so special about the env vars?
---------	---

`spark-class` uses [org.apache.spark.launcher.Main](#) command-line application to compute the Spark command to launch. The `Main` class programmatically computes the command that `spark-class` executes afterwards.

Tip	Use <code>JAVA_HOME</code> to point at the JVM to use.
-----	--

org.apache.spark.launcher.Main Standalone Application

`org.apache.spark.launcher.Main` is a Scala standalone application used in `spark-class` to prepare the Spark command to execute.

`Main` expects that the first parameter is the class name that is the "operation mode":

1. `org.apache.spark.deploy.SparkSubmit` — `Main` uses [SparkSubmitCommandBuilder](#) to parse command-line arguments. This is the mode `spark-submit` uses.
2. *anything* — `Main` uses [SparkClassCommandBuilder](#) to parse command-line arguments.

```
$ ./bin/spark-class org.apache.spark.launcher.Main
Exception in thread "main" java.lang.IllegalArgumentException: Not enough arguments: m
issing class name.
        at org.apache.spark.launcher.CommandBuilderUtils.checkArgument(CommandBuilderU
tils.java:241)
        at org.apache.spark.launcher.Main.main(Main.java:51)
```

`Main` uses `buildCommand` method on the builder to build a Spark command.

If `SPARK_PRINT_LAUNCH_COMMAND` environment variable is enabled, `Main` prints the final Spark command to standard error.

```
Spark Command: [cmd]
=====
```

If on Windows it calls `prepareWindowsCommand` while on non-Windows OSes

`prepareBashCommand` with tokens separated by `\0`.

Caution	FIXME What's <code>prepareWindowsCommand</code> ? <code>prepareBashCommand</code> ?
---------	---

`Main` uses the following 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`.

AbstractCommandBuilder

`AbstractCommandBuilder` is the base command builder for [SparkSubmitCommandBuilder](#) and [SparkClassCommandBuilder](#) specialized command builders.

`AbstractCommandBuilder` expects that command builders define `buildCommand`.

Table 1. `AbstractCommandBuilder` Methods

Method	Description
<code>buildCommand</code>	The only abstract method that subclasses have to define.
<code>buildJavaCommand</code>	
<code>getConfDir</code>	
<code>loadPropertiesFile</code>	Loads the configuration file for a Spark application, be it the user-specified properties file or <code>spark-defaults.conf</code> file under the Spark configuration directory.

buildJavaCommand Internal Method

```
List<String> buildJavaCommand(String extraClassPath)
```

`buildJavaCommand` builds the Java command for a Spark application (which is a collection of elements with the path to `java` executable, JVM options from `java-opts` file, and a class path).

If `javaHome` is set, `buildJavaCommand` adds `[javaHome]/bin/java` to the result Java command. Otherwise, it uses `JAVA_HOME` or, when no earlier checks succeeded, falls through to `java.home` Java's system property.

Caution

[**FIXME**](#) Who sets `javaHome` internal property and when?

`buildJavaCommand` loads extra Java options from the `java-opts` file in [configuration directory](#) if the file exists and adds them to the result Java command.

Eventually, `buildJavaCommand` [builds the class path](#) (with the extra class path if non-empty) and adds it as `-cp` to the result Java command.

buildClassPath method

```
List<String> buildClassPath(String appClassPath)
```

`buildClassPath` builds the classpath for a Spark application.

Note

Directories always end up with the OS-specific file separator at the end of their paths.

`buildClassPath` adds the following in that order:

1. `SPARK_CLASSPATH` environment variable
2. The input `appClassPath`
3. The [configuration directory](#)
4. (only with `SPARK_PREPEND_CLASSES` set or `SPARK_TESTING` being `1`) Locally compiled Spark classes in `classes`, `test-classes` and Core's jars.

Caution

[FIXME](#) Elaborate on "locally compiled Spark classes".

5. (only with `SPARK_SQL_TESTING` being `1`) ...

Caution

[FIXME](#) Elaborate on the SQL testing case

6. `HADOOP_CONF_DIR` environment variable
7. `YARN_CONF_DIR` environment variable
8. `SPARK_DIST_CLASSPATH` environment variable

Note

`childEnv` is queried first before System properties. It is always empty for `AbstractCommandBuilder` (and `SparkSubmitCommandBuilder`, too).

Loading Properties File — `loadPropertiesFile` Internal Method

```
Properties loadPropertiesFile()
```

`loadPropertiesFile` is a part of `AbstractCommandBuilder` *private* API that loads Spark settings from a properties file (when specified on the command line) or `spark-defaults.conf` in the [configuration directory](#).

It loads the settings from the following files starting from the first and checking every location until the first properties file is found:

1. `propertiesFile` (if specified using `--properties-file` command-line option or set by `AbstractCommandBuilder.setPropertiesFile`).
2. `[SPARK_CONF_DIR]/spark-defaults.conf`
3. `[SPARK_HOME]/conf/spark-defaults.conf`

Note

`loadPropertiesFile` reads a properties file using `UTF-8`.

Spark's Configuration Directory — `getConfDir` Internal Method

`AbstractCommandBuilder` uses `getConfDir` to compute the current configuration directory of a Spark application.

It uses `SPARK_CONF_DIR` (from `childEnv` which is always empty anyway or as a environment variable) and falls through to `[SPARK_HOME]/conf` (with `SPARK_HOME` from [getSparkHome internal method](#)).

Spark's Home Directory — `getSparkHome` Internal Method

`AbstractCommandBuilder` uses `getSparkHome` to compute Spark's home directory for a Spark application.

It uses `SPARK_HOME` (from `childEnv` which is always empty anyway or as a environment variable).

If `SPARK_HOME` is not set, Spark throws a `IllegalStateException`:

Spark home not found; set it explicitly or use the `SPARK_HOME` environment variable.

SparkLauncher — Launching Spark Applications Programmatically

`SparkLauncher` is an interface to launch Spark applications programmatically, i.e. from a code (not [spark-submit](#) directly). It uses a builder pattern to configure a Spark application and launch it as a child process using [spark-submit](#).

`SparkLauncher` belongs to `org.apache.spark.launcher` Scala package in `spark-launcher` build module.

`SparkLauncher` uses [SparkSubmitCommandBuilder](#) to build the Spark command of a Spark application to launch.

Table 1. `SparkLauncher`'s Builder Methods to Set Up Invocation of Spark Application

Setter	Description
<code>addAppArgs(String... args)</code>	Adds command line arguments for a Spark application.
<code>addFile(String file)</code>	Adds a file to be submitted with a Spark application.
<code>addJar(String jar)</code>	Adds a jar file to be submitted with the application.
<code>addPyFile(String file)</code>	Adds a python file / zip / egg to be submitted with a Spark application.
<code>addSparkArg(String arg)</code>	Adds a no-value argument to the Spark invocation.
<code>addSparkArg(String name, String value)</code>	Adds an argument with a value to the Spark invocation. It recognizes known command-line arguments, i.e. <code>--master</code> , <code>--properties-file</code> , <code>--conf</code> , <code>--class</code> , <code>-jars</code> , <code>--files</code> , and <code>--py-files</code> .
<code>directory(File dir)</code>	Sets the working directory of spark-submit.
<code>redirectError()</code>	Redirects stderr to stdout.
<code>redirectError(File errFile)</code>	Redirects error output to the specified <code>errFile</code> file.

<code>redirectError(ProcessBuilder.Redirect to)</code>	Redirects error output to the specified <code>to</code> Redirect.
<code>redirectOutput(File outFile)</code>	Redirects output to the specified <code>outFile</code> file.
<code>redirectOutput(ProcessBuilder.Redirect to)</code>	Redirects standard output to the specified <code>to</code> Redirect.
<code>redirectToLog(String loggerName)</code>	Sets all output to be logged and redirected to a logger with the specified name.
<code>setAppName(String appName)</code>	Sets the name of an Spark application
<code>setAppResource(String resource)</code>	Sets the main application resource, i.e. the location of a jar file for Scala/Java applications.
<code>setConf(String key, String value)</code>	Sets a Spark property. Expects <code>key</code> starting with <code>spark.</code> prefix.
<code>setDeployMode(String mode)</code>	Sets the deploy mode.
<code>setJavaHome(String javaHome)</code>	Sets a custom <code>JAVA_HOME</code> .
<code>setMainClass(String mainClass)</code>	Sets the main class.
<code>setMaster(String master)</code>	Sets the master URL.
<code>setPropertiesFile(String path)</code>	Sets the internal <code>propertiesFile</code> . See loadPropertiesFile Internal Method .
<code>setSparkHome(String sparkHome)</code>	Sets a custom <code>SPARK_HOME</code> .
<code>setVerbose(boolean verbose)</code>	Enables verbose reporting for <code>SparkSubmit</code> .

After the invocation of a Spark application is set up, use `launch()` method to launch a subprocess that will start the configured Spark application. It is however recommended to use `startApplication` method instead.

```
import org.apache.spark.launcher.SparkLauncher

val command = new SparkLauncher()
  .setAppResource("SparkPi")
  .setVerbose(true)

val appHandle = command.startApplication()
```

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.

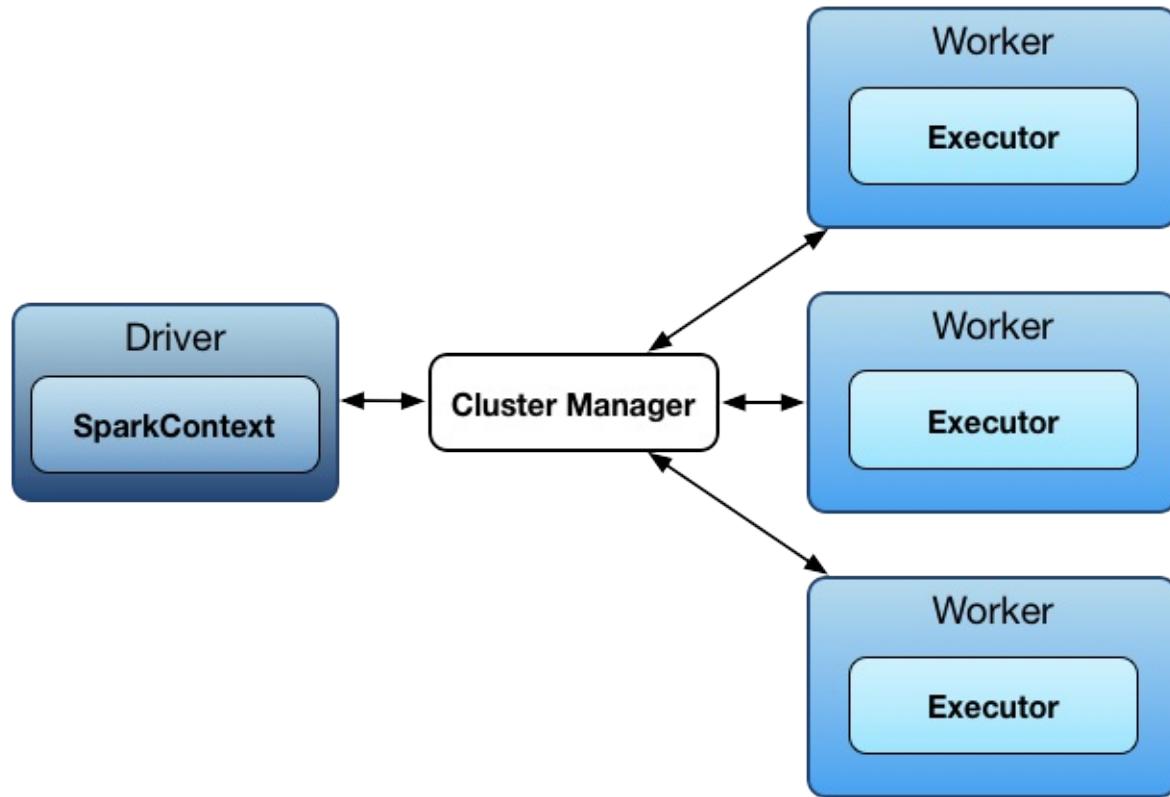


Figure 1. Spark architecture

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.

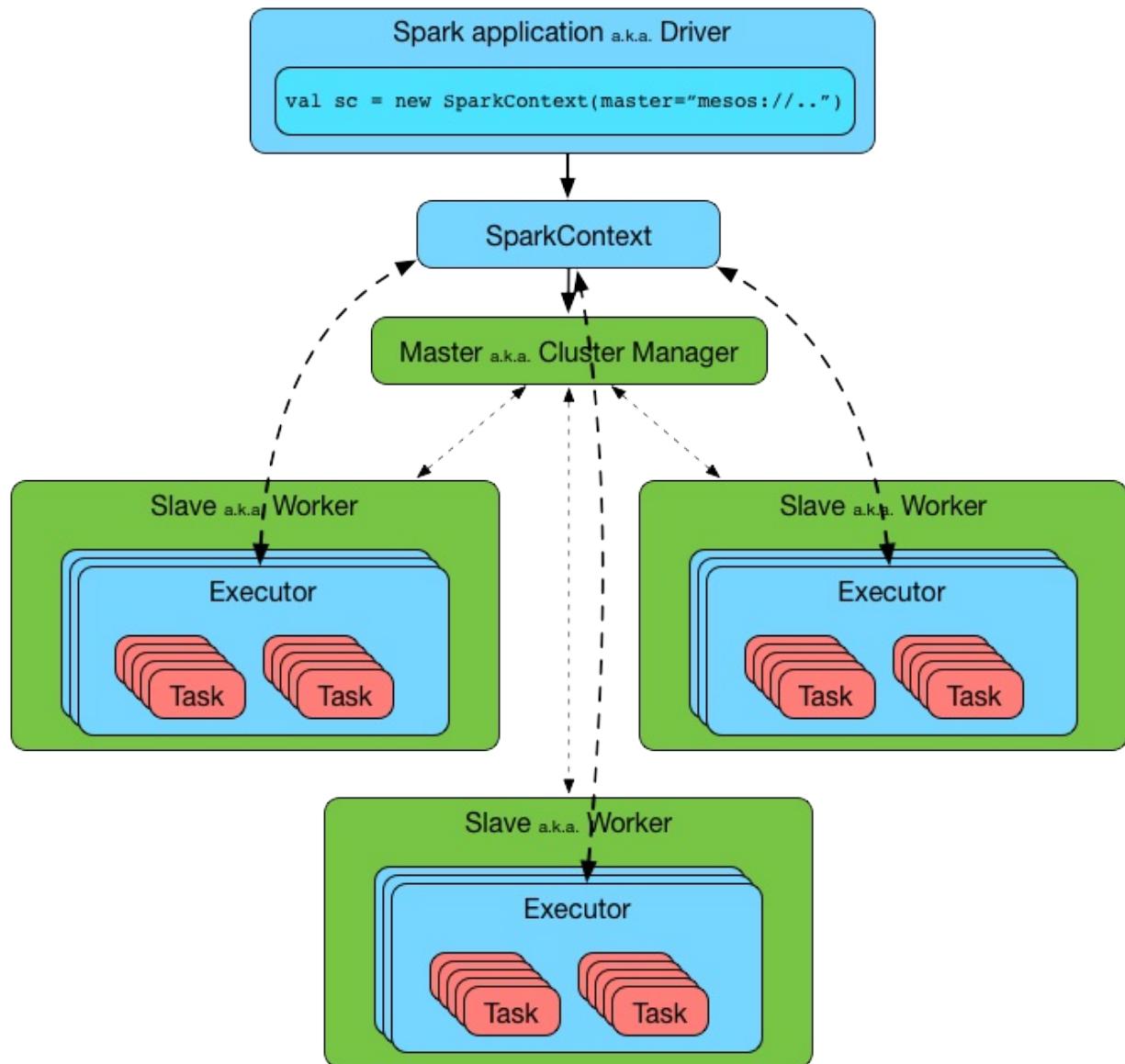


Figure 2. Spark architecture in detail

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 master node in a Spark application.

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

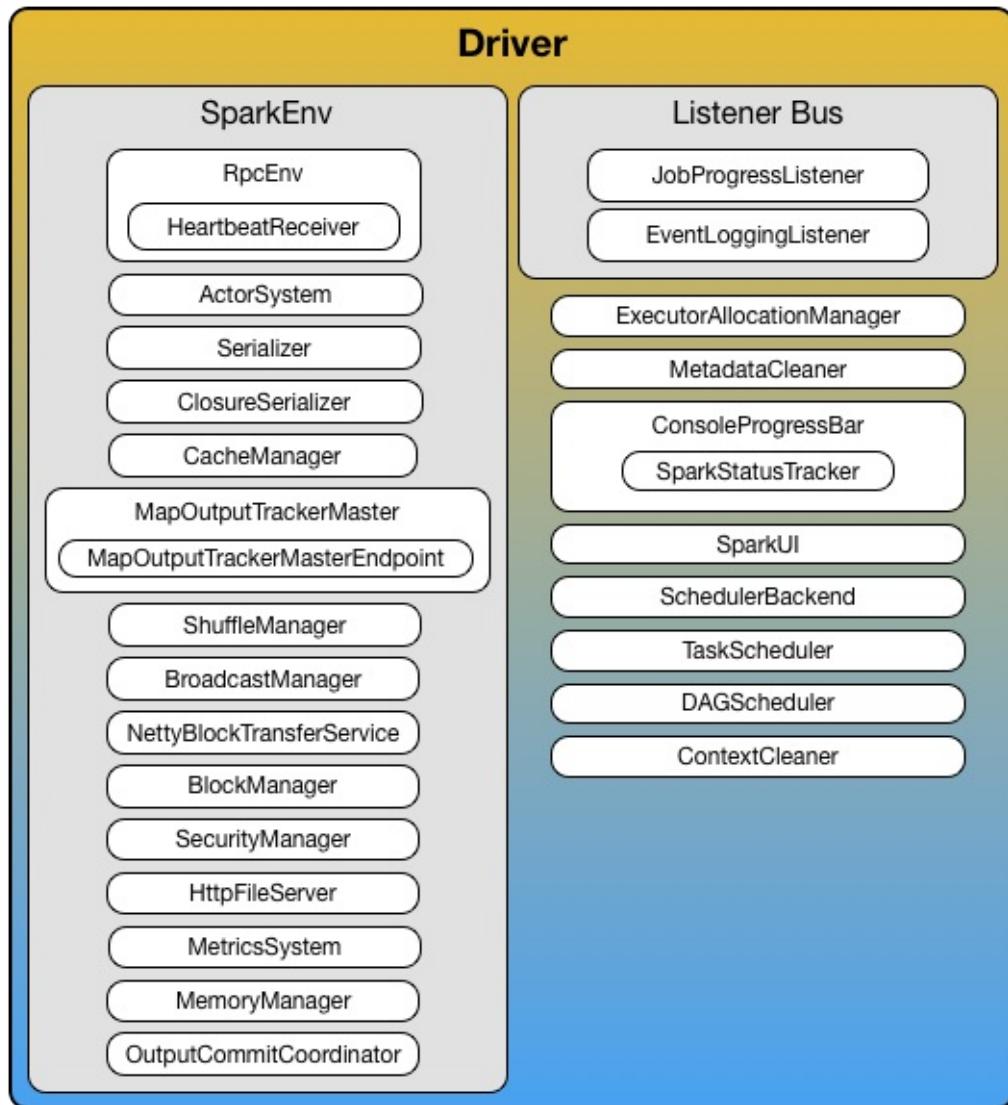


Figure 1. Driver with the services

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	FIXME 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 spark-submit '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 client deploy mode 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 spark-submit 's verbose mode.
------	---

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.driver.blockManager.port</code>	<code>spark.blockManager.port</code>	Port to use for the BlockManager on the driver. More precisely, <code>spark.driver.blockManager.port</code> is used when NettyBlockTransferService is created (while SparkEnv is created for the driver).
<code>spark.driver.host</code>	<code>localHostName</code>	The address of the node the driver runs on. Set when SparkContext is created.
<code>spark.driver.port</code>	<code>0</code>	The port the driver listener is first set to <code>0</code> in the driver when SparkContext is initialized. Set to the port of RpcEndpoint of the driver (in SparkEnv.createDriver) when client-mode . ApplicationMaster connects to the driver (in Spark or YARN).
<code>spark.driver.memory</code>	<code>1g</code>	The driver's memory size in MiBs. Refer to Driver's Memory .
<code>spark.driver.cores</code>	<code>1</code>	The number of CPU cores assigned to the driver in deploy mode . NOTE: When Client is chosen (for Spark on YARN in client mode only), it sets the number of cores for the Application using <code>spark.driver.cores</code> . Refer to Driver's Cores .
<code>spark.driver.extraLibraryPath</code>		

<code>spark.driver.extraJavaOptions</code>	Additional JVM options for the driver.
<p><code>spark.driver.appUIAddress</code></p> <p><code>spark.driver.appUIAddress</code> is used exclusively in Spark on YARN. It is set when YarnClientSchedulerBackend starts to run ExecutorLauncher (and register ApplicationMaster for the Spark application).</p>	<code>spark.driver.libraryPath</code>

spark.driver.extraClassPath

`spark.driver.extraClassPath` system property sets the additional classpath entries (e.g. jars and directories) that should be added to the driver's classpath in [cluster deploy mode](#).

Note	For client deploy mode you can use a properties file or command line to set <code>spark.driver.extraClassPath</code> .
	Do not use SparkConf since it is too late for client deploy mode given the JVM has already been set up to start a Spark application.
	Refer to buildSparkSubmitCommand Internal Method for the very low-level details of how it is handled internally.

`spark.driver.extraClassPath` uses a OS-specific path separator.

Note	Use <code>spark-submit</code> 's --driver-class-path command-line option on command line to override <code>spark.driver.extraClassPath</code> from a Spark properties file .
------	--

Executor

Executor is a distributed agent that is responsible for executing tasks.

Executor is created when:

- CoarseGrainedExecutorBackend receives RegisteredExecutor message (for Spark Standalone and YARN)
- Spark on Mesos's MesosExecutorBackend does registered
- LocalEndpoint is created (for local mode)

Executor typically runs for the entire lifetime of a Spark application which is called **static allocation of executors** (but you could also opt in for [dynamic allocation](#)).

Note

Executors are managed exclusively by [executor backends](#).

Executors reports heartbeat and partial metrics for active tasks to [HeartbeatReceiver](#) RPC Endpoint on the driver.

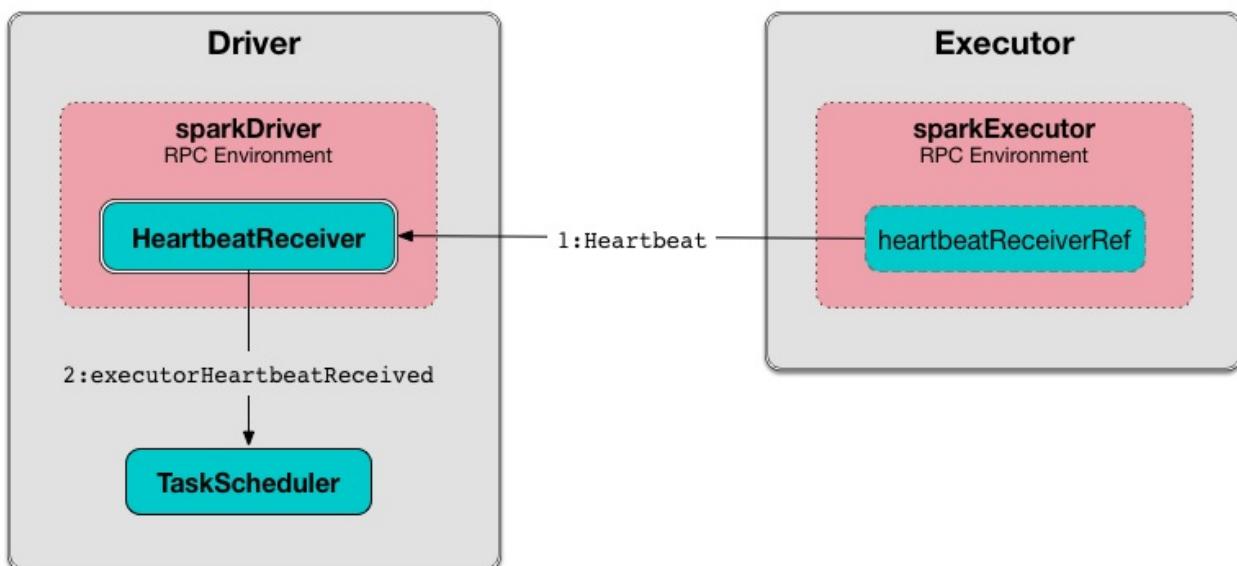


Figure 1. HeartbeatReceiver's Heartbeat Message Handler

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

When an executor starts it first registers with the driver and communicates directly to execute tasks.

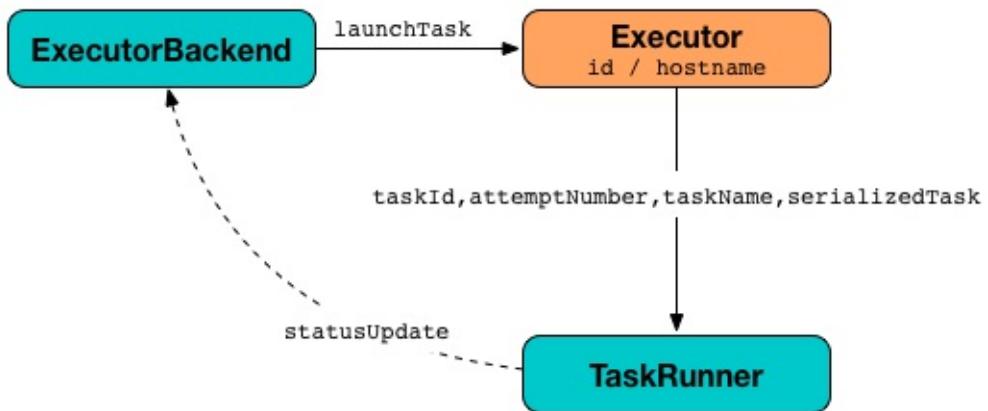


Figure 2. Launching tasks on executor using TaskRunners

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 registry). Consult [Launching Tasks](#) section.

Executors use a [Executor task launch worker thread pool](#) for launching tasks.

Executors send [metrics](#) (and heartbeats) using the [internal heartbeater - Heartbeat Sender Thread](#).

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).

Caution	FIXME How many cores are assigned per executor?
---------	---

Table 1. Executor's Internal Properties

Name	Initial Value	Description
executorSource	ExecutorSource	FIXME

Table 2. Executor's Internal Registries and Counters

Name	Description
heartbeatFailures	
heartbeatReceiverRef	<p>RPC endpoint reference to HeartbeatReceiver on the driver (available on <code>spark.driver.host</code> at <code>spark.driver.port</code> port).</p> <p>Set when <code>Executor</code> is created.</p> <p>Used exclusively when <code>Executor</code> reports heartbeats and partial metrics for active tasks to the driver (that happens every <code>spark.executor.heartbeatInterval</code> interval).</p>
maxDirectResultSize	
maxResultSize	
runningTasks	Lookup table of TaskRunners per... FIXME

Tip	Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.executor.Executor</code> logger to see what happens inside. Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.executor.Executor=DEBUG</code>
	Refer to Logging .

createClassLoader Method

Caution

[FIXME](#)

addRep1ClassLoaderIfNeeded Method

Caution

[FIXME](#)

Creating Executor Instance

`Executor` takes the following when created:

- Executor ID
- Executor's host name

- `SparkEnv`
- Collection of user-defined JARs ([to add to tasks' class path](#)). Empty by default
- Flag whether it runs in local or cluster mode (disabled by default, i.e. cluster is preferred)

Note User-defined JARs are defined using `--user-class-path` command-line option of `CoarseGrainedExecutorBackend` that can be set using `spark.executor.extraClassPath` property.

Note `isLocal` is enabled exclusively for [LocalEndpoint](#) (for [Spark in local mode](#)).

When created, you should see the following INFO messages in the logs:

```
INFO Executor: Starting executor ID [executorId] on host [executorHostname]
```

(only for [non-local mode](#)) `Executor` sets `SparkUncaughtExceptionHandler` as the default handler invoked when a thread abruptly terminates due to an uncaught exception.

(only for [non-local mode](#)) `Executor` registers `ExecutorSource` and initializes the local `BlockManager`.

Note `Executor` uses `sparkEnv` to access the local [MetricsSystem](#) and [BlockManager](#).

`Executor` creates a task class loader (optionally with [REPL support](#)) that the current `Serializer` is requested to use (when deserializing task later).

Note `Executor` uses `sparkEnv` to access the local `Serializer`.

`Executor` starts sending heartbeats and active tasks metrics.

`Executor` initializes the internal registries and counters in the meantime (not necessarily at the very end).

Launching Task — `launchTask` Method

```
launchTask(  
    context: ExecutorBackend,  
    taskId: Long,  
    attemptNumber: Int,  
    taskName: String,  
    serializedTask: ByteBuffer): Unit
```

`launchTask` executes the input `serializedTask` task concurrently.

Internally, `launchTask` creates a `TaskRunner`, registers it in `runningTasks` internal registry (by `taskId`), and finally executes it on "Executor task launch worker" thread pool.

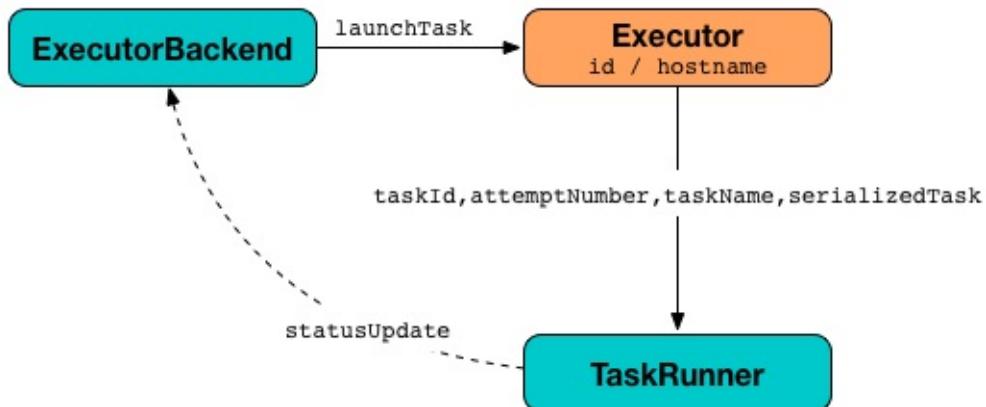


Figure 3. Launching tasks on executor using TaskRunners

Note

`launchTask` is called by `CoarseGrainedExecutorBackend` (when it handles `LaunchTask` message), `MesosExecutorBackend`, and `LocalEndpoint`.

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).

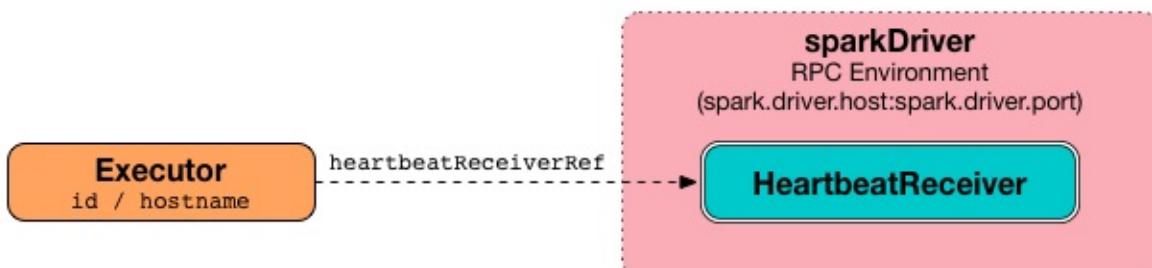


Figure 4. Executors use HeartbeatReceiver endpoint to report task metrics
An executor sends heartbeats using the internal heartbeater — Heartbeat Sender Thread.

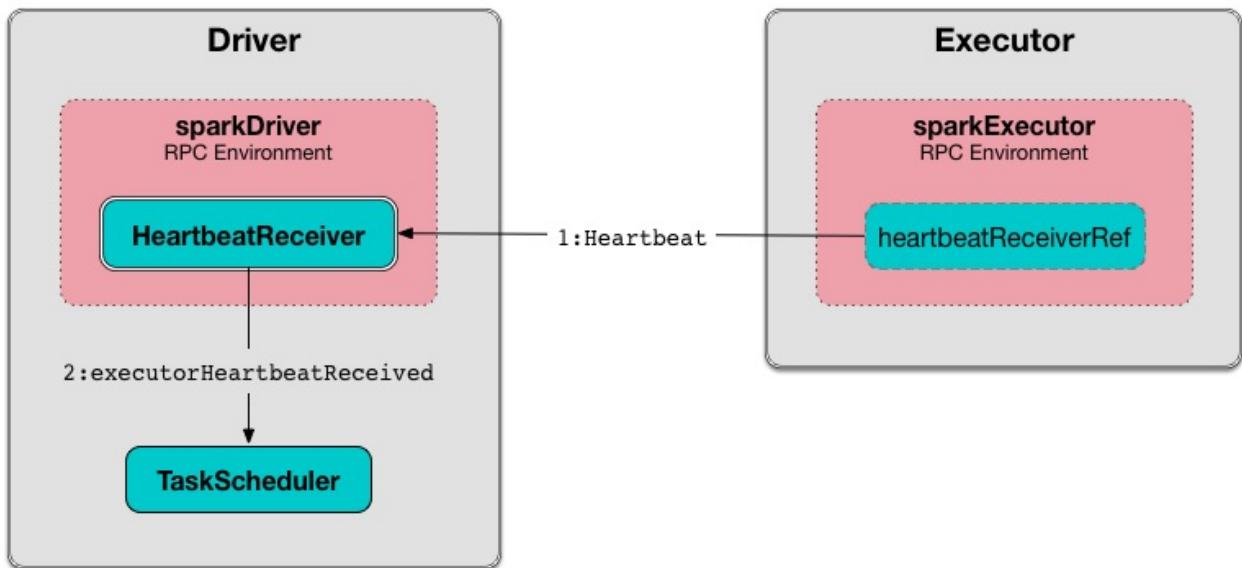


Figure 5. HeartbeatReceiver's Heartbeat Message Handler

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

Caution	FIXME 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	FIXME 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](#) (i.e. `spark.executor.heartbeat.maxFailures` Spark property). 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](#).

Reporting Heartbeat and Partial Metrics for Active Tasks to Driver — `reportHeartBeat` Internal Method

`reportHeartBeat(): Unit`

`reportHeartBeat` collects [TaskRunners](#) for [currently running tasks](#) (aka *active tasks*) with their [tasks](#) deserialized (i.e. either ready for execution or already started).

Note

[TaskRunner](#) has [task](#) deserialized when it [runs the task](#).

For every running task, `reportHeartBeat` takes its [TaskMetrics](#) and:

- Requests [ShuffleRead](#) metrics to be merged
- Sets [jvmGCTime](#) metrics

`reportHeartBeat` then records the latest values of [internal and external accumulators](#) for every task.

Note

Internal accumulators are a task's metrics while external accumulators are a Spark application's accumulators that a user has created.

`reportHeartBeat` sends a blocking [Heartbeat](#) message to [HeartbeatReceiver endpoint](#) (running on the driver). `reportHeartBeat` uses `spark.executor.heartbeatInterval` for the RPC timeout.

Note

A [Heartbeat](#) message contains the executor identifier, the accumulator updates, and the identifier of the [BlockManager](#).

Note

`reportHeartBeat` uses `sparkEnv` to access the current `BlockManager`.

If the response (from `HeartbeatReceiver endpoint`) is to re-register the `BlockManager`, you should see the following INFO message in the logs and `reportHeartBeat requests BlockManager to re-register` (which will register the blocks the `BlockManager` manages with the driver).

```
INFO Told to re-register on heartbeat
```

Note

`HeartbeatResponse requests BlockManager to re-register when either TaskScheduler or HeartbeatReceiver know nothing about the executor.`

When posting the `Heartbeat` was successful, `reportHeartBeat` resets `heartbeatFailures` internal counter.

In case of a non-fatal exception, you should see the following WARN message in the logs (followed by the stack trace).

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

Every failure `reportHeartBeat` increments `heartbeat failures` up to `spark.executor.heartbeat.maxFailures` Spark property. When the heartbeat failures reaches the maximum, you should see the following ERROR message in the logs and the executor terminates with the error code: `56`.

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

Note

`reportHeartBeat` is used when `Executor schedules reporting heartbeat and partial metrics for active tasks to the driver` (that happens every `spark.executor.heartbeatInterval` Spark property).

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.

Resource Offers

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

"Executor task launch worker" Thread Pool — `threadPool` Property

`Executor` uses `threadPool` daemon cached thread pool with the name **Executor task launch worker-[ID]** (with `ID` being the task id) for [launching tasks](#).

`threadPool` is created when `Executor` is created and shut down when it stops.

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 [SparkContext is created](#).

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

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

Spark 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

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 6. 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

Every executor registers its own [ExecutorSource](#) to report metrics.

Stopping Executor— stop Method

`stop(): Unit`

`stop` requests [MetricsSystem](#) for a report.

Note	<code>stop</code> uses SparkEnv to access the current MetricsSystem .
------	---

`stop` shuts driver-heartbeater thread down (and waits at most 10 seconds).

`stop` shuts Executor task launch worker thread pool down.

(only when [not local](#)) `stop` requests [SparkEnv](#) to stop.

Note	<code>stop</code> is used when CoarseGrainedExecutorBackend and LocalEndpoint are requested to stop their managed executors.
------	--

Settings

Table 3. Spark Properties

Spark Property	Default Value	Description
<code>spark.executor.cores</code>		Number of cores for an executor.
<code>spark.executor.extraClassPath</code>	(empty)	List of URLs representing defined class path entries are added to an executor's path. Each entry is separated by system-dependent path separator, i.e. : on

		Unix/MacOS systems and on Microsoft Windows.
<code>spark.executor.extraJavaOptions</code>		Extra Java options for executors. Used to prepare the command to launch CoarseGrainedExecutorBackend in a YARN container .
<code>spark.executor.extraLibraryPath</code>		Extra library paths separated by system-dependent path separator, i.e. <code>:</code> on Unix/MacOS systems and <code>\</code> on Microsoft Windows. Used to prepare the command to launch CoarseGrainedExecutorBackend in a YARN container .
<code>spark.executor.heartbeat.maxFailures</code>	60	Number of times an executor tries to send heartbeats to the driver before it gives up and exits (with exit code <code>56</code>). NOTE: It was introduced in SPARK-13522 . Executor will kill itself when it's unable to send heartbeat to the driver more than N times.
<code>spark.executor.heartbeatInterval</code>	10s	Interval after which an executor reports heartbeat and metrics active tasks to the driver. Refer to Sending heartbeats and partial metrics for active tasks in this document.
<code>spark.executor.id</code>		
<code>spark.executor.instances</code>	0	Number of executors to use.
<code>spark.executor.logs.rolling.maxSize</code>		
<code>spark.executor.logs.rolling.maxRetainedFiles</code>		
<code>spark.executor.logs.rolling.strategy</code>		
<code>spark.executor.logs.rolling.time.interval</code>		

spark.executor.memory	1g	Amount of memory to use executor process. Equivalent to SPARK_EXECUTOR_ME environment variable. Refer to Executor Memory spark.executor.memory or SPARK_EXECUTOR_ME settings in this document.
spark.executor.port		
spark.executor.port		
spark.executor.userClassPathFirst	false	Flag to control whether to classes in user jars before in Spark jars.
spark.executor.uri		Equivalent to SPARK_EXECUTOR_URI
spark.task.maxDirectResultSize	1048576B	

TaskRunner

TaskRunner is a thread of execution that manages a single individual task.

TaskRunner is created exclusively when Executor is requested to launch a task.

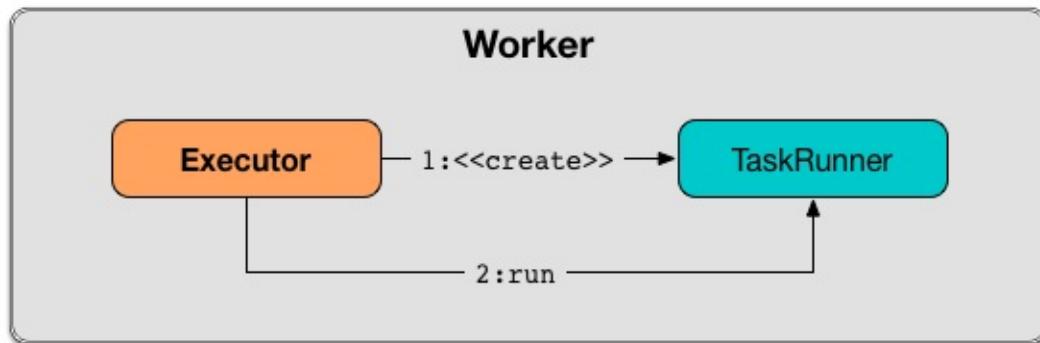


Figure 1. Executor creates TaskRunner and runs (almost) immediately
TaskRunner can be run or killed that simply means running or killing the task this
TaskRunner object manages, respectively.

Table 1. TaskRunner's Internal Registries and Counters

Name	Description
taskID	FIXME Used when... FIXME
threadName	FIXME Used when... FIXME
taskName	FIXME Used when... FIXME
finished	FIXME Used when... FIXME
killed	FIXME Used when... FIXME
threadId	FIXME Used when... FIXME
startGCTime	FIXME Used when... FIXME
task	FIXME Used when... FIXME
replClassLoader	FIXME Used when... FIXME
Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.executor.Executor</code> logger to see what happens inside <code>TaskRunner</code> (since <code>TaskRunner</code> is an internal class of <code>Executor</code>).</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.executor.Executor=DEBUG</pre> <p>Refer to Logging.</p>

Creating TaskRunner Instance

`TaskRunner` takes the following when created:

1. `ExecutorBackend`
2. `TaskDescription`

`TaskRunner` initializes the [internal registries and counters](#).

computeTotalGcTime Method

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

updateDependencies Method

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

setTaskFinishedAndClearInterruptStatus Method

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

Lifecycle

Caution	FIXME 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 java.lang.Runnable contract that <code>TaskRunner</code> follows.
------	---

When executed, `run` initializes `threadId` as the current thread identifier (using Java's [Thread](#))

`run` then sets the name of the current thread as `threadName` (using Java's [Thread](#)).

`run` creates a `TaskMemoryManager` (using the current `MemoryManager` and `taskId`).

Note	<code>run</code> uses <code>SparkEnv</code> to access the current <code>MemoryManager</code> .
------	--

`run` starts tracking the time to deserialize a task.

`run` sets the current thread's context classloader (with `repClassLoader`).

`run` creates a closure `Serializer`.

Note

`run` uses `SparkEnv` to access the current closure `Serializer`.

You should see the following INFO message in the logs:

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

`run` notifies `ExecutorBackend` that `taskId` is in `TaskState.RUNNING` state.

Note

`run` uses `ExecutorBackend` that was specified when `TaskRunner` was created.

`run` computes `startGCTime`.

`run` updates dependencies.

Note

`run` uses `TaskDescription` that is specified when `TaskRunner` is created.

`run` deserializes the task (using the context class loader) and sets its `localProperties` and `TaskMemoryManager`. `run` sets the `task` internal reference to hold the deserialized task.

Note

`run` uses `TaskDescription` to access serialized task.

If `killed` flag is enabled, `run` throws a `TaskKilledException`.

You should see the following DEBUG message in the logs:

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

`run` notifies `MapOutputTracker` about the epoch of the task.

Note

`run` uses `SparkEnv` to access the current `MapOutputTracker`.

`run` records the current time as the task's start time (as `taskStart`).

`run` runs the task (with `taskAttemptId` as `taskId`, `attemptNumber` from `TaskDescription`, and `metricsSystem` as the current `MetricsSystem`).

Note

`run` uses `SparkEnv` to access the current `MetricsSystem`.

Note

The task runs inside a "monitored" block (i.e. `try-finally` block) to detect any memory and lock leaks after the task's `run` finishes regardless of the final outcome - the computed value or an exception thrown.

After the task's `run` has finished (inside the "finally" block of the "monitored" block), `run` [requests `BlockManager`](#) to release all locks of the task (for the task's `taskId`). The locks are later used for lock leak detection.

`run` then [requests `TaskMemoryManager`](#) to clean up allocated memory (that helps finding memory leaks).

If `run` detects memory leak of the managed memory (i.e. the memory freed is greater than `0`) and [spark.unsafe.exceptionOnMemoryLeak](#) Spark property is enabled (it is not by default) and no exception was reported while the task ran, `run` reports a `SparkException`:

```
Managed memory leak detected; size = [freedMemory] bytes, TID = [taskId]
```

Otherwise, if [spark.unsafe.exceptionOnMemoryLeak](#) is disabled, you should see the following ERROR message in the logs instead:

```
ERROR Executor: Managed memory leak detected; size = [freedMemory] bytes, TID = [taskId]
```

Note

If `run` detects a memory leak, it leads to a `SparkException` or ERROR message in the logs.

If `run` detects lock leaking (i.e. the number of locks released) and [spark.storage.exceptionOnPinLeak](#) Spark property is enabled (it is not by default) and no exception was reported while the task ran, `run` reports a `SparkException`:

```
[releasedLocks] block locks were not released by TID = [taskId]:  
[releasedLocks separated by comma]
```

Otherwise, if [spark.storage.exceptionOnPinLeak](#) is disabled or the task reported an exception, you should see the following INFO message in the logs instead:

```
INFO Executor: [releasedLocks] block locks were not released by TID = [taskId]:  
[releasedLocks separated by comma]
```

Note

If `run` detects any lock leak, it leads to a `SparkException` or INFO message in the logs.

Righth after the "monitored" block, `run` records the current time as the task's finish time (as `taskFinish`).

If the `task was killed` (while it was running), `run` reports a `TaskKilledException` (and the `TaskRunner` exits).

`run` creates a `Serializer` and serializes the task's result. `run` measures the time to serialize the result.

Note

`run` uses `SparkEnv` to access the current `Serializer`. `SparkEnv` was specified when the owning `Executor` was created.

Important

This is when `TaskExecutor` serializes the computed value of a task to be sent back to the driver.

`run` records the `task metrics`:

- `executorDeserializeTime`
- `executorDeserializeCpuTime`
- `executorRunTime`
- `executorCpuTime`
- `jvmGCTime`
- `resultSerializationTime`

`run` collects the latest values of internal and external accumulators used in the task.

`run` creates a `DirectTaskResult` (with the serialized result and the latest values of accumulators).

`run` serializes the `DirectTaskResult` and gets the byte buffer's limit.

Note

A serialized `DirectTaskResult` is Java's `java.nio.ByteBuffer`.

`run` selects the proper serialized version of the result before sending it to `ExecutorBackend`.

`run` branches off based on the serialized `DirectTaskResult` byte buffer's limit.

When `maxResultSize` is greater than `0` and the serialized `DirectTaskResult` buffer limit 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.
```

Tip

Read about [spark.driver.maxResultSize](#).

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

In this case, `run` creates a [IndirectTaskResult](#) (with a `TaskResultBlockId` for the task's `taskId` and `resultSize`) and serializes it.

When `maxResultSize` is not positive or `resultSize` is smaller than `maxResultSize` but greater than `maxDirectResultSize`, `run` creates a `TaskResultBlockId` for the task's `taskId` and stores the serialized `DirectTaskResult` in `BlockManager` (as the `TaskResultBlockId` with `MEMORY_AND_DISK_SER` storage level).

You should see the following INFO message in the logs:

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

In this case, `run` creates a [IndirectTaskResult](#) (with a `TaskResultBlockId` for the task's `taskId` and `resultSize`) and serializes it.

Note

The difference between the two above cases is that the result is dropped or stored in `BlockManager` with `MEMORY_AND_DISK_SER` storage level.

When the two cases above do not hold, you should see the following INFO message in the logs:

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

`run` uses the serialized `DirectTaskResult` byte buffer as the final `serializedResult`.

Note	The final <code>serializedResult</code> is either a <code>IndirectTaskResult</code> (possibly with the block stored in <code>BlockManager</code>) or a <code>DirectTaskResult</code> .
------	---

`run` notifies `ExecutorBackend` that `taskId` is in `TaskState.FINISHED` state with the serialized result and removes `taskId` from the owning executor's `runningTasks` registry.

Note	<code>run</code> uses <code>ExecutorBackend</code> that is specified when <code>TaskRunner</code> is created.
------	---

Note	<code>TaskRunner</code> is Java's <code>Runnable</code> and the contract requires that once a <code>TaskRunner</code> has completed execution it must not be restarted.
------	---

When `run` catches a exception while executing the task, `run` acts according to its type (as presented in the following "run's Exception Cases" table and the following sections linked from the table).

Table 2. run's Exception Cases, TaskState and Serialized ByteBuffer

Exception Type	TaskState	Serialized ByteBuffer
<code>FetchFailedException</code>	<code>FAILED</code>	<code>TaskFailedReason</code>
<code>TaskKilledException</code>	<code>KILLED</code>	<code>TaskKilled</code>
<code>InterruptedException</code>	<code>KILLED</code>	<code>TaskKilled</code>
<code>CommitDeniedException</code>	<code>FAILED</code>	<code>TaskFailedReason</code>
<code>Throwable</code>	<code>FAILED</code>	<code>ExceptionFailure</code>

FetchFailedException

When `FetchFailedException` is reported while running a task, `run` `setTaskFinishedAndClearInterruptStatus`.

`run` requests `FetchFailedException` for the `TaskFailedReason`, serializes it and notifies `ExecutorBackend` that the task has failed (with `taskId`, `TaskState.FAILED`, and a serialized reason).

Note	<code>ExecutorBackend</code> was specified when <code>TaskRunner</code> was created.
------	--

Note

`run` uses a closure `Serializer` to serialize the failure reason. The `serializer` was created before `run` ran the task.

TaskKilledException

When `TaskKilledException` is reported while running a task, you should see the following INFO message in the logs:

```
INFO Executor killed [taskName] (TID [taskId])
```

`run` then `setTaskFinishedAndClearInterruptStatus` and notifies `ExecutorBackend` that the task has been killed (with `taskId`, `TaskState.KILLED`, and a serialized `TaskKilled` object).

InterruptedException (with Task Killed)

When `InterruptedException` is reported while running a task, and the task has been killed, you should see the following INFO message in the logs:

```
INFO Executor interrupted and killed [taskName] (TID [taskId])
```

`run` then `setTaskFinishedAndClearInterruptStatus` and notifies `ExecutorBackend` that the task has been killed (with `taskId`, `TaskState.KILLED`, and a serialized `TaskKilled` object).

Note

The difference between this `InterruptedException` and `TaskKilledException` is the INFO message in the logs.

CommitDeniedException

When `CommitDeniedException` is reported while running a task, `run` `setTaskFinishedAndClearInterruptStatus` and notifies `ExecutorBackend` that the task has failed (with `taskId`, `TaskState.FAILED`, and a serialized `TaskKilled` object).

Note

The difference between this `CommitDeniedException` and `FetchFailedException` is just the reason being sent to `ExecutorBackend`.

Throwable

When `run` catches a `Throwable`, you should see the following ERROR message in the logs (followed by the exception).

```
ERROR Exception in [taskName] (TID [taskId])
```

`run` then records the following task metrics (only when [Task](#) is available):

- [executorRunTime](#)
- [jvmGCTime](#)

`run` then [collects the latest values of internal and external accumulators](#) (with `taskFailed` flag enabled to inform that the collection is for a failed task).

Otherwise, when [Task](#) is not available, the accumulator collection is empty.

`run` converts the task accumulators to collection of `AccumulableInfo`, creates a `ExceptionFailure` (with the accumulators), and [serializes them](#).

Note	<code>run</code> uses a closure Serializer to serialize the <code>ExceptionFailure</code> .
------	---

Caution	FIXME Why does <code>run</code> create <code>new ExceptionFailure(t, accUpdates).withAccums(accums)</code> , i.e. accumulators occur twice in the object.
---------	---

`run` [setTaskFinishedAndClearInterruptStatus](#) and [notifies](#) `ExecutorBackend` that the task [has failed](#) (with `taskId`, `TaskState.FAILED`, and the serialized `ExceptionFailure`).

`run` may also trigger `SparkUncaughtExceptionHandler.uncaughtException(t)` if this is a fatal error.

Note	The difference between this most <code>Throwable</code> case and other <code>FAILED</code> cases (i.e. FetchFailedException and CommitDeniedException) is just the serialized <code>ExceptionFailure</code> vs a reason being sent to <code>ExecutorBackend</code> , respectively.
------	---

Killing Task — `kill` Method

```
kill(interruptThread: Boolean): Unit
```

`kill` marks the `TaskRunner` as [killed](#) and [kills the task](#) (if available and not [finished](#) already).

Note	<code>kill</code> passes the input <code>interruptThread</code> on to the task itself while killing it.
------	---

When executed, you should see the following INFO message in the logs:

```
INFO TaskRunner: Executor is trying to kill [taskName] (TID [taskId])
```

Note	<p>killed flag is checked periodically in run to stop executing the task. Once killed, the task will eventually stop.</p>
------	---

Settings

Table 3. Spark Properties

Spark Property	Default Value	Description
<code>spark.unsafe.exceptionOnMemoryLeak</code>	false	FIXME
<code>spark.storage.exceptionOnPinLeak</code>	false	FIXME

ExecutorSource

`ExecutorSource` is a [Source](#) of metrics for an [Executor](#). It uses an executor's [threadPool](#) for calculating the gauges.

Note

Every executor has its own separate `ExecutorSource` that is registered when [CoarseGrainedExecutorBackend](#) receives a [RegisteredExecutor](#).

The name of a `ExecutorSource` is **executor**.

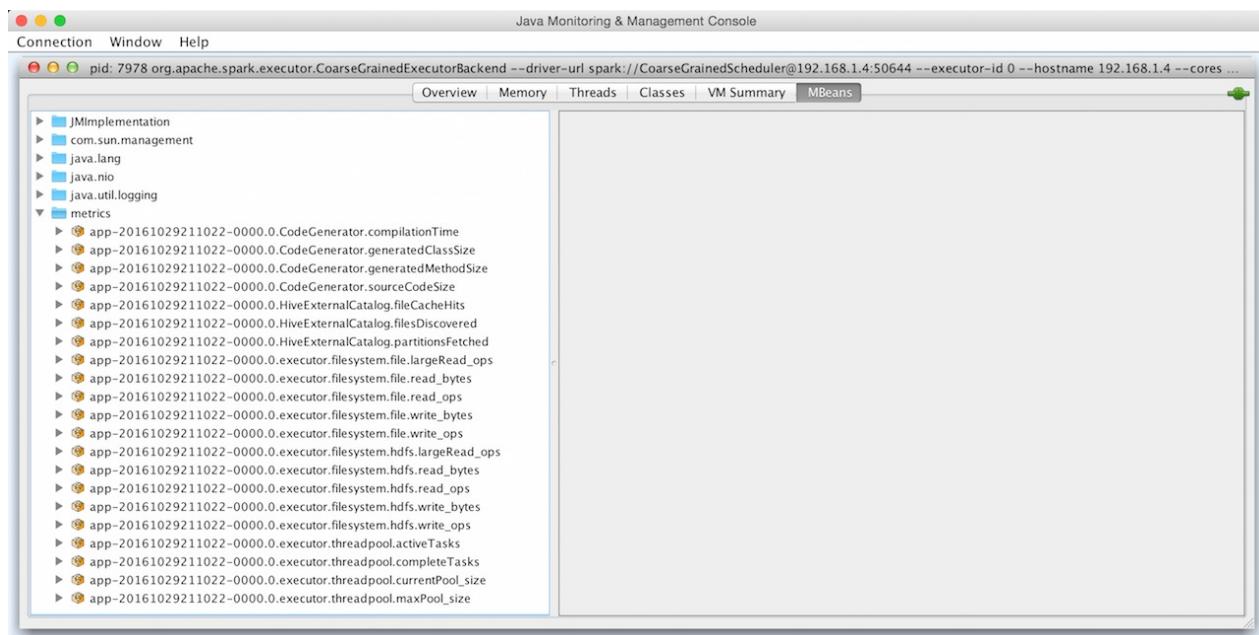


Figure 1. ExecutorSource in JConsole (using Spark Standalone)

Table 1. ExecutorSource Gauges

Gauge	Description
threadpool.activeTasks	Approximate number of threads that are actively executing tasks. Uses ThreadPoolExecutor.getActiveCount() .
threadpool.completeTasks	Approximate total number of tasks that have completed execution. Uses ThreadPoolExecutor.getCompletedTaskCount() .
threadpool.currentPool_size	Current number of threads in the pool. Uses ThreadPoolExecutor.getPoolSize() .
threadpool.maxPool_size	Maximum allowed number of threads that have ever simultaneously been in the pool Uses ThreadPoolExecutor.getMaximumPoolSize() .
filesystem.hdfs.read_bytes	Uses Hadoop's FileSystem.getAllStatistics() and getBytesRead() .
filesystem.hdfs.write_bytes	Uses Hadoop's FileSystem.getAllStatistics() and getBytesWritten() .
filesystem.hdfs.read_ops	Uses Hadoop's FileSystem.getAllStatistics() and getReadOps()
filesystem.hdfs.largeRead_ops	Uses Hadoop's FileSystem.getAllStatistics() and getLargeReadOps() .
filesystem.hdfs.write_ops	Uses Hadoop's FileSystem.getAllStatistics() and getWriteOps() .
filesystem.file.read_bytes	The same as hdfs but for file scheme.
filesystem.file.write_bytes	The same as hdfs but for file scheme.
filesystem.file.read_ops	The same as hdfs but for file scheme.
filesystem.file.largeRead_ops	The same as hdfs but for file scheme.
filesystem.file.write_ops	The same as hdfs but for file scheme.

Master

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

The master acquires cluster nodes to run executors.

Caution

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

Workers

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

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

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

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

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

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

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

New vocabulary often faced in Spark UI

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

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

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

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

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

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

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

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

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

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

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

Once stages are defined, Spark will generate **tasks** from **stages**. The first stage will create **ShuffleMapTasks** with the last stage creating **ResultTasks** because in the last stage, one action operation is included to produce results.

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

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

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

Cleanup

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

Settings

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

Anatomy of Spark Application

Every Spark application starts from creating [SparkContext](#).

Note Without [SparkContext](#) no computation (as a Spark job) can be started.

Note A Spark application is an instance of [SparkContext](#). Or, put it differently, a Spark context constitutes a Spark application.

A Spark application is uniquely identified by a pair of the [application](#) and [application attempt](#) ids.

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](#).

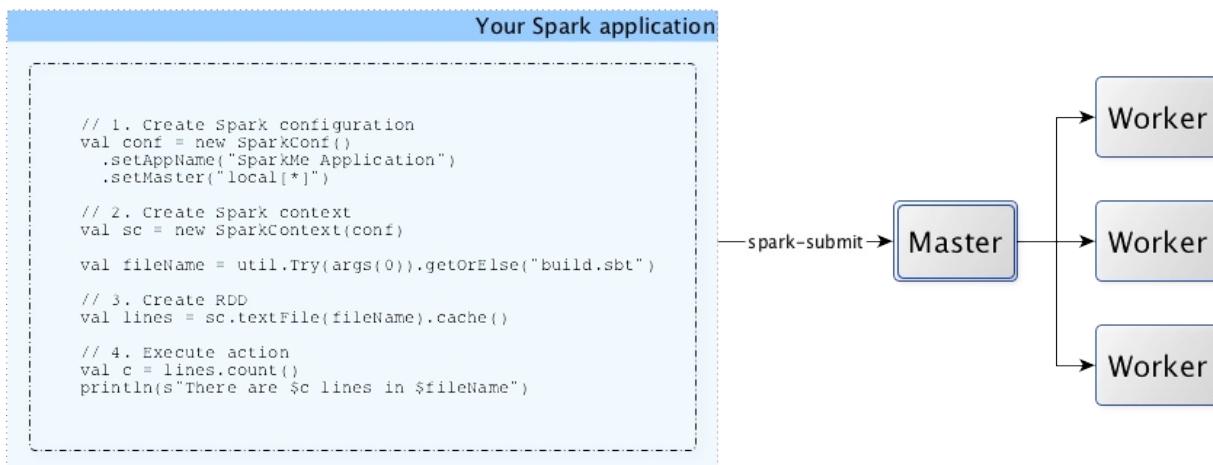


Figure 1. Submitting Spark application to master using master URL

Note

Your Spark application can run locally or on the cluster which is based on the cluster manager and the deploy mode (`--deploy-mode`). Refer to [Deployment Modes](#).

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

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

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

SparkConf — Spark Application's Configuration

Tip	Refer to Spark Configuration 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"> • Describe <code>SparkConf</code> object for the application configuration. • the default configs • system properties

There are three ways to configure Spark and user programs:

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

`setIfMissing` Method

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

`isExecutorStartupConf` Method

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

`set` Method

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

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 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](#).

Tip 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
...
```

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 Spark Properties

There are the following places where a Spark application looks for Spark properties (in the order of importance from the least important to the most important):

- `conf/spark-defaults.conf` - the configuration file with the default Spark properties.
Read [spark-defaults.conf](#).
- `--conf` or `-c` - the command-line option used by [spark-submit](#) (and other shell scripts that use `spark-submit` or `spark-class` under the covers, e.g. `spark-shell`)
- `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
```

Unique Identifier of Spark Application — `getAppId` Method

```
getAppId: String
```

`getAppId` gives `spark.app.id` Spark property or reports `NoSuchElementException` if not set.

Note

`getAppId` is used when:

- `NettyBlockTransferService` is initialized (and creates a `NettyBlockRpcServer` as well as saves the identifier for later use).
- `Executor` is created (in non-local mode and requests `BlockManager` to initialize).

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
spark.master		Master URL
spark.app.id	TaskScheduler.applicationId()	Unique identifier of a Spark application that Spark uses to uniquely identify metric sources . Set when <code>sparkContext</code> is created (right after <code>TaskScheduler</code> is started that actually gives the identifier).
spark.app.name		Application Name

Spark Properties and spark-defaults.conf Properties File

Spark properties are the means of tuning the execution environment for your Spark applications.

The default Spark properties file is `$SPARK_HOME/conf/spark-defaults.conf` that could be overridden using `spark-submit`'s `--properties-file` command-line option.

Table 1. Environment Variables

Environment Variable	Default Value	Description
<code>SPARK_CONF_DIR</code>	<code> \${SPARK_HOME}/conf</code>	Spark's configuration directory (with <code>spark-defaults.conf</code>)

Tip Read the official documentation of Apache Spark on [Spark Configuration](#).

spark-defaults.conf — Default Spark Properties File

`spark-defaults.conf` (under `SPARK_CONF_DIR` or `$SPARK_HOME/conf`) is the default properties file with the Spark properties of your Spark applications.

Note `spark-defaults.conf` is loaded by [AbstractCommandBuilder's](#) `loadPropertiesFile` internal method.

Calculating Path of Default Spark Properties — `Utils.getDefaultPropertiesFile` method

```
getDefaultPropertiesFile(env: Map[String, String] = sys.env): String
```

`getDefaultPropertiesFile` calculates the absolute path to `spark-defaults.conf` properties file that can be either in directory specified by `SPARK_CONF_DIR` environment variable or `$SPARK_HOME/conf` directory.

Note `getDefaultPropertiesFile` is a part of `private[spark]` `org.apache.spark.util.Utils` object.

Environment Variables

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 non-local cluster deployments .
------	--

You can control the deploy mode of a Spark application using `spark-submit`'s [--deploy-mode command-line option](#) or `spark.submit.deployMode` [Spark property](#).

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

Client Deploy Mode

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

Cluster Deploy Mode

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

spark.submit.deployMode

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

SparkContext — Entry Point to Spark Core

`SparkContext` (aka **Spark context**) is the heart of a Spark application.

Note You could also assume that a `SparkContext` instance *is* a Spark application.

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

Once a `SparkContext` is created you can use it to [create RDDs, accumulators and broadcast variables](#), access Spark services and [run jobs](#) (until `SparkContext` is stopped).

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).

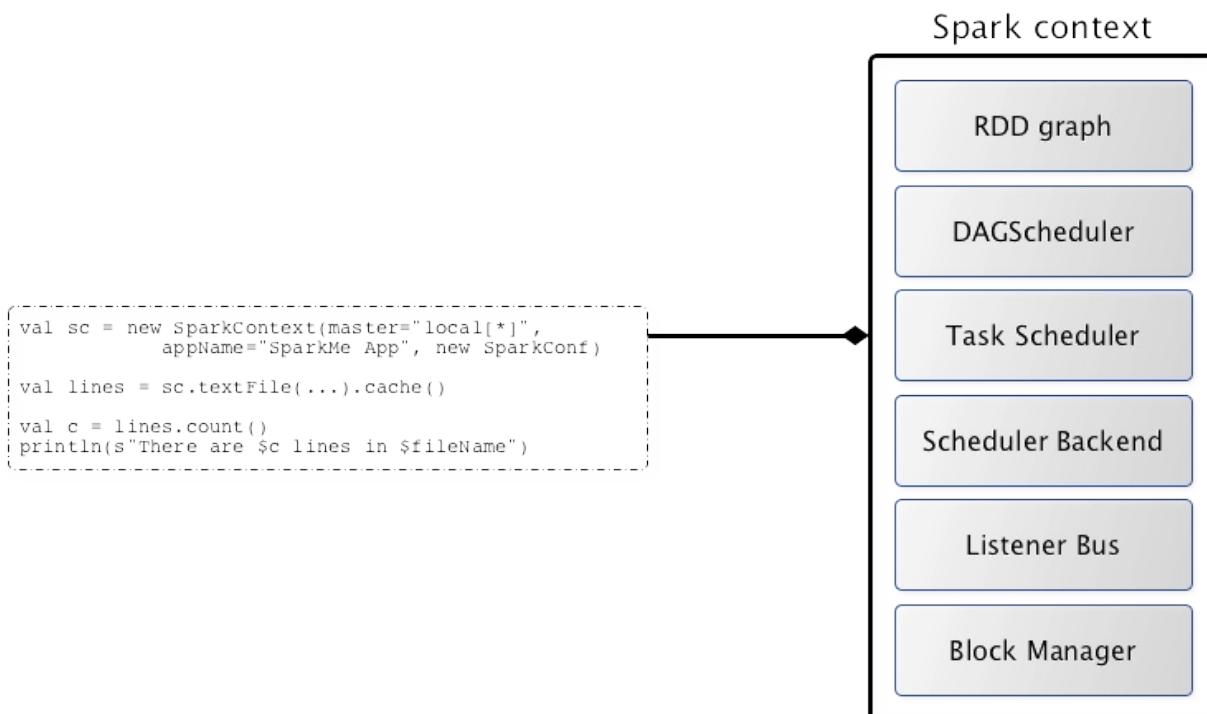


Figure 1. Spark context acts as the master of your Spark application

`SparkContext` offers the following functions:

- Getting current status of a Spark application
 - [SparkEnv](#)
 - [SparkConf](#)
 - [deployment environment \(as master URL\)](#)

- application name
 - unique identifier of execution attempt
 - deploy mode
 - default level of parallelism that specifies the number of partitions in RDDs when they are created without specifying the number explicitly by a user.
 - Spark user
 - the time (in milliseconds) when `SparkContext` was created
 - Spark version
 - Storage status
- Setting Configuration
 - master URL
 - Local Properties — Creating Logical Job Groups
 - Default Logging Level
 - Creating Distributed Entities
 - RDDs
 - Accumulators
 - Broadcast variables
 - Accessing services, e.g. TaskScheduler, LiveListenerBus, BlockManager, SchedulerBackends, ShuffleManager and the optional ContextCleaner.
 - Running jobs synchronously
 - Submitting jobs asynchronously
 - Cancelling a job
 - Cancelling a stage
 - Assigning custom Scheduler Backend, TaskScheduler and DAGScheduler
 - Closure cleaning
 - Accessing persistent RDDs
 - Unpersisting RDDs, i.e. marking RDDs as non-persistent

- Registering SparkListener
- Programmable Dynamic Allocation

Table 1. SparkContext's Internal Registries and Counters

Name	Description
<code>persistentRdds</code>	<p>Lookup table of persistent/cached RDDs per their ids.</p> <p>Used when <code>SparkContext</code> is requested to:</p> <ul style="list-style-type: none"> • <code>persistRDD</code> • <code>getRDDStorageInfo</code> • <code>getPersistentRDDs</code> • <code>unpersistRDD</code>

Table 2. SparkContext's Internal Properties

Name	Initial Value	Description
<code>_taskScheduler</code>	(uninitialized)	<code>TaskScheduler</code>

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

Tip Enable `INFO` logging level for `org.apache.spark.SparkContext` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.SparkContext=INFO
```

Refer to [Logging](#).

Removing RDD Blocks from BlockManagerMaster — `unpersistRDD` Internal Method

```
unpersistRDD(rddId: Int, blocking: Boolean = true): Unit
```

`unpersistRDD` requests `BlockManagerMaster` to remove the blocks for the RDD (given `rddId`).

Note `unpersistRDD` uses `SparkEnv` to access the current `BlockManager` that is in turn used to access the current `BlockManagerMaster`.

`unpersistRDD` removes `rddId` from `persistentRdds` registry.

In the end, `unpersistRDD` posts a `SparkListenerUnpersistRDD` (with `rddId`) to [LiveListenerBus Event Bus](#).

	<code>unpersistRDD</code> is used when:
Note	<ul style="list-style-type: none"> • <code>ContextCleaner</code> does <code>doCleanupRDD</code> • <code>SparkContext</code> unpersists an RDD (i.e. marks an RDD as non-persistent)

Unique Identifier of Spark Application — `applicationId` Method

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

`postApplicationStart` Internal Method

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

`postApplicationEnd` Method

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

`clearActiveContext` Method

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

Accessing persistent RDDs — `getPersistentRDDs` Method

```
getPersistentRDDs: Map[Int, RDD[_]]
```

`getPersistentRDDs` returns the collection of RDDs that have marked themselves as persistent via [cache](#).

Internally, `getPersistentRDDs` returns `persistentRdds` internal registry.

Cancelling Job — `cancelJob` Method

```
cancelJob(jobId: Int)
```

`cancelJob` requests `DAGScheduler` to cancel a Spark job.

Cancelling Stage — `cancelStage` Methods

```
cancelStage(stageId: Int): Unit
cancelStage(stageId: Int, reason: String): Unit
```

`cancelStage` simply requests `DAGScheduler` to cancel a Spark stage (with an optional `reason`).

Note

`cancelStage` is used when `StagesTab` handles a kill request (from a user in web UI).

Programmable Dynamic Allocation

`SparkContext` offers the following methods as the developer API for dynamic allocation of executors:

- `requestExecutors`
- `killExecutors`
- `requestTotalExecutors`
- (private!) `getExecutorIds`

Requesting New Executors — `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

Requesting Total Executors — `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 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
```

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 (!)

Creating SparkContext Instance

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

Note

You may want to read [Inside Creating SparkContext](#) to learn what happens behind the scenes when `SparkContext` is created.

Getting Existing or Creating New SparkContext — `getOrCreate` Methods

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

Accessing Current SparkEnv — `env` Method

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

Getting Current SparkConf — `getConf` Method

<code>getConf: SparkConf</code>

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

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

Deployment Environment — `master` Method

<code>master: String</code>

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

Application Name — `appName` Method

<code>appName: String</code>

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

Note	<code>appName</code> is used when SparkDeploySchedulerBackend starts , SparkUI creates a web UI , when <code>postApplicationStart</code> is executed, and for Mesos and checkpointing in Spark Streaming.
------	---

Unique Identifier of Execution Attempt — `applicationAttemptId` Method

<code>applicationAttemptId: Option[String]</code>

`applicationAttemptId` gives the unique identifier of the execution attempt of a Spark application.

Note

`applicationAttemptId` is used when:

- `ShuffleMapTask` and `ResultTask` are created
- `SparkContext` announces that a Spark application has started

Storage Status (of All BlockManagers)

— `getExecutorStorageStatus` Method

```
getExecutorStorageStatus: Array[StorageStatus]
```

`getExecutorStorageStatus` requests `BlockManagerMaster` for storage status (of all `BlockManagers`).

Note

`getExecutorStorageStatus` is a developer API.

Note

`getExecutorStorageStatus` is used when:

- `SparkContext` is requested for storage status of cached RDDs
- `SparkStatusTracker` is requested for information about all known executors

Deploy Mode — `deployMode` Method

```
deployMode: String
```

`deployMode` returns the current value of `spark.submit.deployMode` setting or `client` if not set.

Scheduling Mode — `getSchedulingMode` Method

```
getSchedulingMode: SchedulingMode.SchedulingMode
```

`getSchedulingMode` returns the current `Scheduling Mode`.

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).

All Pools — `getAllPools` Method

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

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

Note	<code>TaskScheduler.rootPool</code> is part of the TaskScheduler Contract .
------	---

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

Caution	FIXME Where is the method used?
---------	---

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

Default Level of Parallelism

```
defaultParallelism: Int
```

`defaultParallelism` requests [TaskScheduler](#) for the [default level of parallelism](#).

Note	Default level of parallelism specifies the number of partitions in RDDs when created without specifying them explicitly by a user.
------	---

Note	<p><code>defaultParallelism</code> is used in SparkContext.parallelize, SparkContext.range and SparkContext.makeRDD (as well as Spark Streaming's <code>DStream.countByValue</code> and <code>DStream.countByValueAndWindow</code> et al.).</p> <p><code>defaultParallelism</code> is also used to instantiate HashPartitioner and for the minimum number of partitions in HadoopRDDs.</p>
------	--

Current Spark Scheduler (aka TaskScheduler)

— taskScheduler Property

```
taskScheduler: TaskScheduler
taskScheduler_=(ts: TaskScheduler): Unit
```

`taskScheduler` manages (i.e. reads or writes) [_taskScheduler](#) internal property.

Getting Spark Version — version Property

```
version: String
```

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

makeRDD Method

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

Submitting Jobs Asynchronously — submitJob Method

```
submitJob[T, U, R](
  rdd: RDD[T],
  processPartition: Iterator[T] => U,
  partitions: Seq[Int],
  resultHandler: (Int, U) => Unit,
  resultFunc: => R): SimpleFutureAction[R]
```

`submitJob` submits a job in an asynchronous, non-blocking way to [DAGScheduler](#).

It cleans the `processPartition` input function argument and returns an instance of [SimpleFutureAction](#) that holds the [JobWaiter](#) instance.

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.

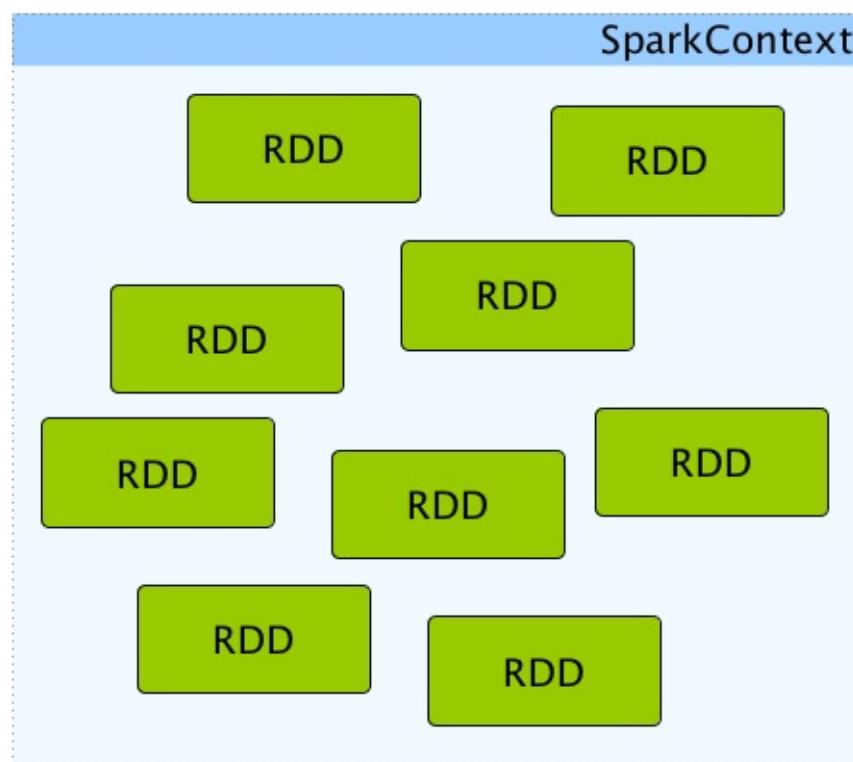


Figure 2. A Spark context creates a living space for RDDs.

Creating RDD — `parallelize` Method

`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 RDD (Marking RDD as Non-Persistent) — `unpersist` Method

Caution

FIXME

`unpersist` 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 Accumulator— `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 `acc` [accumulator](#) (with the current `SparkContext`).

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

Accumulable	Value
counter	45

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 3. Accumulators in the Spark UI

Tip

You can register custom accumulators using [register](#) methods.

Creating Broadcast Variable — `broadcast` Method

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

`broadcast` method creates a [broadcast variable](#). It is a shared memory with `value` (as broadcast blocks) on the driver and later on all Spark executors.

```
val sc: SparkContext = ???
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 the broadcast variable is used multiple times.

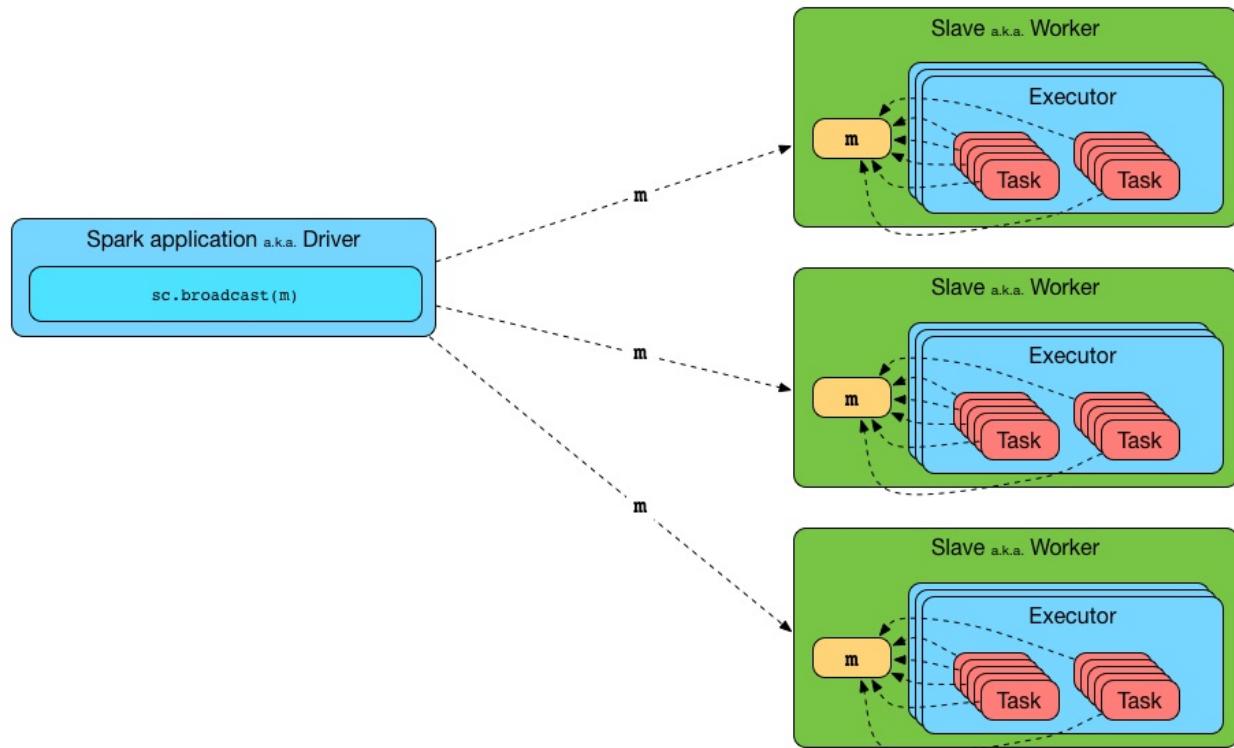


Figure 4. Broadcasting a value to executors

Internally, `broadcast` requests the [current BroadcastManager](#) to create a new broadcast variable.

Note	The current <code>BroadcastManager</code> is available using <code>SparkEnv.broadcastManager</code> attribute and is always <code>BroadcastManager</code> (with few internal configuration changes to reflect where it runs, i.e. inside the driver or executors).
------	--

You should see the following INFO message in the logs:

```
INFO SparkContext: Created broadcast [id] from [callSite]
```

If `contextCleaner` is defined, the new broadcast variable is registered for cleanup.

	Spark does not support broadcasting RDDs.
Note	<pre>scala> sc.broadcast(sc.range(0, 10)) java.lang.IllegalArgumentException: requirement failed: Can not directly broadcast at scala.Predef\$.require(Predef.scala:224) at org.apache.spark.SparkContext.broadcast(SparkContext.scala:1392) ... 48 elided</pre>

Once created, the broadcast variable (and other blocks) are displayed per executor and the driver in web UI (under [Executors tab](#)).

Broadcast and RDD blocks (after cache and persist)													Search:		
Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Shuffle Input	Shuffle Read	Shuffle Write	Thread Dump
driver	10.1.15.114:62791	Active	3	10.4 KB / 384.1 MB	0.0 B	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	Thread Dump	
0	10.1.15.114:62799	Active	8	8.2 KB / 384.1 MB	0.0 B	2	0	0	7	7	2 s (0.2 s)	0.0 B	118 B	236 B	Thread Dump
1	10.1.15.114:62801	Active	9	12 KB / 384.1 MB	0.0 B	2	0	0	7	7	2 s (0.2 s)	0.0 B	118 B	236 B	Thread Dump

Figure 5. Broadcast Variables In web UI's Executors Tab

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	FIXME Why is HttpFileServer used for addJar?
---------	--

SparkContext as Application-Wide Counter

SparkContext keeps track of:

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

Running Job Synchronously — `runJob` Methods

RDD actions run `jobs` using one of `runJob` methods.

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

`runJob` executes a function on one or many partitions of a RDD (in a `SparkContext` space) to produce a collection of values per partition.

Note

`runJob` can only work when a `SparkContext` is *not stopped*.

Internally, `runJob` first makes sure that the `SparkContext` is not `stopped`. If it is, you should see the following `IllegalStateException` exception in the logs:

```
java.lang.IllegalStateException: SparkContext has been shutdown
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1893)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1914)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1934)
  ... 48 elided
```

`runJob` then calculates the call site and cleans a `func` closure.

You should see the following INFO message in the logs:

```
INFO SparkContext: Starting job: [callSite]
```

With `spark.logLineage` enabled (which is not by default), you should see the following INFO message with `toDebugString` (executed on `rdd`):

```
INFO SparkContext: RDD's recursive dependencies:  
[toDebugString]
```

`runJob` requests `DAGScheduler` to run a job.

Tip	<code>runJob</code> just prepares input parameters for <code>DAGScheduler</code> to run a job.
-----	--

After `DAGScheduler` is done and the job has finished, `runJob` stops `consoleProgressBar` and performs `RDD checkpointing` of `rdd`.

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

```
// RDD to work with  
val lines = sc.parallelize(Seq("hello world", "nice to see you"))  
  
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 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).

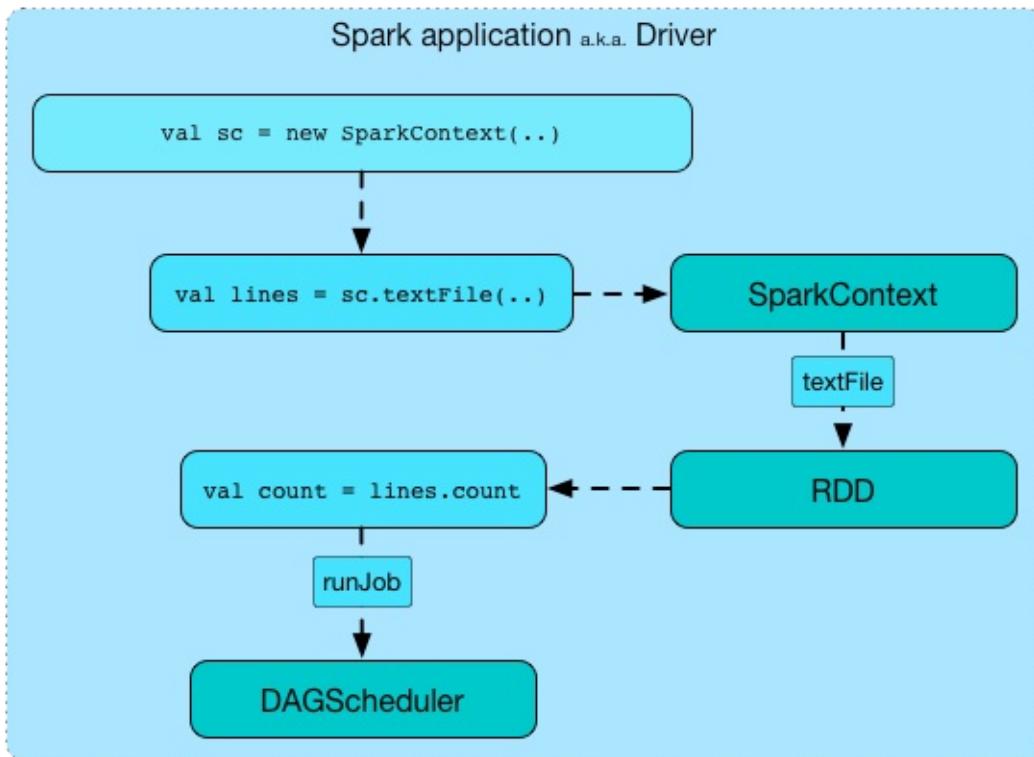


Figure 6. Executing action

Stopping SparkContext — stop Method

```
stop(): Unit
```

`stop` stops the `SparkContext`.

Internally, `stop` enables `stopped` internal flag. If already stopped, you should see the following INFO message in the logs:

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

`stop` then does the following:

1. Removes `_shutdownHookRef` from `ShutdownHookManager`.
2. Posts a `SparkListenerApplicationEnd` (to `LiveListenerBus Event Bus`).
3. Stops web UI
4. Requests `MetricSystem` to report metrics (from all registered sinks).
5. Stops `ContextCleaner`.
6. Requests `ExecutorAllocationManager` to stop.

7. If `LiveListenerBus` was started, requests `LiveListenerBus` to stop.
8. Requests `EventLoggingListener` to stop.
9. Requests `DAGScheduler` to stop.
10. Requests `RpcEnv` to stop `HeartbeatReceiver` endpoint.
11. Requests `ConsoleProgressBar` to stop.
12. Clears the reference to `TaskScheduler`, i.e. `_taskScheduler` is `null`.
13. Requests `SparkEnv` to stop and clears `SparkEnv`.
14. Clears `SPARK_YARN_MODE` flag.
15. Clears an active `SparkContext`.

Ultimately, 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 <code>spark.extraListeners</code> 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	<code>FIXME</code> 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 Logging 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\(logLevel\)](#) that it then uses to set using [org.apache.log4j.LogManager.getLogger\(\).setLevel\(level\)](#).

You can directly set the logging level using [org.apache.log4j.LogManager.getLogger\(\)](#).

Tip

```
LogManager.getLogger("org").setLevel(Level.OFF)
```

Closure Cleaning — `clean` Method

```
clean(f: F, checkSerializable: Boolean = true): F
```

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.

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

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

Tip

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

Refer to [Logging](#).

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

[FIXME](#) 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

[SparkHadoopUtil.get.newConfiguration](#) is used.

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

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

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

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

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

listenerBus — LiveListenerBus Event Bus

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

Note	It is created and started when SparkContext starts and, since it is a single-JVM event bus, 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 Property

<code>startTime: Long</code>

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

<code>scala> sc.startTime</code> <code>res0: Long = 1464425605653</code>
--

Spark User — sparkUser Property

<code>sparkUser: String</code>

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

Note	It is computed when SparkContext is created using <code>Utils.getCurrentUserName</code> .
------	---

Submitting ShuffleDependency for Execution — submitMapStage Internal Method

<code>submitMapStage[K, V, C](</code> <code>dependency: ShuffleDependency[K, V, C]): SimpleFutureAction[MapOutputStatistics]</code>
--

`submitMapStage` submits the input `shuffleDependency` to `DAGScheduler` for execution and returns a `SimpleFutureAction`.

Internally, `submitMapStage` calculates the call site first and submits it with `localProperties`.

Note

Interestingly, `submitMapStage` is used exclusively when Spark SQL's `ShuffleExchange` physical operator is executed.

Note

`submitMapStage` seems related to [Adaptive Query Planning / Adaptive Scheduling](#).

Calculating Call Site — `getCallSite` Method

Caution

FIXME

Cancelling Job Group — `cancelJobGroup` Method

```
cancelJobGroup(groupId: String)
```

`cancelJobGroup` requests `DAGScheduler` to cancel a group of active Spark jobs.

Note

`cancelJobGroup` is used exclusively when `SparkExecuteStatementOperation` does `cancel`.

Cancelling All Running and Scheduled Jobs — `cancelAllJobs` Method

Caution

FIXME

Note

`cancelAllJobs` is used when `spark-shell` is terminated (e.g. using Ctrl+C, so it can in turn terminate all active Spark jobs) or `SparksQLCLIDriver` is terminated.

`setJobGroup` Method

```
setJobGroup(
  groupId: String,
  description: String,
  interruptOnCancel: Boolean = false): Unit
```

Caution

FIXME

`cleaner` Method

```
cleaner: Option[ContextCleaner]
```

`cleaner` is a `private[spark]` method to get the optional application-wide [ContextCleaner](#).

Note

`ContextCleaner` is created when `SparkContext` is created with `spark.cleaner.referenceTracking` [Spark property enabled](#) (which it is by default).

Finding Preferred Locations (Placement Preferences) for RDD Partition — `getPreferredLocs` Method

```
getPreferredLocs(rdd: RDD[_], partition: Int): Seq[TaskLocation]
```

`getPreferredLocs` simply [requests DAGScheduler](#) for the preferred locations for `partition`.

Note

Preferred locations of a partition of a RDD are also called **placement preferences** or **locality preferences**.

Note

`getPreferredLocs` is used in `CoalescedRDDPartition`, `DefaultPartitionCoalescer` and `PartitionerAwareUnionRDD`.

Registering RDD in `persistentRdds` Internal Registry — `persistRDD` Internal Method

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

`persistRDD` registers `rdd` in `persistentRdds` internal registry.

Note

`persistRDD` is used exclusively when `RDD` is [persisted or locally checkpointed](#).

Getting Storage Status of Cached RDDs (as `RDDInfos`) — `getRDDStorageInfo` Methods

```
getRDDStorageInfo: Array[RDDInfo] (1)
getRDDStorageInfo(filter: RDD[_] => Boolean): Array[RDDInfo] (2)
```

1. Part of Spark's Developer API that uses <2> filtering no RDDs

`getRDDStorageInfo` takes all the RDDs (from `persistentRdds` registry) that match `filter` and creates a collection of `RDDInfos`.

`getRDDStorageInfo` then updates the `RDDInfos` with the current status of all BlockManagers (in a Spark application).

In the end, `getRDDStorageInfo` gives only the RDD that are cached (i.e. the sum of memory and disk sizes as well as the number of partitions cached are greater than `0`).

Note

`getRDDStorageInfo` is used when `RDD` is requested for RDD lineage graph.

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

Table 3. Environment Variables

Environment Variable	Default Value	Description
SPARK_EXECUTOR_MEMORY	1024	Amount of memory to allocate for a Spark executor in MB. See Executor Memory .
SPARK_USER		The user who is running <code>SparkContext</code> . Available later as <code>sparkUser</code> .

HeartbeatReceiver RPC Endpoint

`HeartbeatReceiver` is a [ThreadSafeRpcEndpoint](#) registered on the driver under the name `HeartbeatReceiver`.

`HeartbeatReceiver` receives `Heartbeat` messages from executors that Spark uses as the mechanism to receive accumulator updates (with task metrics and a Spark application's accumulators) and [pass them along to](#) `TaskScheduler`.

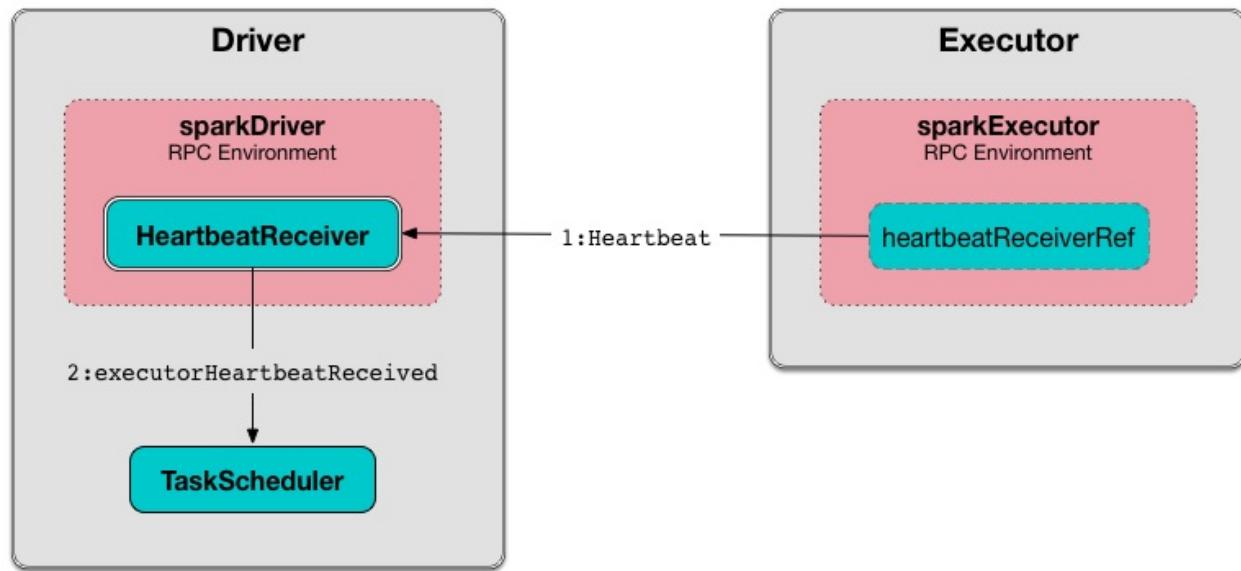


Figure 1. HeartbeatReceiver RPC Endpoint and Heartbeats from Executors

Note

`HeartbeatReceiver` is registered immediately after a Spark application is started, i.e. when `SparkContext` is created.

`HeartbeatReceiver` is a [SparkListener](#) to get notified when [a new executor is added](#) to or [no longer available](#) in a Spark application. `HeartbeatReceiver` tracks executors (in `executorLastSeen` registry) to handle `Heartbeat` and `ExpireDeadHosts` messages from executors that are assigned to the Spark application.

Table 1. HeartbeatReceiver RPC Endpoint's Messages (in alphabetical order)

Message	Description
ExecutorRemoved	Posted when <code>HeartbeatReceiver</code> is notified that an executor is no longer available (to a Spark application).
ExecutorRegistered	Posted when <code>HeartbeatReceiver</code> is notified that a new executor has been registered (with a Spark application).
ExpireDeadHosts	FIXME
Heartbeat	Posted when <code>Executor</code> informs that it is alive and reports task metrics.
TaskSchedulerIsSet	Posted when <code>SparkContext</code> informs that <code>TaskScheduler</code> is available.

Table 2. HeartbeatReceiver's Internal Registries and Counters

Name	Description
<code>executorLastSeen</code>	Executor ids and the timestamps of when the last heartbeat was received.
<code>scheduler</code>	<code>TaskScheduler</code>

Tip	<p>Enable <code>DEBUG</code> or <code>TRACE</code> logging levels for <code>org.apache.spark.HeartbeatReceiver</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.HeartbeatReceiver=TRACE</pre> <p>Refer to Logging.</p>
-----	---

Creating HeartbeatReceiver Instance

`HeartbeatReceiver` takes the following when created:

- `SparkContext`
- `Clock`

`HeartbeatReceiver` registers itself as a `SparkListener`.

`HeartbeatReceiver` initializes the internal registries and counters.

Starting HeartbeatReceiver RPC Endpoint — 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.

ExecutorRegistered

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

When received, `HeartbeatReceiver` registers the `executorId` executor and the current time (in `executorLastSeen` internal registry).

Note

`HeartbeatReceiver` uses the internal [Clock](#) to know the current time.

ExecutorRemoved

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

When `ExecutorRemoved` arrives, `HeartbeatReceiver` removes `executorId` from `executorLastSeen` internal registry.

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](#).

Heartbeat

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

When received, `HeartbeatReceiver` finds the `executorId` executor (in [executorLastSeen](#) registry).

When the executor is found, `HeartbeatReceiver` updates the time the heartbeat was received (in [executorLastSeen](#)).

Note	<code>HeartbeatReceiver</code> uses the internal Clock to know the current time.
------	--

`HeartbeatReceiver` then submits an asynchronous task to notify `TaskScheduler` that the [heartbeat was received from the executor](#) (using [TaskScheduler](#) internal reference).

`HeartbeatReceiver` posts a `HeartbeatResponse` back to the executor (with the response from `TaskScheduler` whether the executor has been registered already or not so it may eventually need to re-register).

If however the executor was not found (in [executorLastSeen](#) registry), i.e. the executor was not registered before, you should see the following DEBUG message in the logs and the response is to notify the executor to re-register.

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

In a very rare case, when [TaskScheduler](#) is not yet assigned to `HeartbeatReceiver`, you should see the following WARN message in the logs and the response is to notify the executor to re-register.

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

Note `TaskScheduler` can be unassigned when no `TaskSchedulerIsSet` has not been received yet.

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

TaskSchedulerIsSet

`TaskSchedulerIsSet`

When received, `HeartbeatReceiver` sets the internal reference to `TaskScheduler`.

Note `HeartbeatReceiver` uses `SparkContext` that is given when `HeartbeatReceiver` is created.

onExecutorAdded Method

`onExecutorAdded(executorAdded: SparkListenerExecutorAdded): Unit`

`onExecutorAdded` simply sends a `ExecutorRegistered` message to itself (that in turn registers an executor).

Note `onExecutorAdded` is a part of `SparkListener contract` to announce that a new executor was registered with a Spark application.

Sending ExecutorRegistered Message to Itself — addExecutor Internal Method

`addExecutor(executorId: String): Option[Future[Boolean]]`

`addExecutor` sends a `ExecutorRegistered` message (to register `executorId` executor).

Note `addExecutor` is used when `HeartbeatReceiver` is notified that a new executor was added.

onExecutorRemoved Method

`onExecutorRemoved(executorRemoved: SparkListenerExecutorRemoved): Unit`

`onExecutorRemoved` simply passes the call to `removeExecutor` (that in turn unregisters an executor).

Note

`onExecutorRemoved` is a part of [SparkListener contract](#) to announce that an executor is no longer available for a Spark application.

Sending ExecutorRemoved Message to Itself — `removeExecutor` Method

```
removeExecutor(executorId: String): Option[Future[Boolean]]
```

`removeExecutor` sends a [ExecutorRemoved](#) message to itself (passing in `executorId`).

Note

`removeExecutor` is used when `HeartbeatReceiver` is notified that an executor is no longer available.

Stopping HeartbeatReceiver RPC Endpoint — `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**.

expireDeadHosts Internal Method

```
expireDeadHosts(): Unit
```

Caution

FIXME

Note

`expireDeadHosts` is used when `HeartbeatReceiver` receives a `ExpireDeadHosts` message.

Settings

Table 3. Spark Properties

Spark Property	Default Value
<code>spark.storage.blockManagerTimeoutIntervalMs</code>	<code>60s</code>
<code>spark.storage.blockManagerSlaveTimeoutMs</code>	<code>120s</code>
<code>spark.network.timeout</code>	<code>spark.storage.blockManagerSlaveTimeoutMs</code>
<code>spark.network.timeoutInterval</code>	<code>spark.storage.blockManagerTimeoutIntervalMs</code>

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](#).

`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` flag 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 enabled (it is not by default), the short name of the [CompressionCodec](#) is assigned to `_eventLogCodec`. The config key is `spark.io.compression.codec` (default: `lz4`).

Tip	Read about compression codecs in Compression .
-----	--

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

Caution	FIXME : What's <code>externalBlockStoreFolderName</code> ?
---------	--

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 <code>MetadataCleaner</code> ?
---------	---

Creating SparkStatusTracker

`SparkContext` creates a [SparkStatusTracker](#).

Creating Optional ConsoleProgressBar

`SparkContext` creates the optional [ConsoleProgressBar](#) when `spark.ui.showConsoleProgress` property is enabled and the `INFO` logging level for `SparkContext` is disabled.

`SparkUI` creates a web UI (as `_ui`) if the property `spark.ui.enabled` is enabled (i.e. `true`).

Caution	FIXME Where's <code>_ui</code> used?
---------	--

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

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

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

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

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

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

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

`SPARK_EXECUTOR_MEMORY`, `_conf.getExecutorEnv`, and `SPARK_USER`.

`SparkContext` registers `HeartbeatReceiver` RPC endpoint.

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

Note	The internal <code>_schedulerBackend</code> and <code>_taskScheduler</code> are used by <code>schedulerBackend</code> and <code>taskScheduler</code> 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).

Starting TaskScheduler

`SparkContext` starts `TaskScheduler`.

Setting Unique Identifiers of Spark Application and Its Execution Attempt—`_applicationId` and `_applicationAttemptId`

`SparkContext` sets the internal fields — `_applicationId` and `_applicationAttemptId` — (using `applicationId` and `applicationAttemptId` methods from the [TaskSchedulerContract](#)).

Note	<code>SparkContext</code> requests <code>TaskScheduler</code> for the unique identifier of a Spark application (that is currently only implemented by <code>TaskSchedulerImpl</code> that uses <code>SchedulerBackend</code> to request the identifier).
------	--

Note	The unique identifier of a Spark application is used to initialize SparkUI and BlockManager .
------	---

Note	<code>_applicationAttemptId</code> is used when <code>SparkContext</code> is requested for the unique identifier of execution attempt of a Spark application and when <code>EventLoggingListener</code> is created.
------	---

Setting `spark.app.id` Spark Property in `SparkConf`

`SparkContext` sets `spark.app.id` property to be the unique identifier of a Spark application and, if enabled, passes it on to `SparkUI`.

Initializing BlockManager

The `BlockManager` (for the driver) is initialized (with `_applicationId`).

Starting MetricsSystem

`SparkContext` starts `MetricsSystem`.

Note	<code>SparkContext</code> starts <code>MetricsSystem</code> after setting <code>spark.app.id</code> Spark property as <code>MetricsSystem</code> uses it to build unique identifiers for metrics sources.
------	---

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 <code>_eventLogger</code> required to be the internal field of <code>SparkContext</code> ? Where is this used?
---------	--

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

Note	<code>_executorAllocationManager</code> is exposed (as a method) to YARN scheduler backends to reset their state to the initial state.
------	--

If `spark.cleaner.referenceTracking` Spark property is enabled (i.e. `true`), `SparkContext` creates `ContextCleaner` (as `_cleaner`) and `started` immediately. Otherwise, `_cleaner` is empty.

Note	<code>spark.cleaner.referenceTracking</code> Spark property is enabled by default.
------	--

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

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` is notified that `SparkContext` is almost fully initialized.

Note	<code>TaskScheduler.postStartHook</code> does nothing by default, but custom implementations offer more advanced features, i.e. <code>TaskSchedulerImpl</code> blocks the current thread until <code>SchedulerBackend</code> is ready. There is also <code>YarnClusterScheduler</code> for Spark on YARN in <code>cluster</code> deploy mode.
------	---

Registering Metrics Sources

`SparkContext` requests `MetricsSystem` to register metrics sources for the following services:

1. `DAGScheduler`
2. `BlockManager`
3. `ExecutorAllocationManager` (if dynamic allocation is enabled)

Adding Shutdown Hook

`SparkContext` adds a shutdown hook (using `shutdownHookManager.addShutdownHook()`).

You should see the following DEBUG message in the logs:

```
DEBUG Adding shutdown hook
```

Caution	FIXME ShutdownHookManager.addShutdownHook()
---------	---

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

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

Caution	FIXME Finish me
---------	---------------------------------

Initializing nextShuffleId and nextRddId Internal Counters

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

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

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

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.

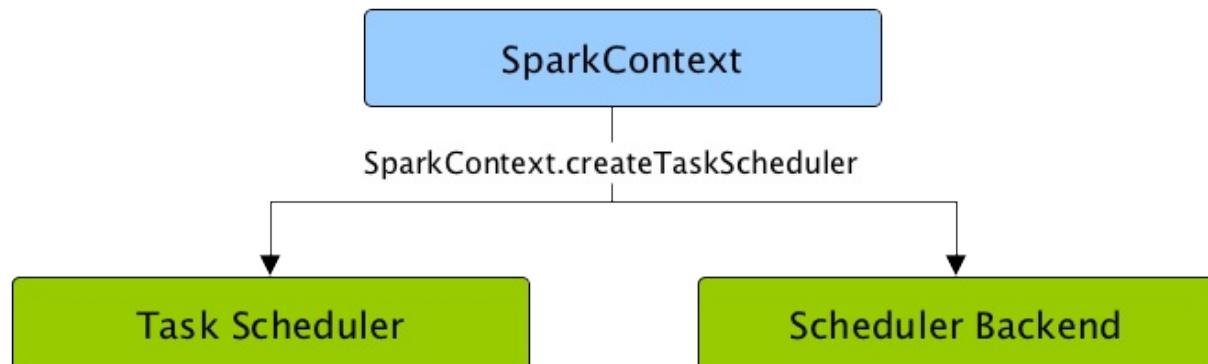


Figure 1. SparkContext creates Task Scheduler and Scheduler Backend
`createTaskScheduler` understands the following master URLs:

- `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	<code>getClusterManager</code> uses Java's <code>ServiceLoader.load</code> method.
------	--

Note	<code>getClusterManager</code> is used to find a cluster manager for a master URL when creating a <code>SchedulerBackend</code> and a <code>TaskScheduler</code> 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` setting, 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 <code>INFO</code> on <code>org.apache.spark.SparkContext</code> logger to see the extra listeners being registered.
	<pre>INFO SparkContext: Registered listener pl.japila.spark.CustomSparkListener</pre>

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.getCurrentUserUserName

```
getCurrentUserName(): String
```

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

Note	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 <code>SparkContext</code> is created and also to compute the default value of <code>spark.driver.host</code> Spark property.
------	---

Caution	FIXME Review the rest.
---------	-------------------------------

stopped flag

Caution	FIXME Where is this used?
---------	----------------------------------

ConsoleProgressBar

`ConsoleProgressBar` shows the progress of active stages to standard error, i.e. `stderr`. It uses [SparkStatusTracker](#) to poll the status of stages periodically and print 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 id, the number of completed, active, and total tasks.

Tip

`ConsoleProgressBar` may be useful when you `ssh` to workers and want to see the progress of active stages.

`ConsoleProgressBar` is created when `SparkContext` starts with `spark.ui.showConsoleProgress` enabled and the logging level of `org.apache.spark.SparkContext` logger as `WARN` or higher (i.e. less messages are printed out and so there is a "space" for `ConsoleProgressBar`).

```
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 `500` milliseconds every `spark.ui.consoleProgress.update.interval` milliseconds.

Note

The initial delay of `500` milliseconds before `ConsoleProgressBar` show the progress is 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)

import org.apache.log4j._
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.

Tip

[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()
```

Creating ConsoleProgressBar Instance

`ConsoleProgressBar` requires a [SparkContext](#).

When being created, `ConsoleProgressBar` reads `spark.ui.consoleProgress.update.interval` Spark property to set up the update interval and `COLUMNS` environment variable for the terminal width (or assumes `80` columns).

`ConsoleProgressBar` starts the internal timer `refresh` that does `refresh` and shows progress.

Note

`ConsoleProgressBar` is created when `SparkContext` starts, `spark.ui.showConsoleProgress` is enabled, and the logging level of `org.apache.spark.SparkContext` logger is `WARN` or higher (i.e. less messages are printed out and so there is a "space" for `ConsoleProgressBar`).

Note

Once created, `ConsoleProgressBar` is available internally as `_progressBar` .

refresh Method**Caution**

[FIXME](#)

finishAll Method**Caution**

[FIXME](#)

stop Method

`stop(): Unit`

`stop` cancels (stops) the internal timer.

Note

`stop` is executed when `SparkContext` stops.

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.ui.showConsoleProgress</code>	<code>true</code>	Controls whether to create <code>ConsoleProgressBar</code> (<code>true</code>) or not (<code>false</code>).
<code>spark.ui.consoleProgress.update.interval</code>	<code>200 (ms)</code>	Update interval, i.e. how often to show the progress.

SparkStatusTracker

SparkStatusTracker is created when [SparkContext is created](#).

Creating SparkStatusTracker Instance

SparkStatusTracker takes the following when created:

- [SparkContext](#)

Local Properties — Creating Logical Job Groups

The purpose of **local properties** concept is to create logical groups of jobs by means of properties that (regardless of the threads used to submit the jobs) makes the separate jobs launched from different threads belong to a single logical group.

You can [set a local property](#) that will affect Spark jobs submitted from a thread, such as the Spark fair scheduler pool. You can use your own custom properties. The properties are propagated through to worker tasks and can be accessed there via [TaskContext.getLocalProperty](#).

Note	Propagating local properties to workers starts when <code>sparkContext</code> is requested to run or submit a Spark job that in turn passes them along to DAGScheduler .
------	--

Note	Local properties is used to group jobs into pools in FAIR job scheduler by spark.scheduler.pool per-thread property and in SQLExecution.withNewExecutionId Helper Methods
------	---

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

```

Local Properties — `localProperties` Property

```
localProperties: InheritableThreadLocal[Properties]
```

`localProperties` is a `protected[spark]` property of a [SparkContext](#) that are the properties through which you can create logical job groups.

TipRead up on Java's [java.lang.InheritableThreadLocal](#).

Setting Local Property — `setLocalProperty` Method

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

`setLocalProperty` sets `key` local property to `value`.

TipWhen `value` is `null` the `key` property is removed from [localProperties](#).

Getting Local Property — `getLocalProperty` Method

```
getLocalProperty(key: String): String
```

`getLocalProperty` gets a local property by `key` in this thread. It returns `null` if `key` is missing.

Getting Local Properties — `getLocalProperties` Method

```
getLocalProperties: Properties
```

`getLocalProperties` is a `private[spark]` method that gives access to [localProperties](#).

`setLocalProperties` Method

```
setLocalProperties(props: Properties): Unit
```

`setLocalProperties` is a `private[spark]` method that sets `props` as [localProperties](#).

RDD — Resilient Distributed Dataset

Resilient Distributed Dataset (aka **RDD**) is the primary data abstraction in Apache Spark and the core of Spark (that I often refer to as "Spark Core").

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.

A RDD is a resilient and distributed collection of records spread over [one or many partitions](#).

Note	One could compare RDDs to collections in Scala, i.e. a RDD is computed on many JVMs while a Scala collection lives on a single JVM.
------	---

Using RDD Spark hides 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.

The features of RDDs (decomposing the 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).

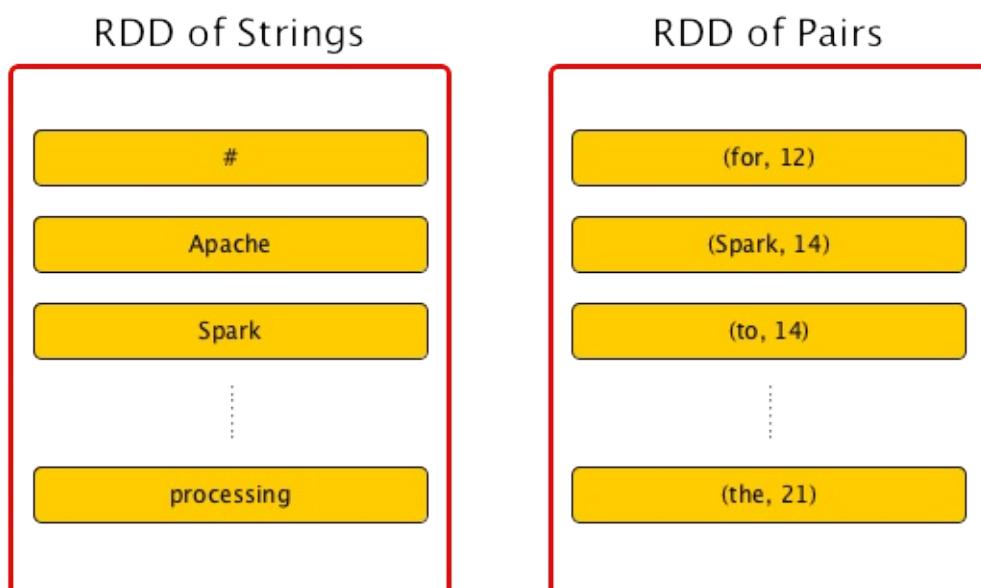


Figure 1. RDDs

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** — RDD records have types, e.g. `Long` in `RDD[Long]` or `(Int, String)` in `RDD[(Int, String)]`.
- **Partitioned** — records are partitioned (split into logical partitions) and distributed across nodes in a cluster.
- **Location-Stickiness** — `RDD` can define [placement preferences](#) to compute partitions (as close to the records as possible).

Note

Preferred location (aka *locality preferences* or *placement preferences* or *locality info*) is information about the locations of RDD records (that Spark's [DAGScheduler](#) uses to place computing partitions on to have the tasks as close to the data as possible).

Computing partitions in a RDD is a distributed process 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](#).

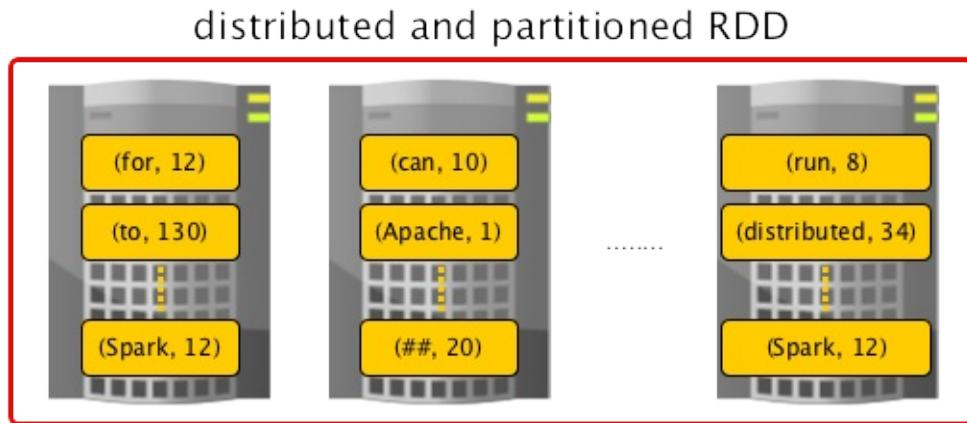


Figure 2. RDDs

Partitions are the units of parallelism. You can control the number of partitions of a RDD using [repartition](#) or [coalesce](#) transformations. 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.

Technically, RDDs follow the [contract](#) defined by the five main intrinsic properties:

- List of [parent RDDs](#) that are the dependencies of the RDD.
- 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 records live or are the closest to read from.

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 in a [SparkContext](#) (available as `context` property).

RDDs live in one and only one [SparkContext](#) that creates a logical boundary.

Note

RDDs cannot 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).

checkpointRDD Internal Method

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

isCheckpointedAndMaterialized Method

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

getNarrowAncestors Method

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

toLocalIterator Method

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

cache Method

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

persist Methods

```
persist(): this.type
persist(newLevel: StorageLevel): this.type
```

Refer to [Persisting RDD — persist Methods](#).

persist Internal Method

```
persist(newLevel: StorageLevel, allowOverride: Boolean): this.type
```

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

Note	persist is used when RDD is requested to persist itself and marks itself for local checkpointing.
------	---

unpersist Method

Caution

FIXME

localCheckpoint Method

```
localCheckpoint(): this.type
```

Refer to [Marking RDD for Local Checkpointing](#) — `localCheckpoint` Method.

RDD Contract

```
abstract class RDD[T] {
    def compute(split: Partition, context: TaskContext): Iterator[T]
    def getPartitions: Array[Partition]
    def getDependencies: Seq[Dependency[_]]
    def getPreferredLocations(split: Partition): Seq[String] = Nil
    val partitioner: Option[Partitioner] = None
}
```

Note

`RDD` is an abstract class in Scala.

Table 1. RDD Contract

Method	Description
<code>compute</code>	Used exclusively when <code>RDD</code> computes a partition (possibly by reading from a checkpoint).
<code>getPartitions</code>	Used exclusively when <code>RDD</code> is requested for its partitions (called only once as the value is cached).
<code>getDependencies</code>	Used when <code>RDD</code> is requested for its dependencies (called only once as the value is cached).
<code>getPreferredLocations</code>	Defines placement preferences of a partition. Used exclusively when <code>RDD</code> is requested for the preferred locations of a partition.
<code>partitioner</code>	Defines the Partitioner of a <code>RDD</code> .

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 [repartition](#) or [coalesce](#) transformations.
- [ShuffledRDD](#) - a result of shuffling, e.g. after [repartition](#) or [coalesce](#) transformations.
- **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 [Transformations](#).

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 [Actions](#).

Creating RDDs

SparkContext.parallelize

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

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

FIXME What's the use case for `makeRDD` ?

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

SparkContext.textFile

One of the easiest ways to create an RDD is to use `sparkContext.textFile` to read files.

You can use the local `README.md` file (and then `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 hundred 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:

The screenshot shows the Spark shell application UI with the 'Storage' tab selected. A table displays the details of an RDD named 'Hundreds ints'. The table has columns for RDD Name, Storage Level, Cached Partitions, Fraction Cached, Size in Memory, Size in ExternalBlockStore, and Size on Disk. The data shows 8 cached partitions at 100% fraction, with memory usage of 2.1 KB and disk usage of 0.0 B.

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

Storage

RDDs

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

Figure 3. RDD with custom name

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

The screenshot shows the Spark shell application UI with the 'RDD Storage Info' page for the 'Hundreds ints' RDD. It displays storage-level details: Memory Deserialized 1x Replicated, 8 cached partitions, 2.1 KB memory size, and 0.0 B disk size. Below this, it shows data distribution on 1 executor, with all data in memory (530.0 MB Remaining) and 0.0 B on disk.

RDD Storage Info for Hundreds 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 interface with the title "Spark shell - Stages for All Jobs". The "Stages" tab is selected. Below it, a table lists completed stages:

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

Figure 5. Number of tasks after `repartition`

Accessing RDD Partitions — `partitions` Final Method

```
partitions: Array[Partition]
```

`partitions` returns the [Partitions](#) of a [RDD](#).

`partitions` [requests](#) [CheckpointRDD](#) for [partitions](#) (if the [RDD](#) is [checkpointed](#)) or [finds them itself](#) and [cache](#) (in [partitions_](#) internal registry that is used next time).

Note

Partitions have the property that their internal index should be equal to their position in the owning [RDD](#).

Computing Partition (in TaskContext) — `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`).

`compute` 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](#).

Note

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

Defining Placement Preferences of RDD Partition — `preferredLocations` Final Method

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

`preferredLocations` requests `CheckpointRDD` for placement preferences (if the RDD is checkpointed) or calculates them itself.

Note

`preferredLocations` is a template method that uses `getPreferredLocations` that custom RDDs can override to specify placement preferences for a partition. `getPreferredLocations` defines no placement preferences by default.

Note

`preferredLocations` is mainly used when `DAGScheduler` computes preferred locations for missing partitions.

The other usages are to define the locations by custom RDDs, e.g.

- (Spark Core) `BlockRDD`, `CoalescedRDD`, `HadoopRDD`, `NewHadoopRDD`, `ParallelCollectionRDD`, `ReliableCheckpointRDD`, `ShuffledRDD`
- (Spark SQL) `KafkaSourceRDD`, `ShuffledRowRDD`, `FileScanRDD`, `StateStoreRDD`
- (Spark Streaming) `KafkaRDD`, `WriteAheadLogBackedBlockRDD`

Getting Number of Partitions — `getNumPartitions` Method

```
getNumPartitions: Int
```

`getNumPartitions` gives the number of partitions of a RDD.

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

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

Computing Partition (Possibly by Reading From Checkpoint) — `computeOrReadCheckpoint` Method

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

`computeOrReadCheckpoint` reads `split` partition from a checkpoint (if available already) or computes it yourself.

Note`computeOrReadCheckpoint` is a `private[spark]` method.**Note**`computeOrReadCheckpoint` is used when `RDD` computes records for a partition or `getOrCompute`.

Accessing Records For Partition Lazily — `iterator` Final Method

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

`iterator` gets (or computes) `split` partition when `cached` or computes it (possibly by reading from checkpoint).

Note`iterator` is a `final` method that, despite being public, considered private and only available for implementing custom RDDs.

Computing RDD Partition — `getOrCompute` Method

`getOrCompute(partition: Partition, context: TaskContext): Iterator[T]`**Caution****FIXME**

`getOrCompute` requests `BlockManager` for a block and returns a `InterruptibleIterator`.

Note`InterruptibleIterator` delegates to a wrapped `Iterator` and allows for task killing functionality.**Note**`getOrCompute` is called on Spark executors.

Internally, `getOrCompute` creates a `RDDBlockId` (for the partition in the RDD) that is then used to retrieve it from `BlockManager` or compute, persist and return its values.

Note`getOrCompute` is a `private[spark]` method that is exclusively used when iterating over partition when a RDD is cached.

RDD Dependencies — `dependencies` Final Template Method

`dependencies: Seq[Dependency[_]]`

`dependencies` returns the [dependencies of a RDD](#).

Note	<code>dependencies</code> is a final method that no class in Spark can ever override.
------	---

Internally, `dependencies` checks out whether the RDD is [checkpointed](#) and acts accordingly.

For a RDD being checkpointed, `dependencies` returns a single-element collection with a [OneToOneDependency](#).

For a non-checkpointed RDD, `dependencies` collection is computed using [getDependencies method](#).

Note	<code>getDependencies</code> method is an abstract method that custom RDDs are required to provide.
------	---

RDD Lineage — Logical Execution Plan

RDD Lineage (aka *RDD operator graph* or *RDD dependency 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 **execution DAG** or **physical execution plan** is the [DAG of stages](#).

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

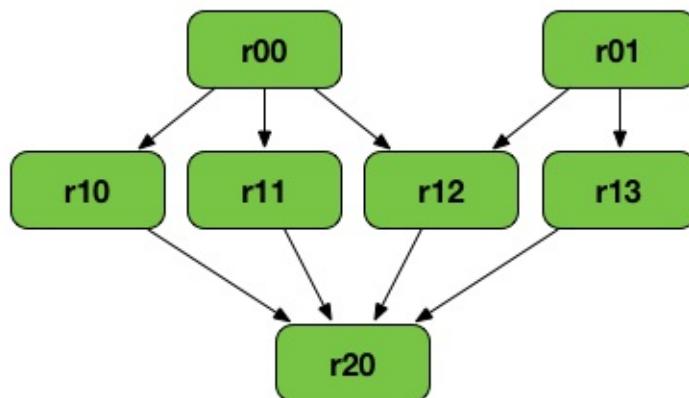


Figure 1. RDD lineage

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.

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, i.e. a DAG, is materialized and executed when `SparkContext` is requested to run a Spark job.

Getting RDD Lineage Graph — `toDebugString` Method

```
toDebugString: String
```

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

```
scala> val wordCount = sc.textFile("README.md").flatMap(_.split("\\s+")).map((_, 1)).reduceByKey(_ + _)
wordCount: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[21] at reduceByKey at <console>:24

scala> wordCount.toDebugString
res13: String =
(2) ShuffledRDD[21] at reduceByKey at <console>:24 []
+- (2) MapPartitionsRDD[20] at map at <console>:24 []
  |  MapPartitionsRDD[19] at flatMap at <console>:24 []
  |  README.md MapPartitionsRDD[18] at textFile at <console>:24 []
  |  README.md HadoopRDD[17] 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, e.g. `(2)` in the above output.

```
scala> wordCount.getNumPartitions
res14: Int = 2
```

With `spark.logLineage` property enabled, `toDebugString` is included when executing an action.

```
$ ./bin/spark-shell --conf 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 []
```

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.logLineage</code>	<code>false</code>	When enabled (i.e. <code>true</code>), executing an action (and hence running a job) will also print out the RDD lineage graph using RDD.toDebugString .

TaskLocation

`TaskLocation` is a location where a [task](#) should run.

`TaskLocation` can either be a host alone or a (host, executorID) pair (as [ExecutorCacheTaskLocation](#)).

With [ExecutorCacheTaskLocation](#) the Spark scheduler prefers to launch the task on the given executor, but the next level of preference is any executor on the same host if this is not possible.

Note

`TaskLocation` is a Scala `private[spark] sealed trait` (i.e. all the available implementations of `TaskLocation` trait are in a single Scala file).

Table 1. Available TaskLocations

Name	Description
<code>HostTaskLocation</code>	A location on a host.
<code>ExecutorCacheTaskLocation</code>	A location that includes both a host and an executor id on that host.
<code>HDFSCacheTaskLocation</code>	A location on a host that is cached by Hadoop HDFS. Used exclusively when HadoopRDD and NewHadoopRDD are requested for their placement preferences (aka <i>preferred locations</i>).

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`

OrderedRDDFunctions

repartitionAndSortWithinPartitions Operator

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

sortByKey Operator

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

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.

Computing Partition (in TaskContext) — `compute` Method

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

`getDependencies` Method

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

SubtractedRDD

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

Computing Partition (in `TaskContext`)— `compute` Method

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

`getDependencies` Method

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

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.

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

getPreferredLocations Method

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

getPartitions Method

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

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

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

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

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

NewHadoopRDD

NewHadoopRDD is an RDD of κ keys and v values.

NewHadoopRDD is created when:

- `SparkContext.newAPIHadoopFile`
- `SparkContext.newAPIHadoopRDD`
- (indirectly) `SparkContext.binaryFiles`
- (indirectly) `SparkContext.wholeTextFiles`

Note NewHadoopRDD is the base RDD of `BinaryFileRDD` and `WholeTextFileRDD`.

getPreferredLocations Method

Caution

FIXME

Creating NewHadoopRDD Instance

NewHadoopRDD takes the following when created:

- `SparkContext`
- HDFS' `InputFormat[K, V]`
- κ class name
- v class name
- transient HDFS' `Configuration`

NewHadoopRDD initializes the internal registries and counters.

ShuffledRDD

`ShuffledRDD` is an [RDD](#) of key-value pairs that represents the **shuffle step** in a [RDD lineage](#). It uses custom [ShuffledRDDPartition](#) partitions.

A `shuffledRDD` is created for RDD transformations that trigger a [data shuffling](#):

1. `coalesce` transformation (with `shuffle` flag enabled).
2. `PairRDDFunctions`'s `combineByKeyWithClassTag` and `partitionBy` (when the parent RDD's and specified `Partitioners` are different).
3. `OrderedRDDFunctions`'s `sortByKey` and `repartitionAndSortWithinPartitions` ordered operators.

```
scala> val rdd = sc.parallelize(0 to 9)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> rdd.getNumPartitions
res0: Int = 8

// ShuffledRDD and coalesce Example

scala> rdd.coalesce(numPartitions = 4, shuffle = true).toDebugString
res1: String =
(4) MapPartitionsRDD[4] at coalesce at <console>:27 []
| CoalescedRDD[3] at coalesce at <console>:27 []
| ShuffledRDD[2] at coalesce at <console>:27 []
+- (8) MapPartitionsRDD[1] at coalesce at <console>:27 []
   | ParallelCollectionRDD[0] at parallelize at <console>:24 []

// ShuffledRDD and sortByKey Example

scala> val grouped = rdd.groupBy(_ % 2)
grouped: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[6] at groupBy at <console>:26

scala> grouped.sortByKey(numPartitions = 2).toDebugString
res2: String =
(2) ShuffledRDD[9] at sortByKey at <console>:29 []
+- (8) ShuffledRDD[6] at groupBy at <console>:26 []
   +- (8) MapPartitionsRDD[5] at groupBy at <console>:26 []
      | ParallelCollectionRDD[0] at parallelize at <console>:24 []
```

`ShuffledRDD` takes a parent RDD and a [Partitioner](#) when created.

`getDependencies` returns a single-element collection of [RDD dependencies](#) with a [ShuffleDependency](#) (with the `Serializer` according to [map-side combine internal flag](#)).

Map-Side Combine `mapSideCombine` Internal Flag

`mapSideCombine: Boolean`

`mapSideCombine` internal flag is used to select the `Serializer` (for shuffling) when [ShuffleDependency](#) is created (which is the [one and only Dependency](#) of a [ShuffledRDD](#)).

Note `mapSideCombine` is only used when `userSpecifiedSerializer` optional `Serializer` is not specified explicitly (which is the default).

Note `mapSideCombine` uses [SparkEnv](#) to access the current `SerializerManager`.

If enabled (i.e. `true`), `mapSideCombine` directs to [find the Serializer](#) for the types `k` and `c`. Otherwise, `getDependencies` finds the `Serializer` for the types `k` and `v`.

Note The types `k`, `c` and `v` are specified when [ShuffledRDD](#) is created.

Note `mapSideCombine` is disabled (i.e. `false`) when [ShuffledRDD](#) is created and can be set using `setMapSideCombine` method.
`setMapSideCombine` method is only used in the experimental [PairRDDFunctions.combineByKeyWithClassTag](#) transformations.

Computing Partition (in `TaskContext`)— `compute` Method

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

Note `compute` is a part of [RDD contract](#) to compute a given partition in a [TaskContext](#).

Internally, `compute` makes sure that the input `split` is a [ShuffleDependency](#). It then requests `ShuffleManager` for a `ShuffleReader` to read key-value pairs (as `Iterator[(K, C)]`) for the `split`.

Note `compute` uses [SparkEnv](#) to access the current `shuffleManager`.

Note A Partition has the `index` property to specify `startPartition` and `endPartition` partition offsets.

Getting Placement Preferences of Partition — `getPreferredLocations` Method

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

Note	<code>getPreferredLocations</code> is a part of RDD contract to specify placement preferences (aka <i>preferred task locations</i>), i.e. where tasks should be executed to be as close to the data as possible.
------	---

Internally, `getPreferredLocations` requests [MapOutputTrackerMaster](#) for the preferred locations, i.e. [BlockManagers](#) with the most map outputs, for the input `partition` (of the one and only [ShuffleDependency](#)).

Note	<code>getPreferredLocations</code> uses SparkEnv to access the current MapOutputTrackerMaster (which runs on the driver).
------	---

ShuffledRDDPartition

`ShuffledRDDPartition` gets an `index` when it is created (that in turn is the index of partitions as calculated by the [Partitioner](#) of a [ShuffledRDD](#)).

BlockRDD

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

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

Note	It appears that <code>BlockRDD</code> is used in Spark Streaming exclusively.
------	---

Computing Partition (in `TaskContext`)— `compute` Method

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

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. `sortBy` , [zipWithIndex](#), etc.

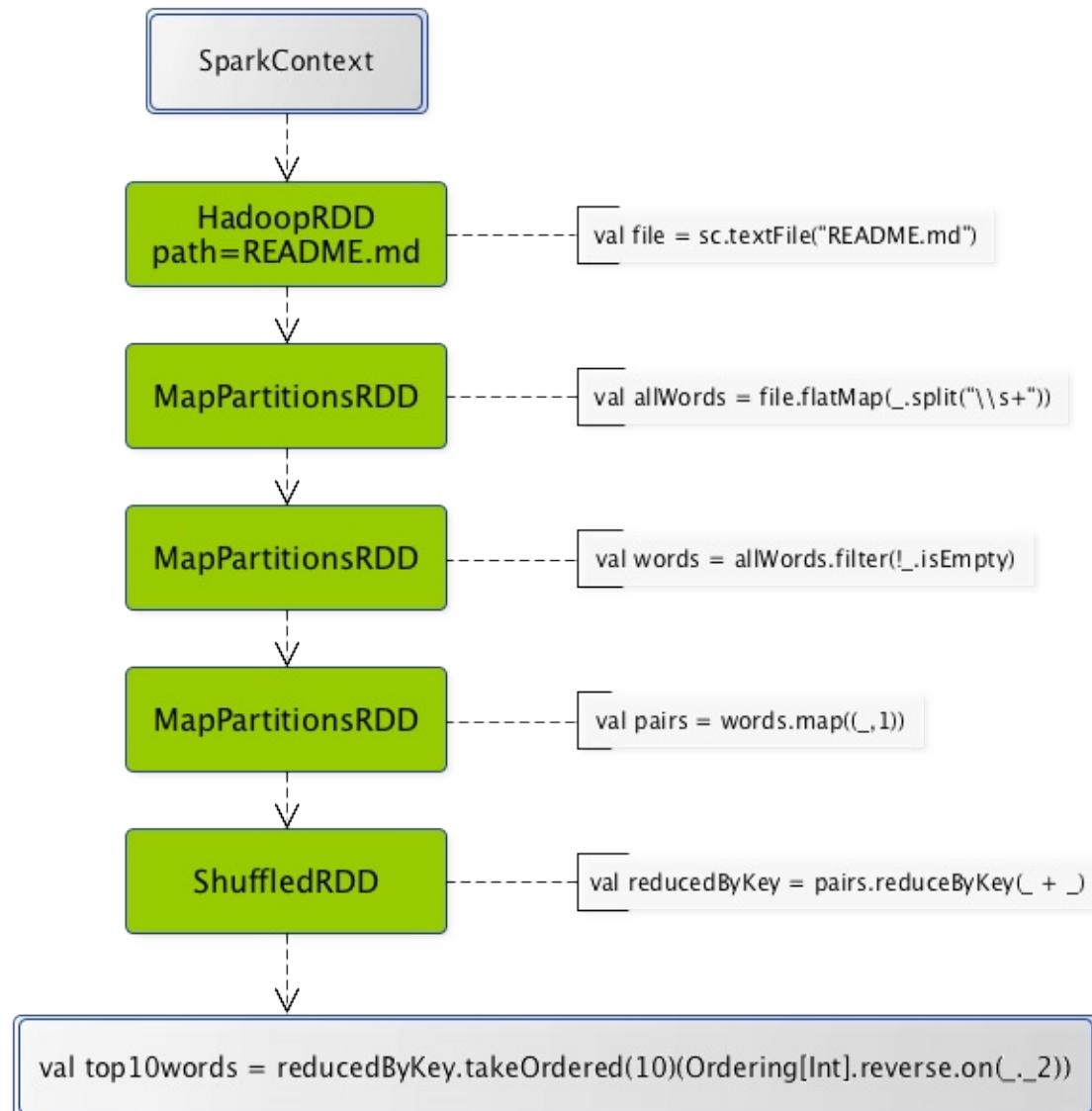


Figure 1. From `SparkContext` by transformations to the result

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.

map

Caution[FIXME](#)

flatMap

Caution[FIXME](#)

filter

Caution[FIXME](#)

randomSplit

Caution[FIXME](#)

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.

Caution

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

```
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
```



Jobs Stages Storage Environment Executors [Spark shell application UI](#)

Spark Jobs (?)

User: jacek
 Total Uptime: 23 s
 Scheduling Mode: FIFO
Completed Jobs: 1

[Event Timeline](#)

Completed Jobs (1)

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

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 <code>PairRDDFunctions</code> . Read up about it in Partitions and Partitioning .
-----	---

countApproxDistinctByKey Transformation

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

foldByKey Transformation

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

aggregateByKey Transformation

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

combineByKey Transformation

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

partitionBy Operator

```
partitionBy(partitioner: Partitioner): RDD[(K, V)]
```

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

groupByKey and reduceByKey Transformations

`reduceByKey` is sort of a particular case of [aggregateByKey](#).

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 Transformations

```
combineByKeyWithClassTag[C](
  createCombiner: V => C,
  mergeValue: (C, V) => C,
  mergeCombiners: (C, C) => C)(implicit ct: ClassTag[C]): RDD[(K, C)] (1)
combineByKeyWithClassTag[C](
  createCombiner: V => C,
  mergeValue: (C, V) => C,
  mergeCombiners: (C, C) => C,
  numPartitions: Int)(implicit ct: ClassTag[C]): RDD[(K, C)] (2)
combineByKeyWithClassTag[C](
  createCombiner: V => C,
  mergeValue: (C, V) => C,
  mergeCombiners: (C, C) => C,
  partitioner: Partitioner,
  mapSideCombine: Boolean = true,
  serializer: Serializer = null)(implicit ct: ClassTag[C]): RDD[(K, C)]
```

1. [FIXME](#)
2. [FIXME](#) too

`combineByKeyWithClassTag` transformations use `mapSideCombine` enabled (i.e. `true`) by default. They create a [ShuffledRDD](#) with the value of `mapSideCombine` when the input partitioner is different from the current one in an RDD.

Note

`combineByKeyWithClassTag` is a base transformation for [combineByKey](#)-based transformations, [aggregateByKey](#), [foldByKey](#), [reduceByKey](#), [countApproxDistinctByKey](#), and [groupByKey](#).

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	FIXME
---------	-----------------------

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 `MEMORY_ONLY` storage level.

Persisting RDD — `persist` Methods

```
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.

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 `NONE` storage level as the current storage level.

StorageLevel

`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 for RDDs)
- `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)
```


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 a logical chunk of a large distributed data set.

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

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

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

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

Features:

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

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

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

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

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

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

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

Start `spark-shell` and see it yourself!

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

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

The screenshot shows the 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
```

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 RDD — `repartition` Transformation

```
repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]
```

`repartition` is [coalesce](#) with `numPartitions` and `shuffle` enabled.

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 `shuffle` flag (disabled by default, i.e. `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).

Tip

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 purposes. Note the number of partitions that remains the same as the number of partitions in the source RDD `rdd`.

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.default.parallelism</code>	(varies per deployment environment)	<p>Sets up the number of partitions to use for HashPartitioner. It corresponds to default parallelism of a scheduler backend.</p> <p>More specifically, <code>spark.default.parallelism</code> corresponds to:</p> <ul style="list-style-type: none"> • The number of threads for LocalSchedulerBackend. • the number of CPU cores in Spark on Mesos and defaults to <code>8</code>. • Maximum of <code>totalCoreCount</code> and <code>2</code> in CoarseGrainedSchedulerBackend.

Partition

Caution	FIXME
Note	A partition is missing when it has not be computed yet.

Partitioner

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

`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.

The contract of partitioner ensures that records for a given key have to reside on a single partition.

numPartitions Method

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

getPartition Method

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

HashPartitioner

`HashPartitioner` is a `Partitioner` that uses `partitions` configurable number of partitions to shuffle data around.

Table 1. `HashPartitioner` Attributes and Method

Property	Description
<code>numPartitions</code>	Exactly <code>partitions</code> number of partitions
<code>getPartition</code>	<code>0</code> for <code>null</code> keys and Java's <code>Object.hashCode</code> for non- <code>null</code> keys (modulo <code>partitions</code> number of partitions or <code>0</code> for negative hashes).
<code>equals</code>	<code>true</code> for <code>HashPartitioner</code> s with <code>partitions</code> number of partitions. Otherwise, <code>false</code> .
<code>hashCode</code>	Exactly <code>partitions</code> number of partitions

Note

`HashPartitioner` is the default `Partitioner` for `coalesce` transformation with `shuffle` enabled, e.g. calling `repartition`.

It is possible to re-shuffle data despite all the records for the key `k` being already on a single Spark executor (i.e. `BlockManager` to be precise). When `HashPartitioner`'s result for `k1` is `3` the key `k1` will go to the third executor.

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.

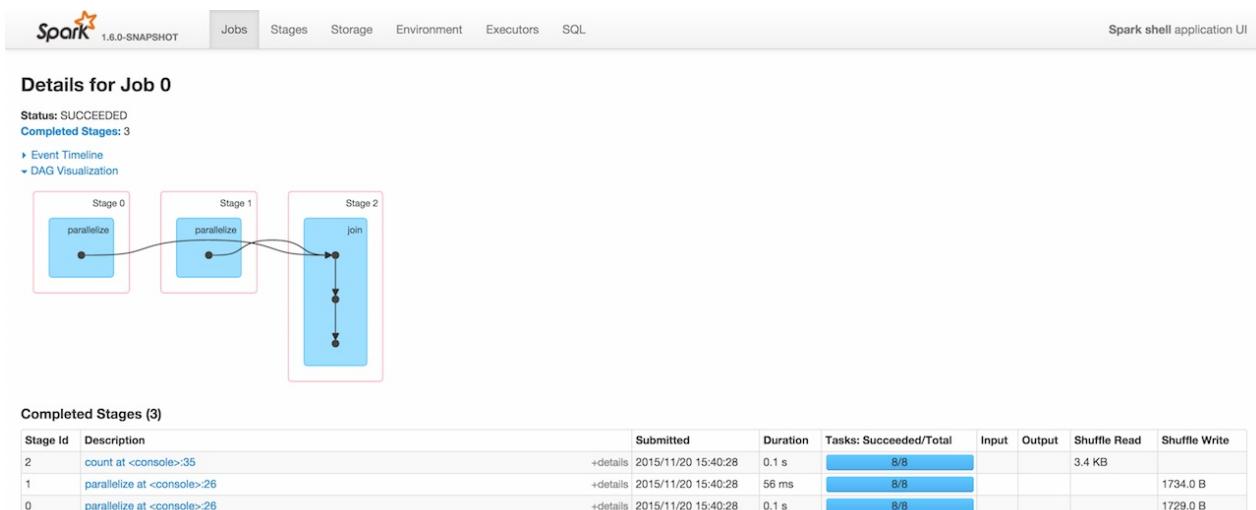


Figure 1. Executing `joined.count`

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` property 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

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()
```

Before checkpointing is used, a Spark developer has to set the checkpoint directory using

```
SparkContext.setCheckpointDir(directory: String)
```

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`.

ReliableCheckpointRDD

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

Marking RDD for Local Checkpointing — `localCheckpoint` Method

```
localCheckpoint(): this.type
```

`localCheckpoint` marks a RDD for **local checkpointing** using Spark's caching layer.

`localCheckpoint` 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.

Note

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

LocalRDDCheckpointData

[FIXME](#)

LocalCheckpointRDD

[FIXME](#)

doCheckpoint Method

Caution

[FIXME](#)

CheckpointRDD

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

RDD Dependencies

`Dependency` class is the base (abstract) class to model a dependency relationship between two or more RDDs.

`Dependency` has a single method `rdd` to access the RDD that is behind a dependency.

```
def rdd: RDD[T]
```

Whenever you apply a [transformation](#) (e.g. `map`, `flatMap`) to a RDD you build the so-called [RDD lineage graph](#). `Dependency`-ies represent the edges in a lineage graph.

Note	<code>NarrowDependency</code> and <code>ShuffleDependency</code> are the two top-level subclasses of <code>Dependency</code> abstract class.
------	--

Table 1. Kinds of Dependencies

Name	Description
<code>NarrowDependency</code>	
<code>ShuffleDependency</code>	
<code>OneToOneDependency</code>	
<code>PruneDependency</code>	
<code>RangeDependency</code>	

The dependencies of a RDD are available using [dependencies](#) method.

```
// A demo RDD  
scala> val myRdd = sc.parallelize(0 to 9).groupBy(_ % 2)  
myRdd: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[8] at group  
  
scala> myRdd.foreach(println)  
(0,CompactBuffer(0, 2, 4, 6, 8))  
(1,CompactBuffer(1, 3, 5, 7, 9))  
  
scala> myRdd.dependencies  
res5: Seq[org.apache.spark.Dependency[_]] = List(org.apache.spark.ShuffleDepend  
  
// Access all RDDs in the demo RDD lineage  
scala> myRdd.dependencies.map(_.rdd).foreach(println)  
MapPartitionsRDD[7] at groupBy at <console>:24
```

Note

You use [toDebugString](#) method to print out the RDD lineage in a user-friendly way.

```
scala> myRdd.toDebugString  
res6: String =  
(8) ShuffledRDD[8] at groupBy at <console>:24 []  
 +- (8) MapPartitionsRDD[7] at groupBy at <console>:24 []  
   | ParallelCollectionRDD[6] at parallelize at <console>:24 []
```

NarrowDependency — Narrow Dependencies

`NarrowDependency` is a base (abstract) [Dependency](#) with *narrow* (limited) number of [partitions](#) of the parent RDD that are required to compute a partition of the child RDD.

Note	Narrow dependencies allow for pipelined execution.
------	--

Table 1. Concrete `NarrowDependency`-ies

Name	Description
OneToOneDependency	
PruneDependency	
RangeDependency	

NarrowDependency Contract

`NarrowDependency` contract assumes that extensions implement `getParents` method.

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

`getParents` returns the partitions of the parent RDD that the input `partitionId` depends on.

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(_,_)
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 RDD.

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 []

```


ShuffleDependency — Shuffle Dependency

`ShuffleDependency` is a [RDD Dependency](#) on the output of a [ShuffleMapStage](#) for a [key-value pair RDD](#).

`ShuffleDependency` uses the [RDD](#) to know the number of (map-side/pre-shuffle) partitions and the [Partitioner](#) for the number of (reduce-size/post-shuffle) partitions.

`ShuffleDependency` is a dependency of [ShuffledRDD](#) as well as [CoGroupedRDD](#) and [SubtractedRDD](#) but only when [partitioners](#) (of the RDD's and after transformations) are different.

A `shuffleDependency` is [created](#) for a key-value pair RDD, i.e. `RDD[Product2[K, V]]` with `K` and `V` being the types of keys and values, respectively.

Tip	Use <code>dependencies</code> method on an RDD to know the dependencies.
-----	--

```
scala> val rdd = sc.parallelize(0 to 8).groupBy(_ % 3)
rdd: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[2] at groupBy at <console>:24

scala> rdd.dependencies
res0: Seq[org.apache.spark.Dependency[_]] = List(org.apache.spark.ShuffleDependency@45
4f6cc5)
```

Every `shuffleDependency` has a unique application-wide **shuffleId** number that is assigned when `ShuffleDependency` is [created](#) (and is used throughout Spark's code to reference a `ShuffleDependency`).

Note	Shuffle ids are tracked by <code>SparkContext</code> .
------	--

keyOrdering Property

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

serializer Property

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

Creating ShuffleDependency Instance

ShuffleDependency takes the following when created:

1. A single key-value pair RDD, i.e. `RDD[Product2[K, V]]`,
2. Partitioner (available as `partitioner` property),
3. Serializer,
4. Optional key ordering (of Scala's `scala.math.Ordering` type),
5. Optional Aggregator,
6. `mapSideCombine` flag which is disabled (i.e. `false`) by default.

Note	ShuffleDependency uses <code>SparkEnv</code> to access the current Serializer .
------	---

When created, ShuffleDependency gets shuffle id (as `shuffleId`).

Note	ShuffleDependency uses the input RDD to access <code>SparkContext</code> and so the <code>shuffleId</code> .
------	--

ShuffleDependency registers itself with `shuffleManager` and gets a `ShuffleHandle` (available as `shuffleHandle` property).

Note	ShuffleDependency accesses <code>shuffleManager</code> using <code>SparkEnv</code> .
------	--

In the end, ShuffleDependency registers itself for cleanup with `ContextCleaner`.

Note	ShuffleDependency accesses the optional <code>ContextCleaner</code> through <code>SparkContext</code> .
------	---

Note	ShuffleDependency is created when <code>ShuffledRDD</code> , <code>CoGroupedRDD</code> , and <code>SubtractedRDD</code> return their RDD dependencies.
------	--

rdd Property

rdd: <code>RDD[Product2[K, V]]</code>

rdd returns a key-value pair RDD this ShuffleDependency was created for.

Note

- `rdd` is used when:
1. `MapOutputTrackerMaster` finds preferred `BlockManagers` with most map outputs for a `ShuffleDependency`,
 2. `DAGScheduler` finds or creates new `ShuffleMapStage` stages for a `ShuffleDependency`,
 3. `DAGScheduler` creates a `ShuffleMapStage` for a `ShuffleDependency` and a `ActiveJob`,
 4. `DAGScheduler` finds missing `ShuffleDependencies` for a `RDD`,
 5. `DAGScheduler` submits a `ShuffleDependency` for execution.

partitioner Property

`partitioner` property is a [Partitioner](#) that is used to partition the shuffle output.

`partitioner` is specified when `ShuffleDependency` is created.

Note

- `partitioner` is used when:
1. `MapOutputTracker` computes the statistics for a `ShuffleDependency` (and is the size of the array with the total sizes of shuffle blocks),
 2. `MapOutputTrackerMaster` finds preferred `BlockManagers` with most map outputs for a `ShuffleDependency`,
 3. `ShuffledRowRDD.adoc#numPreShufflePartitions`,
 4. `SortShuffleManager` checks if `SerializedShuffleHandle` can be used (for a `ShuffleHandle`).
 5. [FIXME](#)

shuffleHandle Property

```
shuffleHandle: ShuffleHandle
```

`shuffleHandle` is the `ShuffleHandle` of a `ShuffleDependency` as assigned eagerly when `ShuffleDependency` was created.

Note

`shuffleHandle` is used to compute `CoGroupedRDDs`, `ShuffledRDD`, `SubtractedRDD`, and `ShuffledRowRDD` (to get a `ShuffleReader` for a `ShuffleDependency`) and when a `shuffleMapTask` runs (to get a `ShuffleWriter` for a `ShuffleDependency`).

Map-Size Combine Flag — `mapSideCombine` Attribute

`mapSideCombine` is a flag to control whether to use **partial aggregation** (aka **map-side combine**).

`mapSideCombine` is by default disabled (i.e. `false`) when creating a `shuffleDependency`.

When enabled, `SortShuffleWriter` and `BlockStoreShuffleReader` assume that an `Aggregator` is also defined.

Note	<code>mapSideCombine</code> is exclusively set (and hence can be enabled) when <code>ShuffledRDD</code> returns the dependencies (which is a single <code>shuffleDependency</code>).
------	---

aggregator Property

```
aggregator: Option[Aggregator[K, V, C]] = None
```

`aggregator` is a `map/reduce-side Aggregator` (for a RDD's shuffle).

`aggregator` is by default undefined (i.e. `None`) when `shuffleDependency` is created.

Note	<code>aggregator</code> is used when <code>SortShuffleWriter</code> writes records and <code>BlockStoreShuffleReader</code> reads combined key-values for a reduce task.
------	--

Usage

The places where `shuffleDependency` is used:

- `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.

Map/Reduce-side Aggregator

`Aggregator` is a set of functions used to aggregate distributed data sets:

```
createCombiner: V => C
mergeValue: (C, V) => C
mergeCombiners: (C, C) => C
```

Note

`Aggregator` is created in `combineByKeyWithClassTag` transformations to `createShuffledRDDs` and is eventually passed on to `ShuffleDependency`. It is also used in `ExternalSorter`.

updateMetrics Internal Method

Caution**FIXME**

combineValuesByKey Method

Caution**FIXME**

combineCombinersByKey Method

Caution**FIXME**

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.

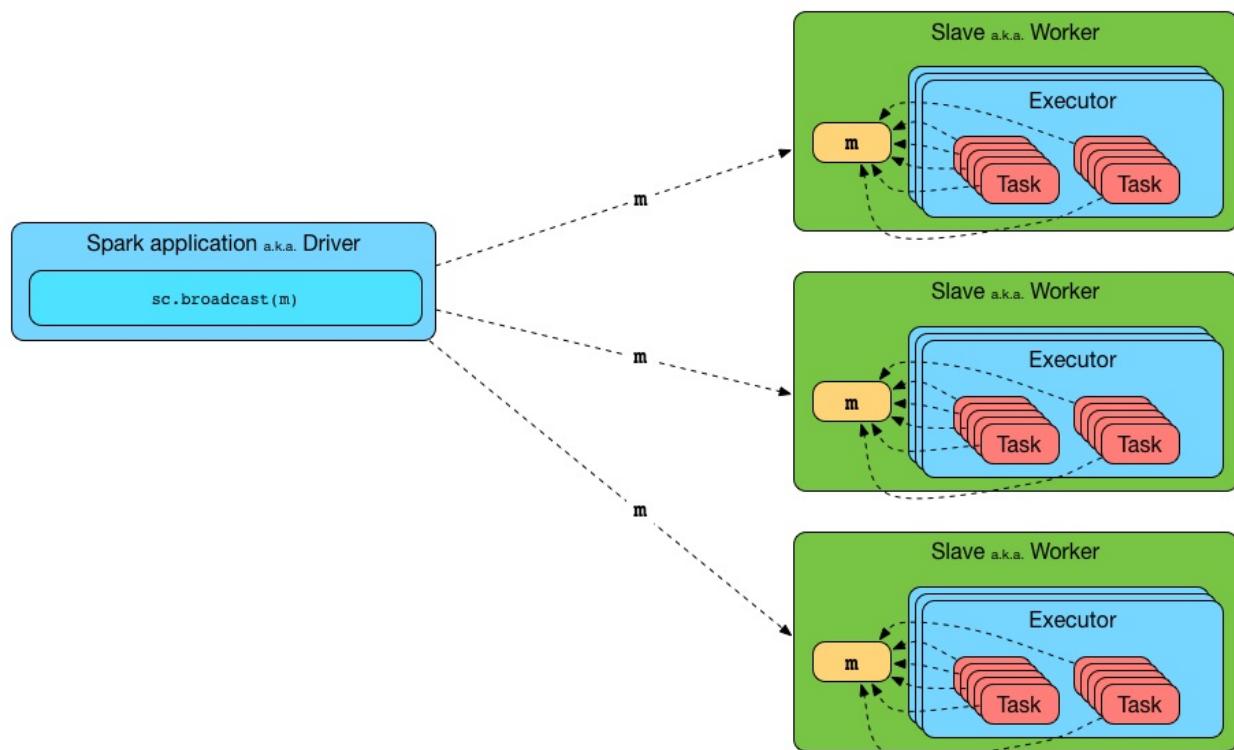


Figure 1. Broadcasting a value to executors

To use a broadcast value in a Spark 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.

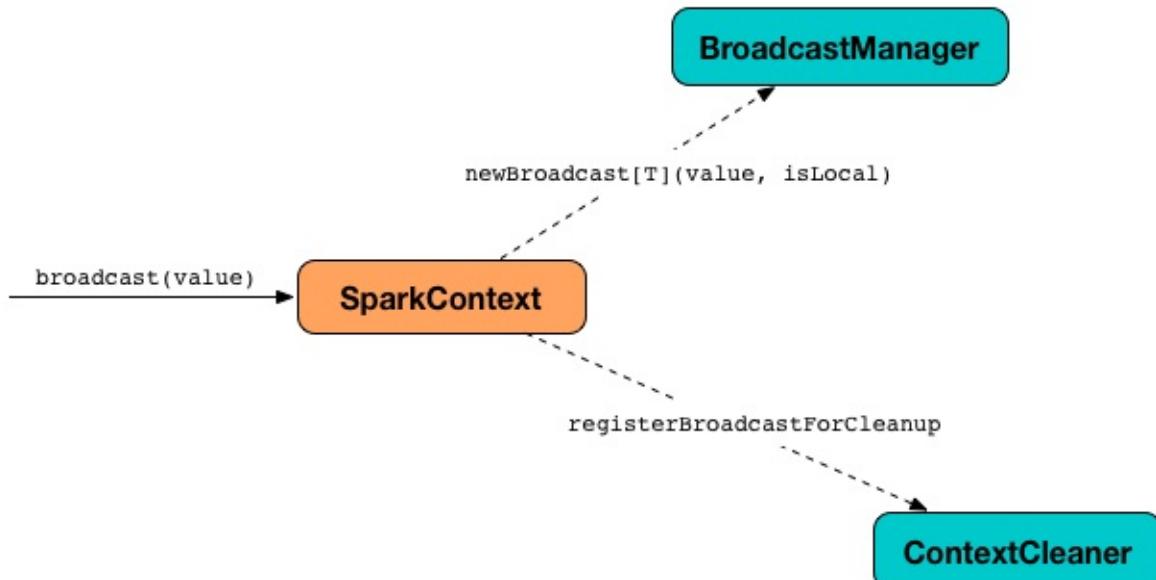


Figure 2. SparkContext to broadcast using BroadcastManager and ContextCleaner

Tip

Not only can Spark developers use broadcast variables for efficient data distribution, but Spark itself uses them quite often. A very notable use case is when [Spark distributes tasks to executors for their execution](#). That does change my perspective on the role of broadcast variables in Spark.

Broadcast Spark Developer-Facing Contract

The developer-facing `Broadcast` contract allows Spark developers to use it in their applications.

Table 1. Broadcast API

Method Name	Description
<code>id</code>	The unique identifier
<code>value</code>	The value
<code>unpersist</code>	Asynchronously deletes cached copies of this broadcast on the executors.
<code>destroy</code>	Destroys all data and metadata related to this broadcast variable.
<code>toString</code>	The string representation

Lifecycle of Broadcast Variable

You can create a broadcast variable of type `T` using `SparkContext.broadcast` method.

```
scala> val b = sc.broadcast(1)
b: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(0)
```

Tip

Enable `DEBUG` logging level for `org.apache.spark.storage.BlockManager` logger to debug `broadcast` method.

Read [BlockManager](#) to find out how to enable the logging level.

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

```
DEBUG BlockManager: Put block broadcast_0 locally took 430 ms
DEBUG BlockManager: Putting block broadcast_0 without replication took 431 ms
DEBUG BlockManager: Told master about block broadcast_0_piece0
DEBUG BlockManager: Put block broadcast_0_piece0 locally took 4 ms
DEBUG BlockManager: Putting block broadcast_0_piece0 without replication took 4 ms
```

After creating an instance of a broadcast variable, you can then reference the value using `value` method.

```
scala> b.value
res0: Int = 1
```

Note

`value` method is the only way to access the value of a broadcast variable.

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

```
DEBUG BlockManager: Getting local block broadcast_0
DEBUG BlockManager: Level for block broadcast_0 is StorageLevel(disk, memory, deserialized, 1 replicas)
```

When you are done with a broadcast variable, you should [destroy](#) it to release memory.

```
scala> b.destroy
```

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

```
DEBUG BlockManager: Removing broadcast 0
DEBUG BlockManager: Removing block broadcast_0_piece0
DEBUG BlockManager: Told master about block broadcast_0_piece0
DEBUG BlockManager: Removing block broadcast_0
```

Before [destroying](#) a broadcast variable, you may want to [unpersist](#) it.

```
scala> b.unpersist
```

Getting the Value of Broadcast Variable — `value` Method

```
value: T
```

`value` returns the value of a broadcast variable. You can only access the value until it is [destroyed](#) after which you will see the following `SparkException` exception in the logs:

```
org.apache.spark.SparkException: Attempted to use Broadcast(0) after it was destroyed
(destroy at <console>:27)
  at org.apache.spark.broadcast.Broadcast.assertValid(Broadcast.scala:144)
  at org.apache.spark.broadcast.Broadcast.value(Broadcast.scala:69)
  ... 48 elided
```

Internally, `value` makes sure that the broadcast variable is **valid**, i.e. [destroy](#) was not called, and, if so, calls the abstract `getValue` method.

Note

`getValue` is abstracted and broadcast variable implementations are supposed to provide a concrete behaviour.
Refer to [TorrentBroadcast](#).

Unpersisting Broadcast Variable — `unpersist` Methods

```
unpersist(): Unit
unpersist(blocking: Boolean): Unit
```

Destroying Broadcast Variable — `destroy` Method

```
destroy(): Unit
```

`destroy` removes a broadcast variable.

Note

Once a broadcast variable has been destroyed, it cannot be used again.

If you try to destroy a broadcast variable more than once, you will see the following `SparkException` exception in the logs:

```
scala> b.destroy
org.apache.spark.SparkException: Attempted to use Broadcast(0) after it was destroyed
(destroy at <console>:27)
  at org.apache.spark.broadcast.Broadcast.assertValid(Broadcast.scala:144)
  at org.apache.spark.broadcast.Broadcast.destroy(Broadcast.scala:107)
  at org.apache.spark.broadcast.Broadcast.destroy(Broadcast.scala:98)
... 48 elided
```

Internally, `destroy` executes the internal `destroy` (with `blocking` enabled).

Removing Persisted Data of Broadcast Variable — `destroy` Internal Method

```
destroy(blocking: Boolean): Unit
```

`destroy` destroys all data and metadata of a broadcast variable.

Note	<code>destroy</code> is a <code>private[spark]</code> method.
------	---

Internally, `destroy` marks a broadcast variable destroyed, i.e. the internal `_isValid` flag is disabled.

You should see the following INFO message in the logs:

```
INFO TorrentBroadcast: Destroying Broadcast([id]) (from [destroySite])
```

In the end, `doDestroy` method is executed (that broadcast implementations are supposed to provide).

Note	<code>doDestroy</code> is a part of the Broadcast contract for broadcast implementations so they can provide their own custom behaviour.
------	--

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.

Broadcast Contract

The `Broadcast` contract is made up of the following methods that custom `Broadcast` implementations are supposed to provide:

1. `getValue`
2. `doUnpersist`
3. `doDestroy`

Note	TorrentBroadcast is the only implementation of the <code>Broadcast</code> contract.
------	---

Note	Broadcast Spark Developer-Facing Contract is the developer-facing <code>Broadcast</code> contract that allows Spark developers to use it in their applications.
------	---

Further Reading or Watching

- [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 (e.g. [task metrics](#)).

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										
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 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 .
-----	--

Table 1. AccumulatorV2's Internal Registries and Counters

Name	Description
metadata	AccumulatorMetadata Used when... FIXME
atDriverSide	Flag whether... FIXME Used when... FIXME

merge Method

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

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` creates a [AccumulatorMetadata](#) metadata object for the accumulator (with a [new unique identifier](#)) that is then used to register the accumulator with.

In the end, `register` registers the accumulator for cleanup (only when `ContextCleaner` is defined in the `SparkContext`).

`register` reports a `IllegalStateException` if `metadata` is already defined (which means that `register` was called already).

Cannot register an Accumulator twice.

Note	<code>register</code> is a <code>private[spark]</code> method.
------	--

Note	<code>register</code> is used when: <ul style="list-style-type: none"> • <code>SparkContext</code> registers accumulators • <code>TaskMetrics</code> registers the internal accumulators • <code>SQLMetrics</code> creates metrics.
------	--

AccumulatorMetadata

`AccumulatorMetadata` is a container object with the metadata of an accumulator:

- Accumulator ID
- (optional) name
- Flag whether to include the latest value of an accumulator on failure

Note	<code>countFailedValues</code> is used exclusively when <code>Task</code> collects the latest values of accumulators (irrespective of task status — a success or a failure).
------	--

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

FIXME Code with failing transformations (tasks) that update accumulator (`Map`) with `TaskContext` info.

Example: Custom Accumulator

Caution	FIXME Improve the earlier example
---------	---

Example: Distributed Stopwatch

Note	This is <i>almost</i> a raw copy of org.apache.spark.ml.util.DistributedStopwatch.
------	--

<pre>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 } }</pre>
--

Further reading or watching

- [Performance and Scalability of Broadcast in Spark](#)

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.

register Method

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

newId Method

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

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.

SerializerManager

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

When `SparkEnv` is created (either for the driver or executors), it instantiates `SerializerManager` that is then used to create a `BlockManager`.

`SerializerManager` automatically selects the "best" serializer for shuffle blocks that could either be `KryoSerializer` when a RDD's types are known to be compatible with Kryo or the default `Serializer`.

The common idiom in Spark's code is to access the current `SerializerManager` using `SparkEnv`.

```
SparkEnv.get.serializerManager
```

Note	<code>SerializerManager</code> was introduced in SPARK-13926.
------	---

Creating SerializerManager Instance

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

wrapStream Method

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

dataDeserializeStream Method

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

Automatic Selection of Best Serializer

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

`SerializerManager` will automatically pick a Kryo serializer for ShuffledRDDs whose key, value, and/or combiner types are primitives, arrays of primitives, or strings.

Selecting "Best" Serializer — `getSerializer` Method

```
getSerializer(keyClassTag: ClassTag[_], valueClassTag: ClassTag[_]): Serializer
```

`getSerializer` selects the "best" `Serializer` given the input types for keys and values (in a RDD).

`getSerializer` returns `KryoSerializer` when the types of keys and values are compatible with `Kryo` or the default `Serializer`.

Note	The default <code>Serializer</code> is defined when <code>SerializerManager</code> is created.
------	--

Note	<code>getSerializer</code> is used when <code>ShuffledRDD</code> returns the single-element dependency list (with <code>ShuffleDependency</code>).
------	---

Settings

Table 1. Spark Properties

Name	Default value	Description
<code>spark.shuffle.compress</code>	<code>true</code>	The flag to control whether to compress shuffle output when stored
<code>spark.rdd.compress</code>	<code>false</code>	The flag to control whether to compress RDD partitions when stored serialized.
<code>spark.shuffle.spill.compress</code>	<code>true</code>	The flag to control whether to compress shuffle output temporarily spilled to disk.
<code>spark.block.failures.beforeLocationRefresh</code>	<code>5</code>	
<code>spark.io.encryption.enabled</code>	<code>false</code>	The flag to enable IO encryption

MemoryManager — Memory Management System

`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:

1. [UnifiedMemoryManager](#) — the default memory manager since Spark 1.6.
2. `StaticMemoryManager` (legacy)

Note

`org.apache.spark.memory.MemoryManager` is a `private[spark]` 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

FIXME

`acquireStorageMemory` is used in [MemoryStore](#) to put bytes.

maxOffHeapStorageMemory Attribute

```
maxOffHeapStorageMemory: Long
```

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

maxOnHeapStorageMemory Attribute

```
maxOnHeapStorageMemory: Long
```

`maxOnHeapStorageMemory` is the total amount of memory available for storage, in bytes. It can vary over time.

Caution	FIXME 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	FIXME Describe <code>org.apache.spark.unsafe.Platform.unaligned</code> .
---------	--

UnifiedMemoryManager

`UnifiedMemoryManager` is the default `MemoryManager` with `onHeapStorageMemory` being ??? and `onHeapExecutionMemory` being ???

Calculate Maximum Memory to Use — `getMaxMemory` Method

```
getMaxMemory(conf: SparkConf): Long
```

`getMaxMemory` calculates the maximum memory to use for execution and storage.

```
// local mode with --conf spark.driver.memory=2g
scala> sc.getConf.getSizeAsBytes("spark.driver.memory")
res0: Long = 2147483648

scala> val systemMemory = Runtime.getRuntime.maxMemory

// fixed amount of memory for non-storage, non-execution purposes
val reservedMemory = 300 * 1024 * 1024

// minimum system memory required
val minSystemMemory = (reservedMemory * 1.5).ceil.toLong

val usableMemory = systemMemory - reservedMemory

val memoryFraction = sc.getConf.getDouble("spark.memory.fraction", 0.6)
scala> val maxMemory = (usableMemory * memoryFraction).toLong
maxMemory: Long = 956615884

import org.apache.spark.network.util.JavaUtils
scala> JavaUtils.byteStringAsMb(maxMemory + "b")
res1: Long = 912
```

`getMaxMemory` reads the maximum amount of memory that the Java virtual machine will attempt to use and decrements it by reserved system memory (for non-storage and non-execution purposes).

`getMaxMemory` makes sure that the following requirements are met:

1. System memory is not smaller than about 1,5 of the reserved system memory.
2. `spark.executor.memory` is not smaller than about 1,5 of the reserved system memory.

Ultimately, `getMaxMemory` returns `spark.memory.fraction` of the maximum amount of memory for the JVM (minus the reserved system memory).

Caution	FIXME omnigraffle it.
---------	---------------------------------------

Creating UnifiedMemoryManager Instance

```
class UnifiedMemoryManager(
    conf: SparkConf,
    val maxHeapMemory: Long,
    onHeapStorageRegionSize: Long,
    numCores: Int)
```

UnifiedMemoryManager requires a `SparkConf` and the following values:

- `maxHeapMemory` — the maximum on-heap memory to manage. It is assumed that `onHeapExecutionMemoryPool` with `onHeapStorageMemoryPool` is exactly `maxHeapMemory`.
- `onHeapStorageRegionSize`
- `numCores`

UnifiedMemoryManager makes sure that the sum of `offHeapExecutionMemoryPool` and `offHeapStorageMemoryPool` pool sizes is exactly `maxOffHeapMemory`.

Caution	FIXME Describe the pools
---------	--

apply Factory Method

```
apply(conf: SparkConf, numCores: Int): UnifiedMemoryManager
```

apply factory method creates an instance of `UnifiedMemoryManager`.

Internally, apply calculates the maximum memory to use (given `conf`). It then creates a `UnifiedMemoryManager` with the following values:

1. `maxHeapMemory` being the maximum memory just calculated.
2. `onHeapStorageRegionSize` being `spark.memory.storageFraction` of maximum memory.
3. `numCores` as configured.

Note	<code>apply</code> is used when <code>SparkEnv</code> is created.
------	---

acquireStorageMemory Method

```
acquireStorageMemory(
    blockId: BlockId,
    numBytes: Long,
    memoryMode: MemoryMode): Boolean
```

`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	FIXME Where are they used?
---------	--

Note	<code>acquireStorageMemory</code> is a part of the MemoryManager Contract .
------	---

In `MemoryMode.ON_HEAP`, `onHeapExecutionMemoryPool`, `onHeapStorageMemoryPool`, and `maxOnHeapStorageMemory` are used.

In `MemoryMode.OFF_HEAP`, `offHeapExecutionMemoryPool`, `offHeapStorageMemoryPool`, and `maxOffHeapMemory` are used.

Caution	FIXME 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`.

Note	<code>acquireStorageMemory</code> is used when <code>MemoryStore</code> acquires storage memory to <code>putBytes</code> or <code>putIteratorAsValues</code> and <code>putIteratorAsBytes</code> .
------	--

	It is also used internally when <code>UnifiedMemoryManager</code> acquires unroll memory.
--	---

acquireUnrollMemory Method

Note `acquireUnrollMemory` is a part of the [MemoryManager Contract](#).

`acquireUnrollMemory` simply forwards all the calls to [acquireStorageMemory](#).

acquireExecutionMemory Method

```
acquireExecutionMemory(  
    numBytes: Long,  
    taskAttemptId: Long,  
    memoryMode: MemoryMode): Long
```

`acquireExecutionMemory` does...[FIXME](#)

Internally, `acquireExecutionMemory` varies per `MemoryMode`, i.e. `ON_HEAP` and `OFF_HEAP`.

Table 1. `acquireExecutionMemory` and `MemoryMode`

	ON_HEAP	OFF_HEAP
<code>executionPool</code>	<code>onHeapExecutionMemoryPool</code>	<code>offHeapExecutionMemoryPool</code>
<code>storagePool</code>	<code>onHeapStorageMemoryPool</code>	<code>offHeapStorageMemoryPool</code>
<code>storageRegionSize</code>	<code>onHeapStorageRegionSize <1></code>	<code>offHeapStorageMemory</code>
<code>maxMemory</code>	<code>maxHeapMemory <2></code>	<code>maxOffHeapMemory</code>

- Defined when [UnifiedMemoryManager](#) is created.
- Defined when [UnifiedMemoryManager](#) is created.

Note `acquireExecutionMemory` is a part of the [MemoryManager Contract](#).

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

maxOnHeapStorageMemory Method

```
maxOnHeapStorageMemory: Long
```

`maxOnHeapStorageMemory` is the difference between `maxHeapMemory` of the `UnifiedMemoryManager` and the memory currently in use in `onHeapExecutionMemoryPool` execution memory pool.

Note

`maxOnHeapStorageMemory` is a part of the [MemoryManager Contract](#).

Settings

Table 2. Spark Properties

Spark Property	Default Value	Description
<code>spark.memory.fraction</code>	0.6	Fraction of JVM heap space used for execution and storage.
<code>spark.memory.storageFraction</code>	0.5	
<code>spark.testing.memory</code>	Java's Runtime.getRuntime.maxMemory	System memory
<code>spark.testing.reservedMemory</code>	300M or 0 (with <code>spark.testing</code> enabled)	

SparkEnv — Spark Runtime Environment

Spark Runtime Environment (`SparkEnv`) is the runtime environment with Spark's public services that interact with each other to establish a distributed computing platform for a Spark application.

Spark Runtime Environment is represented by a `SparkEnv` object that holds all the required runtime services for a running Spark application with separate environments for the `driver` and `executors`.

The idiomatic way in Spark to access the current `SparkEnv` when on the driver or executors is to use `get` method.

```
import org.apache.spark._  
scala> SparkEnv.get  
res0: org.apache.spark.SparkEnv = org.apache.spark.SparkEnv@49322d04
```

Table 1. `SparkEnv` Services

Property	Service	Description
<code>rpcEnv</code>	<code>RpcEnv</code>	
<code>serializer</code>	<code>Serializer</code>	
<code>closureSerializer</code>	<code>Serializer</code>	
<code>serializerManager</code>	<code>SerializerManager</code>	
<code>mapOutputTracker</code>	<code>MapOutputTracker</code>	
<code>shuffleManager</code>	<code>ShuffleManager</code>	
<code>broadcastManager</code>	<code>BroadcastManager</code>	
<code>blockManager</code>	<code>BlockManager</code>	
<code>securityManager</code>	<code>SecurityManager</code>	
<code>metricsSystem</code>	<code>MetricsSystem</code>	
<code>memoryManager</code>	<code>MemoryManager</code>	
<code>outputCommitCoordinator</code>	<code>OutputCommitCoordinator</code>	

Table 2. `SparkEnv`'s Internal Properties

Name	Initial Value	Description
<code>isStopped</code>	Disabled, i.e. <code>false</code>	Used to mark <code>SparkEnv</code> stopped. FIXME
<code>driverTmpDir</code>		

Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.SparkEnv</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.SparkEnv=DEBUG</pre> <p>Refer to Logging.</p>

SparkEnv Factory Object

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, i.e. a driver or an executor.

Table 3. `create`'s Input Arguments and Their Usage

Input Argument	Usage
<code>bindAddress</code>	Used to create <code>RpcEnv</code> and <code>NettyBlockTransferService</code> .
<code>advertiseAddress</code>	Used to create <code>RpcEnv</code> and <code>NettyBlockTransferService</code> .
<code>numUsableCores</code>	Used to create <code>MemoryManager</code> , <code>NettyBlockTransferService</code> and <code>BlockManager</code> .

When executed, `create` creates a `Serializer` (based on `spark.serializer` setting). You should see the following `DEBUG` message in the logs:

```
DEBUG SparkEnv: Using serializer: [serializer]
```

It creates another `Serializer` (based on `spark.closure.serializer`).

It creates a `ShuffleManager` based on `spark.shuffle.manager` Spark property.

It creates a `MemoryManager` based on `spark.memory.useLegacyMode` setting (with `UnifiedMemoryManager` being the default and `numCores` the input `numUsableCores`).

`create` creates a `NettyBlockTransferService`. It uses `spark.driver.blockManager.port` for the port on the driver and `spark.blockManager.port` for the port on executors.

Caution

FIXME A picture with `SparkEnv`, `NettyBlockTransferService` and the ports "armed".

`create` 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.

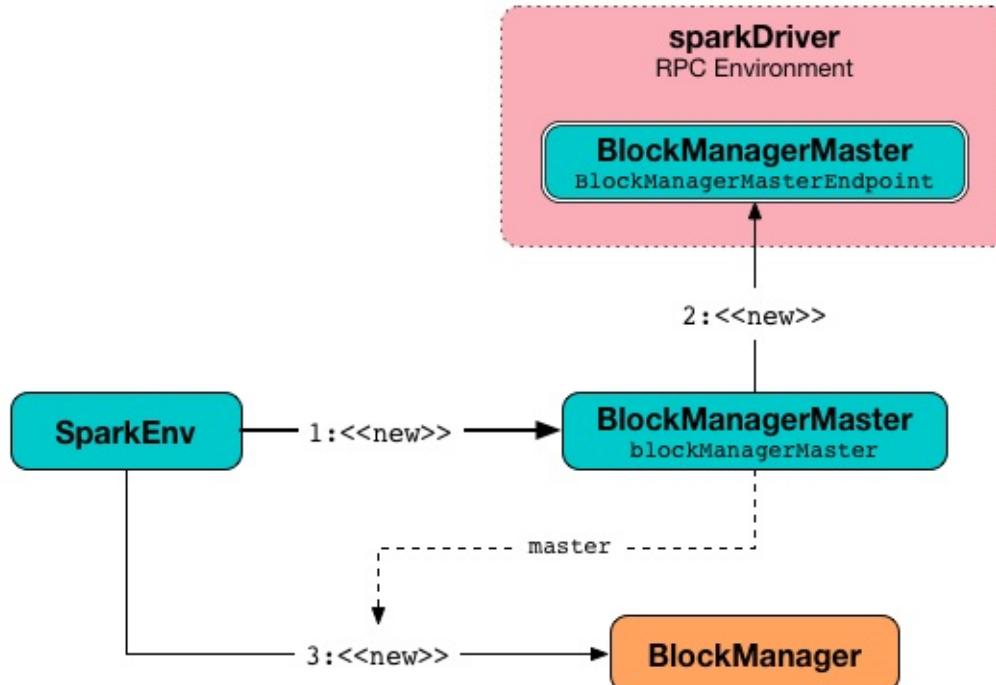


Figure 1. Creating BlockManager for the Driver

Note

`create` registers the `BlockManagerMaster` RPC endpoint for the driver and looks it up for executors.

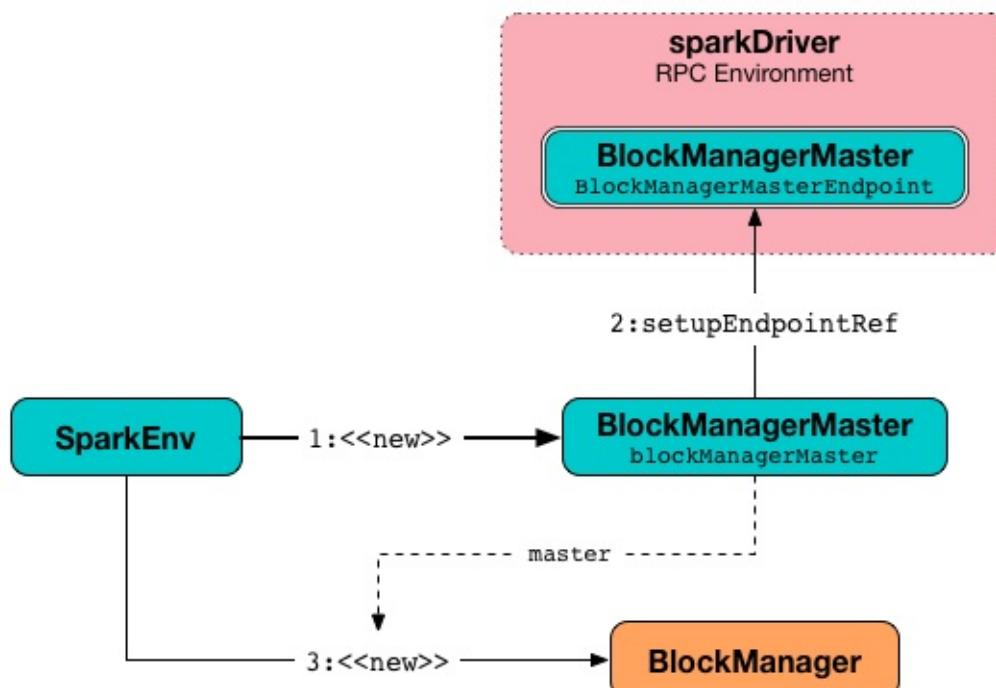


Figure 2. Creating BlockManager for Executor

It creates a `BlockManager` (using the above `BlockManagerMaster`, `NettyBlockTransferService` and other services).

`create` creates a [BroadcastManager](#).

`create` creates a [MapOutputTrackerMaster](#) or [MapOutputTrackerWorker](#) for the driver and executors, respectively.

Note

The choice of the real implementation of [MapOutputTracker](#) is based on whether the input `executorId` is **driver** or not.

`create` registers or looks up `RpcEndpoint` as **MapOutputTracker**. It registers [MapOutputTrackerMasterEndpoint](#) on the driver and creates a RPC endpoint reference on executors. The RPC endpoint reference gets assigned as the [MapOutputTracker](#) RPC endpoint.

Caution

[FIXME](#)

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

`create` is called by [createDriverEnv](#) and [createExecutorEnv](#).

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.

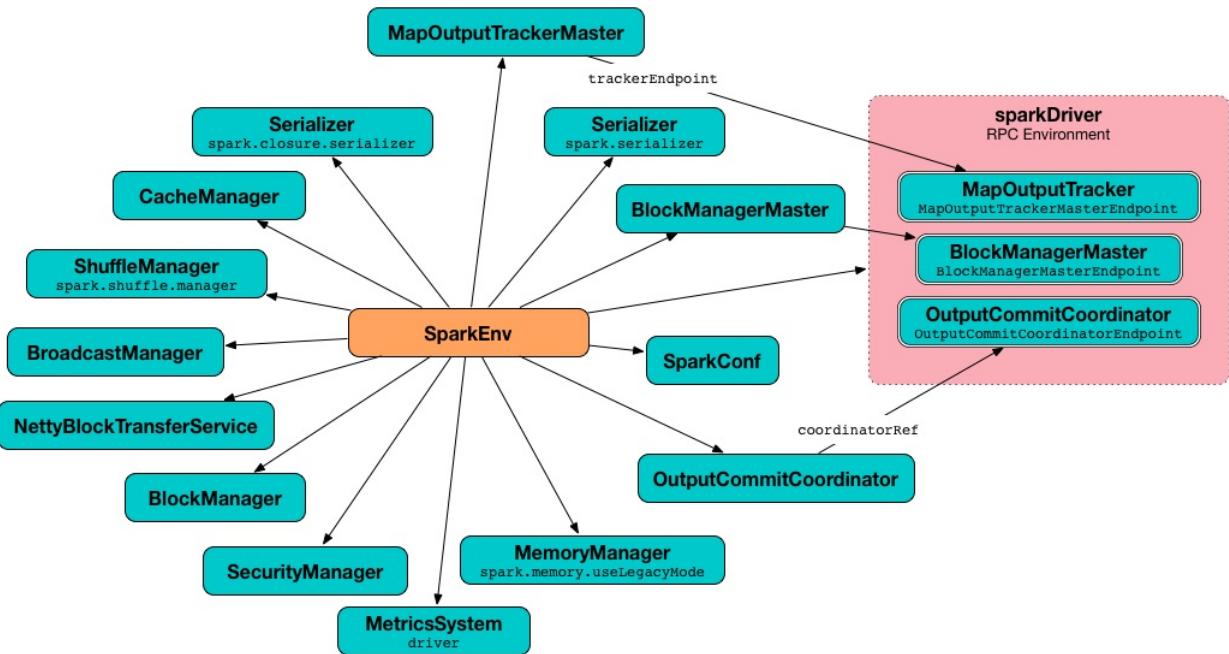


Figure 3. Spark Environment for driver

`createDriverEnv` accepts an instance of `SparkConf`, whether it runs in local mode or not, `LiveListenerBus`, the number of 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 defined.

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

```
createExecutorEnv(
    conf: SparkConf,
    executorId: String,
    hostname: String,
    port: Int,
    numCores: Int,
    ioEncryptionKey: Option[Array[Byte]],
    isLocal: Boolean): SparkEnv
```

`createExecutorEnv` creates an **executor's (execution) environment** that is the Spark execution environment for an executor.

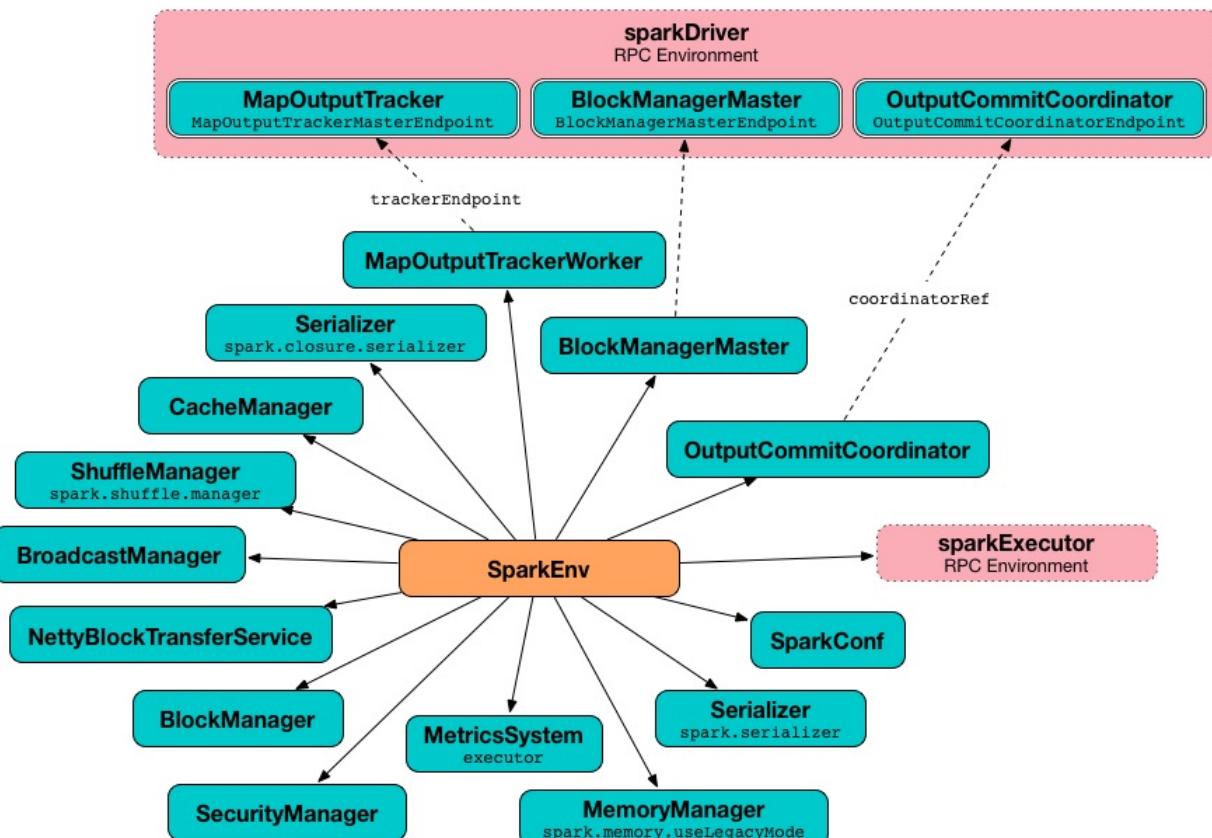


Figure 4. Spark Environment for executor

Note	<code>createExecutorEnv</code> is a <code>private[spark]</code> method.
------	---

`createExecutorEnv` simply **creates the base `SparkEnv`** (passing in all the input parameters) and **sets it as the current `SparkEnv`**.

Note	The number of cores <code>numcores</code> is configured using <code>--cores</code> command-line option of <code>CoarseGrainedExecutorBackend</code> and is specific to a cluster manager.
------	---

Note	<code>createExecutorEnv</code> is used when <code>CoarseGrainedExecutorBackend</code> runs and <code>MesosExecutorBackend</code> registers a Spark executor.
------	--

Getting Current SparkEnv — `get` Method

```
get: SparkEnv
```

`get` returns the current `SparkEnv`.

```
import org.apache.spark._  
scala> SparkEnv.get  
res0: org.apache.spark.SparkEnv = org.apache.spark.SparkEnv@49322d04
```

Stopping SparkEnv — `stop` Method

```
stop(): Unit
```

`stop` checks `isStopped` internal flag and does nothing when enabled.

Note	<code>stop</code> is a <code>private[spark]</code> method.
------	--

Otherwise, `stop` turns `isStopped` flag on, stops all `pythonWorkers` and requests the following services to stop:

1. `MapOutputTracker`
2. `ShuffleManager`
3. `BroadcastManager`
4. `BlockManager`
5. `BlockManagerMaster`
6. `MetricsSystem`
7. `OutputCommitCoordinator`

`stop` requests `RpcEnv` to shut down and waits till it terminates.

Only on the driver, `stop` deletes the [temporary directory](#). You can see the following WARN message in the logs if the deletion fails.

```
WARN Exception while deleting Spark temp dir: [path]
```

Note	<code>stop</code> is used when <code>SparkContext</code> stops (on the driver) and <code>Executor</code> stops.
------	---

Settings

Table 4. Spark Properties

Spark Property	Default Value	Description
<code>spark.serializer</code>	<code>org.apache.spark.serializer.JavaSerializer</code>	<p>Serializer</p> <p>TIP: Enabling logging level <code>INFO</code> on <code>org.apache.spark.serializer</code> logger to see what is being serialized.</p> <pre>` DEBUG org.apache.spark.serializer`</pre>
<code>spark.closure.serializer</code>	<code>org.apache.spark.serializer.JavaSerializer</code>	Serializer
<code>spark.memory.useLegacyMode</code>	<code>false</code>	<p>Controls whether to use legacy memory management mode.</p> <p>When enabled, it is the legacy mode, using <code>StaticMemoryManager</code>. When disabled, it uses <code>UnifiedMemoryManager</code>, otherwise.</p>

DAGScheduler — Stage-Oriented Scheduler

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**. It transforms a **logical execution plan** (i.e. [RDD lineage](#) of dependencies built using [RDD transformations](#)) to a **physical execution plan** (using [stages](#)).

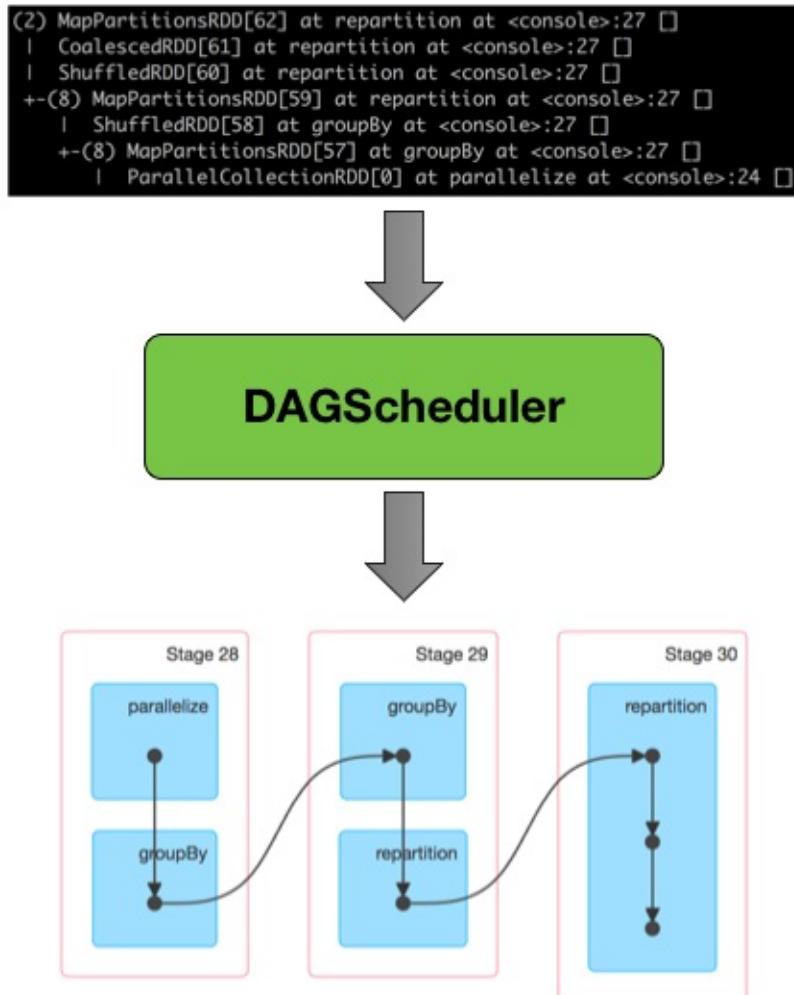


Figure 1. **DAGScheduler** Transforming RDD Lineage Into Stage DAG

After an [action](#) has been called, [SparkContext](#) hands over a logical plan to [DAGScheduler](#) that it in turn translates to a set of stages that are submitted as [TaskSets](#) for execution (see [Execution Model](#)).

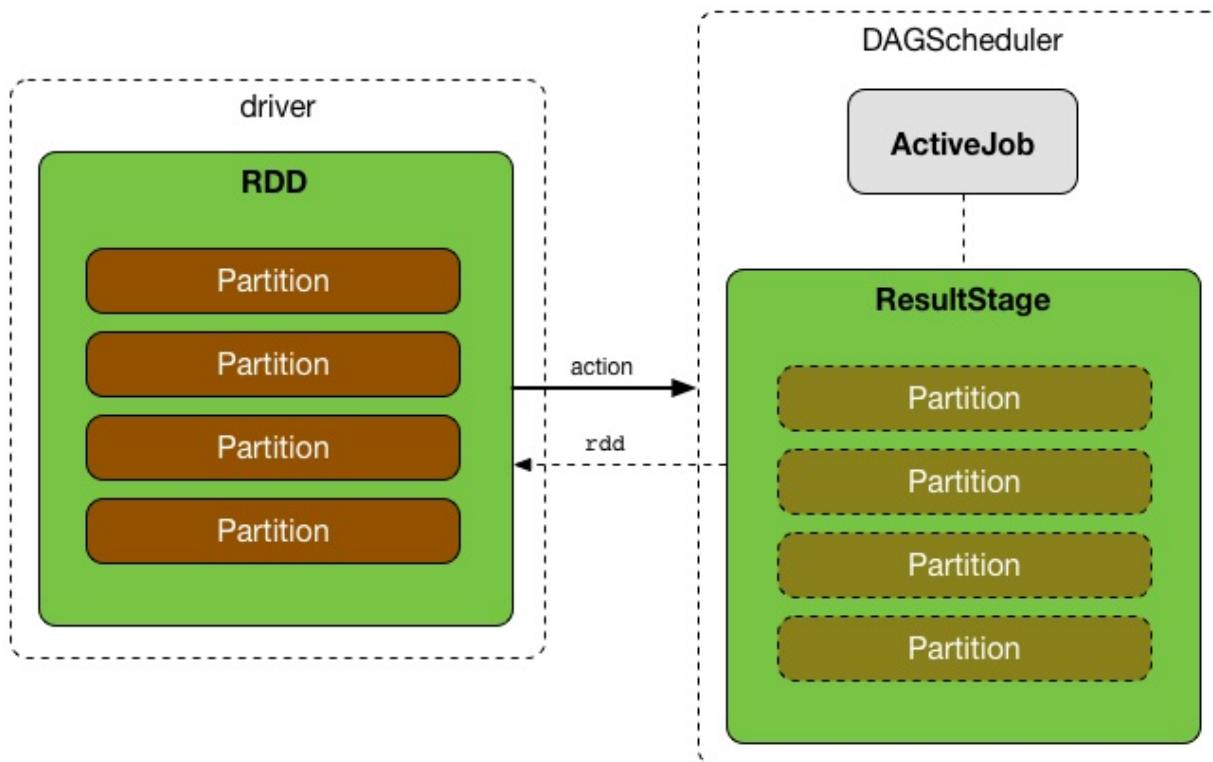


Figure 2. Executing action leads to new ResultStage and ActiveJob in DAGScheduler
The fundamental concepts of [DAGScheduler](#) are **jobs** and **stages** (refer to [Jobs](#) and [Stages](#) respectively) that it tracks through [internal registries and counters](#).

DAGScheduler works solely on the driver and is created as part of [SparkContext's initialization](#) (right after [TaskScheduler](#) and [SchedulerBackend](#) are ready).

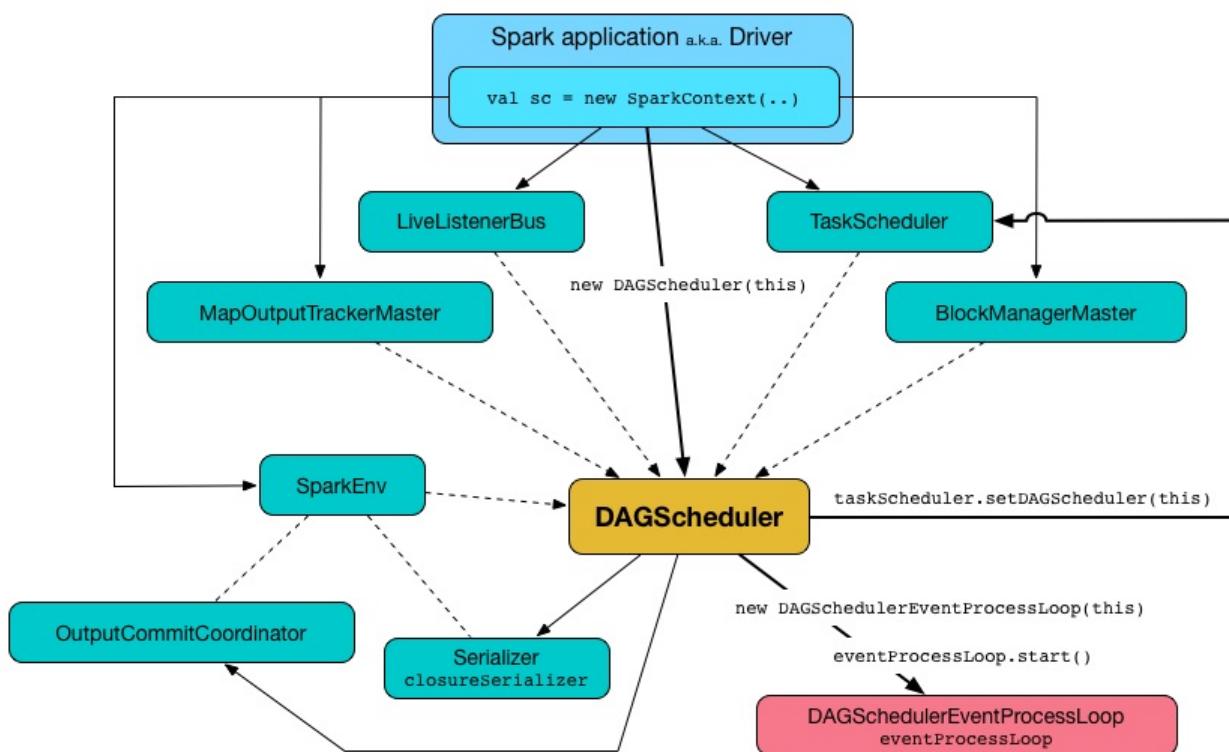


Figure 3. DAGScheduler as created by SparkContext with other services
DAGScheduler does three things in Spark (thorough explanations follow):

- 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.

`DAGScheduler` 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`.

`DAGScheduler` tracks which **RDDs are cached (or persisted)** to avoid "recomputing" them, i.e. redoing the map side of a shuffle. `DAGScheduler` remembers what **ShuffleMapStages** have already produced output files (that are stored in **BlockManagers**).

DAGScheduler is only interested in cache location coordinates, i.e. host and executor id, per partition of a RDD.

Caution

FIXME: A diagram, please

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.

Table 1. DAGScheduler's Internal Properties

Name	Initial Value	Description
metricsSource	<code>DAGSchedulerSource</code>	<code>FIXME</code>

Table 2. DAGScheduler's Internal Registries and Counters

Name	Description
<code>activeJobs</code>	<code>ActiveJob</code> instances
<code>cacheLocs</code>	<p>Block locations per RDD and partition.</p> <p>Uses <code>TaskLocation</code> that includes a host name and an executor id on that host (as <code>ExecutorCacheTaskLocation</code>).</p> <p>The keys are RDDs (their ids) and the values are arrays indexed by partition numbers.</p> <p>Each entry is a set of block locations where a RDD partition is cached, i.e. the <code>BlockManager</code>s of the blocks.</p> <p>Initialized empty when <code>DAGScheduler</code> is created.</p> <p>Used when <code>DAGScheduler</code> is requested for the locations of the cache blocks of a RDD or clear them.</p>
<code>failedEpoch</code>	The lookup table of lost executors and the epoch of the event.
<code>failedStages</code>	Stages that failed due to fetch failures (when a task fails with FetchFailed exception).
<code> jobIdToActiveJob</code>	The lookup table of <code>ActiveJob</code> s per job id.
<code> jobIdToStageIds</code>	The lookup table of all stages per <code>ActiveJob</code> id
	The next job id counting from <code>0</code> .

	Used when <code>DAGScheduler</code> submits a job and a map stage, and runs an approximate job.
<code>nextStageId</code>	The next stage id counting from 0 . Used when <code>DAGScheduler</code> creates a shuffle map stage and a result stage. It is the key in <code>stageIdToStage</code> .
<code>runningStages</code>	The set of stages that are currently "running". A stage is added when <code>submitMissingTasks</code> gets executed (without first checking if the stage has not already been added).
<code>shuffleIdToMapStage</code>	The lookup table of <code>ShuffleMapStages</code> per <code>ShuffleDependency</code> .
<code>stageIdToStage</code>	The lookup table for stages per their ids. Used when <code>DAGScheduler</code> creates a shuffle map stage, creates a result stage, cleans up job state and independent stages, is informed that a task is started, a taskset has failed, a job is submitted (to compute a <code>ResultStage</code>), a map stage was submitted, a task has completed or a stage was cancelled, updates accumulators, aborts a stage and fails a job and independent stages.
<code>waitingStages</code>	The stages with parents to be computed

Tip	Enable <code>INFO</code> , <code>DEBUG</code> or <code>TRACE</code> logging levels for <code>org.apache.spark.scheduler.DAGScheduler</code> logger to see what happens inside <code>DAGScheduler</code> . Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.scheduler.DAGScheduler=TRACE</code>
	Refer to Logging .

DAGScheduler uses `SparkContext`, `TaskScheduler`, `LiveListenerBus`, `MapOutputTracker` and `BlockManager` for its services. However, at the very minimum, `DAGScheduler` takes a `SparkContext` only (and requests `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.

Running Approximate Job — runApproximateJob Method

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

createResultStage Internal Method

```
createResultStage(  
    rdd: RDD[_],  
    func: (TaskContext, Iterator[_]) => _,  
    partitions: Array[Int],  
    jobId: Int,  
    callSite: CallSite): ResultStage
```

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

updateJobIdStageIdMaps Method

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

Creating DAGScheduler Instance

DAGScheduler takes the following when created:

- [SparkContext](#)
- [TaskScheduler](#)
- [LiveListenerBus](#)
- [MapOutputTrackerMaster](#)
- [BlockManagerMaster](#)
- [SparkEnv](#)
- [Clock](#) (defaults to [SystemClock](#))

DAGScheduler initializes the [internal registries and counters](#).

DAGScheduler sets itself in the given TaskScheduler and in the end starts DAGScheduler Event Bus.

Note

DAGScheduler can reference all the services through a single SparkContext with or without specifying explicit TaskScheduler.

LiveListenerBus Event Bus for SparkListenerEvents — listenerBus Property

```
listenerBus: LiveListenerBus
```

listenerBus is a LiveListenerBus to post scheduling events and is passed in when DAGScheduler is created.

executorHeartbeatReceived Method

```
executorHeartbeatReceived(  
  execId: String,  
  accumUpdates: Array[(Long, Int, Int, Seq[AccumulableInfo])],  
  blockManagerId: BlockManagerId): Boolean
```

executorHeartbeatReceived posts a SparkListenerExecutorMetricsUpdate (to listenerBus) and informs BlockManagerMaster that blockManagerId block manager is alive (by posting BlockManagerHeartbeat).

Note

executorHeartbeatReceived is called when TaskSchedulerImpl handles executorHeartbeatReceived .

Cleaning Up After ActiveJob and Independent Stages — cleanupStateForJobAndIndependentStages Method

```
cleanupStateForJobAndIndependentStages(job: ActiveJob): Unit
```

cleanupStateForJobAndIndependentStages cleans up the state for job and any stages that are *not* part of any other job.

cleanupStateForJobAndIndependentStages looks the job up in the internal jobIdToStageIds registry.

If no stages are found, the following ERROR is printed out to the logs:

```
ERROR No stages registered for job [jobId]
```

Otherwise, `cleanupStateForJobAndIndependentStages` uses `stageIdToStage` registry to find the stages (the real objects not ids!).

For each stage, `cleanupStateForJobAndIndependentStages` reads the jobs the stage belongs to.

If the `job` does not belong to the jobs of the stage, the following ERROR is printed out to the logs:

```
ERROR Job [jobId] not registered for stage [stageId] even though that stage was registered for the job
```

If the `job` was the only job for the stage, the stage (and the stage id) gets cleaned up from the registries, i.e. `runningStages`, `shuffleIdToMapStage`, `waitingStages`, `failedStages` and `stageIdToStage`.

While removing from `runningStages`, you should see the following DEBUG message in the logs:

```
DEBUG Removing running stage [stageId]
```

While removing from `waitingStages`, you should see the following DEBUG message in the logs:

```
DEBUG Removing stage [stageId] from waiting set.
```

While removing from `failedStages`, you should see the following DEBUG message in the logs:

```
DEBUG Removing stage [stageId] from failed set.
```

After all cleaning (using `stageIdToStage` as the source registry), if the stage belonged to the one and only `job`, you should see the following DEBUG message in the logs:

```
DEBUG After removal of stage [stageId], remaining stages = [stageIdToStage.size]
```

The `job` is removed from `jobIdToStageIds`, `jobIdToActiveJob`, `activeJobs` registries.

The final stage of the `job` is removed, i.e. `ResultStage` or `ShuffleMapStage`.

Note

`cleanupStateForJobAndIndependentStages` is used in `handleTaskCompletion` when a `ResultTask` has completed successfully, `failJobAndIndependentStages` and `markMapStageJobAsFinished`.

Marking ShuffleMapStage Job Finished — `markMapStageJobAsFinished` Method

```
markMapStageJobAsFinished(job: ActiveJob, stats: MapOutputStatistics): Unit
```

`markMapStageJobAsFinished` marks the active `job` finished and notifies Spark listeners.

Internally, `markMapStageJobAsFinished` marks the zeroth partition finished and increases the number of tasks finished in `job`.

The `job` listener is notified about the 0th task succeeded.

The state of the `job` and independent stages are cleaned up.

Ultimately, `SparkListenerJobEnd` is posted to `LiveListenerBus` (as `listenerBus`) for the `job`, the current time (in millis) and `JobSucceeded` job result.

Note

`markMapStageJobAsFinished` is used in `handleMapStageSubmitted` and `handleTaskCompletion`.

Submitting Job — `submitJob` method

```
submitJob[T, U](
  rdd: RDD[T],
  func: (TaskContext, Iterator[T]) => U,
  partitions: Seq[Int],
  callSite: CallSite,
  resultHandler: (Int, U) => Unit,
  properties: Properties): JobWaiter[U]
```

`submitJob` creates a `JobWaiter` and posts a `JobSubmitted` event.

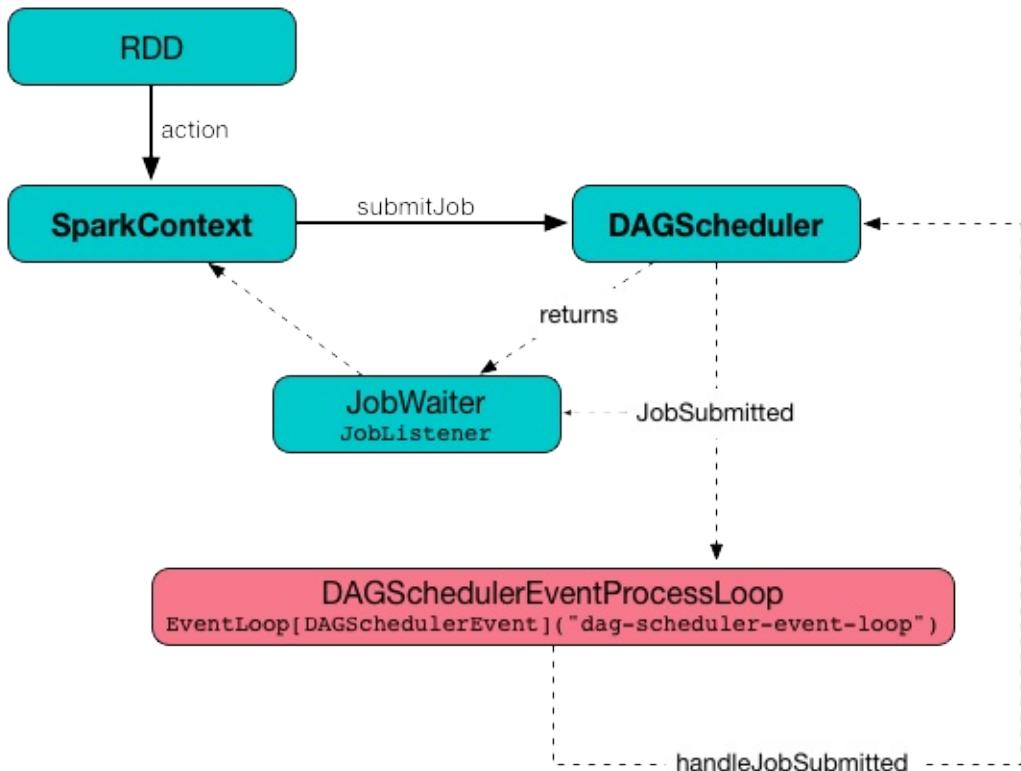


Figure 4. DAGScheduler.submitJob

Internally, `submitJob` does the following:

1. Checks whether `partitions` reference available partitions of the input `rdd`.
2. Increments `nextJobId` internal job counter.
3. Returns a 0-task `JobWaiter` when the number of `partitions` is zero.
4. Posts a `JobSubmitted` event and returns a `JobWaiter`.

You may see a `IllegalArgumentException` thrown when the input `partitions` references partitions not in the input `rdd`:

```
Attempting to access a non-existent partition: [p]. Total number of partitions: [maxPartitions]
```

Note	<code>submitJob</code> is called when <code>SparkContext</code> submits a job and <code>DAGScheduler</code> runs a job.
Note	<code>submitJob</code> assumes that the partitions of a RDD are indexed from 0 onwards in sequential order.

Submitting ShuffleDependency for Execution — `submitMapStage` Method

```
submitMapStage[K, V, C](
    dependency: ShuffleDependency[K, V, C],
    callback: MapOutputStatistics => Unit,
    callSite: CallSite,
    properties: Properties): JobWaiter[MapOutputStatistics]
```

`submitMapStage` creates a [JobWaiter](#) (that it eventually returns) and posts a [MapStageSubmitted](#) event to [DAGScheduler Event Bus](#)).

Internally, `submitMapStage` increments [nextJobId](#) internal counter to get the job id.

`submitMapStage` then creates a [JobWaiter](#) (with the job id and with one artificial task that will however get completed only when the entire stage finishes).

`submitMapStage` announces the map stage submission application-wide (by posting a [MapStageSubmitted](#) to [LiveListenerBus](#)).

Note	A <code>MapStageSubmitted</code> holds the newly-created job id and <code>JobWaiter</code> with the input <code>dependency</code> , <code>callSite</code> and <code>properties</code> parameters.
------	---

`submitMapStage` returns the `JobWaiter`.

If the number of partition to compute is `0`, `submitMapStage` throws a `SparkException`:

```
Can't run submitMapStage on RDD with 0 partitions
```

Note	<code>submitMapStage</code> is used when SparkContext submits a map stage for execution.
------	--

Relaying Stage Cancellation From SparkContext (by Posting StageCancelled to DAGScheduler Event Bus)

— `cancelStage` Method

```
cancelStage(stageId: Int)
```

`cancelJobGroup` merely posts a [StageCancelled](#) event to the [DAGScheduler Event Bus](#).

Note	<code>cancelStage</code> is used exclusively when SparkContext cancels a stage.
------	---

Relaying Job Group Cancellation From SparkContext (by Posting JobGroupCancelled to DAGScheduler Event Bus)

— `cancelJobGroup` Method

```
cancelJobGroup(groupId: String): Unit
```

`cancelJobGroup` prints the following INFO message to the logs followed by posting a [JobGroupCancelled](#) event to the [DAGScheduler Event Bus](#).

```
INFO Asked to cancel job group [groupId]
```

Note	<code>cancelJobGroup</code> is used exclusively when <code>SparkContext</code> cancels a job group .
------	--

Relaying All Jobs Cancellation From SparkContext (by Posting AllJobsCancelled to DAGScheduler Event Bus) — `cancelAllJobs` Method

```
cancelAllJobs(): Unit
```

`cancelAllJobs` merely posts a [AllJobsCancelled](#) event to the [DAGScheduler Event Bus](#).

Note	<code>cancelAllJobs</code> is used exclusively when <code>SparkContext</code> cancels all running or scheduled Spark jobs .
------	---

Relaying Task Started From TaskSetManager (by Posting BeginEvent to DAGScheduler Event Bus) — `taskStarted` Method

```
taskStarted(task: Task[_], taskInfo: TaskInfo)
```

`taskStarted` merely posts a [BeginEvent](#) event to the [DAGScheduler Event Bus](#).

Note	<code>taskStarted</code> is used exclusively when a <code>TaskSetManager</code> starts a task .
------	---

Relaying Task Fetching/Getting Result From TaskSetManager (by Posting GettingResultEvent to DAGScheduler Event Bus) — `taskGettingResult` Method

```
taskGettingResult(taskInfo: TaskInfo)
```

`taskGettingResult` merely posts a [GettingResultEvent](#) event to the [DAGScheduler Event Bus](#).

Note	<code>taskGettingResult</code> is used exclusively when a <code>TaskSetManager</code> gets notified about a task fetching result.
------	---

Relaying Task End From TaskSetManager (by Posting CompletionEvent to DAGScheduler Event Bus)

— `taskEnded` Method

```
taskEnded(  
    task: Task[_],  
    reason: TaskEndReason,  
    result: Any,  
    accumUpdates: Map[Long, Any],  
    taskInfo: TaskInfo,  
    taskMetrics: TaskMetrics): Unit
```

`taskEnded` simply posts a [CompletionEvent](#) event to the [DAGScheduler Event Bus](#).

Note	<code>taskEnded</code> is used exclusively when a <code>TaskSetManager</code> reports task completions, i.e. success or failure .
------	---

Tip	Read TaskMetrics .
-----	------------------------------------

Relaying TaskSet Failed From TaskSetManager (by Posting TaskSetFailed to DAGScheduler Event Bus)

— `taskSetFailed` Method

```
taskSetFailed(  
    taskSet: TaskSet,  
    reason: String,  
    exception: Option[Throwable]): Unit
```

`taskSetFailed` simply posts a [TaskSetFailed](#) to [DAGScheduler Event Bus](#).

Note	The input arguments of <code>taskSetFailed</code> are exactly the arguments of TaskSetFailed .
------	--

Note	<code>taskSetFailed</code> is used exclusively when a <code>TaskSetManager</code> is aborted.
------	---

Relying Executor Lost From TaskSchedulerImpl (by Posting ExecutorLost to DAGScheduler Event Bus)

— executorLost Method

```
executorLost(execId: String, reason: ExecutorLossReason): Unit
```

`executorLost` simply posts a [ExecutorLost](#) event to [DAGScheduler](#) Event Bus.

Note

`executorLost` is used when [TaskSchedulerImpl](#) [gets task status update](#) (and a task gets lost which is used to indicate that the executor got broken and hence should be considered lost) or [executorLost](#).

Relying Executor Added From TaskSchedulerImpl (by Posting ExecutorAdded to DAGScheduler Event Bus)

— executorAdded Method

```
executorAdded(execId: String, host: String): Unit
```

`executorAdded` simply posts a [ExecutorAdded](#) event to [DAGScheduler](#) Event Bus.

Note

`executorAdded` is used exclusively when [TaskSchedulerImpl](#) [is offered resources on executors](#) (and a new executor is found in the resource offers).

Relying Job Cancellation From SparkContext or JobWaiter (by Posting JobCancelled to DAGScheduler Event Bus)

— cancelJob Method

```
cancelJob(jobId: Int): Unit
```

`cancelJob` prints the following INFO message and posts a [JobCancelled](#) to [DAGScheduler](#) Event Bus.

```
INFO DAGScheduler: Asked to cancel job [id]
```

Note

`cancelJob` is used when [SparkContext](#) or [JobWaiter](#) cancel a Spark job.

Finding Or Creating Missing Direct Parent ShuffleMapStages (For ShuffleDependencies of Input RDD) — `getOrCreateParentStages` Internal Method

```
getOrCreateParentStages(rdd: RDD[_], firstJobId: Int): List[Stage]
```

`getOrCreateParentStages` finds all direct parent `ShuffleDependencies` of the input `rdd` and then finds `shuffleMapStage` stages for each `ShuffleDependency`.

Note

`getOrCreateParentStages` is used when `DAGScheduler` `createShuffleMapStage` and `createResultStage`.

Marking Stage Finished — `markStageAsFinished` Internal Method

```
markStageAsFinished(stage: Stage, errorMessage: Option[String] = None): Unit
```

Caution

[FIXME](#)

Running Job — `runJob` Method

```
runJob[T, U](
  rdd: RDD[T],
  func: (TaskContext, Iterator[T]) => U,
  partitions: Seq[Int],
  callSite: CallSite,
  resultHandler: (Int, U) => Unit,
  properties: Properties): Unit
```

`runJob` submits an action job to the `DAGScheduler` and waits for a result.

Internally, `runJob` executes `submitJob` and then waits until a result comes using `JobWaiter`.

When the job succeeds, you should see the following INFO message in the logs:

```
INFO Job [jobId] finished: [callSite], took [time] s
```

When the job fails, you should see the following INFO message in the logs and the exception (that led to the failure) is thrown.

```
INFO Job [jobId] failed: [callSite], took [time] s
```

Note	<code>runJob</code> is used when <code>SparkContext</code> runs a job.
------	--

Finding or Creating New ShuffleMapStages for ShuffleDependency — `getOrCreateShuffleMapStage` Internal Method

```
getOrCreateShuffleMapStage(  
    shuffleDep: ShuffleDependency[_, _, _],  
    firstJobId: Int): ShuffleMapStage
```

`getOrCreateShuffleMapStage` finds or creates the `ShuffleMapStage` for the input `ShuffleDependency`.

Internally, `getOrCreateShuffleMapStage` finds the `ShuffleDependency` in `shuffleIdToMapStage` internal registry and returns one when found.

If no `ShuffleDependency` was available, `getOrCreateShuffleMapStage` finds all the missing shuffle dependencies and creates corresponding `ShuffleMapStage` stages (including one for the input `shuffleDep`).

Note	All the new <code>ShuffleMapStage</code> stages are associated with the input <code>firstJobId</code> .
------	---

Note	<code>getOrCreateShuffleMapStage</code> is used when DAGScheduler finds or creates missing direct parent <code>ShuffleMapStages</code> (for <code>ShuffleDependencies</code> of given RDD), <code>getMissingParentStages</code> (for <code>ShuffleDependencies</code>), is notified that <code>ShuffleDependency</code> was submitted, and checks if a stage depends on another.
------	---

Creating ShuffleMapStage for ShuffleDependency (Copying Shuffle Map Output Locations From Previous Jobs) — `createShuffleMapStage` Method

```
createShuffleMapStage(  
    shuffleDep: ShuffleDependency[_, _, _],  
    jobId: Int): ShuffleMapStage
```

`createShuffleMapStage` creates a `ShuffleMapStage` for the input `ShuffleDependency` and `jobId` (of a `ActiveJob`) possibly copying shuffle map output locations from previous jobs to avoid recomputing records.

Note

When a `ShuffleMapStage` is created, the `id` is generated (using `nextStageId internal counter`), `rdd` is from `ShuffleDependency`, `numTasks` is the number of partitions in the RDD, all `parents` are looked up (and possibly created), the `jobId` is given, `callSite` is the `creationSite` of the RDD, and `shuffleDep` is the input `ShuffleDependency`.

Internally, `createShuffleMapStage` first finds or creates missing parent `ShuffleMapStage` stages of the associated RDD.

Note

`ShuffleDependency` is associated with exactly one `RDD[Product2[K, V]]`.

`createShuffleMapStage` creates a `ShuffleMapStage` (with the stage id from `nextStageId internal counter`).

Note

The RDD of the new `ShuffleMapStage` is from the input `ShuffleDependency`.

`createShuffleMapStage` registers the `ShuffleMapStage` in `stageIdToStage` and `shuffledIdToMapStage` internal registries.

`createShuffleMapStage` calls `updateJobIdStageIdMaps`.

If `MapOutputTrackerMaster` tracks the input `shuffleDependency` (because other jobs have already computed it), `createShuffleMapStage` requests the serialized `ShuffleMapStage` outputs, deserializes them and registers with the new `ShuffleMapStage`.

Note

`MapOutputTrackerMaster` was defined when `DAGScheduler` was created.

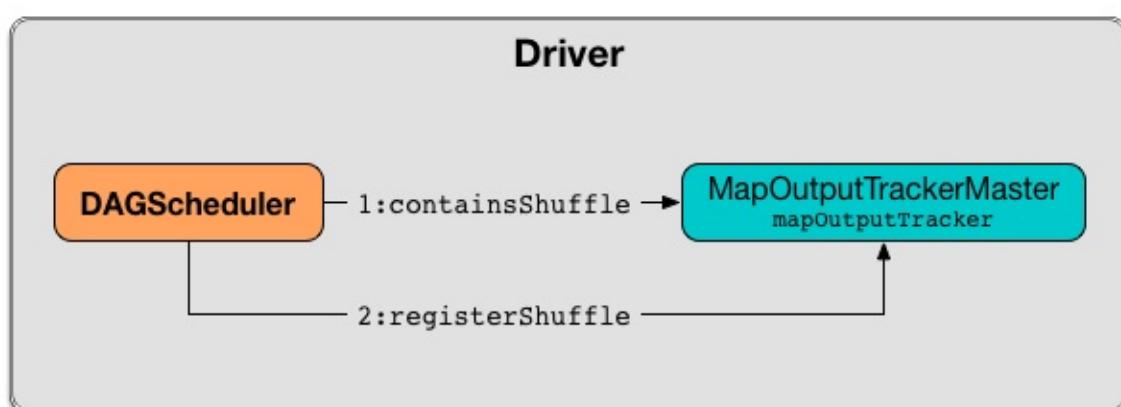


Figure 5. `DAGScheduler` Asks `MapOutputTrackerMaster` Whether Shuffle Map Output Is Already Tracked

If however `MapOutputTrackerMaster` does not track the input `shuffleDependency`, you should see the following INFO message in the logs and `createShuffleMapStage` registers the `ShuffleDependency` with `MapOutputTrackerMaster`.

```
INFO Registering RDD [id] ([creationSite])
```

`createShuffleMapStage` returns the new `ShuffleMapStage`.

Note	<code>createShuffleMapStage</code> is executed only when <code>DAGScheduler</code> finds or creates parent <code>ShuffleMapStage</code> stages for a <code>ShuffleDependency</code> .
------	---

Clearing Cache of RDD Block Locations — `clearCacheLocs` Internal Method

```
clearCacheLocs(): Unit
```

`clearCacheLocs` clears the internal registry of the partition locations per RDD.

Note	<code>DAGScheduler</code> clears the cache while resubmitting failed stages, and as a result of <code>JobSubmitted</code> , <code>MapStageSubmitted</code> , <code>CompletionEvent</code> , <code>ExecutorLost</code> events.
------	---

Finding Missing ShuffleDependencies For RDD — `getMissingAncestorShuffleDependencies` Internal Method

```
getMissingAncestorShuffleDependencies(rdd: RDD[_]): Stack[ShuffleDependency[_, _, _]]
```

`getMissingAncestorShuffleDependencies` finds all missing `shuffle` dependencies for the given RDD traversing its `dependency chain` (aka *RDD lineage*).

Note	A missing shuffle dependency of a RDD is a dependency not registered in <code>shuffleIdToMapStage</code> internal registry.
------	--

Internally, `getMissingAncestorShuffleDependencies` finds direct parent shuffle dependencies of the input RDD and collects the ones that are not registered in `shuffleIdToMapStage` internal registry. It repeats the process for the RDDs of the parent shuffle dependencies.

Note	<code>getMissingAncestorShuffleDependencies</code> is used when <code>DAGScheduler</code> finds all <code>ShuffleMapStage</code> stages for a <code>ShuffleDependency</code> .
------	--

Finding Direct Parent Shuffle Dependencies of RDD — `getShuffleDependencies` Internal Method

```
getShuffleDependencies(rdd: RDD[_]): HashSet[ShuffleDependency[_, _, _]]
```

`getShuffleDependencies` finds direct parent `shuffle dependencies` for the given RDD.

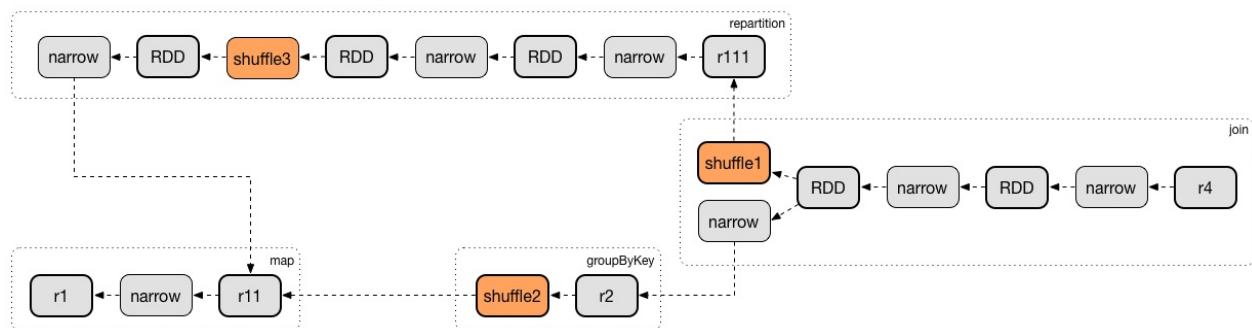


Figure 6. `getShuffleDependencies` Finds Direct Parent ShuffleDependencies (shuffle1 and shuffle2)

Internally, `getShuffleDependencies` takes the direct [shuffle dependencies of the input RDD](#) and direct shuffle dependencies of all the parent non-[shuffleDependencies](#) in the [dependency chain](#) (aka *RDD lineage*).

Note

`getShuffleDependencies` is used when [DAGScheduler finds or creates missing direct parent ShuffleMapStages](#) (for [ShuffleDependencies](#) of given [RDD](#)) and [finds all missing shuffle dependencies for a given \[RDD\]\(#\)](#).

Failing Job and Independent Single-Job Stages — `failJobAndIndependentStages` Internal Method

```
failJobAndIndependentStages(  
    job: ActiveJob,  
    failureReason: String,  
    exception: Option[Throwable] = None): Unit
```

The internal `failJobAndIndependentStages` method fails the input `job` and all the stages that are only used by the job.

Internally, `failJobAndIndependentStages` uses [jobIdToStageIds internal registry](#) to look up the stages registered for the job.

If no stages could be found, you should see the following ERROR message in the logs:

```
ERROR No stages registered for job [id]
```

Otherwise, for every stage, `failJobAndIndependentStages` finds the job ids the stage belongs to.

If no stages could be found or the job is not referenced by the stages, you should see the following ERROR message in the logs:

```
ERROR Job [id] not registered for stage [id] even though that stage was registered for
the job
```

Only when there is exactly one job registered for the stage and the stage is in RUNNING state (in `runningStages` internal registry), [TaskScheduler](#) is requested to cancel the stage's tasks and marks the stage finished.

Note	<code>failJobAndIndependentStages</code> is called from handleJobCancellation and <code>abortStage</code> .
------	---

Note	<code>failJobAndIndependentStages</code> uses <code>jobIdToStageIds</code> , <code>stageIdToStage</code> , and <code>runningStages</code> internal registries.
------	--

Aborting Stage — `abortStage` Internal Method

```
abortStage(
  failedStage: Stage,
  reason: String,
  exception: Option[Throwable]): Unit
```

`abortStage` is an internal method that finds all the active jobs that depend on the `failedStage` stage and fails them.

Internally, `abortStage` looks the `failedStage` stage up in the internal `stageIdToStage` registry and exits if there the stage was not registered earlier.

If it was, `abortStage` finds all the active jobs (in the internal `activeJobs` registry) with the final stage depending on the `failedStage` stage.

At this time, the `completionTime` property (of the failed stage's [StageInfo](#)) is assigned to the current time (millis).

All the active jobs that depend on the failed stage (as calculated above) and the stages that do not belong to other jobs (aka *independent stages*) are [failed](#) (with the failure reason being "Job aborted due to stage failure: [reason]" and the input `exception`).

If there are no jobs depending on the failed stage, you should see the following INFO message in the logs:

```
INFO Ignoring failure of [failedStage] because all jobs depending on it are done
```

Note	<code>abortStage</code> is used to handle <code>TaskSetFailed</code> event, when submitting a stage with no active job
------	--

Checking Out Stage Dependency on Given Stage

— `stageDependsOn` Method

```
stageDependsOn(stage: Stage, target: Stage): Boolean
```

`stageDependsOn` compares two stages and returns whether the `stage` depends on `target` stage (i.e. `true`) or not (i.e. `false`).

Note	A stage <code>A</code> depends on stage <code>B</code> if <code>B</code> is among the ancestors of <code>A</code> .
------	---

Internally, `stageDependsOn` walks through the graph of RDDs of the input `stage`. For every RDD in the RDD's dependencies (using `RDD.dependencies`) `stageDependsOn` adds the RDD of a [NarrowDependency](#) to a stack of RDDs to visit while for a [ShuffleDependency](#) it finds [ShuffleMapStage](#) stages for a [ShuffleDependency](#) for the dependency and the `stage`'s first job id that it later adds to a stack of RDDs to visit if the map stage is ready, i.e. all the partitions have shuffle outputs.

After all the RDDs of the input `stage` are visited, `stageDependsOn` checks if the `target`'s RDD is among the RDDs of the `stage`, i.e. whether the `stage` depends on `target` `stage`.

`dag-scheduler-event-loop` — DAGScheduler Event Bus

`eventProcessLoop` is [DAGScheduler's event bus](#) to which Spark (by `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](#)

Caution	FIXME statistics? <code>MapOutputStatistics</code> ?
---------	--

Submitting Waiting Child Stages for Execution

— `submitWaitingChildStages` Internal Method

```
submitWaitingChildStages(parent: Stage): Unit
```

`submitWaitingChildStages` submits for execution all waiting stages for which the input parent `Stage` is the direct parent.

Note	Waiting stages are the stages registered in <code>waitingStages</code> internal registry.
------	--

When executed, you should see the following `TRACE` messages in the logs:

```
TRACE DAGScheduler: Checking if any dependencies of [parent] are now runnable
TRACE DAGScheduler: running: [runningStages]
TRACE DAGScheduler: waiting: [waitingStages]
TRACE DAGScheduler: failed: [failedStages]
```

`submitWaitingChildStages` finds child stages of the input `parent` stage, removes them from `waitingStages` internal registry, and `submits` one by one sorted by their job ids.

Note	<code>submitWaitingChildStages</code> is executed when <code>DAGScheduler</code> submits missing tasks for stage and handles successful <code>ShuffleMapTask</code> completion.
------	---

Submitting Stage or Its Missing Parents for Execution — `submitStage` Internal Method

```
submitStage(stage: Stage)
```

`submitStage` is an internal method that `DAGScheduler` uses to submit the input `stage` or its missing parents (if there any stages not computed yet before the input `stage` could).

Note	<code>submitStage</code> is also used to resubmit failed stages .
------	---

`submitStage` recursively submits any missing parents of the `stage`.

Internally, `submitStage` first finds the earliest-created job id that needs the `stage`.

Note	A stage itself tracks the jobs (their ids) it belongs to (using the internal <code>jobIds</code> registry).
------	---

The following steps depend on whether there is a job or not.

If there are no jobs that require the `stage`, `submitStage` [aborts it](#) with the reason:

```
No active job for stage [id]
```

If however there is a job for the `stage`, you should see the following DEBUG message in the logs:

```
DEBUG DAGScheduler: submitStage([stage])
```

`submitStage` checks the status of the `stage` and continues when it was not recorded in `waiting`, `running` or `failed` internal registries. It simply exits otherwise.

With the `stage` ready for submission, `submitStage` calculates the [list of missing parent stages of the stage](#) (sorted by their job ids). You should see the following DEBUG message in the logs:

```
DEBUG DAGScheduler: missing: [missing]
```

When the `stage` has no parent stages missing, you should see the following INFO message in the logs:

```
INFO DAGScheduler: Submitting [stage] ([stage.rdd]), which has no missing parents
```

`submitStage` submits the `stage` (with the earliest-created job id) and finishes.

If however there are missing parent stages for the `stage`, `submitStage` submits all the [parent stages](#), and the `stage` is recorded in the internal `waitingStages` registry.

Note

`submitStage` is executed when `DAGScheduler` submits [missing parent map stages \(of a stage\)](#) recursively or [waiting child stages](#), resubmits failed stages, and handles `JobSubmitted`, `MapStageSubmitted`, or `CompletionEvent` events.

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 the 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`.

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 / Adaptive Scheduling

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 a [ResubmitFailedStages](#) event when `FetchFailed` is reported.

They are created using `ThreadUtils.newDaemonSingleThreadScheduledExecutor` method that uses Guava DSL to instantiate a ThreadFactory.

Finding Missing Parent ShuffleMapStages For Stage — `getMissingParentStages` Internal Method

```
getMissingParentStages(stage: Stage): List[Stage]
```

`getMissingParentStages` finds missing parent [ShuffleMapStages](#) in the dependency graph of the input `stage` (using the [breadth-first search algorithm](#)).

Internally, `getMissingParentStages` starts with the `stage`'s RDD and walks up the tree of all parent RDDs to find [uncached partitions](#).

Note	A <code>stage</code> tracks the associated RDD using <code>rdd</code> property.
------	---

Note	An uncached partition of a RDD is a partition that has <code>Nil</code> in the internal registry of partition locations per RDD (which results in no RDD blocks in any of the active <code>BlockManager</code> s on executors).
------	--

`getMissingParentStages` traverses the [parent dependencies](#) of the RDD and acts according to their type, i.e. [ShuffleDependency](#) or [NarrowDependency](#).

Note

[ShuffleDependency](#) and [NarrowDependency](#) are the main top-level Dependencies.

For each `NarrowDependency`, `getMissingParentStages` simply marks the corresponding RDD to visit and moves on to a next dependency of a RDD or works on another unvisited parent RDD.

Note

[NarrowDependency](#) is a RDD dependency that allows for pipelined execution.

`getMissingParentStages` focuses on `ShuffleDependency` dependencies.

Note

[ShuffleDependency](#) is a RDD dependency that represents a dependency on the output of a [ShuffleMapStage](#), i.e. **shuffle map stage**.

For each `ShuffleDependency`, `getMissingParentStages` finds `ShuffleMapStage` stages. If the `ShuffleMapStage` is not *available*, it is added to the set of missing (map) stages.

Note

A `ShuffleMapStage` is **available** when all its partitions are computed, i.e. results are available (as blocks).

Caution

[FIXME](#)...IMAGE with ShuffleDependencies queried

Note

`getMissingParentStages` is used when `DAGScheduler` submits missing parent `shuffleMapStage`s (of a stage) and handles `JobSubmitted` and `MapStageSubmitted` events.

Submitting Missing Tasks of Stage (in a Spark Job) — `submitMissingTasks` Internal Method

```
submitMissingTasks(stage: Stage, jobId: Int): Unit
```

`submitMissingTasks` ...[FIXME](#)

Caution

[FIXME](#)

When executed, you should see the following DEBUG message in the logs:

```
DEBUG DAGScheduler: submitMissingTasks([stage])
```

The input `stage`'s `pendingPartitions` internal field is cleared (it is later filled out with the partitions to run tasks for).

`submitMissingTasks` requests the `stage` for **missing partitions**, i.e. the indices of the partitions to compute.

`submitMissingTasks` marks the `stage` as running (i.e. adds it to `runningStages` internal registry).

`submitMissingTasks` notifies `OutputCommitCoordinator` that the stage is started.

Note

The input `maxPartitionId` argument handed over to `OutputCommitCoordinator` depends on the type of the stage, i.e. `ShuffleMapStage` OR `ResultStage`. `ShuffleMapStage` tracks the number of partitions itself (as `numPartitions` property) while `ResultStage` uses the internal `RDD` to find out the number.

For the missing partitions, `submitMissingTasks` computes their **task locality preferences**, i.e. pairs of missing partition ids and **their task locality information**. HERE NOTE: The locality information of a RDD is called **preferred locations**.

In case of *non-fatal* exceptions at this time (while getting the locality information), `submitMissingTasks` creates a new stage attempt.

Note

A stage attempt is an internal property of a stage.

Despite the failure to submit any tasks, `submitMissingTasks` does announce that at least there was an attempt on `LiveListenerBus` by posting a `SparkListenerStageSubmitted` message.

Note

The Spark application's `LiveListenerBus` is given when `DAGScheduler` is created.

`submitMissingTasks` then **aborts the stage** (with the reason being "Task creation failed" followed by the exception).

The `stage` is removed from the internal `runningStages` collection of stages and `submitMissingTasks` exits.

When no exception was thrown (while computing the locality information for tasks), `submitMissingTasks` creates a new stage attempt and announces it on `LiveListenerBus` by posting a `SparkListenerStageSubmitted` message.

Note

Yes, that is correct. Whether there was a task submission failure or not, `submitMissingTasks` creates a new stage attempt and posts a `SparkListenerStageSubmitted`. That makes sense, doesn't it?

At that time, `submitMissingTasks` serializes the RDD (of the stage for which tasks are submitted for) and, depending on the type of the stage, the `shuffleDependency` (for `ShuffleMapStage`) or the `function` (for `ResultStage`).

Note

`submitMissingTasks` uses a closure `Serializer` that `DAGScheduler` creates for the entire lifetime when it is created. The closure serializer is available through `SparkEnv`.

The serialized so-called *task binary bytes* are "wrapped" as a broadcast variable (to make it available for executors to execute later on).

Note

That exact moment should make clear how important broadcast variables are for Spark itself that you, a Spark developer, can use, too, to distribute data across the nodes in a Spark application in a very efficient way.

Any `NotSerializableException` exceptions lead to [aborting the stage](#) (with the reason being "Task not serializable: [exception]") and removing the stage from the internal `runningStages` collection of stages. `submitMissingTasks` exits.

Any *non-fatal* exceptions lead to [aborting the stage](#) (with the reason being "Task serialization failed" followed by the exception) and removing the stage from the internal `runningStages` collection of stages. `submitMissingTasks` exits.

With no exceptions along the way, `submitMissingTasks` computes a collection of tasks to execute for the missing partitions (of the `stage`).

`submitMissingTasks` creates a `ShuffleMapTask` or `ResultTask` for every missing partition of the `stage` being `ShuffleMapStage` or `ResultStage`, respectively. `submitMissingTasks` uses the preferred locations (computed earlier) per partition.

Caution

[FIXME](#) Image with creating tasks for partitions in the stage.

Any *non-fatal* exceptions lead to [aborting the stage](#) (with the reason being "Task creation failed" followed by the exception) and removing the stage from the internal `runningStages` collection of stages. `submitMissingTasks` exits.

If there are tasks to submit for execution (i.e. there are missing partitions in the stage), you should see the following INFO message in the logs:

```
INFO DAGScheduler: Submitting [size] missing tasks from [stage] ([rdd])
```

`submitMissingTasks` records the partitions (of the tasks) in the `stage`'s `pendingPartitions` property.

Note

`pendingPartitions` property of the `stage` was cleared when `submitMissingTasks` started.

You should see the following DEBUG message in the logs:

```
DEBUG DAGScheduler: New pending partitions: [pendingPartitions]
```

`submitMissingTasks` submits the tasks to `TaskScheduler` for execution (with the id of the stage, attempt id, the input `jobId`, and the properties of the `ActiveJob` with `jobId`).

Note	A <code>TaskScheduler</code> was given when <code>DAGScheduler</code> was created.
------	--

Caution	<code>FIXME</code> What are the <code>ActiveJob</code> properties for? Where are they used?
---------	---

`submitMissingTasks` records the submission time in the stage's `StageInfo` and exits.

If however there are no tasks to submit for execution, `submitMissingTasks` marks the stage as finished (with no `errorMessage`).

You should see a DEBUG message that varies per the type of the input `stage` which are:

```
DEBUG DAGScheduler: Stage [stage] is actually done; (available: [isAvailable], available outputs: [numAvailableOutputs], partitions: [numPartitions])
```

or

```
DEBUG DAGScheduler: Stage [stage] is actually done; (partitions: [numPartitions])
```

for `ShuffleMapStage` and `ResultStage`, respectively.

In the end, with no tasks to submit for execution, `submitMissingTasks` submits waiting child stages for execution and exits.

Note	<code>submitMissingTasks</code> is called when <code>DAGScheduler</code> submits a stage for execution.
------	---

Computing Preferred Locations for Missing Partitions — `getPreferredLocs` Method

```
getPreferredLocs(rdd: RDD[_], partition: Int): Seq[TaskLocation]
```

`getPreferredLocs` is simply an alias for the internal (recursive) `getPreferredLocsInternal`.

Note	<code>getPreferredLocs</code> is used when <code>SparkContext</code> gets the locality information for a RDD partition and <code>DAGScheduler</code> submits missing tasks for a stage.
------	---

Finding BlockManagers (Executors) for Cached RDD Partitions (aka Block Location Discovery)

— `getCacheLocs` Internal Method

```
getCacheLocs(rdd: RDD[_]): IndexedSeq[Seq[TaskLocation]]
```

`getCacheLocs` gives [TaskLocations](#) (block locations) for the partitions of the input `rdd`.

`getCacheLocs` caches lookup results in [cacheLocs](#) internal registry.

Note	The size of the collection from <code>getCacheLocs</code> is exactly the number of partitions in <code>rdd</code> RDD.
------	--

Note	The size of every TaskLocation collection (i.e. every entry in the result of <code>getCacheLocs</code>) is exactly the number of blocks managed using BlockManagers on executors.
------	--

Internally, `getCacheLocs` finds `rdd` in the [cacheLocs](#) internal registry (of partition locations per RDD).

If `rdd` is not in [cacheLocs](#) internal registry, `getCacheLocs` branches per its [storage level](#).

For `NONE` storage level (i.e. no caching), the result is an empty locations (i.e. no location preference).

For other non-`NONE` storage levels, `getCacheLocs` requests [BlockManagerMaster](#) for block locations that are then mapped to [TaskLocations](#) with the hostname of the owning [BlockManager](#) for a block (of a partition) and the executor id.

Note	<code>getCacheLocs</code> uses BlockManagerMaster that was defined when DAGScheduler was created.
------	---

`getCacheLocs` records the computed block locations per partition (as [TaskLocation](#)) in [cacheLocs](#) internal registry.

Note	<code>getCacheLocs</code> requests locations from BlockManagerMaster using RDDBlockId with the RDD id and the partition indices (which implies that the order of the partitions matters to request proper blocks).
------	--

Note	DAGScheduler uses TaskLocations (with host and executor) while BlockManagerMaster uses BlockManagerId (to track similar information, i.e. block locations).
------	---

Note	<code>getCacheLocs</code> is used when DAGScheduler finds missing parent MapStages and getPreferredLocsInternal .
------	---

Finding Placement Preferences for RDD Partition (recursively) — `getPreferredLocsInternal` Internal Method

```
getPreferredLocsInternal(  
    rdd: RDD[_],  
    partition: Int,  
    visited: HashSet[(RDD[_], Int)]): Seq[TaskLocation]
```

`getPreferredLocsInternal` first finds the `TaskLocations` for the `partition` of the `rdd` (using `cacheLocs` internal cache) and returns them.

Otherwise, if not found, `getPreferredLocsInternal` requests `rdd` for the preferred locations of `partition` and returns them.

Note	Preferred locations of the partitions of a RDD are also called placement preferences or locality preferences .
------	--

Otherwise, if not found, `getPreferredLocsInternal` finds the first parent `NarrowDependency` and (recursively) finds `TaskLocations`.

If all the attempts fail to yield any non-empty result, `getPreferredLocsInternal` returns an empty collection of `TaskLocations`.

Note	<code>getPreferredLocsInternal</code> is used exclusively when <code>DAGScheduler</code> computes preferred locations for missing partitions.
------	---

Stopping DAGScheduler — `stop` Method

```
stop(): Unit
```

`stop` stops the internal `dag-scheduler-message` thread pool, `dag-scheduler-event-loop`, and `TaskScheduler`.

DAGSchedulerSource Metrics Source

`DAGScheduler` uses `Spark Metrics System` (via `DAGSchedulerSource`) to report metrics about internal status.

Caution	FIXME 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` Internal Method

```
updateAccumulators(event: CompletionEvent): Unit
```

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 handleTaskCompletion .
------	--

For each `AccumulableInfo` in the `CompletionEvent`, a partial value from a task is obtained (from `AccumulableInfo.update`) and added to the driver's accumulator (using `Accumulable.++=` method).

For named accumulators with the update value being a non-zero value, i.e. not

`Accumulable.zero`:

- `stage.latestInfo.accumulables` for the `AccumulableInfo.id` is set
- `CompletionEvent.taskInfo.accumulables` has a new `AccumulableInfo` added.

Caution	FIXME Where are <code>Stage.latestInfo.accumulables</code> and <code>CompletionEvent.taskInfo.accumulables</code> used?
---------	---

Settings

Table 3. Spark Properties

Spark Property	Default Value	Description
<code>spark.test.noStageRetry</code>	<code>false</code>	When enabled (i.e. <code>true</code>), task failures with FetchFailed exceptions will not cause stage retries, in order to surface the problem. Used for testing.

ActiveJob

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 (or for Adaptive Query Planning / Adaptive Scheduling).

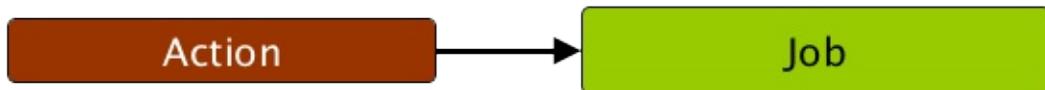


Figure 1. RDD actions submit jobs to DAGScheduler

Computing a job is equivalent to computing the partitions of the RDD the action has been executed upon. The number of partitions in a job depends on the type of a stage - [ResultStage](#) or [ShuffleMapStage](#).

A job starts with a single target RDD, but can ultimately include other RDDs that are all part of [the target RDD's lineage graph](#).

The parent stages are the instances of [ShuffleMapStage](#).

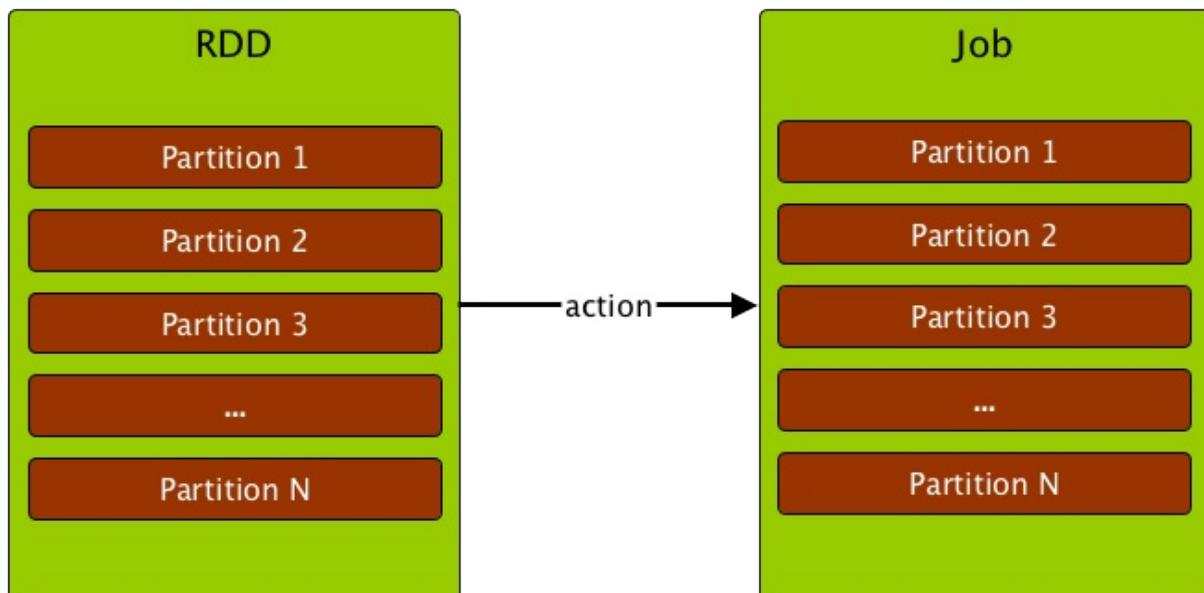


Figure 2. Computing a job is computing the partitions of an RDD

Note	Note that not all partitions have always to be computed for ResultStages for actions like <code>first()</code> and <code>lookup()</code> .
------	--

Internally, a job is represented by an instance of [private\[spark\]](#) class `org.apache.spark.scheduler.ActiveJob`.

Caution	<p>FIXME</p> <ul style="list-style-type: none">• Where are instances of ActiveJob used?
---------	--

A job can be one of two logical types (that are only distinguished by an internal `finalStage` field of `ActiveJob`):

- **Map-stage job** that computes the map output files for a [ShuffleMapStage](#) (for `submitMapStage`) before any downstream stages are submitted.

It is also used for [Adaptive Query Planning / Adaptive Scheduling](#), 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).

Stage — Physical Unit Of Execution

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 task per partition (of an RDD that computes partial results of a function executed as part of a Spark job).

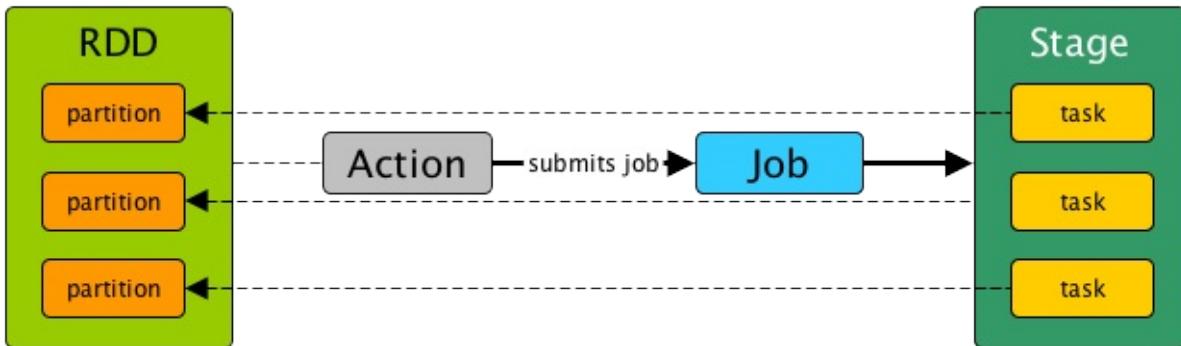


Figure 1. Stage, tasks and submitting a job

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](#)).

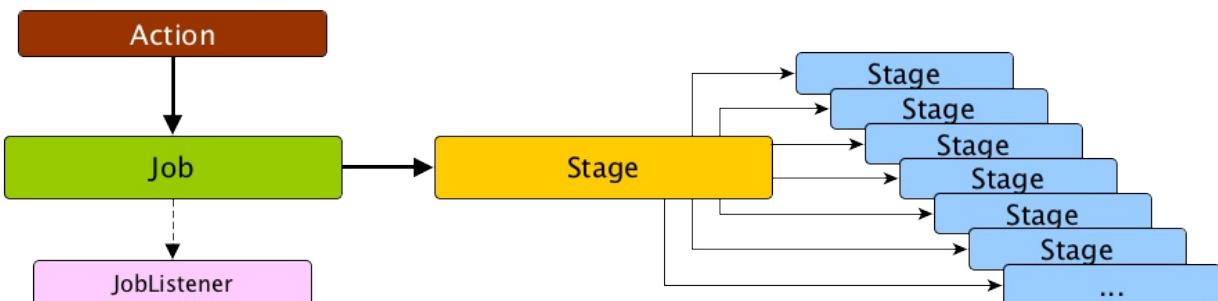


Figure 2. Submitting a job triggers execution of the stage and its parent stages

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 / Adaptive Scheduling](#).
- **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.

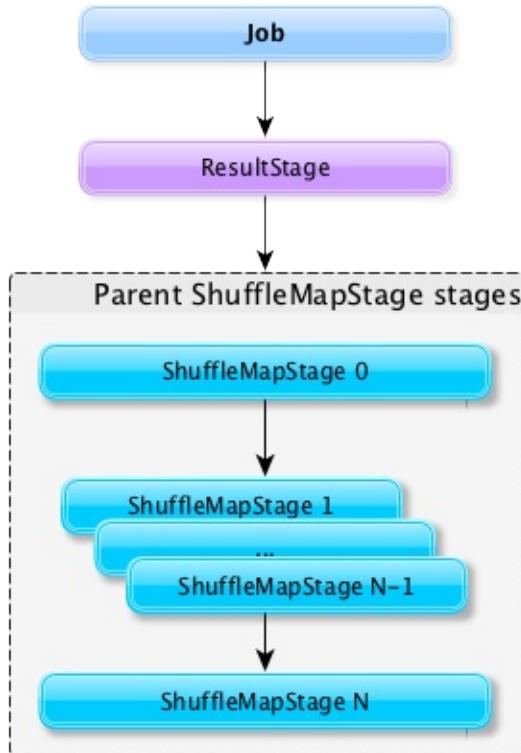


Figure 3. DAGScheduler and Stages for a job

A stage tracks the jobs (their ids) it belongs to (using the internal `jobIds` registry).

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.

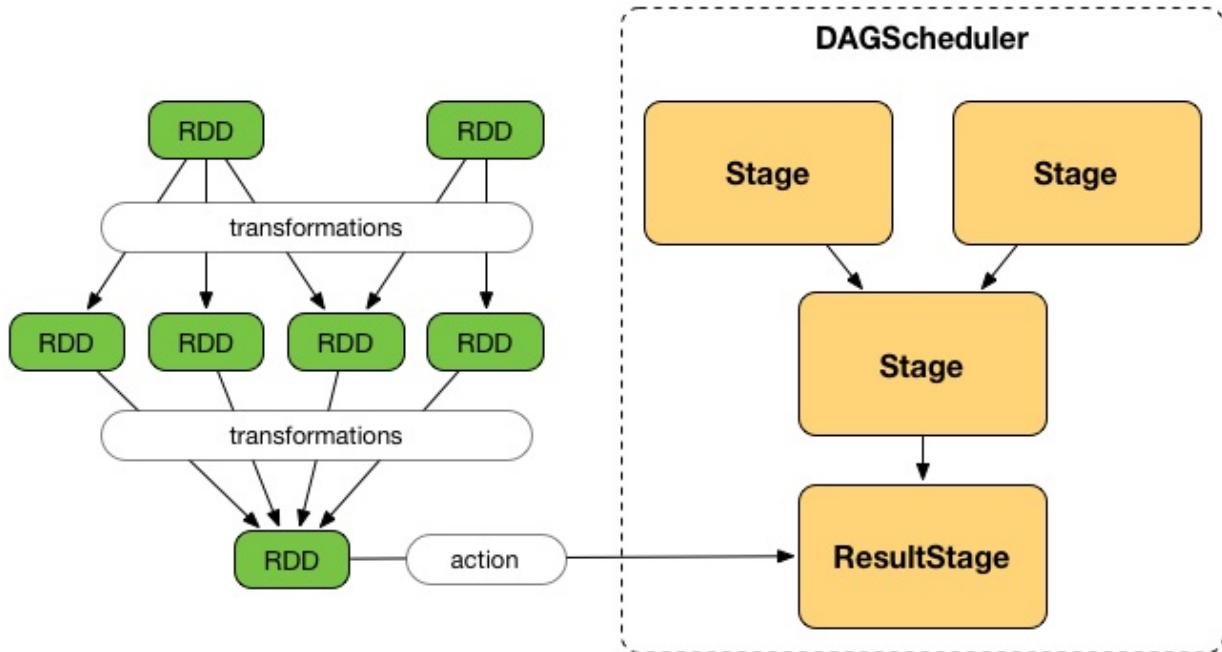


Figure 4. Graph of Stages

Shuffle boundaries introduce a barrier where stages/tasks must wait for the previous stage to finish before they fetch map outputs.

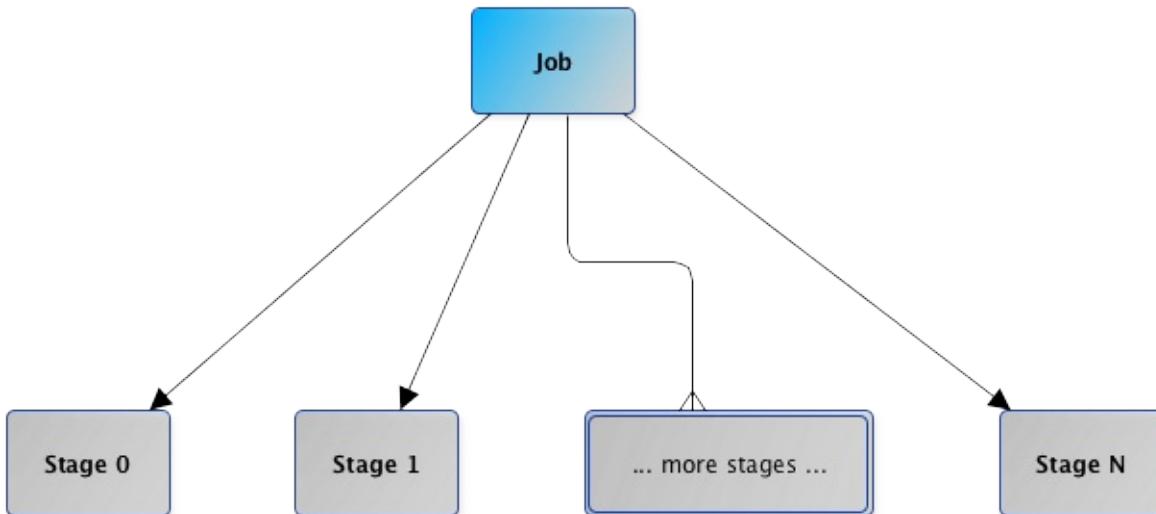


Figure 5. DAGScheduler splits a job into stages

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
```

Table 1. Stage's Internal Registries and Counters

Name	Description
details	Long description of the stage Used when... FIXME
fetchFailedAttemptIds	FIXME Used when... FIXME
jobIds	Set of jobs the stage belongs to. Used when... FIXME
name	Name of the stage Used when... FIXME
nextAttemptId	The ID for the next attempt of the stage. Used when... FIXME
numPartitions	Number of partitions Used when... FIXME
pendingPartitions	Set of pending partitions Used when... FIXME
_latestInfo	Internal cache with... FIXME Used when... FIXME

Stage Contract

```
abstract class Stage {
  def findMissingPartitions(): Seq[Int]
}
```

Note

Stage is a `private[scheduler]` abstract contract.

Table 2. Stage Contract

Method	Description
findMissingPartitions	Used when...

findMissingPartitions Method

`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).

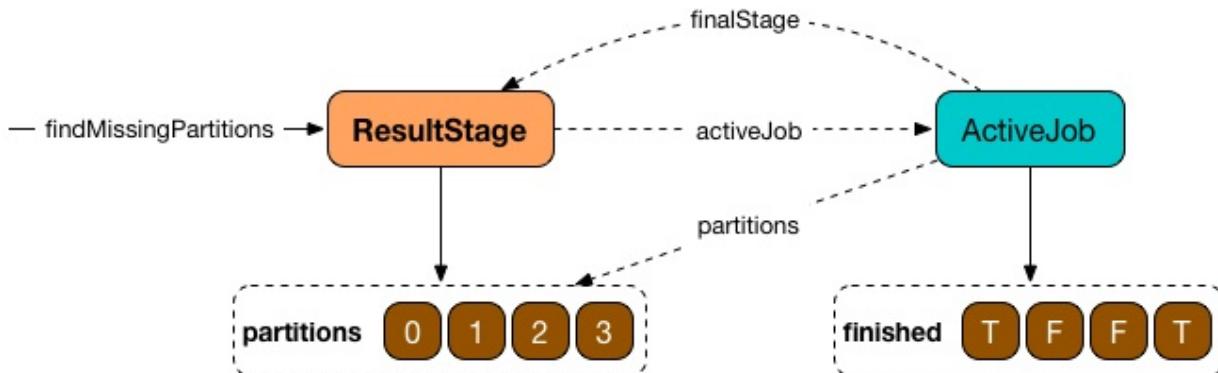


Figure 6. `ResultStage.findMissingPartitions` and `ActiveJob`

In the above figure, partitions 1 and 2 are not finished (`F` is false while `T` is true).

failedOnFetchAndShouldAbort Method

`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.
------	---

Getting StageInfo For Most Recent Attempt — latestInfo Method

```
latestInfo: StageInfo
```

`latestInfo` simply returns the [most recent](#) `StageInfo` (i.e. makes it accessible).

Creating New Stage Attempt (as StageInfo) — makeNewStageAttempt Method

```
makeNewStageAttempt(
  numPartitionsToCompute: Int,
  taskLocalityPreferences: Seq[Seq[TaskLocation]] = Seq.empty): Unit
```

`makeNewStageAttempt` creates a new `TaskMetrics` and registers the internal accumulators (using the RDD's `SparkContext`).

Note	<code>makeNewStageAttempt</code> uses <code>rdd</code> that was defined when <code>Stage</code> was created.
------	--

`makeNewStageAttempt` sets `_latestInfo` to be a `StageInfo` from the current stage (with `nextAttemptId`, `numPartitionsToCompute`, and `taskLocalityPreferences`).

`makeNewStageAttempt` increments `nextAttemptId` counter.

Note	<code>makeNewStageAttempt</code> is used exclusively when <code>DAGScheduler</code> submits missing tasks for a stage.
------	--

ShuffleMapStage — Intermediate Stage in Execution DAG

`ShuffleMapStage` (aka **shuffle map stage** or simply **map stage**) is an [intermediate stage](#) in the **physical execution DAG** that corresponds to a [ShuffleDependency](#).

Note

`ShuffleMapStage` can also be submitted independently as a Spark job for Adaptive Query Planning / Adaptive Scheduling.

Note

The **logical DAG** or **logical execution plan** is the [RDD lineage](#).

When executed, a `shuffleMapStage` saves **map output files** that can later be fetched by reduce tasks. When all map outputs are available, the `shuffleMapStage` is considered **available** (or **ready**).

Output locations can be missing, i.e. partitions have not been calculated or are lost.

`ShuffleMapStage` uses [outputLocs](#) and [_numAvailableOutputs](#) internal registries to track how many shuffle map outputs are available.

`ShuffleMapStage` is an input for the other following stages in the DAG of stages and is also called a **shuffle dependency's map side**.

A `shuffleMapStage` may contain multiple **pipelined operations**, e.g. `map` and `filter`, before shuffle operation.

A single `ShuffleMapStage` can be shared across different jobs.

Table 1. `ShuffleMapStage` Internal Registries and Counters

Name	Description
<code>_mapStageJobs</code>	<p><code>ActiveJobs</code> associated with the <code>ShuffleMapStage</code>.</p> <p>A new <code>ActiveJob</code> can be registered and deregistered.</p> <p>The list of <code>ActiveJobs</code> registered are available using <code>mapStageJobs</code>.</p>
<code>outputLocs</code>	<p>Tracks <code>MapStatuses</code> for each partition.</p> <p>There could be many <code>MapStatus</code> entries per partition due to Speculative Execution of Tasks.</p> <p>When <code>ShuffleMapStage</code> is created, <code>outputLocs</code> is empty, i.e. all elements are empty lists.</p> <p>The size of <code>outputLocs</code> is exactly the number of partitions of the RDD the stage runs on.</p>
<code>_numAvailableOutputs</code>	<p>The number of available outputs for the partitions of the <code>ShuffleMapStage</code>.</p> <p><code>_numAvailableOutputs</code> is incremented when the first MapStatus is registered for a partition (that could be more tasks per partition) and decrements when the last MapStatus is removed for a partition.</p> <p><code>_numAvailableOutputs</code> should not be greater than the number of partitions (and hence the number of <code>MapStatus</code> collections in <code>outputLocs</code> internal registry).</p>

Creating `ShuffleMapStage` Instance

`ShuffleMapStage` takes the following when created:

1. `id` identifier
2. `rdd` — the [RDD](#) of `ShuffleDependency`
3. `numTasks` — the number of tasks (that is exactly the [number of partitions in the `rdd`](#))
4. `parents` — the collection of parent [Stages](#)
5. `firstJobId` — the [ActiveJob](#) that created it
6. `callSite` — the creationSite of the [RDD](#)
7. `shuffleDep` — `ShuffleDependency` (from the [logical execution plan](#))

`ShuffleMapStage` initializes the [internal registries and counters](#).

Note	DAGScheduler tracks the number of ShuffleMapStage created so far.
------	---

Note	ShuffleMapStage is created only when DAGScheduler creates one for a ShuffleDependency .
------	---

Registering MapStatus For Partition — addOutputLoc Method

```
addOutputLoc(partition: Int, status: MapStatus): Unit
```

`addOutputLoc` adds the input `status` to the `output locations` for the input `partition`.

`addOutputLoc` increments `_numAvailableOutputs` internal counter if the input `MapStatus` is the first result for the `partition`.

Note	<code>addOutputLoc</code> is used when DAGScheduler creates a <code>ShuffleMapStage</code> for a <code>ShuffleDependency</code> and a <code>ActiveJob</code> (and <code>MapOutputTrackerMaster</code> tracks some output locations of the <code>ShuffleDependency</code>) and when <code>ShuffleMapTask</code> has finished.
------	---

Removing MapStatus For Partition And BlockManager — removeOutputLoc Method

```
removeOutputLoc(partition: Int, bmAddress: BlockManagerId): Unit
```

`removeOutputLoc` removes the `MapStatus` for the input `partition` and `bmAddress` `BlockManager` from the `output locations`.

`removeOutputLoc` decrements `_numAvailableOutputs` internal counter if the removed `MapStatus` was the last result for the `partition`.

Note	<code>removeOutputLoc</code> is exclusively used when a <code>Task</code> has failed with <code>FetchFailed</code> exception.
------	---

Finding Missing Partitions — findMissingPartitions Method

```
findMissingPartitions(): Seq[Int]
```

Note

`findMissingPartitions` is a part of `Stage contract` that returns the partitions that are missing, i.e. are yet to be computed.

Internally, `findMissingPartitions` uses `outputLocs` internal registry to find indices with empty lists of `MapStatus`.

ShuffleMapStage Sharing

A `shuffleMapStage` 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.

```
scala> val rdd = sc.parallelize(0 to 5).map(_,_).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

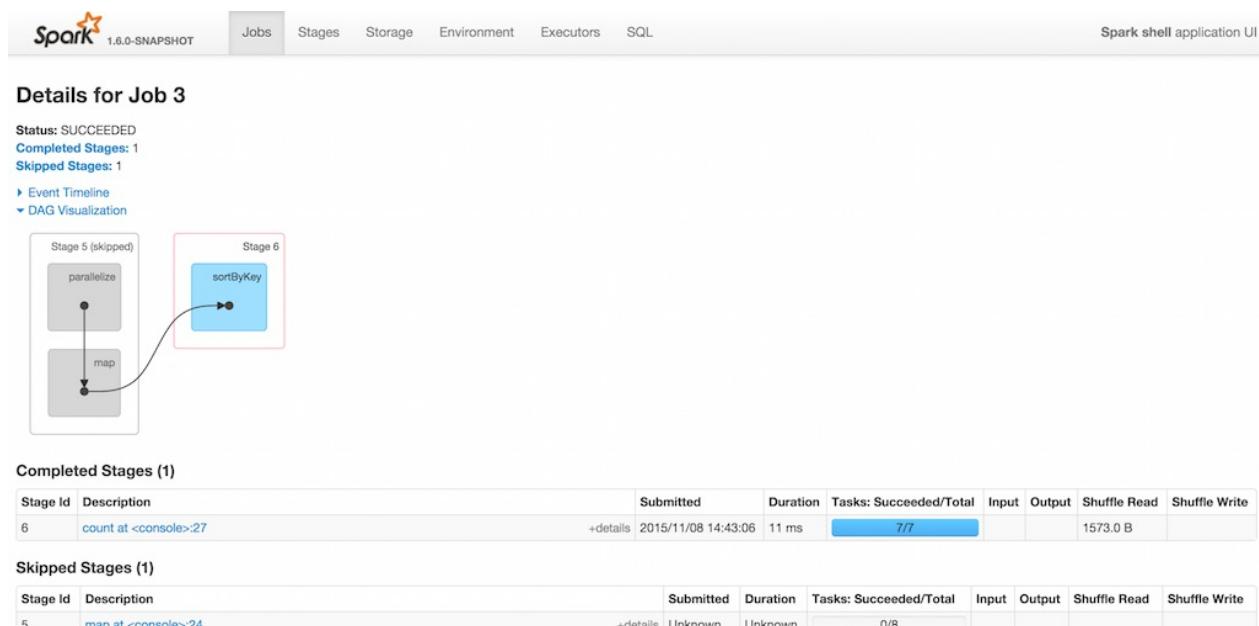


Figure 1. Skipped Stages are already-computed ShuffleMapStages

Returning Number of Available Shuffle Map Outputs — `numAvailableOutputs` Method

```
numAvailableOutputs: Int
```

`numAvailableOutputs` returns `_numAvailableOutputs` internal registry.

Note

`numAvailableOutputs` is used exclusively when `DAGScheduler` submits missing tasks for `ShuffleMapStage` (and only to print a DEBUG message when the `ShuffleMapStage` is finished).

Returning Collection of Active Jobs — `mapStageJobs` Method

```
mapStageJobs: Seq[ActiveJob]
```

`mapStageJobs` returns `_mapStageJobs` internal registry.

Note

`mapStageJobs` is used exclusively when `DAGScheduler` is notified that a `shuffleMapTask` has finished successfully (and the task made `ShuffleMapStage` completed and so marks any map-stage jobs waiting on this stage as finished).

Registering Job (that Computes ShuffleDependency) — `addActiveJob` Method

```
addActiveJob(job: ActiveJob): Unit
```

`addActiveJob` registers the input `ActiveJob` in `_mapStageJobs` internal registry.

Note

The `ActiveJob` is added as the first element in `_mapStageJobs`.

Note

`addActiveJob` is used exclusively when `DAGScheduler` is notified that a `ShuffleDependency` was submitted (and so a new `ActiveJob` is created to compute it).

Deregistering Job — `removeActiveJob` Method

```
removeActiveJob(job: ActiveJob): Unit
```

`removeActiveJob` removes a `ActiveJob` from `_mapStageJobs` internal registry.

Note

`removeActiveJob` is used exclusively when `DAGScheduler` cleans up after `ActiveJob` has finished (regardless of the outcome).

Removing All Shuffle Outputs Registered for Lost Executor — `removeOutputsOnExecutor` Method

```
removeOutputsOnExecutor(execId: String): Unit
```

`removeOutputsOnExecutor` removes all `MapStatuses` with the input `execId` executor from the `outputLocs` internal registry (of `MapStatuses` per partition).

If the input `execId` had the last registered `MapStatus` for a partition, `removeOutputsOnExecutor` decrements `_numAvailableOutputs` counter and you should see the following INFO message in the logs:

```
INFO [stage] is now unavailable on executor [execId] ([_numAvailableOutputs]/[numPartitions], [isAvailable])
```

Note	<code>removeOutputsOnExecutor</code> is used exclusively when <code>DAGScheduler</code> cleans up after a lost executor.
------	--

Preparing Shuffle Map Outputs in MapOutputTracker Format — `outputLocInMapOutputTrackerFormat` Method

```
outputLocInMapOutputTrackerFormat(): Array[MapStatus]
```

`outputLocInMapOutputTrackerFormat` returns the first (if available) element for every partition from `outputLocs` internal registry. If there is no entry for a partition, that position is filled with `null`.

Note	<code>outputLocInMapOutputTrackerFormat</code> is used when <code>DAGScheduler</code> is notified that a <code>ShuffleMapTask</code> has finished successfully (and the corresponding <code>ShuffleMapStage</code> is complete) and cleans up after a lost executor.
------	--

In both cases, `outputLocInMapOutputTrackerFormat` is used to register the shuffle map outputs (of the `ShuffleDependency`) with `MapOutputTrackerMaster`.

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.

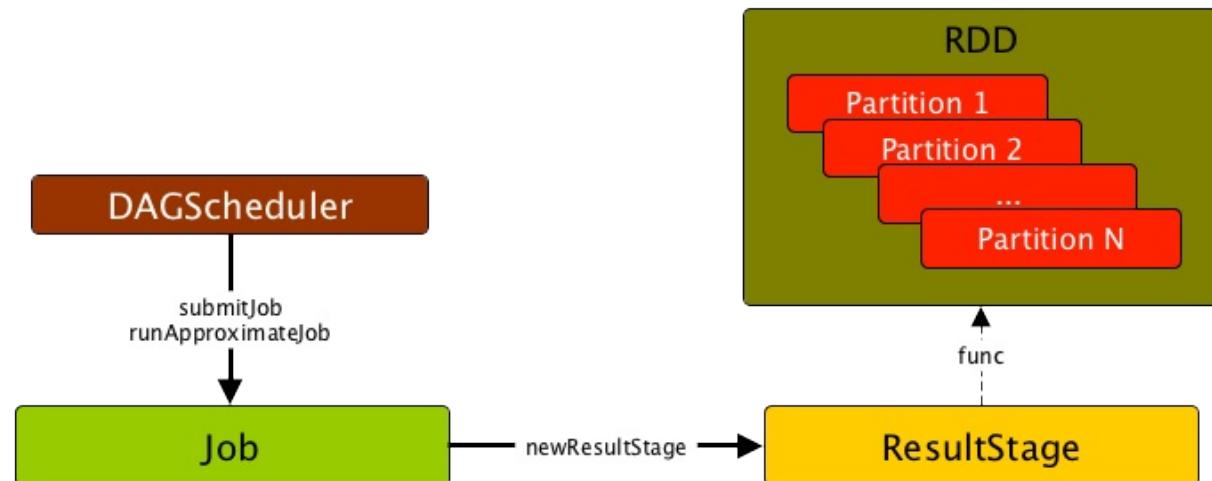


Figure 1. Job creates ResultStage as the first stage

The partitions are given as a collection of partition ids (`partitions`) and the function `func`:
`(TaskContext, Iterator[_]) ⇒ _`.

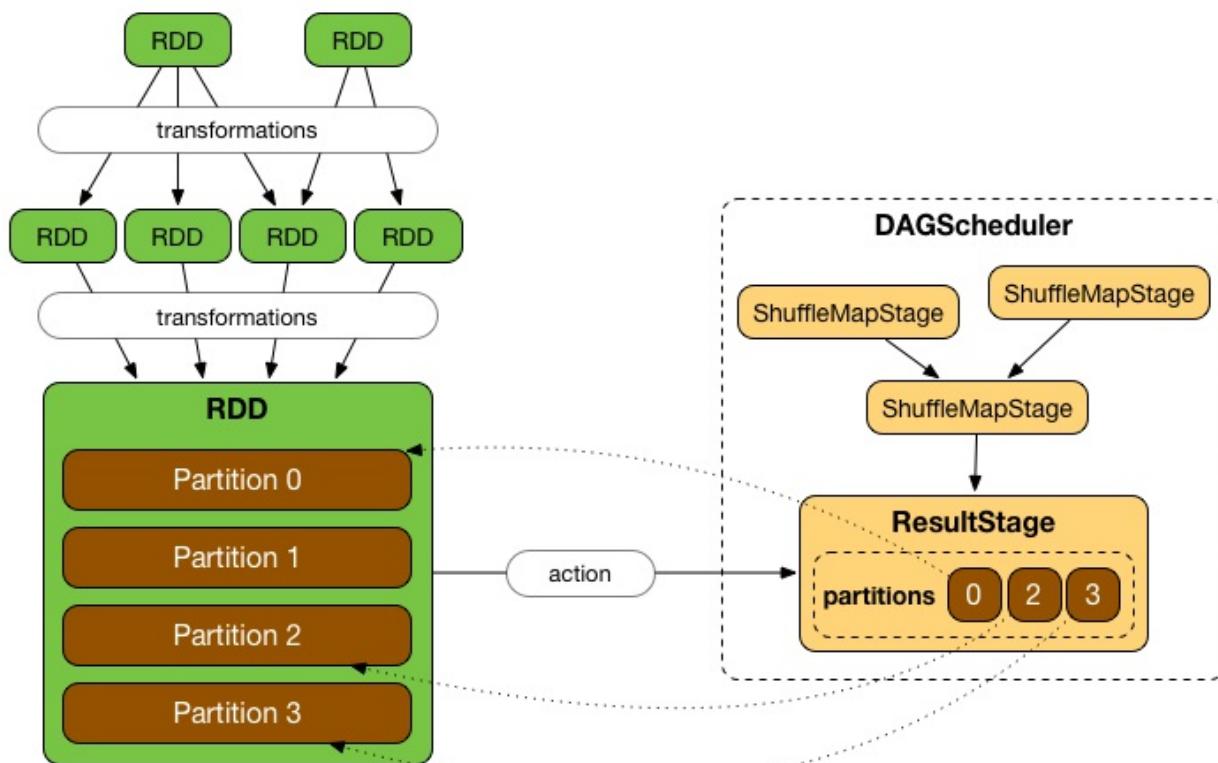


Figure 2. `ResultStage` and partitions

Tip

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

func Property

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

setActiveJob Method

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

removeActiveJob Method

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

activeJob Method

```
activeJob: Option[ActiveJob]
```

`activeJob` returns the optional `ActiveJob` associated with a `ResultStage`.

Caution	FIXME When/why would that be <code>NONE</code> (empty)?
---------	---

StageInfo

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

fromStage Method

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

DAGSchedulerEventProcessLoop — DAGScheduler Event Bus

`DAGSchedulerEventProcessLoop` (**dag-scheduler-event-loop**) is an `EventLoop` single "business logic" thread for processing `DAGSchedulerEvent` events.

Note	The purpose of the <code>DAGSchedulerEventProcessLoop</code> is to have a separate thread to process events asynchronously and serially, i.e. one by one, and let <code>DAGScheduler</code> do its work on the main thread.
------	---

Table 1. DAGSchedulerEvents and Event Handlers (in alphabetical order)

DAGSchedulerEvent	Event Handler	Trigger
AllJobsCancelled		<code>DAGScheduler</code> was requested to cancel all running or waiting jobs.
BeginEvent	handleBeginEvent	<code>TaskSetManager</code> informs <code>DAGScheduler</code> that a task is starting (through <code>taskStarted</code>).
CompletionEvent	handleTaskCompletion	<p>Posted to inform <code>DAGScheduler</code> that a task has completed (successfully or not).</p> <p><code>completionEvent</code> conveys the following information:</p> <ol style="list-style-type: none"> 1. Completed <code>Task</code> instance (as <code>task</code>) 2. <code>TaskEndReason</code> (as <code>reason</code>) 3. Result of the task (as <code>result</code>) 4. <code>Accumulator</code> updates 5. <code>TaskInfo</code>
ExecutorAdded	handleExecutorAdded	<code>DAGScheduler</code> was informed (through <code>executorAdded</code>) that an executor was spun up on a host.
		Posted to notify <code>DAGScheduler</code> that an executor was lost.

<code>ExecutorLost</code>	<code>handleExecutorLost</code>	<p><code>ExecutorLost</code> conveys the following information:</p> <ol style="list-style-type: none"> 1. <code>execId</code> 2. <code>ExecutorLossReason</code> <p>NOTE: The input <code>filesLost</code> for <code>handleExecutorLost</code> is enabled when <code>ExecutorLossReason</code> is <code>SlaveLost</code> with <code>workerLost</code> enabled (it is disabled by default).</p> <p>NOTE: <code>handleExecutorLost</code> is also called when <code>DAGScheduler</code> is informed that a task has failed due to <code>FetchFailed</code> exception.</p>
<code>GettingResultEvent</code>		<code>TaskSetManager</code> informs <code>DAGScheduler</code> (through <code>taskGettingResult</code>) that a task has completed and results are being fetched remotely.
<code>JobCancelled</code>	<code>handleJobCancellation</code>	<code>DAGScheduler</code> was requested to cancel a job.
<code>JobGroupCancelled</code>	<code>handleJobGroupCancelled</code>	<code>DAGScheduler</code> was requested to cancel a job group.
<code>JobSubmitted</code>	<code>handleJobSubmitted</code>	<p>Posted when <code>DAGScheduler</code> is requested to submit a job or run an approximate job.</p> <p><code>JobSubmitted</code> conveys the following information:</p> <ol style="list-style-type: none"> 1. A job identifier (as <code>jobId</code>) 2. A <code>RDD</code> (as <code>finalRDD</code>) 3. The function to execute (as <code>func: (TaskContext, Iterator[_]) => _</code>) 4. The partitions to compute (as <code>partitions</code>) 5. A <code>callsite</code> (as <code>callSite</code>) 6. The <code>JobListener</code> to inform about the status of the stage.

		7. Properties of the execution
MapStageSubmitted	handleMapStageSubmitted	<p>Posted to inform DAGScheduler that SparkContext submitted a MapStage for execution (through <code>submitMapStage</code>).</p> <p><code>MapStageSubmitted</code> conveys the following information:</p> <ol style="list-style-type: none"> 1. A job identifier (as <code>jobId</code>) 2. The <code>ShuffleDependency</code> 3. A callsite (as <code>callSite</code>) 4. The <code>JobListener</code> to inform about the status of the stage. 5. Properties of the execution
ResubmitFailedStages	resubmitFailedStages	DAGScheduler was informed that a task has failed due to <code>FetchFailed</code> exception.
StageCancelled	handleStageCancellation	DAGScheduler was requested to cancel a stage.
TaskSetFailed	handleTaskSetFailed	DAGScheduler was requested to cancel a TaskSet

When created, `DAGSchedulerEventProcessLoop` gets the reference to the owning `DAGScheduler` that it uses to call event handler methods on.

Note	DAGSchedulerEventProcessLoop uses <code>java.util.concurrent.LinkedBlockingDeque</code> blocking deque that grows indefinitely, i.e. up to <code>Integer.MAX_VALUE</code> events.
------	---

AllJobsCancelled Event and...

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

GettingResultEvent Event and handleGetTaskResult Handler

```
GettingResultEvent(taskInfo: TaskInfo) extends DAGSchedulerEvent
```

`GettingResultEvent` is a `DAGSchedulerEvent` that triggers `handleGetTaskResult` (on a separate thread).

Note

`GettingResultEvent` is posted to inform `DAGScheduler` (through `taskGettingResult`) that a task fetches results.

handleGetTaskResult Handler

```
handleGetTaskResult(taskInfo: TaskInfo): Unit
```

`handleGetTaskResult` merely posts `SparkListenerTaskGettingResult` (to `LiveListenerBus` Event Bus).

BeginEvent Event and handleBeginEvent Handler

```
BeginEvent(task: Task[_], taskInfo: TaskInfo) extends DAGSchedulerEvent
```

`BeginEvent` is a `DAGSchedulerEvent` that triggers `handleBeginEvent` (on a separate thread).

Note

`BeginEvent` is posted to inform `DAGScheduler` (through `taskStarted`) that a `TaskSetManager` starts a task.

handleBeginEvent Handler

```
handleBeginEvent(task: Task[_], taskInfo: TaskInfo): Unit
```

`handleBeginEvent` looks the stage of `task` up in `stageIdToStage` internal registry to compute the last attempt id (or `-1` if not available) and posts `SparkListenerTaskStart` (to `listenerBus` event bus).

JobGroupCancelled Event and handleJobGroupCancelled Handler

```
JobGroupCancelled(groupId: String) extends DAGSchedulerEvent
```

`JobGroupCancelled` is a `DAGSchedulerEvent` that triggers `handleJobGroupCancelled` (on a separate thread).

Note

`JobGroupCancelled` is posted when `DAGScheduler` is informed (through `cancelJobGroup`) that `SparkContext` was requested to cancel a job group.

handleJobGroupCancelled Handler

```
handleJobGroupCancelled(groupId: String): Unit
```

`handleJobGroupCancelled` finds active jobs in a group and cancels them.

Internally, `handleJobGroupCancelled` computes all the active jobs (registered in the internal collection of active jobs) that have `spark.jobGroup.id` scheduling property set to `groupId`.

`handleJobGroupCancelled` then cancels every active job in the group one by one and the cancellation reason: "part of cancelled job group [groupId]."

Getting Notified that ShuffleDependency Was Submitted — handleMapStageSubmitted Handler

```
handleMapStageSubmitted(
    jobId: Int,
    dependency: ShuffleDependency[_, _, _],
    callSite: CallSite,
    listener: JobListener,
    properties: Properties): Unit
```

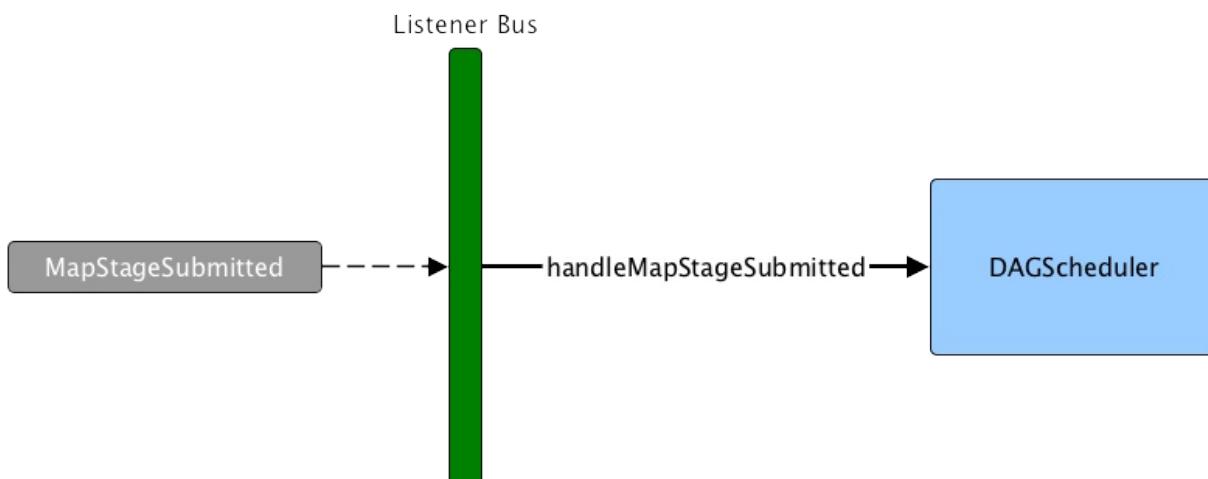


Figure 1. `MapStageSubmitted` Event Handling

`handleMapStageSubmitted` finds or creates a new `ShuffleMapStage` for the input `ShuffleDependency` and `jobId`.

`handleMapStageSubmitted` creates an [ActiveJob](#) (with the input `jobId`, `callSite`, `listener` and `properties`, and the `ShuffleMapStage`).

`handleMapStageSubmitted` [clears the internal cache of RDD partition locations](#).

Caution	FIXME Why is this clearing here so important?
---------	---

You should see the following INFO messages in the logs:

```
INFO DAGScheduler: Got map stage job [id] ([callSite]) with [number] output partitions
INFO DAGScheduler: Final stage: [stage] ([name])
INFO DAGScheduler: Parents of final stage: [parents]
INFO DAGScheduler: Missing parents: [missingStages]
```

`handleMapStageSubmitted` registers the new job in `jobIdToActiveJob` and `activeJobs` internal registries, and [with the final `ShuffleMapStage`](#).

Note	<code>ShuffleMapStage</code> can have multiple <code>ActiveJob</code> s registered.
------	---

`handleMapStageSubmitted` finds all the registered stages for the input `jobId` and collects their [latest `StageInfo`](#).

Ultimately, `handleMapStageSubmitted` posts [SparkListenerJobStart](#) message to [LiveListenerBus](#) and submits the `ShuffleMapStage`.

In case the `ShuffleMapStage` could be available already, `handleMapStageSubmitted` marks the job finished.

Note	<code>DAGScheduler</code> requests <code>MapOutputTrackerMaster</code> for statistics for <code>ShuffleDependency</code> that it uses for <code>handleMapStageSubmitted</code> .
------	--

Note	<code>MapOutputTrackerMaster</code> is passed in when <code>DAGScheduler</code> is created.
------	---

When `handleMapStageSubmitted` could not find or create a `ShuffleMapStage`, you should see the following WARN message in the logs.

```
WARN Creating new stage failed due to exception - job: [id]
```

`handleMapStageSubmitted` notifies `listener` about the job failure and exits.

Note	<code>MapStageSubmitted</code> event processing is very similar to JobSubmitted events.
------	---

The difference between `handleMapStageSubmitted` and `handleJobSubmitted`:

- `handleMapStageSubmitted` has a `ShuffleDependency` among the input parameters while `handleJobSubmitted` has `finalRDD`, `func`, and `partitions`.
 - `handleMapStageSubmitted` initializes `finalStage` as
`getShuffleMapStage(dependency, jobId)` while `handleJobSubmitted` as `finalStage = newResultStage(finalRDD, func, partitions, jobId, callSite)`
 - `handleMapStageSubmitted` INFO logs Got map stage job %s (%s) with %d output partitions with `dependency.rdd.partitions.length` while `handleJobSubmitted` does Got job %s (%s) with %d output partitions with `partitions.length`.
- Tip**
- **FIXME:** Could the above be cut to `ActiveJob.numPartitions`?
 - `handleMapStageSubmitted` adds a new job with `finalStage.addActiveJob(job)` while `handleJobSubmitted` sets with `finalStage.setActiveJob(job)`.
 - `handleMapStageSubmitted` checks if the final stage has already finished, tells the listener and removes it using the code:

```
if (finalStage.isAvailable) {
    markMapStageJobAsFinished(job, mapOutputTracker.getStatistics(dependency))
}
```

TaskSetFailed Event and handleTaskSetFailed Handler

```
TaskSetFailed(
  taskSet: TaskSet,
  reason: String,
  exception: Option[Throwable])
extends DAGSchedulerEvent
```

`TaskSetFailed` is a `DAGSchedulerEvent` that triggers `handleTaskSetFailed` method.

Note

`TaskSetFailed` is posted when `DAGScheduler` is requested to cancel a `TaskSet`.

handleTaskSetFailed Handler

```
handleTaskSetFailed(
  taskSet: TaskSet,
  reason: String,
  exception: Option[Throwable]): Unit
```

`handleTaskSetFailed` looks the stage (of the input `taskSet`) up in the internal `stageIdToStage` registry and [aborts](#) it.

ResubmitFailedStages Event and resubmitFailedStages Handler

`ResubmitFailedStages` [extends](#) `DAGSchedulerEvent`

`ResubmitFailedStages` [is a](#) `DAGSchedulerEvent` that triggers [resubmitFailedStages](#) method.

Note

`ResubmitFailedStages` [is posted](#) for `FetchFailed` case in `handleTaskCompletion`.

resubmitFailedStages Handler

`resubmitFailedStages(): Unit`

`resubmitFailedStages` [iterates](#) over the internal [collection of failed stages](#) and submits them.

Note

`resubmitFailedStages` [does nothing](#) when there are no [failed stages reported](#).

You should see the following INFO message in the logs:

INFO Resubmitting failed stages

`resubmitFailedStages` [clears](#) the internal cache of RDD partition locations first. It then makes a copy of the [collection of failed stages](#) so `DAGScheduler` can track failed stages afresh.

Note

At this point `DAGScheduler` has no failed stages reported.

The previously-reported failed stages are sorted by the corresponding job ids in incremental order and [resubmitted](#).

Getting Notified that Executor Is Lost — handleExecutorLost Handler

```
handleExecutorLost(
  execId: String,
  filesLost: Boolean,
  maybeEpoch: Option[Long] = None): Unit
```

`handleExecutorLost` checks whether the input optional `maybeEpoch` is defined and if not requests the current epoch from `MapOutputTrackerMaster`.

Note

`MapOutputTrackerMaster` is passed in (as `mapOutputTracker`) when `DAGScheduler` is created.

Caution

FIXME When is `maybeEpoch` passed in?

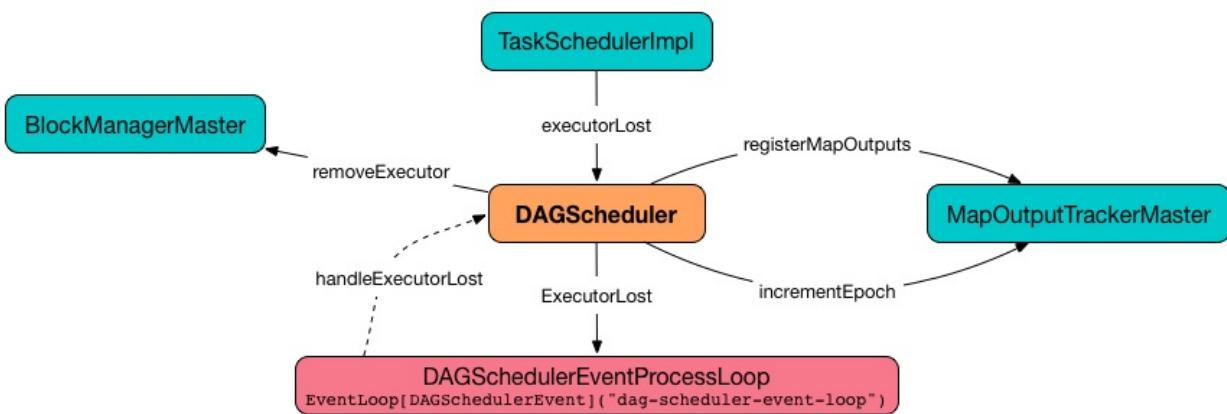


Figure 2. `DAGScheduler.handleExecutorLost`

Recurring `ExecutorLost` events lead to the following repeating DEBUG message in the logs:

```
DEBUG Additional executor lost message for [execId] (epoch [currentEpoch])
```

Note

`handleExecutorLost` handler uses `DAGScheduler`'s `failedEpoch` and **FIXME** internal registries.

Otherwise, when the executor `execId` is not in the [list of executor lost](#) or the executor failure's epoch is smaller than the input `maybeEpoch`, the executor's lost event is recorded in [failedEpoch](#) internal registry.

Caution

FIXME Describe the case above in simpler non-technical words. Perhaps change the order, too.

You should see the following INFO message in the logs:

```
INFO Executor lost: [execId] (epoch [epoch])
```

`BlockManagerMaster` is requested to remove the lost executor `execId`.

Caution

FIXME Review what's `filesLost`.

`handleExecutorLost` exits unless the `ExecutorLost` event was for a map output fetch operation (and the input `filesLost` is `true`) or `external shuffle service` is *not* used.

In such a case, you should see the following INFO message in the logs:

```
INFO Shuffle files lost for executor: [execId] (epoch [epoch])
```

`handleExecutorLost` walks over all `ShuffleMapStages` in `DAGScheduler`'s `shuffleToMapStage` internal registry and do the following (in order):

1. `ShuffleMapStage.removeOutputsOnExecutor(execId)` is called
2. `MapOutputTrackerMaster.registerMapOutputs(shuffleId, stage.outputLocInMapOutputTrackerFormat(), changeEpoch = true)` is called.

In case `DAGScheduler`'s `shuffleToMapStage` internal registry has no shuffles registered, `MapOutputTrackerMaster` is requested to increment epoch.

Ultimately, `DAGScheduler` clears the internal cache of RDD partition locations.

JobCancelled Event and handleJobCancellation Handler

```
JobCancelled(jobId: Int) extends DAGSchedulerEvent
```

`JobCancelled` is a `DAGSchedulerEvent` that triggers `handleJobCancellation` method (on a separate thread).

Note

`JobCancelled` is posted when `DAGScheduler` is requested to cancel a job.

handleJobCancellation Handler

```
handleJobCancellation(jobId: Int, reason: String = "")
```

`handleJobCancellation` first makes sure that the input `jobId` has been registered earlier (using `jobIdToStageIds` internal registry).

If the input `jobId` is not known to `DAGScheduler`, you should see the following DEBUG message in the logs:

```
DEBUG DAGScheduler: Trying to cancel unregistered job [jobId]
```

Otherwise, `handleJobCancellation` fails the active job and all independent stages (by looking up the active job using `jobIdToActiveJob`) with failure reason:

```
Job [jobId] cancelled [reason]
```

Getting Notified That Task Has Finished — `handleTaskCompletion` Handler

```
handleTaskCompletion(event: CompletionEvent): Unit
```

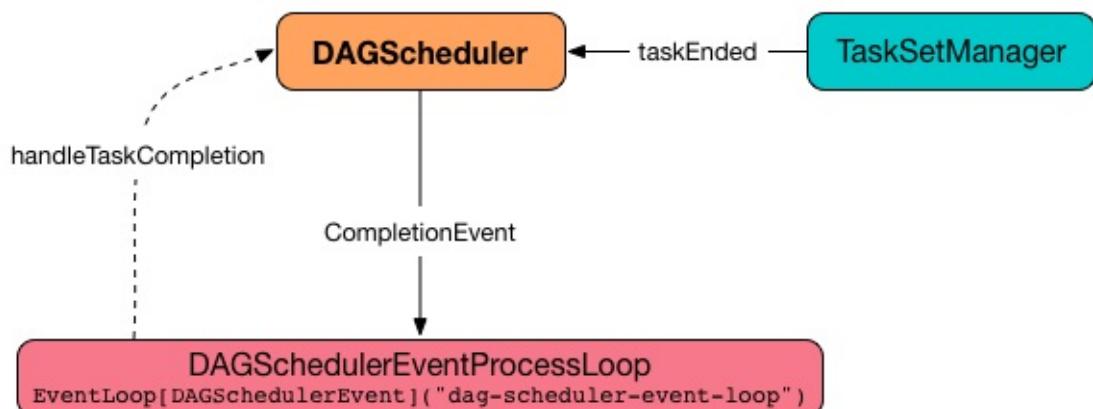


Figure 3. DAGScheduler and CompletionEvent

Note

`CompletionEvent` holds contextual information about the completed task.

Table 2. `CompletionEvent` Properties

Property	Description
<code>task</code>	Completed <code>Task</code> instance for a stage, partition and stage attempt.
<code>reason</code>	<code>TaskEndReason</code> ... FIXME
<code>result</code>	Result of the task
<code>accumUpdates</code>	<code>Accumulators</code> with... FIXME
<code>taskInfo</code>	<code>TaskInfo</code>

`handleTaskCompletion` starts by notifying `outputCommitCoordinator` that a task completed.

`handleTaskCompletion` re-creates `TaskMetrics` (using `accumUpdates` accumulators of the input `event`).

Note	<code>TaskMetrics</code> can be empty when the task has failed.
------	---

`handleTaskCompletion` announces task completion application-wide (by posting a `SparkListenerTaskEnd` to `LiveListenerBus`).

`handleTaskCompletion` checks the stage of the task out in the `stageIdToStage` internal registry and if not found, it simply exits.

`handleTaskCompletion` branches off per `TaskEndReason` (as `event.reason`).

Table 3. `handleTaskCompletion` Branches per `TaskEndReason`

TaskEndReason	Description
<code>Success</code>	Acts according to the type of the task that completed, i.e. <code>ShuffleMapTask</code> and <code>ResultTask</code> .
<code>Resubmitted</code>	
<code>FetchFailed</code>	
<code>ExceptionFailure</code>	Updates accumulators (with partial values from the task).
<code>ExecutorLostFailure</code>	Does nothing
<code>TaskCommitDenied</code>	Does nothing
<code>TaskKilled</code>	Does nothing
<code>TaskResultLost</code>	Does nothing
<code>UnknownReason</code>	Does nothing

Handling Successful Task Completion

When a task has finished successfully (i.e. `success` end reason), `handleTaskCompletion` marks the partition as no longer pending (i.e. the partition the task worked on is removed from `pendingPartitions` of the stage).

Note	A Stage tracks its own pending partitions using <code>pendingPartitions</code> property.
------	--

`handleTaskCompletion` branches off given the type of the task that completed, i.e. `ShuffleMapTask` and `ResultTask`.

Handling Successful `ResultTask` Completion

For `ResultTask`, the stage is assumed a `ResultStage`.

`handleTaskCompletion` finds the `ActiveJob` associated with the `ResultStage`.

Note	<code>ResultStage</code> tracks the optional <code>ActiveJob</code> as <code>activeJob</code> property. There could only be one active job for a <code>ResultStage</code> .
------	---

If there is *no* job for the `ResultStage`, you should see the following INFO message in the logs:

```
INFO DAGScheduler: Ignoring result from [task] because its job has finished
```

Otherwise, when the `ResultStage` has a `ActiveJob`, `handleTaskCompletion` checks the status of the partition output for the partition the `ResultTask` ran for.

Note	<code>ActiveJob</code> tracks task completions in <code>finished</code> property with flags for every partition in a stage. When the flag for a partition is enabled (i.e. <code>true</code>), it is assumed that the partition has been computed (and no results from any <code>ResultTask</code> are expected and hence simply ignored).
------	---

Caution	<code>FIXME</code> Describe why could a partition have more <code>ResultTask</code> running.
---------	--

`handleTaskCompletion` ignores the `CompletionEvent` when the partition has already been marked as completed for the stage and simply exits.

`handleTaskCompletion` updates accumulators.

The partition for the `ActiveJob` (of the `ResultStage`) is marked as computed and the number of partitions calculated increased.

Note	<code>ActiveJob</code> tracks what partitions have already been computed and their number.
------	--

If the `ActiveJob` has finished (when the number of partitions computed is exactly the number of partitions in a stage) `handleTaskCompletion` does the following (in order):

1. Marks `ResultStage` computed.
2. Cleans up after `ActiveJob` and independent stages.
3. Announces the job completion application-wide (by posting a `SparkListenerJobEnd` to `LiveListenerBus`).

In the end, `handleTaskCompletion` notifies `JobListener` of the `ActiveJob` that the task succeeded.

Note	A task succeeded notification holds the output index and the result.
------	--

When the notification throws an exception (because it runs user code),

`handleTaskCompletion` notifies `JobListener` about the failure (wrapping it inside a `SparkDriverExecutionException` exception).

Handling Successful `shuffleMapTask` Completion

For `ShuffleMapTask`, the stage is assumed a `ShuffleMapStage`.

`handleTaskCompletion` updates accumulators.

The task's result is assumed `MapStatus` that knows the executor where the task has finished.

You should see the following DEBUG message in the logs:

```
DEBUG DAGScheduler: ShuffleMapTask finished on [execId]
```

If the executor is registered in `failedEpoch` internal registry and the epoch of the completed task is not greater than that of the executor (as in `failedEpoch` registry), you should see the following INFO message in the logs:

```
INFO DAGScheduler: Ignoring possibly bogus [task] completion from executor [executorId]
```

Otherwise, `handleTaskCompletion` registers the `MapStatus` result for the partition with the stage (of the completed task).

`handleTaskCompletion` does more processing only if the `shuffleMapStage` is registered as still running (in `runningStages` internal registry) and the `shuffleMapStage` stage has no pending partitions to compute.

The `ShuffleMapStage` is marked as finished.

You should see the following INFO messages in the logs:

```
INFO DAGScheduler: looking for newly runnable stages
INFO DAGScheduler: running: [runningStages]
INFO DAGScheduler: waiting: [waitingStages]
INFO DAGScheduler: failed: [failedStages]
```

`handleTaskCompletion` registers the shuffle map outputs of the `shuffleDependency` with `MapOutputTrackerMaster` (with the epoch incremented) and clears internal cache of the stage's RDD block locations.

Note	MapOutputTrackerMaster is given when DAGScheduler is created.
------	---

If the `ShuffleMapStage` stage is ready, all active jobs of the stage (aka *map-stage jobs*) are marked as finished (with `MapOutputStatistics` from `MapOutputTrackerMaster` for the `ShuffleDependency`).

Note	A <code>ShuffleMapStage</code> stage is ready (aka <i>available</i>) when all partitions have shuffle outputs, i.e. when their tasks have completed.
------	---

Eventually, `handleTaskCompletion` submits waiting child stages (of the ready `ShuffleMapStage`).

If however the `ShuffleMapStage` is *not* ready, you should see the following INFO message in the logs:

```
INFO DAGScheduler: Resubmitting [shuffleStage] ([shuffleStage.name]) because some of its tasks had failed: [missingPartitions]
```

In the end, `handleTaskCompletion` submits the `ShuffleMapStage` for execution.

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.
-----	--

Task Failed with FetchFailed Exception — TaskEndReason: FetchFailed

```
FetchFailed(
  bmAddress: BlockManagerId,
  shuffleId: Int,
  mapId: Int,
  reduceId: Int,
  message: String)
extends TaskFailedReason
```

Table 4. `FetchFailed` Properties

Name	Description
<code>bmAddress</code>	<code>BlockManagerId</code>
<code>shuffleId</code>	Used when...
<code>mapId</code>	Used when...
<code>reduceId</code>	Used when...
<code>failureMessage</code>	Used when...

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 <code>failedStage.latestInfo.attemptId != task.stageAttemptId</code> 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 <code>markStageAsFinished</code> 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	FIXME Describe <code>disallowStageRetryForTest</code> and <code>abortStage</code> .
---------	---

If the number of fetch failed attempts for the stage exceeds the allowed number, the failed stage is aborted with the reason:

```
[failedStage] ([name]) has failed the maximum allowable number of times: 4. Most recent failure reason: [failureMessage]
```

If there are no failed stages reported (`DAGScheduler.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	FIXME What does the above code do?
---------	--

For all the cases, the failed stage and map stages are both added to the internal [registry of failed stages](#).

If `mapId` (in the `FetchFailed` object for the case) is provided, the map stage output is cleaned up (as it is broken) using `mapStage.removeOutputLoc(mapId, bmAddress)` and `MapOutputTrackerMaster.unregisterMapOutput(shuffleId, mapId, bmAddress)` methods.

Caution	FIXME What does <code>mapStage.removeOutputLoc</code> do?
---------	---

If `BlockManagerId` (as `bmAddress` in the `FetchFailed` object) is defined, `handleTaskCompletion` notifies `DAGScheduler` that an executor was lost (with `filesLost` enabled and `maybeEpoch` from the `Task` that completed).

StageCancelled Event and handleStageCancellation Handler

```
StageCancelled(stageId: Int) extends DAGSchedulerEvent
```

`StageCancelled` is a `DAGSchedulerEvent` that triggers `handleStageCancellation` (on a separate thread).

handleStageCancellation Handler

```
handleStageCancellation(stageId: Int): Unit
```

`handleStageCancellation` checks if the input `stageId` was registered earlier (in the internal `stageIdToStage` registry) and if it was attempts to `cancel the associated jobs` (with "because Stage [stageId] was cancelled" cancellation reason).

Note	A stage tracks the jobs it belongs to using <code>jobIds</code> property.
------	---

If the stage `stageId` was not registered earlier, you should see the following INFO message in the logs:

```
INFO No active jobs to kill for Stage [stageId]
```

Note	<code>handleStageCancellation</code> is the result of executing <code>SparkContext.cancelStage(stageId: Int)</code> that is called from the web UI (controlled by <code>spark.ui.killEnabled</code>).
------	--

handleJobSubmitted Handler

```
handleJobSubmitted(
  jobId: Int,
  finalRDD: RDD[_],
  func: (TaskContext, Iterator[_]) => _,
  partitions: Array[Int],
  callSite: CallSite,
  listener: JobListener,
  properties: Properties)
```

`handleJobSubmitted` creates a new `ResultStage` (as `finalStage` in the picture below) given the input `finalRDD`, `func`, `partitions`, `jobId` and `callSite`.

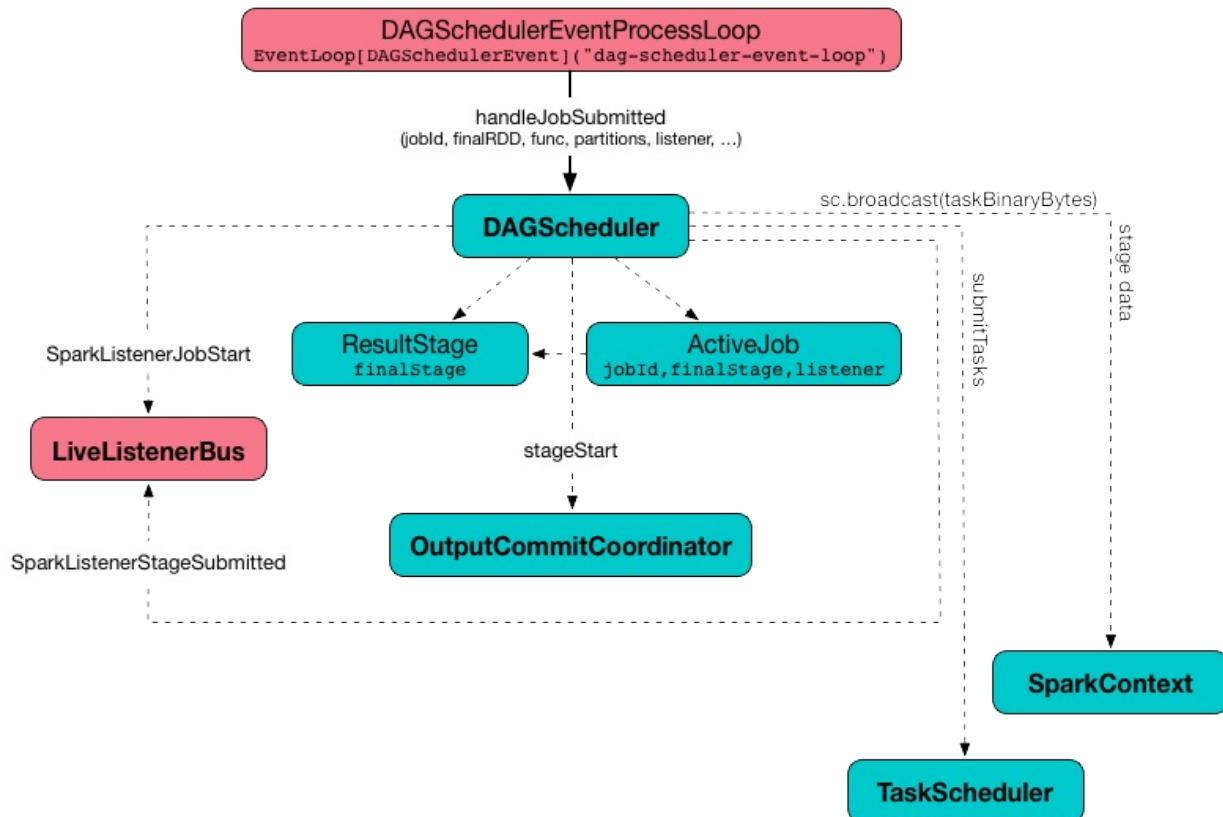


Figure 4. `DAGScheduler.handleJobSubmitted` Method

`handleJobSubmitted` creates an `ActiveJob` (with the input `jobId`, `callSite`, `listener`, `properties`, and the `ResultStage`).

`handleJobSubmitted` clears the internal cache of RDD partition locations.

Caution

FIXME Why is this clearing here so important?

You should see the following INFO messages in the logs:

```

INFO DAGScheduler: Got job [id] ([callSite]) with [number] output partitions
INFO DAGScheduler: Final stage: [stage] ([name])
INFO DAGScheduler: Parents of final stage: [parents]
INFO DAGScheduler: Missing parents: [missingStages]
  
```

`handleJobSubmitted` then registers the new job in `jobIdToActiveJob` and `activeJobs` internal registries, and with the final `ResultStage`.

Note

`ResultStage` can only have one `ActiveJob` registered.

`handleJobSubmitted` finds all the registered stages for the input `jobId` and collects their latest `StageInfo`.

Ultimately, `handleJobSubmitted` posts `SparkListenerJobStart` message to `LiveListenerBus` and submits the stage.

ExecutorAdded Event and handleExecutorAdded Handler

```
ExecutorAdded(execId: String, host: String) extends DAGSchedulerEvent
```

`ExecutorAdded` is a `DAGSchedulerEvent` that triggers `handleExecutorAdded` method (on a separate thread).

Removing Executor From failedEpoch Registry — handleExecutorAdded Handler

```
handleExecutorAdded(execId: String, host: String)
```

`handleExecutorAdded` checks if the input `execId` executor was registered in `failedEpoch` and, if it was, removes it from the `failedEpoch` registry.

You should see the following INFO message in the logs:

```
INFO Host added was in lost list earlier: [host]
```

JobListener

Spark subscribes for job completion or failure events (after submitting a job to [DAGScheduler](#)) using `JobListener` trait.

The following are the job listeners used:

1. [JobWaiter](#) waits until [DAGScheduler](#) completes a job and passes the results of tasks to a `resultHandler` function.
2. [ApproximateActionListener](#) ...[FIXME](#)

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 `JobSubmitted`

JobListener Contract

`JobListener` is a `private[spark]` contract with the following two methods:

```
private[spark] trait JobListener {  
    def taskSucceeded(index: Int, result: Any)  
    def jobFailed(exception: Exception)  
}
```

A `JobListener` object is notified each time a task succeeds (by `taskSucceeded`) and when the whole job fails (by `jobFailed`).

JobWaiter

```
JobWaiter[T](  
    dagScheduler: DAGScheduler,  
    val jobId: Int,  
    totalTasks: Int,  
    resultHandler: (Int, T) => Unit)  
extends JobListener
```

`JobWaiter` is a [JobListener](#) that is used when `DAGScheduler` submits a job or submits a map stage.

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`.

As a `JobListener`, `JobWaiter` gets notified about task completions or failures, using `taskSucceeded` and `jobFailed`, respectively. When the total number of tasks (that equals the number of partitions to compute) equals the number of `taskSucceeded`, the `JobWaiter` instance is marked successful. A `jobFailed` event marks the `JobWaiter` instance failed.

TaskScheduler — Spark Scheduler

`TaskScheduler` is responsible for [submitting tasks for execution](#) in a Spark application (per [scheduling policy](#)).

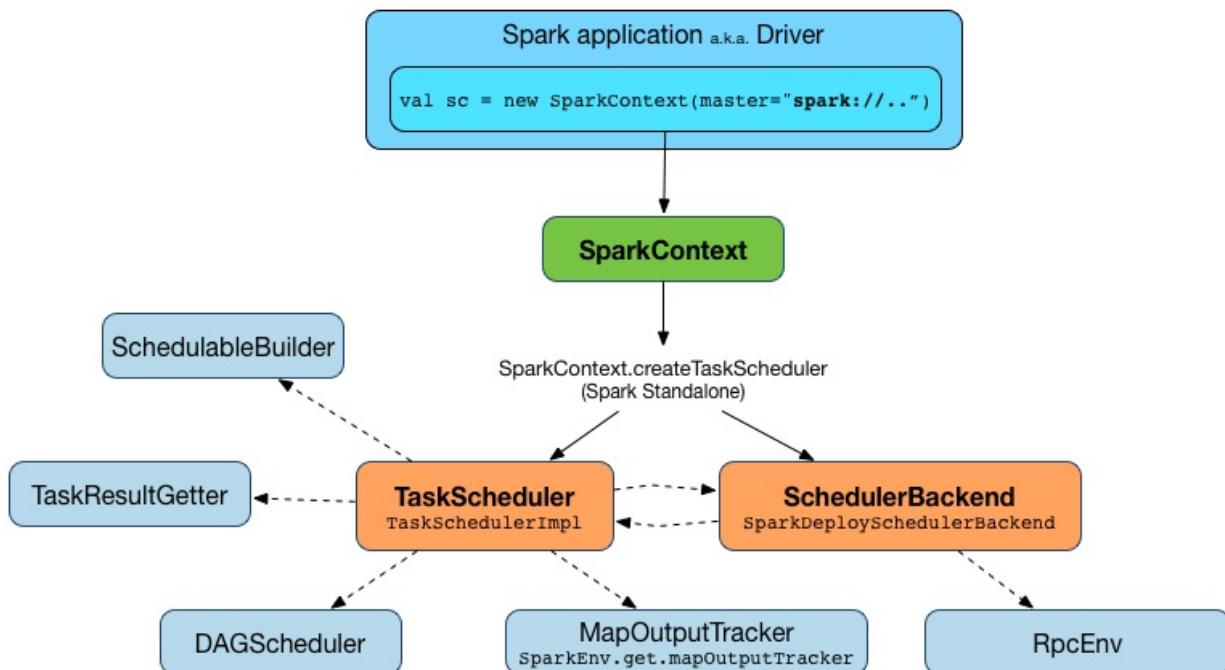


Figure 1. TaskScheduler works for a single SparkContext

Note

`TaskScheduler` works closely with `DAGScheduler` that [submits sets of tasks for execution](#) (for every stage in a Spark job).

`TaskScheduler` tracks the executors in a Spark application using `executorHeartbeatReceived` and `executorLost` methods that are to inform about `active` and `lost` executors, respectively.

Spark comes with the following custom `TaskSchedulers`:

- `TaskSchedulerImpl` — the default `TaskScheduler` (that the following two YARN-specific `TaskSchedulers` extend).
- `YarnScheduler` for Spark on YARN in [client deploy mode](#).
- `YarnClusterScheduler` for Spark on YARN in [cluster deploy mode](#).

Note

The source of `TaskScheduler` is available in `org.apache.spark.scheduler.TaskScheduler`.

TaskScheduler Contract

```

trait TaskScheduler {
  def applicationAttemptId(): Option[String]
  def applicationId(): String
  def cancelTasks(stageId: Int, interruptThread: Boolean): Unit
  def defaultParallelism(): Int
  def executorHeartbeatReceived(
    execId: String,
    accumUpdates: Array[(Long, Seq[AccumulatorV2[_, _]])],
    blockManagerId: BlockManagerId): Boolean
  def executorLost(executorId: String, reason: ExecutorLossReason): Unit
  def postStartHook(): Unit
  def rootPool: Pool
  def schedulingMode: SchedulingMode
  def setDAGScheduler(dagScheduler: DAGScheduler): Unit
  def start(): Unit
  def stop(): Unit
  def submitTasks(taskSet: TaskSet): Unit
}

```

Note	TaskScheduler is a <code>private[spark]</code> contract.
------	--

Table 1. TaskScheduler Contract

Method	Description
applicationAttemptId	<p>Unique identifier of an (execution) attempt of a Spark application.</p> <p>Used exclusively when <code>SparkContext</code> is initialized.</p>
applicationId	<p>Unique identifier of a Spark application.</p> <p>By default, it is in the format <code>spark-application-[System.currentTimeMillis]</code>.</p> <p>Used exclusively when <code>SparkContext</code> is initialized (to set <code>spark.app.id</code>).</p>
cancelTasks	<p>Cancels all tasks of a given stage.</p> <p>Used exclusively when <code>DAGScheduler</code> fails a Spark job and independent single-job stages.</p>
defaultParallelism	<p>Calculates the default level of parallelism.</p> <p>Used when <code>SparkContext</code> is requested for the default level of parallelism.</p>
	<p>Intercepts heartbeats (with task metrics) from executors.</p>

executorHeartbeatReceived	<pre>executorHeartbeatReceived(execId: String, accumUpdates: Array[(Long, Seq[AccumulatorV2[_, _]])], blockManagerId: BlockManagerId): Boolean</pre> <p>Expected to return <code>true</code> when the executor <code>execId</code> is managed by the <code>TaskScheduler</code>. <code>false</code> is to indicate that the block manager (on the executor) should re-register.</p> <p>Used exclusively when <code>HeartbeatReceiver</code> RPC endpoint receives a heartbeat and task metrics from an executor.</p>
executorLost	<p>Intercepts events about executors getting lost.</p> <p>Used when <code>HeartbeatReceiver</code> RPC endpoint gets informed about disconnected executors (i.e. that are no longer available) and when <code>DriverEndpoint</code> forgets or disables malfunctioning executors (i.e. either lost or blacklisted for some reason).</p>
postStartHook	<p>Post-start initialization.</p> <p>Does nothing by default, but allows custom implementations for some additional post-start initialization.</p> <p>Used exclusively when <code>SparkContext</code> is created (right before <code>SparkContext</code> is considered fully initialized).</p>
rootPool	<p>Pool (of Schedulables).</p>
schedulingMode	<p>Scheduling mode.</p> <p>Puts tasks in order according to a scheduling policy (as <code>schedulingMode</code>). It is used in <code>SparkContext.getSchedulingMode</code>.</p>
setDAGScheduler	<p>Assigns DAGScheduler.</p> <p>Used exclusively when <code>DAGScheduler</code> is created (and passes on a reference to itself).</p>
start	<p>Starts <code>TaskScheduler</code>.</p> <p>Used exclusively when <code>SparkContext</code> is created.</p>
stop	<p>Stops <code>TaskScheduler</code>.</p> <p>Used exclusively when <code>DAGScheduler</code> is stopped.</p>

submitTasks	<p>Submits tasks for execution (as TaskSet) of a given stage.</p> <p>Used exclusively when DAGScheduler submits tasks (of a stage) for execution.</p>
--------------------	---

TaskScheduler's Lifecycle

A `TaskScheduler` is created while [SparkContext](#) is being created (by calling `SparkContext.createTaskScheduler` for a given master URL and deploy mode).

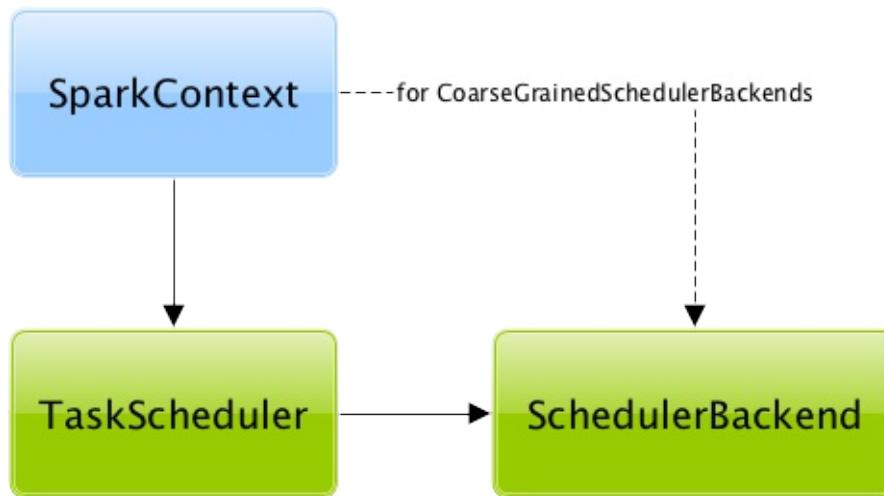


Figure 2. TaskScheduler uses SchedulerBackend to support different clusters

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 `spark.app.id` Spark property, and configure [SparkUI](#), and [BlockManager](#)).

Caution

FIXME 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

FIXME If it is SparkContext to start a TaskScheduler, shouldn't SparkContext stop it too? Why is this the way it is now?

Task

`Task` (aka *command*) is the smallest individual unit of execution that is launched to compute a [RDD partition](#).

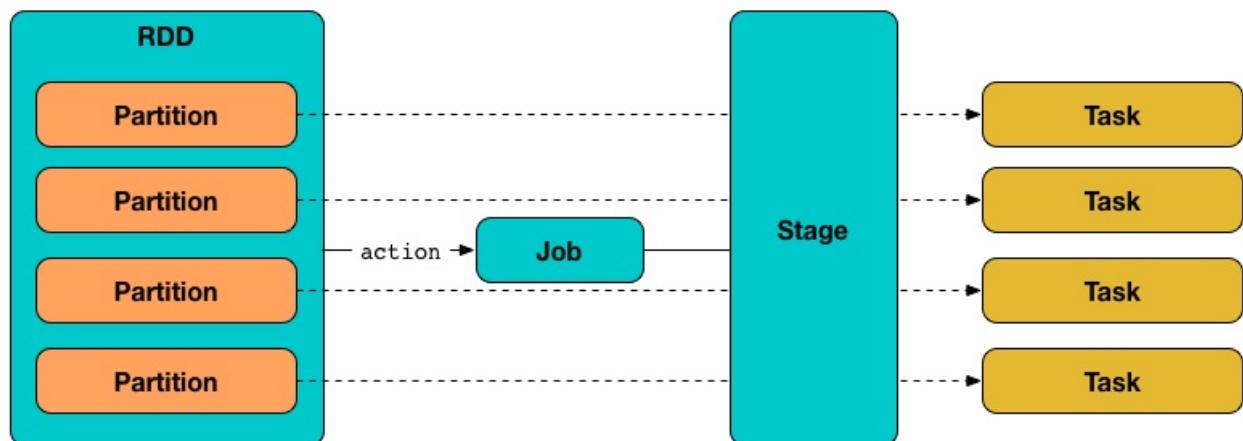


Figure 1. Tasks correspond to partitions in RDD

A task is described by the [Task contract](#) with a single `runTask` to run it and optional [placement preferences](#) to place the computation on right executors.

There are two concrete implementations of `Task` contract:

- [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 Spark job consists of multiple [ResultTasks](#), while earlier stages can only be [ShuffleMapTasks](#).

Caution	FIXME You could have a Spark job with ShuffleMapTask being the last.
---------	--

Tasks are [launched on executors](#) and [ran when](#) `TaskRunner` starts.

In other (more technical) words, a task is a computation on the records in a RDD partition in a stage of a RDD in a Spark job.

Note	<code>T</code> is the type defined when a <code>Task</code> is created.
------	---

Table 1. Task Internal Registries and Counters

Name	Description
context	Used when ???
epoch	Set for a Task when TaskSetManager is created and later used when TaskRunner runs and when DAGScheduler handles a ShuffleMapTask successful completion.
_executorDeserializeTime	Used when ???
_executorDeserializeCpuTime	Used when ???
_killed	Used when ???
metrics	<p>TaskMetrics</p> <p>Created lazily when Task is created from serializedTaskMetrics.</p> <p>Used when ???</p>
taskMemoryManager	<p>TaskMemoryManager that manages the memory allocated by the task.</p> <p>Used when ???</p>
taskThread	Used when ???

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 ?
---------	--

Task Contract

```
def runTask(context: TaskContext): T
def preferredLocations: Seq[TaskLocation] = Nil
```

Note	Task is a private[spark] contract.
------	------------------------------------

Table 2. Task Contract

Method	Description
runTask	Used when a task runs .
preferredLocations	<p>Collection of TaskLocations.</p> <p>Used exclusively when TaskSetManager registers a task as pending execution and dequeueSpeculativeTask.</p> <p>Empty by default and so no task location preferences are defined that says the task could be launched on any executor.</p> <p>Defined by the custom tasks, i.e. ShuffleMapTask and ResultTask.</p>

Creating Task Instance

`Task` takes the following when created:

- [Stage ID](#)
- Stage attempt ID (different per stage execution re-attempt)
- [Partition ID](#)
- Local `Properties` (defaults to empty properties)
- Serialized [TaskMetrics](#) (that [were part of the owning Stage](#))
- (optional) [Job ID](#)
- (optional) Application ID
- (optional) Application attempt ID

`Task` initializes the [internal registries and counters](#).

Running Task Thread — `run` Method

```
run(
  taskAttemptId: Long,
  attemptNumber: Int,
  metricsSystem: MetricsSystem): T
```

`run` registers the task (identified as `taskAttemptId`) with the local `BlockManager`.

Note

`run` uses `SparkEnv` to access the current `BlockManager`.

`run` creates a `TaskContextImpl` that in turn becomes the task's `TaskContext`.

Note

`run` is a `final` method and so must not be overridden.

`run` checks `_killed` flag and, if enabled, kills the task (with `interruptThread` flag disabled).

`run` creates a Hadoop `CallerContext` and sets it.

`run` runs the task.

Note

This is the moment when the custom `Task`'s `runTask` is executed.

In the end, `run` notifies `TaskContextImpl` that the task has completed (regardless of the final outcome—a success or a failure).

In case of any exceptions, `run` notifies `TaskContextImpl` that the task has failed. `run` requests `MemoryStore` to release unroll memory for this task (for both `ON_HEAP` and `OFF_HEAP` memory modes).

Note

`run` uses `SparkEnv` to access the current `BlockManager` that it uses to access `MemoryStore`.

`run` requests `MemoryManager` to notify any tasks waiting for execution memory to be freed to wake up and try to acquire memory again.

`run` unsets the task's `TaskContext`.

Note

`run` uses `SparkEnv` to access the current `MemoryManager`.

Note

`run` is used exclusively when `TaskRunner` starts. The `Task` instance has just been deserialized from `taskBytes` that were sent over the wire to an executor. `localProperties` and `TaskMemoryManager` are already assigned.

Task States

A task can be in one of the following states (as described by `TaskState` enumeration):

- `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`, `CommitDeniedException` or any `Throwable` occurs

- `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 `ExecutorBackend`.

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 `org.apache.mesos.Protos.TaskState`.

Collect Latest Values of (Internal and External) Accumulators — `collectAccumulatorUpdates` Method

```
collectAccumulatorUpdates(taskFailed: Boolean = false): Seq[AccumulableInfo]
```

`collectAccumulatorUpdates` collects the latest values of internal and external accumulators from a task (and returns the values as a collection of `AccumulableInfo`).

Internally, `collectAccumulatorUpdates` takes `TaskMetrics`.

Note

`collectAccumulatorUpdates` uses `TaskContextImpl` to access the task's `TaskMetrics`.

`collectAccumulatorUpdates` collects the latest values of:

- `internal accumulators` whose current value is not the zero value and the `RESULT_SIZE` accumulator (regardless whether the value is its zero or not).
- `external accumulators` when `taskFailed` is disabled (`false`) or which `should be included on failures`.

`collectAccumulatorUpdates` returns an empty collection when `TaskContextImpl` is not initialized.

Note

`collectAccumulatorUpdates` is used when `TaskRunner` runs a task (and sends a task's final results back to the driver).

Killing Task — `kill` Method

```
kill(interruptThread: Boolean)
```

`kill` marks the task to be killed, i.e. it sets the internal `_killed` flag to `true`.

`kill` calls [TaskContextImpl.markInterrupted](#) when `context` is set.

If `interruptThread` is enabled and the internal `taskThread` is available, `kill` interrupts it.

Caution

[FIXME](#) When could `context` and `interruptThread` not be set?

ShuffleMapTask — Task for ShuffleMapStage

`ShuffleMapTask` is a [Task](#) that [computes a `MapStatus`](#), i.e. writes the result of computing records in a RDD partition to the [shuffle system](#) and returns information about the [BlockManager](#) and estimated size of the result shuffle blocks.

`ShuffleMapTask` is created exclusively when [DAGScheduler](#) submits missing tasks for a [ShuffleMapStage](#).

Table 1. ShuffleMapTask's Internal Registries and Counters

Name	Description
<code>preferredLocs</code>	<p>Collection of TaskLocations.</p> <p>Corresponds directly to unique entries in <code>locs</code> with the only rule that when <code>locs</code> is not defined, it is empty, and no task location preferences are defined.</p> <p>Initialized when <code>ShuffleMapTask</code> is created.</p> <p>Used exclusively when <code>ShuffleMapTask</code> is requested for preferred locations.</p>

Note	Spark uses broadcast variables to send (serialized) tasks to executors.
------	---

Creating ShuffleMapTask Instance

`ShuffleMapTask` takes the following when created:

- `stageId` — the [stage](#) of the task
- `stageAttemptId` — the stage's attempt
- `taskBinary` — the [broadcast variable](#) with the serialized task (as an array of bytes)
- [Partition](#)
- Collection of [TaskLocations](#)
- `localProperties` — task-specific local properties
- `serializedTaskMetrics` — the serialized [FIXME](#) (as an array of bytes)
- `jobId` — optional [ActiveJob](#) id (default: undefined)
- `appId` — optional application id (default: undefined)
- `appAttemptId` — optional application attempt id (default: undefined)

`ShuffleMapTask` calculates `preferredLocs` internal attribute that is the input `locs` if defined. Otherwise, it is empty.

Note	<code>preferredLocs</code> and <code>locs</code> are transient so they are not sent over the wire with the task.
------	--

`ShuffleMapTask` initializes the [internal registries and counters](#).

Writing Records (After Computing RDD Partition) to Shuffle System — `runTask` Method

<code>runTask(context: TaskContext): MapStatus</code>	
---	--

Note	<code>runTask</code> is a part of Task contract to... FIXME
------	---

`runTask` computes a [MapStatus](#) (which is the [BlockManager](#) and an estimated size of the result shuffle block) after the records of the [Partition](#) were written to the [shuffle system](#).

Internally, `runTask` uses the [current closure](#) `Serializer` to deserialize the `taskBinary` serialized task (into a pair of [RDD](#) and [ShuffleDependency](#)).

`runTask` measures the thread and CPU time for deserialization (using the System clock and JMX if supported) and stores it in `_executorDeserializeTime` and `_executorDeserializeCpuTime` attributes.

Note	<code>runTask</code> uses <code>SparkEnv</code> to access the current closure <code>Serializer</code> .
------	---

Note	The <code>taskBinary</code> serialized task is given when <code>ShuffleMapTask</code> is created.
------	---

`runTask` requests `ShuffleManager` for a `ShuffleWriter` (given the `ShuffleHandle` of the deserialized `ShuffleDependency`, `partitionId` and input `TaskContext`).

Note	<code>runTask</code> uses <code>SparkEnv</code> to access the current <code>shuffleManager</code> .
------	---

Note	The <code>partitionId</code> partition is given when <code>ShuffleMapTask</code> is created.
------	--

`runTask` gets the records in the RDD partition (as an `Iterator`) and writes them (to the shuffle system).

Note	This is the moment in <code>Task</code> 's lifecycle (and its corresponding RDD) when a RDD partition is computed and in turn becomes a sequence of records (i.e. real data) on a executor.
------	---

`runTask` stops the `ShuffleWriter` (with `success` flag enabled) and returns the `MapStatus`.

When the record writing was not successful, `runTask` stops the `ShuffleWriter` (with `success` flag disabled) and the exception is re-thrown.

You may also see the following DEBUG message in the logs when the `ShuffleWriter` could not be stopped.

```
DEBUG Could not stop writer
```

preferredLocations Method

```
preferredLocations: Seq[TaskLocation]
```

Note

`preferredLocations` is a part of Task contract to...[FIXME](#)

`preferredLocations` simply returns `preferredLocs` internal property.

ResultTask

`ResultTask` is a [Task](#) that [executes a function](#) on the records in a `RDD` partition.

`ResultTask` is created exclusively when `DAGScheduler` submits missing tasks for a `ResultStage`.

`ResultTask` is created with a [broadcast variable](#) with the `RDD` and the function to execute it on and the [partition](#).

Table 1. ResultTask's Internal Registries and Counters

Name	Description
<code>preferredLocs</code>	<p>Collection of TaskLocations. Corresponds directly to unique entries in <code>locs</code> with the only rule that when <code>locs</code> is not defined, it is empty, and no task location preferences are defined.</p> <p>Initialized when <code>ResultTask</code> is created.</p> <p>Used exclusively when <code>ResultTask</code> is requested for preferred locations.</p>

Creating ResultTask Instance

`ResultTask` takes the following when created:

- `stageId` — the stage the task is executed for
- `stageAttemptId` — the stage attempt id
- [Broadcast variable](#) with the serialized task (as `Array[Byte]`). The broadcast contains of a serialized pair of `RDD` and the function to execute.
- [Partition](#) to compute
- Collection of [TaskLocations](#), i.e. preferred locations (executors) to execute the task on
- `outputId`
- `local Properties`
- The stage's serialized [TaskMetrics](#) (as `Array[Byte]`)
- (optional) [Job](#) id
- (optional) Application id

- (optional) Application attempt id

`ResultTask` initializes the [internal registries and counters](#).

preferredLocations Method

```
preferredLocations: Seq[TaskLocation]
```

Note	<code>preferredLocations</code> is a part of Task contract .
------	--

`preferredLocations` simply returns [preferredLocs](#) internal property.

Deserialize RDD and Function (From Broadcast) and Execute Function (on RDD Partition) — runTask Method

```
runTask(context: TaskContext): U
```

Note	<code>U</code> is the type of a result as defined when <code>ResultTask</code> is created.
------	--

`runTask` deserializes a RDD and a function from the [broadcast](#) and then executes the function (on the records from the RDD [partition](#)).

Note	<code>runTask</code> is a part of Task contract to run a task.
------	--

Internally, `runTask` starts by tracking the time required to deserialize a RDD and a function to execute.

`runTask` [creates a new closure](#) `Serializer`.

Note	<code>runTask</code> uses <code>sparkEnv</code> to access the current closure <code>Serializer</code> .
------	---

`runTask` [requests the closure](#) `Serializer` to deserialize an `RDD` and the function to execute (from `taskBinary` broadcast).

Note	<code>taskBinary</code> broadcast is defined when <code>ResultTask</code> is created.
------	---

`runTask` records `_executorDeserializeTime` and `_executorDeserializeCpuTime` properties.

In the end, `runTask` executes the function (passing in the input `context` and the [records](#) from `partition` of the `RDD`).

Note	<code>partition</code> to use to access the records in a deserialized RDD is defined when <code>ResultTask</code> was created.
------	--

TaskDescription

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

encode Method

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

decode Method

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

FetchFailedException

`FetchFailedException` exception may be thrown when a task runs (and `ShuffleBlockFetcherIterator` did not manage to fetch shuffle blocks).

`FetchFailedException` contains the following:

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

When `FetchFailedException` is reported, `TaskRunner` catches it and notifies `ExecutorBackend` (with `TaskState.FAILED` task state).

The root `cause` of the `FetchFailedException` is usually because the executor (with the `BlockManager` for the shuffle blocks) is lost (i.e. no longer available) due to:

1. `OutOfMemoryError` could be thrown (aka OOMed) or some other unhandled exception.
2. The cluster manager that manages the workers with the executors of your Spark application, e.g. YARN, enforces the container memory limits and eventually decided to kill the executor due to excessive memory usage.

You should review the logs of the Spark application using [web UI](#), [Spark History Server](#) or cluster-specific tools like `yarn logs -applicationId` for Hadoop YARN.

A solution is usually to tune the memory of your Spark application.

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

toTaskFailedReason Method

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

MapStatus — Shuffle Map Output Status

`MapStatus` is the result of running a `ShuffleMapTask` that includes information about the `BlockManager` and estimated size of the reduce blocks.

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	<code>FIXME</code> What exactly is 2000? Is this the number of tasks in a job?
---------	--

MapStatus Contract

```
trait MapStatus {
  def location: BlockManagerId
  def getSizeForBlock(reduceId: Int): Long
}
```

Note	<code>MapStatus</code> is a <code>private[spark]</code> contract.
------	---

Table 1. `MapStatus` Contract

Method	Description
<code>location</code>	The <code>BlockManager</code> where a <code>ShuffleMapTask</code> ran and the result is stored.
<code>getSizeForBlock</code>	The estimated size for the reduce block (in bytes).

TaskSet— Set of Tasks for Single Stage

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`
- `TaskSchedulerImpl.createTaskSetManager`

Priority Field and FIFO Scheduling

A TaskSet has `priority` field that turns into the **priority** field's value of [TaskSetManager](#) (which is a [Schedulable](#)).

The `priority` field is used in [FIFOSchedulingAlgorithm](#) in which equal priorities give stages an advantage (not to say *priority*).

Note

`FIFO``SchedulingAlgorithm` is only used for `FIFO` scheduling mode in a `Pool` (i.e. a schedulable collection of `Schedulable` objects).

Effectively, the `priority` field is the job's id of the first job this stage was part of (for FIFO scheduling).

TaskSetManager

`TaskSetManager` is a `Schedulable` that manages scheduling of tasks in a `TaskSet`.

Note A [TaskSet](#) represents a set of [tasks](#) that correspond to missing partitions of a stage.

TaskSetManager is created when TaskSchedulerImpl submits tasks (for a given TaskSet).

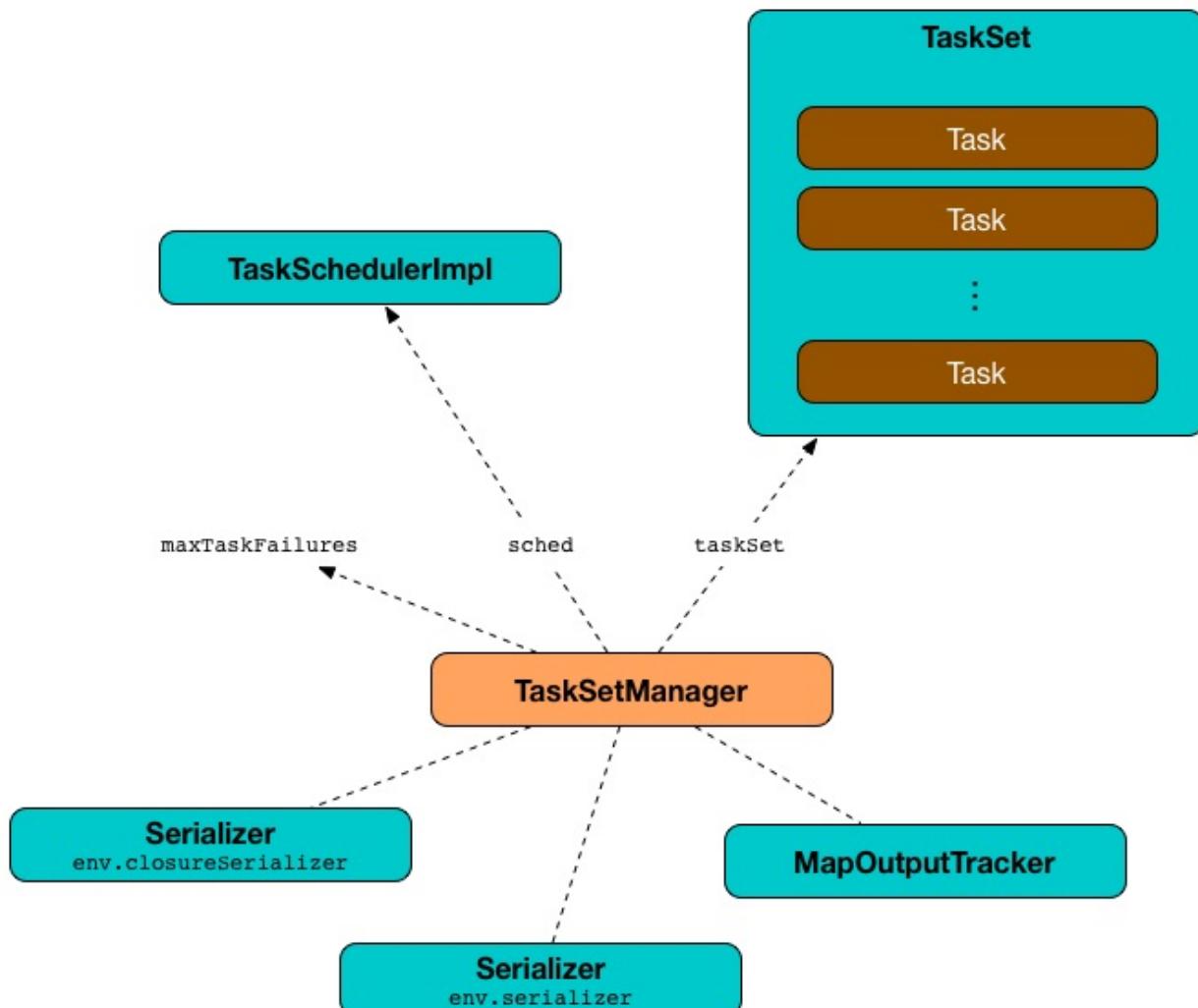


Figure 1. TaskSetManager and its Dependencies

When `TaskSetManager` is created for a `TaskSet`, `TaskSetManager` registers all the tasks as pending execution.

`TaskSetManager` is notified when a task (from the `TaskSet` it manages) finishes — **successfully** or due to a **failure** (in task execution or **an executor being lost**).

`TaskSetManager` uses `maxTaskFailures` to control how many times a `single task can fail before an entire TaskSet gets aborted` that can take the following values:

- 1 for local run mode
- maxFailures in Spark local-with-retries (i.e. local[N, maxFailures])
- spark.task.maxFailures property for Spark local-cluster and Spark clustered (using Spark Standalone, Mesos and YARN)

The responsibilities of a TaskSetManager include:

- Scheduling the tasks in a taskset
- Retrying tasks on failure
- Locality-aware scheduling via delay scheduling

Enable DEBUG logging levels for org.apache.spark.scheduler.TaskSchedulerImpl (or org.apache.spark.scheduler.cluster.YarnScheduler for YARN) and org.apache.spark following two-stage job to see their low-level innerworkings.

A cluster manager is recommended since it gives more task localization choices (with localization).

```
$ ./bin/spark-shell --master yarn --conf spark.ui.showConsoleProgress=false

// Keep # partitions low to keep # messages low
scala> sc.parallelize(0 to 9, 3).groupBy(_ % 3).count
INFO YarnScheduler: Adding task set 0.0 with 3 tasks
DEBUG TaskSetManager: Epoch for TaskSet 0.0: 0
DEBUG TaskSetManager: Valid locality levels for TaskSet 0.0: NO_PREF, ANY
DEBUG YarnScheduler: parentName: , name: TaskSet_0.0, runningTasks: 0
INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, 10.0.2.87, executor
INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1, 10.0.2.87, executor
DEBUG YarnScheduler: parentName: , name: TaskSet_0.0, runningTasks: 1
INFO TaskSetManager: Starting task 2.0 in stage 0.0 (TID 2, 10.0.2.87, executor
DEBUG YarnScheduler: parentName: , name: TaskSet_0.0, runningTasks: 1
DEBUG TaskSetManager: No tasks for locality level NO_PREF, so moving to locality
INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 518 ms on 10.0.2.87
INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1) in 512 ms on 10.0.2.87
DEBUG YarnScheduler: parentName: , name: TaskSet_0.0, runningTasks: 0
INFO TaskSetManager: Finished task 2.0 in stage 0.0 (TID 2) in 51 ms on 10.0.2.87
INFO YarnScheduler: Removed TaskSet 0.0, whose tasks have all completed, from pool
INFO YarnScheduler: Adding task set 1.0 with 3 tasks
DEBUG TaskSetManager: Epoch for TaskSet 1.0: 1
DEBUG TaskSetManager: Valid locality levels for TaskSet 1.0: NODE_LOCAL, RACK_LOCAL
DEBUG YarnScheduler: parentName: , name: TaskSet_1.0, runningTasks: 0
INFO TaskSetManager: Starting task 0.0 in stage 1.0 (TID 3, 10.0.2.87, executor
INFO TaskSetManager: Starting task 1.0 in stage 1.0 (TID 4, 10.0.2.87, executor
DEBUG YarnScheduler: parentName: , name: TaskSet_1.0, runningTasks: 1
INFO TaskSetManager: Starting task 2.0 in stage 1.0 (TID 5, 10.0.2.87, executor
INFO TaskSetManager: Finished task 1.0 in stage 1.0 (TID 4) in 130 ms on 10.0.2.87
DEBUG YarnScheduler: parentName: , name: TaskSet_1.0, runningTasks: 1
DEBUG TaskSetManager: No tasks for locality level NODE_LOCAL, so moving to local
DEBUG TaskSetManager: No tasks for locality level RACK_LOCAL, so moving to local
INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 3) in 133 ms on 10.0.2.87
DEBUG YarnScheduler: parentName: , name: TaskSet_1.0, runningTasks: 0
INFO TaskSetManager: Finished task 2.0 in stage 1.0 (TID 5) in 21 ms on 10.0.2.87
INFO YarnScheduler: Removed TaskSet 1.0, whose tasks have all completed, from pool
res0: Long = 3
```

Tip

Table 1. TaskSetManager's Internal Registries and Counters

Name	Description
allPendingTasks	Indices of all the pending tasks to execute (regardless of their localization preferences). Updated with an task index when <code>TaskSetManager registers a task as pending execution (per preferred locations)</code> .
calculatedTasks	The number of the tasks that have already completed execution. Starts from <code>0</code> when a <code>TaskSetManager</code> is created and is only incremented when the <code>TaskSetManager</code> checks that there is enough memory to fetch a task result.
copiesRunning	The number of task copies currently running per task (index in its task set). The number of task copies of a task is increased when finds a task for execution (given resource offer) or checking for speculatable tasks and decreased when a task fails or an executor is lost (for a shuffle map stage and no external shuffle service).
currentLocalityIndex	
epoch	Current map output tracker epoch.
failedExecutors	Lookup table of <code>TaskInfo</code> indices that failed to executor ids and the time of the failure. Used in <code>handleFailedTask</code> .
isZombie	Disabled, i.e. <code>false</code> , by default. Read Zombie state in this document.
lastLaunchTime	
localityWaits	
myLocalityLevels	<code>TaskLocality</code> locality preferences of the pending tasks in the <code>TaskSet</code> ranging from <code>PROCESS_LOCAL</code> through <code>NODE_LOCAL</code> , <code>NO_PREF</code> , and <code>RACK_LOCAL</code> to <code>ANY</code> . NOTE: <code>myLocalityLevels</code> may contain only a few of all the available <code>TaskLocality</code> preferences with <code>ANY</code> as a mandatory task locality preference. Set immediately when <code>TaskSetManager</code> is created. Recomputed every change in the status of executors.

<code>name</code>	
<code>numFailures</code>	<p>Array of the number of task failures per task.</p> <p>Incremented when <code>TaskSetManager</code> handles a task failure and immediately checked if above acceptable number of task failures.</p>
<code>numTasks</code>	Number of tasks to compute.
<code>pendingTasksForExecutor</code>	<p>Lookup table of the indices of tasks pending execution per executor.</p> <p>Updated with an task index and executor when <code>TaskSetManager</code> registers a task as pending execution (per preferred locations) (and the location is a <code>ExecutorCacheTaskLocation</code> or <code>HDFSCacheTaskLocation</code>).</p>
<code>pendingTasksForHost</code>	<p>Lookup table of the indices of tasks pending execution per host.</p> <p>Updated with an task index and host when <code>TaskSetManager</code> registers a task as pending execution (per preferred locations).</p>
<code>pendingTasksForRack</code>	<p>Lookup table of the indices of tasks pending execution per rack.</p> <p>Updated with an task index and rack when <code>TaskSetManager</code> registers a task as pending execution (per preferred locations).</p>
<code>pendingTasksWithNoPrefs</code>	<p>Lookup table of the indices of tasks pending execution with no location preferences.</p> <p>Updated with an task index when <code>TaskSetManager</code> registers a task as pending execution (per preferred locations).</p>
<code>priority</code>	
<code>recentExceptions</code>	
<code>runningTasksSet</code>	<p>Collection of running tasks that a <code>TaskSetManager</code> manages.</p> <p>Used to implement <code>runningTasks</code> (that is simply the size of <code>runningTasksSet</code> but a required part of any <code>Schedulable</code>). <code>runningTasksSet</code> is expanded when registering a running task and shrunked when unregistering a running task.</p>

	Used in TaskSchedulerImpl to cancel tasks.
<code>speculatableTasks</code>	
<code>stageId</code>	<p>The stage's id a <code>TaskSetManager</code> runs for.</p> <p>Set when TaskSetManager is created.</p> <p>NOTE: <code>stageId</code> is a part of Schedulable contract.</p>
<code>successful</code>	<p>Status of tasks (with a boolean flag, i.e. <code>true</code> or <code>false</code>, per task).</p> <p>All tasks start with their flags disabled, i.e. <code>false</code>, when TaskSetManager is created.</p> <p>The flag for a task is turned on, i.e. <code>true</code>, when a task finishes successfully but also with a failure.</p> <p>A flag is explicitly turned off only for ShuffleMapTask tasks when their executor is lost.</p>
<code>taskAttempts</code>	Registry of TaskInfos per every task attempt per task.
<code>taskInfos</code>	<p>Registry of TaskInfos per task id.</p> <p>Updated with the task (id) and the corresponding <code>TaskInfo</code> when TaskSetManager finds a task for execution (given resource offer).</p> <p>NOTE: It <i>appears</i> that the entries stay forever, i.e. are never removed (perhaps because the maintenance overhead is not needed given a <code>TaskSetManager</code> is a short-lived entity).</p>
<code>tasks</code>	<p>Lookup table of Tasks (per partition id) to schedule execution of.</p> <p>NOTE: The tasks all belong to a single TaskSet that was given when TaskSetManager was created (which actually represent a single Stage).</p>
<code>tasksSuccessful</code>	
<code>totalResultSize</code>	<p>The current total size of the result of all the tasks that have finished.</p> <p>Starts from <code>0</code> when TaskSetManager is created.</p> <p>Only increased with the size of a task result whenever a TaskSetManager checks that there is enough memory to fetch the task result.</p>

Enable `DEBUG` logging level for `org.apache.spark.scheduler.TaskSetManager` logger to see what happens inside.

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

Tip

```
log4j.logger.org.apache.spark.scheduler.TaskSetManager=DEBUG
```

Refer to [Logging](#).

isTaskBlacklistedOnExecOrNode Method

Caution

[FIXME](#)

getLocalityIndex Method

Caution

[FIXME](#)

dequeueSpeculativeTask Method

Caution

[FIXME](#)

executorAdded Method

`executorAdded` simply calls [recomputeLocality](#) method.

abortIfCompletelyBlacklisted Method

Caution

[FIXME](#)

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`.
- `addScheduled` and `removeScheduled` do nothing.
- `getScheduledByName` always returns `null`.
- `getSortedTaskSetQueue` returns a one-element collection with the sole element being itself.
- `executorLost`
- `checkSpeculatableTasks`

Marking Task As Fetching Indirect Result — `handleTaskGettingResult` Method

```
handleTaskGettingResult(tid: Long): Unit
```

`handleTaskGettingResult` finds `TaskInfo` for `tid` task in `taskInfos` internal registry and marks it as fetching indirect task result. It then notifies `DAGScheduler`.

Note	<code>handleTaskGettingResult</code> is executed when <code>TaskSchedulerImpl</code> is notified about fetching indirect task result.
------	---

Registering Running Task — `addRunningTask` Method

```
addRunningTask(tid: Long): Unit
```

`addRunningTask` adds `tid` to `runningTasksSet` internal registry and requests the `parent pool` to increase the number of running tasks (if defined).

Unregistering Running Task — `removeRunningTask` Method

```
removeRunningTask(tid: Long): Unit
```

`removeRunningTask` removes `tid` from `runningTasksSet` internal registry and requests the `parent pool` to decrease the number of running task (if defined).

Checking Speculatable Tasks — `checkSpeculatableTasks` Method

Note

`checkSpeculatableTasks` is part of the [Scheduled Contract](#).

```
checkSpeculatableTasks(minTimeToSpeculation: Int): Boolean
```

`checkSpeculatableTasks` checks whether there are speculatable tasks in a `TaskSet`.

Note

`checkSpeculatableTasks` is called when `TaskSchedulerImpl` checks for speculatable tasks.

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` internal registry) 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 internal registry`) 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.

getAllowedLocalityLevel Method

Caution

[FIXME](#)

Finding Task For Execution (Given Resource Offer) — resourceOffer Method

```
resourceOffer(  
    execId: String,  
    host: String,  
    maxLocality: TaskLocality): Option[TaskDescription]
```

(only if `TaskSetBlacklist` is defined) `resourceOffer` requests `TaskSetBlacklist` to check if the input `execId` `executor` or `host` `node` are blacklisted.

When `TaskSetManager` is a `zombie` or the resource offer (as executor and host) is blacklisted, `resourceOffer` finds no tasks to execute (and returns no `TaskDescription`).

Note

`resourceOffer` finds a task to schedule for a resource offer when neither `TaskSetManager` is a `zombie` nor the resource offer is blacklisted.

`resourceOffer` calculates the allowed task locality for task selection. When the input `maxLocality` is not `NO_PREF` task locality, `resourceOffer` `getAllowedLocalityLevel` (for the current time) and sets it as the current task locality if more localized (specific).

Note

`TaskLocality` can be the most localized `PROCESS_LOCAL`, `NODE_LOCAL` through `NO_PREF` and `RACK_LOCAL` to `ANY`.

`resourceOffer` dequeues a task for execution (given locality information).

If a task (index) is found, `resourceOffer` takes the `Task` (from `tasks` registry).

`resourceOffer` requests `TaskSchedulerImpl` for the id for the new task.

`resourceOffer` increments the number of the copies of the task that are currently running and finds the task attempt number (as the size of `taskAttempts` entries for the task index).

`resourceOffer` creates a `TaskInfo` that is then registered in `taskInfos` and `taskAttempts`.

If the maximum acceptable task locality is not `NO_PREF`, `resourceOffer` `getLocalityIndex` (using the task's locality) and records it as `currentLocalityIndex` with the current time as `lastLaunchTime`.

`resourceOffer` serializes the task.

Note	<code>resourceOffer</code> uses <code>SparkEnv</code> to access the closure <code>Serializer</code> and create an instance thereof.
------	---

If the task serialization fails, you should see the following ERROR message in the logs:

```
Failed to serialize task [taskId], not attempting to retry it.
```

`resourceOffer` aborts the `TaskSet` with the following message and reports a `TaskNotSerializableException`.

```
Failed to serialize task [taskId], not attempting to retry it.  
Exception during serialization: [exception]
```

`resourceOffer` checks the size of the serialized task. If it is greater than `100` kB, you should see the following WARN message in the logs:

```
WARN Stage [id] contains a task of very large size ([size] KB).  
The maximum recommended task size is 100 KB.
```

Note	The size of the serializable task, i.e. <code>100</code> kB, is not configurable.
------	---

If however the serialization went well and the size is fine too, `resourceOffer` registers the task as running.

You should see the following INFO message in the logs:

```
INFO TaskSetManager: Starting [name] (TID [id], [host], executor  
[id], partition [id], [taskLocality], [size] bytes)
```

For example:

```
INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1,
localhost, partition 1, PROCESS_LOCAL, 2054 bytes)
```

`resourceOffer` notifies `DAGScheduler` that the task has been started.

Important	This is the moment when <code>TaskSetManager</code> informs <code>DAGScheduler</code> that a task has started.
-----------	--

Note	<code>resourceOffer</code> is used when <code>TaskSchedulerImpl</code> <code>resourceOfferSingleTaskSet</code> .
------	--

Dequeueing Task For Execution (Given Locality Information) — `dequeueTask` Internal Method

```
dequeueTask(execId: String, host: String, maxLocality: TaskLocality): Option[(Int, TaskLocality, Boolean)]
```

`dequeueTask` tries to find the highest task index (meeting localization requirements) using tasks (indices) registered for execution on `execId` `executor`. If a task is found, `dequeueTask` returns its index, `PROCESS_LOCAL` task locality and the speculative marker disabled.

`dequeueTask` then goes over all the possible task localities and checks what locality is allowed given the input `maxLocality`.

`dequeueTask` checks out `NODE_LOCAL`, `NO_PREF`, `RACK_LOCAL` and `ANY` in that order.

For `NODE_LOCAL` `dequeueTask` tries to find the highest task index (meeting localization requirements) using tasks (indices) registered for execution on `host` `host` and if found returns its index, `NODE_LOCAL` task locality and the speculative marker disabled.

For `NO_PREF` `dequeueTask` tries to find the highest task index (meeting localization requirements) using `pendingTasksWithNoPrefs` internal registry and if found returns its index, `PROCESS_LOCAL` task locality and the speculative marker disabled.

Note	For <code>NO_PREF</code> the task locality is <code>PROCESS_LOCAL</code> .
------	--

For `RACK_LOCAL` `dequeueTask` finds the rack for the input `host` and if available tries to find the highest task index (meeting localization requirements) using tasks (indices) registered for execution on the rack. If a task is found, `dequeueTask` returns its index, `RACK_LOCAL` task locality and the speculative marker disabled.

For `ANY` `dequeueTask` tries to [find the highest task index](#) (meeting localization requirements) using `allPendingTasks` internal registry and if found returns its index, `ANY` task locality and the speculative marker disabled.

In the end, when no task could be found, `dequeueTask` [dequeueSpeculativeTask](#) and if found returns its index, locality and the speculative marker enabled.

Note	The speculative marker is enabled for a task only when <code>dequeueTask</code> did not manage to find a task for the available task localities and did find a speculative task.
------	--

Note	<code>dequeueTask</code> is used exclusively when <code>TaskSetManager</code> finds a task for execution (given resource offer) .
------	---

Finding Highest Task Index (Not Blacklisted, With No Copies Running and Not Completed Already)

— `dequeueTaskFromList` Internal Method

```
dequeueTaskFromList(
  execId: String,
  host: String,
  list: ArrayBuffer[Int]): Option[Int]
```

`dequeueTaskFromList` takes task indices from the input `list` backwards (from the last to the first entry). For every index `dequeueTaskFromList` checks if it is not [blacklisted on the input `execId` executor and `host`](#) and if not, checks that:

- [number of the copies of the task currently running is `0`](#)
- the task has not been marked as [completed](#)

If so, `dequeueTaskFromList` returns the task index.

If `dequeueTaskFromList` has checked all the indices and no index has passed the checks, `dequeueTaskFromList` returns `None` (to indicate that no index has met the requirements).

Note	<code>dequeueTaskFromList</code> is used exclusively when <code>TaskSetManager</code> dequeues a task for execution (given locality information) .
------	--

Finding Tasks (Indices) Registered For Execution on Executor — `getPendingTasksForExecutor` Internal Method

```
getPendingTasksForExecutor(executorId: String): ArrayBuffer[Int]
```

`getPendingTasksForExecutor` finds pending tasks (indices) registered for execution on the input `executorId` executor (in `pendingTasksForExecutor` internal registry).

Note	<code>getPendingTasksForExecutor</code> may find no matching tasks and return an empty collection.
------	--

Note	<code>getPendingTasksForExecutor</code> is used exclusively when <code>TaskSetManager dequeues a task for execution (given locality information)</code> .
------	---

Finding Tasks (Indices) Registered For Execution on Host — `getPendingTasksForHost` Internal Method

```
getPendingTasksForHost(host: String): ArrayBuffer[Int]
```

`getPendingTasksForHost` finds pending tasks (indices) registered for execution on the input `host` host (in `pendingTasksForHost` internal registry).

Note	<code>getPendingTasksForHost</code> may find no matching tasks and return an empty collection.
------	--

Note	<code>getPendingTasksForHost</code> is used exclusively when <code>TaskSetManager dequeues a task for execution (given locality information)</code> .
------	---

Finding Tasks (Indices) Registered For Execution on Rack — `getPendingTasksForRack` Internal Method

```
getPendingTasksForRack(rack: String): ArrayBuffer[Int]
```

`getPendingTasksForRack` finds pending tasks (indices) registered for execution on the input `rack` rack (in `pendingTasksForRack` internal registry).

Note	<code>getPendingTasksForRack</code> may find no matching tasks and return an empty collection.
------	--

Note	<code>getPendingTasksForRack</code> is used exclusively when <code>TaskSetManager dequeues a task for execution (given locality information)</code> .
------	---

Scheduling Tasks in TaskSet

Caution

FIXME

For each submitted [TaskSet](#), a new TaskSetManager is created. The TaskSetManager completely and exclusively owns a TaskSet submitted for execution.

Caution

FIXME A picture with `TaskSetManager` owning TaskSet

Caution

FIXME What component knows about TaskSet and TaskSetManager. Isn't it that TaskSets are **created** by DAGScheduler while TaskSetManager is used by TaskSchedulerImpl only?

TaskSetManager 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

Once a task has finished, `TaskSetManager` informs [DAGScheduler](#).

Caution

FIXME

Recording Successful Task And Notifying DAGScheduler — `handleSuccessfulTask` Method

```
handleSuccessfulTask(tid: Long, result: DirectTaskResult[_]): Unit
```

`handleSuccessfulTask` records the `tid` task as finished, notifies the [DAGScheduler](#) that the task has ended and attempts to mark the `TaskSet` finished.

Note

`handleSuccessfulTask` is executed after `TaskSchedulerImpl` has been informed that `tid` task finished successfully (and the task result was deserialized).

Internally, `handleSuccessfulTask` finds `TaskInfo` (in `taskInfos` internal registry) and marks it as `FINISHED`.

It then removes `tid` task from `runningTasksSet` internal registry.

`handleSuccessfulTask` notifies `DAGScheduler` that `tid` task ended successfully (with the `Task` object from `tasks` internal registry and the result as `Success`).

At this point, `handleSuccessfulTask` finds the other `running task attempts` of `tid` task and requests `SchedulerBackend` to kill them (since they are no longer necessary now when at least one task attempt has completed successfully). You should see the following INFO message in the logs:

```
INFO Killing attempt [attemptNumber] for task [id] in stage [id]
(TID [id]) on [host] as the attempt [attemptNumber] succeeded on
[host]
```

Caution	FIXME Review <code>taskAttempts</code>
---------	--

If `tid` has *not* yet been recorded as `successful`, `handleSuccessfulTask` increases `tasksSuccessful` counter. You should see the following INFO message in the logs:

```
INFO Finished task [id] in stage [id] (TID [taskId]) in
[duration] ms on [host] (executor [executorId])
([tasksSuccessful]/[numTasks])
```

`tid` task is marked as `successful`. If the number of task that have finished successfully is exactly the number of the tasks to execute (in the `Taskset`), the `TaskSetManager` becomes a `zombie`.

If `tid` task was already recorded as `successful`, you should *merely* see the following INFO message in the logs:

```
INFO Ignoring task-finished event for [id] in stage [id] because
task [index] has already completed successfully
```

Ultimately, `handleSuccessfulTask` attempts to mark the `Taskset` finished.

Attempting to Mark TaskSet Finished — `maybeFinishTaskSet` Internal Method

```
maybeFinishTaskSet(): Unit
```

`maybeFinishTaskSet` notifies `TaskSchedulerImpl` that a `TaskSet` has finished when there are no other running tasks and the `TaskSetManager` is not in zombie state.

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.

Tip	In Spark shell with local master, <code>spark.task.maxFailures</code> is fixed to <code>1</code> and you need to use <code>local-with-retries master</code> 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

A `TaskSetManager` is in **zombie** state when all tasks in a taskset have completed successfully (regardless of the number of task attempts), or if the taskset has been [aborted](#).

While in zombie state, a `TaskSetManager` can launch no new tasks and responds with no `TaskDescription` to `resourceOffers`.

A `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 — `abort` Method

```
abort(message: String, exception: Option[Throwable] = None): Unit
```

`abort` informs `DAGScheduler` that the `TaskSet` has been aborted.

Caution	FIXME image with DAGScheduler call
---------	------------------------------------

The `TaskSetManager` enters `zombie state`.

Finally, `abort` attempts to mark the `TaskSet` finished.

Checking Available Memory For Task Result — `canFetchMoreResults` Method

```
canFetchMoreResults(size: Long): Boolean
```

`canFetchMoreResults` checks whether there is enough memory to fetch the result of a task.

Internally, `canFetchMoreResults` increments the internal `totalResultSize` with the input `size` which is the result of a task. It also increments the internal `calculatedTasks`.

If the current internal `totalResultSize` is bigger than `spark.driver.maxResultSize` the following ERROR message is printed out to the logs:

```
ERROR TaskSetManager: Total size of serialized results of [calculatedTasks] tasks ([totalResultSize]) is bigger than spark.driver.maxResultSize ([maxResultSize])
```

The current `TaskSet` is `aborted` and `canFetchMoreResults` returns `false`.

Otherwise, `canFetchMoreResults` returns `true`.

Note	<code>canFetchMoreResults</code> is used in <code>TaskResultGetter.enqueueSuccessfulTask</code> only.
------	---

Creating TaskSetManager Instance

`TaskSetManager` takes the following when created:

- `TaskSchedulerImpl`
- `TaskSet` that the `TaskSetManager` manages scheduling for
- Acceptable number of task failure, i.e. how many times a `single task can fail` before an entire `TaskSet` gets aborted.
- (optional) `BlacklistTracker`
- `Clock` (defaults to `SystemClock`)

`TaskSetManager` initializes the internal registries and counters.

Note

`maxTaskFailures` is `1` for local run mode, `maxFailures` for Spark local-with-retries, and `spark.task.maxFailures` property for Spark local-cluster and Spark with cluster managers (Spark Standalone, Mesos and YARN).

`TaskSetManager` requests the current epoch from `MapOutputTracker` and sets it on all tasks in the taskset.

Note

`TaskSetManager` uses `TaskSchedulerImpl` (that was given when `created`) to access the current `MapOutputTracker`.

You should see the following DEBUG in the logs:

```
DEBUG Epoch for [taskSet]: [epoch]
```

Caution

`FIXME` Why is the epoch important?

Note

`TaskSetManager` requests `MapOutputTracker` from `TaskSchedulerImpl` which is likely for unit testing only since `MapOutputTracker` is available using `SparkEnv`.

`TaskSetManager` adds the tasks as pending execution (in reverse order from the highest partition to the lowest).

Caution

`FIXME` Why is reverse order important? The code says it's to execute tasks with low indices first.

Getting Notified that Task Failed — `handleFailedTask` Method

```
handleFailedTask(
  tid: Long,
  state: TaskState,
  reason: TaskFailedReason): Unit
```

`handleFailedTask` finds `TaskInfo` of `tid` task in `taskInfos` internal registry and simply quits if the task is already marked as failed or killed.

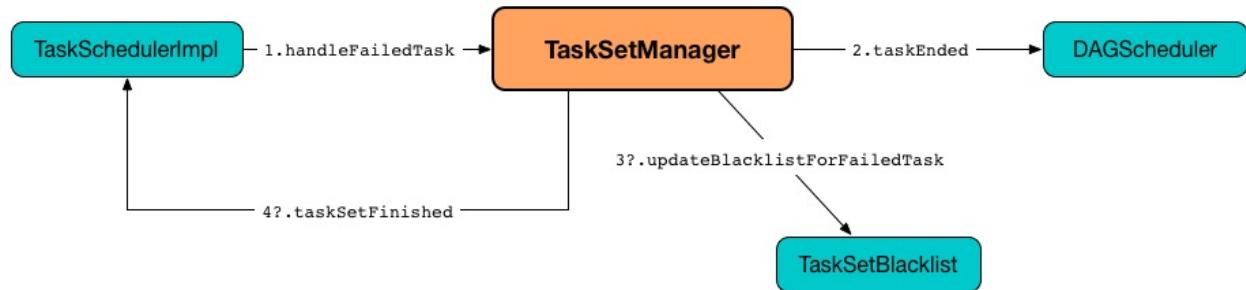


Figure 2. TaskSetManager Gets Notified that Task Has Failed

Note

`handleFailedTask` is executed after `TaskSchedulerImpl` has been informed that `tid` task failed or an executor was lost. In either case, tasks could not finish successfully or could not report their status back.

`handleFailedTask` unregisters `tid` task from the internal registry of running tasks and then marks the corresponding `TaskInfo` as finished (passing in the input `state`).

`handleFailedTask` decrements the number of the running copies of `tid` task (in `copiesRunning` internal registry).

Note

With speculative execution of tasks enabled, there can be many copies of a task running simultaneously.

`handleFailedTask` uses the following pattern as the reason of the failure:

```
Lost task [id] in stage [taskSetId] (TID [tid], [host], executor [executorId]): [reason]
```

`handleFailedTask` then calculates the failure exception per the input `reason` (follow the links for more details):

- [FetchFailed](#)
- [ExceptionFailure](#)
- [ExecutorLostFailure](#)
- other [TaskFailedReasons](#)

Note	Description of how the final failure exception is "computed" was moved to respective sections below to make the reading slightly more pleasant and comprehensible.
------	--

`handleFailedTask` informs `DAGScheduler` that `tid` task has ended (passing on the `Task` instance from `tasks` internal registry, the input `reason`, `null` result, calculated `accumUpdates` per failure, and the `TaskInfo`).

Important	This is the moment when <code>TaskSetManager</code> informs <code>DAGScheduler</code> that a task has ended.
-----------	--

If `tid` task has already been marked as completed (in `successful` internal registry) you should see the following INFO message in the logs:

```
INFO Task [id] in stage [id] (TID [tid]) failed, but the task will not be re-executed (either because the task failed with a shuffle data fetch failure, so the previous stage needs to be re-run, or because a different copy of the task has already succeeded).
```

Tip	Read up on Speculative Execution of Tasks to find out why a single task could be executed multiple times.
-----	---

If however `tid` task was not recorded as `completed`, `handleFailedTask` records it as `pending`.

If the `TaskSetManager` is not a `zombie` and the task failed `reason` should be counted towards the maximum number of times the task is allowed to fail before the stage is aborted (i.e. `TaskFailedReason.countTowardsTaskFailures` attribute is enabled), the optional `TaskSetBlacklist` is notified (passing on the host, executor and the task's index). `handleFailedTask` then increments the `number of failures` for `tid` task and checks if the number of failures is equal or greater than the `allowed number of task failures per TaskSet` (as defined when the `TaskSetManager` was created).

If so, i.e. the number of task failures of `tid` task reached the maximum value, you should see the following ERROR message in the logs:

```
ERROR Task [id] in stage [id] failed [maxTaskFailures] times; aborting job
```

And `handleFailedTask` aborts the `TaskSet` with the following message and then quits:

```
Task [index] in stage [id] failed [maxTaskFailures] times, most recent failure: [failureReason]
```

In the end (except when the number of failures of `tid` task grew beyond the acceptable number), `handleFailedTask` attempts to mark the `TaskSet` as finished.

Note

`handleFailedTask` is used when `TaskSchedulerImpl` is informed that a task has failed or when `TaskSetManager` is informed that an executor has been lost.

FetchFailed TaskFailedReason

For `FetchFailed` you should see the following WARN message in the logs:

```
WARN Lost task [id] in stage [id] (TID [tid], [host], executor [id]): [reason]
```

Unless `tid` has already been marked as successful (in `successful` internal registry), it becomes so and the number of successful tasks in `taskSet` gets increased.

The `TaskSetManager` enters [zombie state](#).

The failure exception is empty.

ExceptionFailure TaskFailedReason

For `ExceptionFailure`, `handleFailedTask` checks if the exception is of type `NotSerializableException`. If so, you should see the following ERROR message in the logs:

```
ERROR Task [id] in stage [id] (TID [tid]) had a not serializable result: [description]
; not retrying
```

And `handleFailedTask` aborts the `TaskSet` and then quits.

Otherwise, if the exception is not of type `NotSerializableException`, `handleFailedTask` accesses accumulators and calculates whether to print the WARN message (with the failure reason) or the INFO message.

If the failure has already been reported (and is therefore a duplication), `spark.logging.exceptionPrintInterval` is checked before reprinting the duplicate exception in its entirety.

For full printout of the `ExceptionFailure`, the following WARN appears in the logs:

```
WARN Lost task [id] in stage [id] (TID [tid], [host], executor [id]): [reason]
```

Otherwise, the following INFO appears in the logs:

```
INFO Lost task [id] in stage [id] (TID [tid]) on [host], executor [id]: [className] ([description]) [duplicate [dupCount]]
```

The exception in `ExceptionFailure` becomes the failure exception.

ExecutorLostFailure TaskFailedReason

For `ExecutorLostFailure` if not `exitCausedByApp`, you should see the following INFO 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.
```

The failure exception is empty.

Other TaskFailedReasons

For the other TaskFailedReasons, you should see the following WARN message in the logs:

```
WARN Lost task [id] in stage [id] (TID [tid], [host], executor [id]): [reason]
```

The failure exception is empty.

Registering Task As Pending Execution (Per Preferred Locations) — addPendingTask Internal Method

```
addPendingTask(index: Int): Unit
```

`addPendingTask` registers a `index` task in the pending-task lists that the task should be eventually scheduled to (per its preferred locations).

Internally, `addPendingTask` takes the [preferred locations of the task](#) (given `index`) and registers the task in the internal pending-task registries for every preferred location:

- `pendingTasksForExecutor` when the `TaskLocation` is `ExecutorCacheTaskLocation`.

- `pendingTasksForHost` for the hosts of a `TaskLocation`.
- `pendingTasksForRack` for the racks from `TaskSchedulerImpl` per the host (of a `TaskLocation`).

For a `TaskLocation` being `HDFSCacheTaskLocation`, `addPendingTask` requests `TaskSchedulerImpl` for the executors on the host (of a preferred location) and registers the task in `pendingTasksForExecutor` for every executor (if available).

You should see the following INFO message in the logs:

```
INFO Pending task [index] has a cached location at [host] , where there are executors [executors]
```

When `addPendingTask` could not find executors for a `HDFSCacheTaskLocation` preferred location, you should see the following DEBUG message in the logs:

```
DEBUG Pending task [index] has a cached location at [host] , but there are no executors alive there.
```

If the task has no location preferences, `addPendingTask` registers it in `pendingTasksWithNoPrefs`.

`addPendingTask` always registers the task in `allPendingTasks`.

Note	<code>addPendingTask</code> is used immediately when <code>TaskSetManager</code> is created and later when handling a task failure or lost executor.
------	--

Re-enqueuing ShuffleMapTasks (with no ExternalShuffleService) and Reporting All Running Tasks on Lost Executor as Failed — `executorLost` Method

```
executorLost(execId: String, host: String, reason: ExecutorLossReason): Unit
```

`executorLost` re-enqueues all the `ShuffleMapTasks` that have completed already on the lost executor (when `external shuffle service` is not in use) and reports all currently-running tasks on the lost executor as failed.

Note	<code>executorLost</code> is a part of the <code>Schedulable</code> contract that <code>TaskSchedulerImpl</code> uses to inform <code>TaskSetManagers</code> about lost executors.
------	--

Note

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`.

Internally, `executorLost` first checks whether the `tasks` are `ShuffleMapTasks` and whether an `external shuffle service` is enabled (that could serve the map shuffle outputs in case of failure).

Note

`executorLost` checks out the first task in `tasks` as it is assumed the other belong to the same stage. If the task is a `ShuffleMapTask`, the entire `TaskSet` is for a `ShuffleMapStage`.

Note

`executorLost` uses `SparkEnv` to access the current `BlockManager` and finds out whether an `external shuffle service` is enabled or not (that is controlled using `spark.shuffle.service.enabled` property).

If `executorLost` is indeed due to an executor lost that executed tasks for a `ShuffleMapStage` (that this `TaskSetManager` manages) and no external shuffle server is enabled, `executorLost` finds `all the tasks` that were scheduled on this lost executor and marks the ones that were already successfully completed as not executed yet.

Note

`executorLost` uses records every tasks on the lost executor in `successful` (as `false`) and decrements `[copiesRunning copiesRunning]`, and `tasksSuccessful` for every task.

`executorLost` registers every task as pending execution (per preferred locations) and informs `DAGScheduler` that the tasks (on the lost executor) have ended (with Resubmitted reason).

Note

`executorLost` uses `TaskSchedulerImpl` to access the `DAGScheduler`. `TaskSchedulerImpl` is given when the `TaskSetManager` was created.

Regardless of whether this `TaskSetManager` manages `ShuffleMapTasks` or not (it could also manage `ResultTasks`) and whether the external shuffle service is used or not, `executorLost` finds all `currently-running tasks` on this lost executor and `reports them as failed` (with the task state `FAILED`).

Note

`executorLost` finds out if the reason for the executor lost is due to application fault, i.e. assumes `ExecutorExited`'s exit status as the indicator, `ExecutorKilled` for non-application's fault and any other reason is an application fault.

`executorLost` recomputes locality preferences.

Recomputing Task Locality Preferences

— `recomputeLocality` Method

```
recomputeLocality(): Unit
```

`recomputeLocality` recomputes the internal caches: `myLocalityLevels`, `localityWaits` and `currentLocalityIndex`.

Caution	<code>FIXME</code> But why are the caches important (and have to be recomputed)?
---------	---

`recomputeLocality` records the current `TaskLocality` level of this `TaskSetManager` (that is `currentLocalityIndex` in `myLocalityLevels`).

Note	TaskLocality is one of <code>PROCESS_LOCAL</code> , <code>NODE_LOCAL</code> , <code>NO_PREF</code> , <code>RACK_LOCAL</code> and <code>ANY</code> values.
------	---

`recomputeLocality` computes locality levels (for scheduled tasks) and saves the result in `myLocalityLevels` internal cache.

`recomputeLocality` computes `localityWaits` (by finding locality wait for every locality level in `myLocalityLevels` internal cache).

In the end, `recomputeLocality` `getLocalityIndex` of the previous locality level and records it in `currentLocalityIndex`.

Note	<code>recomputeLocality</code> is used when <code>TaskSetManager</code> gets notified about status change in executors, i.e. when an executor is <code>lost</code> or <code>added</code> .
------	--

`recomputeLocality` is used when `TaskSetManager` gets notified about status change in executors, i.e. when an executor is `lost` or `added`.

Computing Locality Levels (for Scheduled Tasks)

— `computeValidLocalityLevels` Internal Method

```
computeValidLocalityLevels(): Array[TaskLocality]
```

`computeValidLocalityLevels` computes valid locality levels for tasks that were registered in corresponding registries per locality level.

Note	<code>TaskLocality</code> is a task locality preference and can be the most localized <code>PROCESS_LOCAL</code> , <code>NODE_LOCAL</code> through <code>NO_PREF</code> and <code>RACK_LOCAL</code> to <code>ANY</code> .
------	---

`TaskLocality` is a task locality preference and can be the most localized `PROCESS_LOCAL`, `NODE_LOCAL` through `NO_PREF` and `RACK_LOCAL` to `ANY`.

Table 2. TaskLocalities and Corresponding Internal Registries

TaskLocality	Internal Registry
PROCESS_LOCAL	pendingTasksForExecutor
NODE_LOCAL	pendingTasksForHost
NO_PREF	pendingTasksWithNoPrefs
RACK_LOCAL	pendingTasksForRack

`computeValidLocalityLevels` walks over every internal registry and if it is not empty **computes locality wait** for the corresponding `TaskLocality` and proceeds with it only when the locality wait is not `0`.

For `TaskLocality` with pending tasks, `computeValidLocalityLevels` asks `TaskSchedulerImpl` whether there is at least one executor alive (for `PROCESS_LOCAL`, `NODE_LOCAL` and `RACK_LOCAL`) and if so registers the `TaskLocality`.

Note	<code>computeValidLocalityLevels</code> uses <code>TaskSchedulerImpl</code> that was given when <code>TaskSetManager</code> was created.
------	--

`computeValidLocalityLevels` always registers `ANY` task locality level.

In the end, you should see the following DEBUG message in the logs:

DEBUG TaskSetManager: Valid locality levels for [taskSet]: [comma-separated levels]

Note	<code>computeValidLocalityLevels</code> is used when <code>TaskSetManager</code> is created and later to recompute locality.
------	--

Finding Locality Wait — `getLocalityWait` Internal Method

getLocalityWait(level: TaskLocality): Long
--

`getLocalityWait` finds **locality wait** (in milliseconds) for a given `TaskLocality`.

`getLocalityWait` uses `spark.locality.wait` (default: `3s`) when the `TaskLocality`-specific property is not defined or `0` for `NO_PREF` and `ANY`.

Note	<code>NO_PREF</code> and <code>ANY</code> task localities have no locality wait.
------	--

Table 3. TaskLocalities and Corresponding Spark Properties

TaskLocality	Spark Property
PROCESS_LOCAL	spark.locality.wait.process
NODE_LOCAL	spark.locality.wait.node
RACK_LOCAL	spark.locality.wait.rack

Note	<code>getLocalityWait</code> is used when <code>TaskSetManager</code> calculates <code>localityWaits</code> , computes locality levels (for scheduled tasks) and recomputes locality preferences.
------	---

Settings

Table 4. Spark Properties

Spark Property	Default Value	Description
<code>spark.driver.maxResultSize</code>	<code>1g</code>	The maximum size of the task results in a <code>TaskSet</code> . If the value smaller than <code>1m</code> or <code>1048576</code> (<code>1024 * 1024</code>) it is considered <code>0</code> . Used when <code>TaskSetManager</code> checks available memory for task result and <code>Utils.getMaxResultSize</code> .
<code>spark.scheduler.executorTaskBlacklistTime</code>	<code>0L</code>	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.
<code>spark.logging.exceptionPrintInterval</code>	<code>10000</code>	How frequently to report duplicate exceptions in full (in millis).
<code>spark.locality.wait</code>	<code>3s</code>	For locality-aware delay scheduling for <code>PROCESS_LOCAL</code> , <code>NODE_LOCAL</code> , and <code>RACK_LOCAL</code> <code>TaskLocalities</code> when locality-specific settings not set.
<code>spark.locality.wait.process</code>	The value of <code>spark.locality.wait</code>	Scheduling delay for <code>PROCESS_LOCAL</code> <code>TaskLocality</code>
<code>spark.locality.wait.node</code>	The value of <code>spark.locality.wait</code>	Scheduling delay for <code>NODE_LOCAL</code> <code>TaskLocality</code>
<code>spark.locality.wait.rack</code>	The value of <code>spark.locality.wait</code>	Scheduling delay for <code>RACK_LOCAL</code> <code>TaskLocality</code>

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	schedulableQueue is java.util.concurrent.ConcurrentLinkedQueue.
------	---

- 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	FIXME 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`.

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 TaskSchedulerImpl is initialized.
------	--

Note	The TaskScheduler Contract and Schedulable Contract both require that their entities have <code>rootPool</code> of type <code>Pool</code> .
------	---

increaseRunningTasks Method

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

decreaseRunningTasks Method

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

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 SchedulingMode .
-----	---

Note	<code>taskSetSchedulingAlgorithm</code> is used in getSortedTaskSetQueue .
------	--

Getting TaskSetManagers Sorted

— `getSortedTaskSetQueue` Method

Note	<code>getSortedTaskSetQueue</code> is part of the Schedulable Contract .
------	--

`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`.

addSchedulable Method

Note	<code>addSchedulable</code> is part of the Schedulable Contract .
------	---

`addSchedulable` adds a `Schedulable` to the `schedulableQueue` and `schedulableNameToSchedulable`.

More importantly, it sets the `Schedulable` entity's [parent](#) to itself.

removeSchedulable Method

Note	<code>removeSchedulable</code> is part of the Schedulable Contract .
------	--

`removeSchedulable` removes a `Schedulable` from the `schedulableQueue` and `schedulableNameToSchedulable`.

Note	<code>removeSchedulable</code> is the opposite to <code>addSchedulable</code> method.
------	---

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	priority and stageId are part of Scheduled Contract .
------	---

Caution	FIXME A picture is worth a thousand words. How to picture the algorithm?
---------	--

FairSchedulingAlgorithm

FairSchedulingAlgorithm is a scheduling algorithm that compares Schedulables by their minShare , runningTasks , and weight .

Note	minShare , runningTasks , and weight are part of Scheduled Contract .
------	---

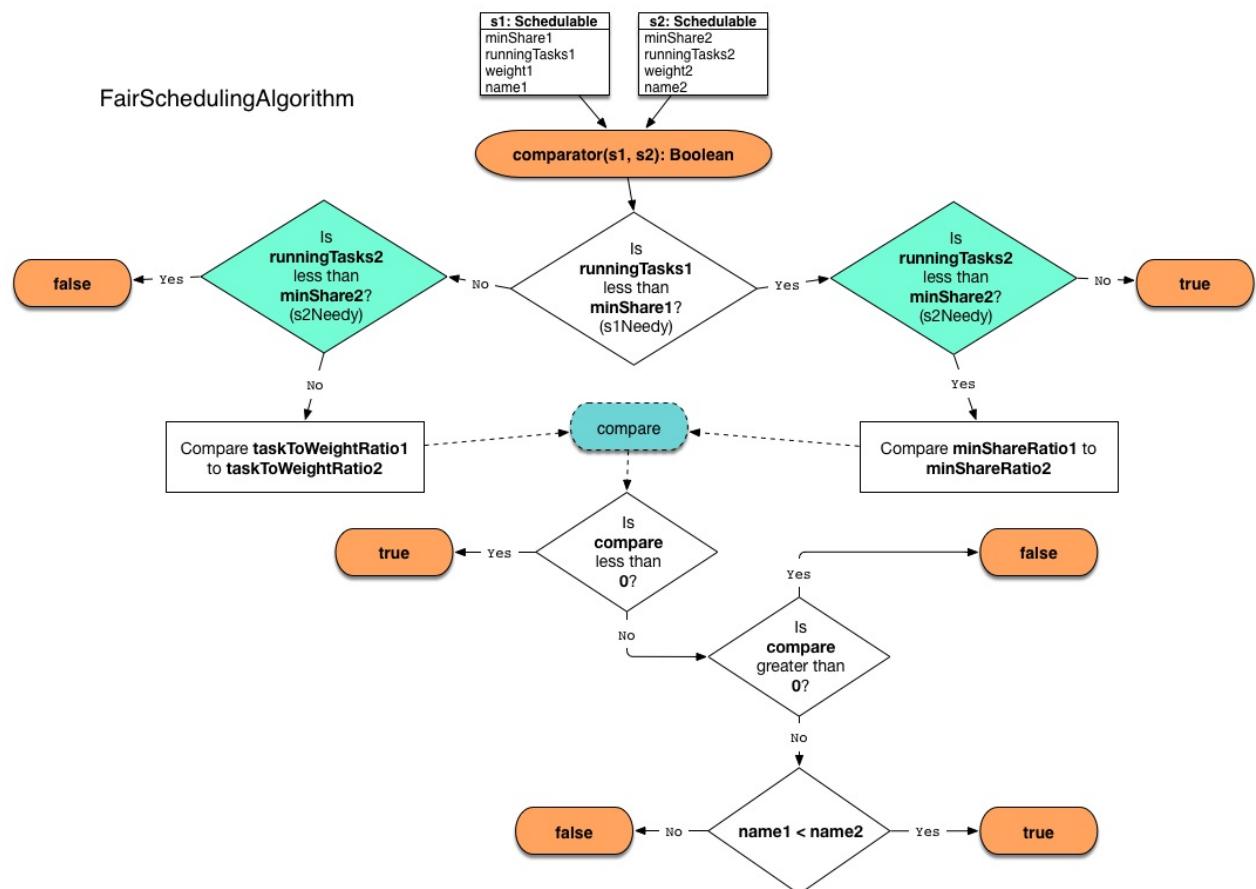


Figure 1. FairSchedulingAlgorithm

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 org.apache.spark.scheduler.SchedulableBuilder .
------	---

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 holds a single `Pool` (that is given when `FIFOSchedulableBuilder` is created).

Note

`FIFOSchedulableBuilder` is the default `SchedulableBuilder` for `TaskSchedulerImpl`.

Note

When `FIFOSchedulableBuilder` is created, the `TaskSchedulerImpl` passes its own `rootPool` (a part of [TaskScheduler Contract](#)).

`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`.

Creating FIFOSchedulableBuilder Instance

`FIFOSchedulableBuilder` takes the following when created:

- `rootPool` `Pool`

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.

Tip	<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 Logging.</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).

Note	<p><code>buildPools</code> is part of the SchedulableBuilder Contract.</p>
Tip	<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`).

Note	<p><code>addTaskSetManager</code> is part of the SchedulableBuilder Contract.</p>
Note	<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 to group jobs in logical groups. This mechanism is used by `FairSchedulableBuilder` to watch for `spark.scheduler.pool` property to group jobs from threads 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` Spark Property

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 <code>SchedulingMode</code> policies only <code>FIFO</code> and <code>FAIR</code> 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 <code>root pool</code> is a <code>Schedulable</code> . Refer to Schedulable .
------	---

Monitoring FAIR Scheduling Mode using Spark UI

Caution	FIXME Describe me...
---------	--------------------------------------

TaskInfo

`TaskInfo` is information about a running task attempt inside a [TaskSet](#).

`TaskInfo` is created when:

- `TaskSetManager` dequeues a task for execution (given resource offer) (and records the task as running)
- TaskUIData does `dropInternalAndSQLAccumulables`
- `JsonProtocol` re-creates a task details from JSON

Note	Back then, at the commit 63051dd2bcc4bf09d413ff7cf89a37967edc33ba, when <code>TaskInfo</code> was first merged to Apache Spark on 07/06/12, <code>TaskInfo</code> was part of <code>spark.scheduler.mesos</code> package—note "Mesos" in the name of the package that shows how much Spark and Mesos influenced each other at that time.
------	--

Table 1. TaskInfo's Internal Registries and Counters

Name	Description
<code>finishTime</code>	Time when <code>TaskInfo</code> was marked as finished. Used when... FIXME

Creating TaskInfo Instance

`TaskInfo` takes the following when created:

- Task ID
- Index of the task within its [TaskSet](#) that may not necessarily be the same as the ID of the RDD partition that the task is computing.
- Task attempt ID
- Time when the task was dequeued for execution
- Executor that has been offered (as a resource) to run the task
- Host of the [executor](#)
- [TaskLocality](#), i.e. locality preference of the task
- Flag whether a task is speculative or not

`TaskInfo` initializes the [internal registries and counters](#).

Marking Task As Finished (Successfully or Not)

— markFinished Method

```
markFinished(state: TaskState, time: Long = System.currentTimeMillis): Unit
```

markFinished records the input time as finishTime.

markFinished marks TaskInfo as failed when the input state is FAILED or killed for state being KILLED .

Note

markFinished is used when TaskSetManager is notified that a task has finished successfully or failed.

TaskSchedulerImpl — Default TaskScheduler

`TaskSchedulerImpl` is the default [TaskScheduler](#).

`TaskSchedulerImpl` can schedule tasks for multiple types of cluster managers by means of [SchedulerBackends](#).

When a Spark application starts (and so an instance of [SparkContext](#) is created) `TaskSchedulerImpl` with a [SchedulerBackend](#) and [DAGScheduler](#) are created and soon started.

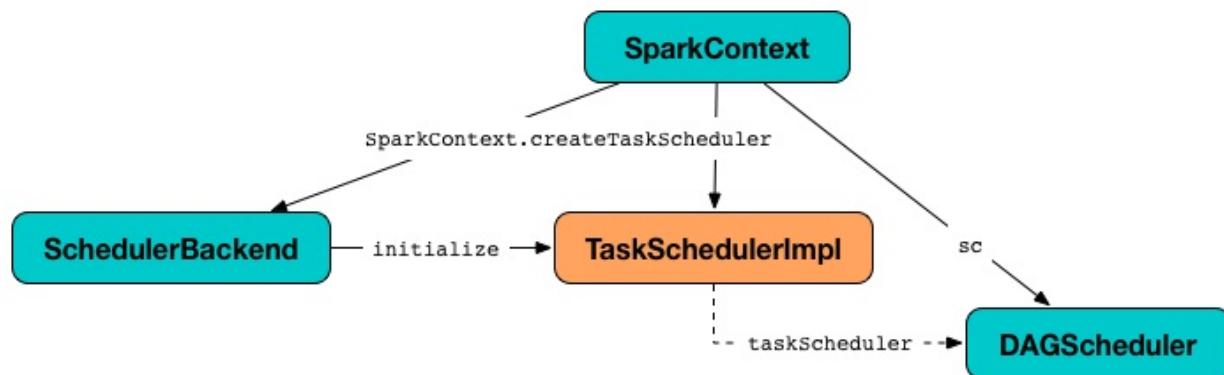


Figure 1. `TaskSchedulerImpl` and Other Services

`TaskSchedulerImpl` generates tasks for executor resource offers.

`TaskSchedulerImpl` can track racks per host and port (that however is only used with Hadoop YARN cluster manager).

Using `spark.scheduler.mode` setting you can select the scheduling policy.

`TaskSchedulerImpl` submits tasks using [SchedulableBuilders](#).

Table 1. `TaskSchedulerImpl`'s Internal Registries and Counters

Name	Description
backend	<code>SchedulerBackend</code> Set when <code>TaskSchedulerImpl</code> is initialized.
dagScheduler	<code>DAGScheduler</code> Used when... FIXME
executorIdToHost	Lookup table of hosts per executor. Used when... FIXME
	Lookup table of running tasks per executor.

	Used when... FIXME
<code>executorIdToTaskCount</code>	Lookup table of the number of running tasks by executor .
<code>executorsByHost</code>	Collection of executors per host
<code>hasLaunchedTask</code>	Flag... FIXME Used when... FIXME
<code>hostToExecutors</code>	Lookup table of executors per hosts in a cluster. Used when... FIXME
<code>hostsByRack</code>	Lookup table of hosts per rack. Used when... FIXME
<code>nextTaskId</code>	The next task id counting from <code>0</code> . Used when <code>TaskSchedulerImpl ...</code>
<code>rootPool</code>	Schedulable Pool Used when <code>TaskSchedulerImpl ...</code>
<code>schedulingMode</code>	SchedulingMode Used when <code>TaskSchedulerImpl ...</code>
<code>taskSetsByStageIdAndAttempt</code>	Lookup table of TaskSet by stage and attempt ids.
<code>taskIdToExecutorId</code>	Lookup table of executor by task id.
<code>taskIdToTaskSetManager</code>	Registry of active TaskSetManager per task id.
Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging levels for <code>org.apache.spark.scheduler.TaskSchedulerImpl</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.TaskSchedulerImpl=DEBUG</pre> <p>Refer to Logging.</p>

Finding Unique Identifier of Spark Application — `applicationId` Method

```
applicationId(): String
```

Note

`applicationId` is a part of [TaskScheduler contract](#) to find the Spark application's id.

`applicationId` simply request [SchedulerBackend](#) for the [Spark application's id](#).

nodeBlacklist Method

Caution

[FIXME](#)

cleanupTaskState Method

Caution

[FIXME](#)

newTaskId Method

Caution

[FIXME](#)

getExecutorsAliveOnHost Method

Caution

[FIXME](#)

isExecutorAlive Method

Caution

[FIXME](#)

hasExecutorsAliveOnHost Method

Caution

[FIXME](#)

hasHostAliveOnRack Method

Caution

[FIXME](#)

executorLost Method

Caution

FIXME

mapOutputTracker

Caution

FIXME

starvationTimer

Caution

FIXME

executorHeartbeatReceived Method

```
executorHeartbeatReceived(
  execId: String,
  accumUpdates: Array[(Long, Seq[AccumulatorV2[_, _]])],
  blockManagerId: BlockManagerId): Boolean
```

executorHeartbeatReceived is...

Caution

FIXME

Note

executorHeartbeatReceived is a part of the [TaskScheduler Contract](#).

Cancelling Tasks for Stage — cancelTasks Method

```
cancelTasks(stageId: Int, interruptThread: Boolean): Unit
```

Note

cancelTasks is a part of [TaskScheduler contract](#).

cancelTasks cancels all tasks submitted for execution in a stage stageId .

Note

cancelTasks is used exclusively when [DAGScheduler cancels a stage](#).

handleSuccessfulTask Method

```
handleSuccessfulTask(
    taskSetManager: TaskSetManager,
    tid: Long,
    taskResult: DirectTaskResult[_]): Unit
```

`handleSuccessfulTask` simply forwards the call to the input `taskSetManager` (passing `tid` and `taskResult`).

Note

`handleSuccessfulTask` is called when `TaskSchedulerGetter` has managed to deserialize the task result of a task that finished successfully.

handleTaskGettingResult Method

```
handleTaskGettingResult(taskSetManager: TaskSetManager, tid: Long): Unit
```

`handleTaskGettingResult` simply forwards the call to the `taskSetManager`.

Note

`handleTaskGettingResult` is used to inform that `TaskResultGetter` enqueues a successful task with `IndirectTaskResult` task result (and so is about to fetch a remote block from a `BlockManager`).

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 `spark.scheduler.mode` setting to select the scheduling policy.

Tracking Racks per Hosts and Ports — `getRackForHost` Method

```
getRackForHost(value: String): Option[String]
```

`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 overriden by the YARN-specific TaskScheduler YarnScheduler .
------	--

`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` Instance

`TaskSchedulerImpl` takes the following when created:

- `SparkContext`
- Acceptable number of task failures
- optional `BlacklistTracker`
- optional `isLocal` flag to differentiate between local and cluster run modes (defaults to `false`)

`TaskSchedulerImpl` initializes the `internal registries and counters`.

Note	There is another <code>TaskSchedulerImpl</code> constructor that requires a <code>SparkContext</code> object only and sets <code>maxTaskFailures</code> to <code>spark.task.maxFailures</code> or, if not set, defaults to <code>4</code> .
------	---

`TaskSchedulerImpl` sets `schedulingMode` to the value of `spark.scheduler.mode` setting (defaults to `FIFO`).

Note	<code>schedulingMode</code> is part of TaskScheduler Contract .
------	---

Failure to set `schedulingMode` results in a `SparkException` :

```
Unrecognized spark.scheduler.mode: [schedulingModeConf]
```

Ultimately, `TaskSchedulerImpl` creates a [TaskResultGetter](#).

Saving SchedulerBackend and Building Schedulable Pools (aka Initializing `TaskSchedulerImpl`) — `initialize` Method

```
initialize(backend: SchedulerBackend): Unit
```

`initialize` initializes `TaskSchedulerImpl`.

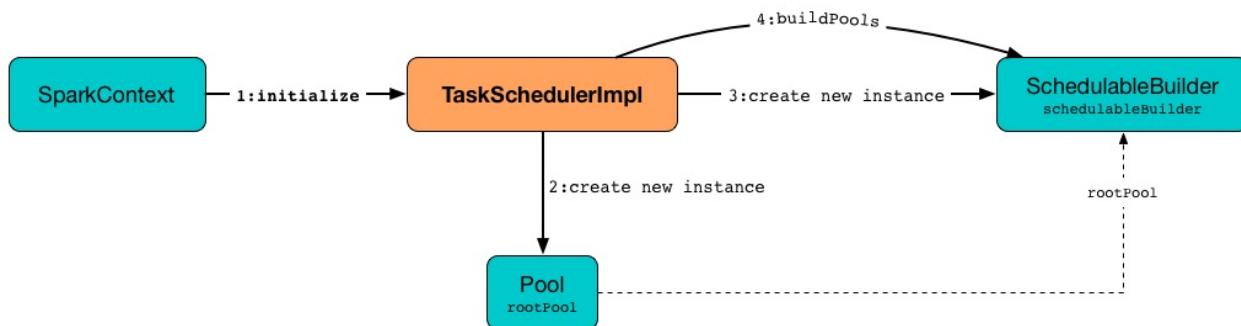


Figure 2. `TaskSchedulerImpl` initialization

`initialize` saves the input [SchedulerBackend](#).

`initialize` then sets [schedulablePool](#) as an empty-named [Pool](#) (passing in [SchedulingMode](#), `initMinShare` and `initWeight` as `0`).

Note	SchedulingMode is defined when <code>TaskSchedulerImpl</code> is created.
------	---

Note	schedulingMode and <code>rootPool</code> are a part of TaskScheduler Contract .
------	---

`initialize` sets [SchedulableBuilder](#) (based on [SchedulingMode](#)):

- [FIFOSchedulableBuilder](#) for `FIFO` scheduling mode
- [FairSchedulableBuilder](#) for `FAIR` scheduling mode

`initialize` requests [SchedulableBuilder](#) to build pools.

Caution	FIXME Why are <code>rootPool</code> and <code>schedulableBuilder</code> created only now? What do they need that it is not available when <code>TaskSchedulerImpl</code> is created?
---------	--

Note	<code>initialize</code> is called while SparkContext is created and creates SchedulerBackend and TaskScheduler .
------	--

Starting TaskSchedulerImpl — start Method

As part of [initialization of a sparkContext](#), TaskSchedulerImpl is started (using `start` from the [TaskScheduler Contract](#)).

```
start(): Unit
```

`start` starts the [scheduler backend](#).

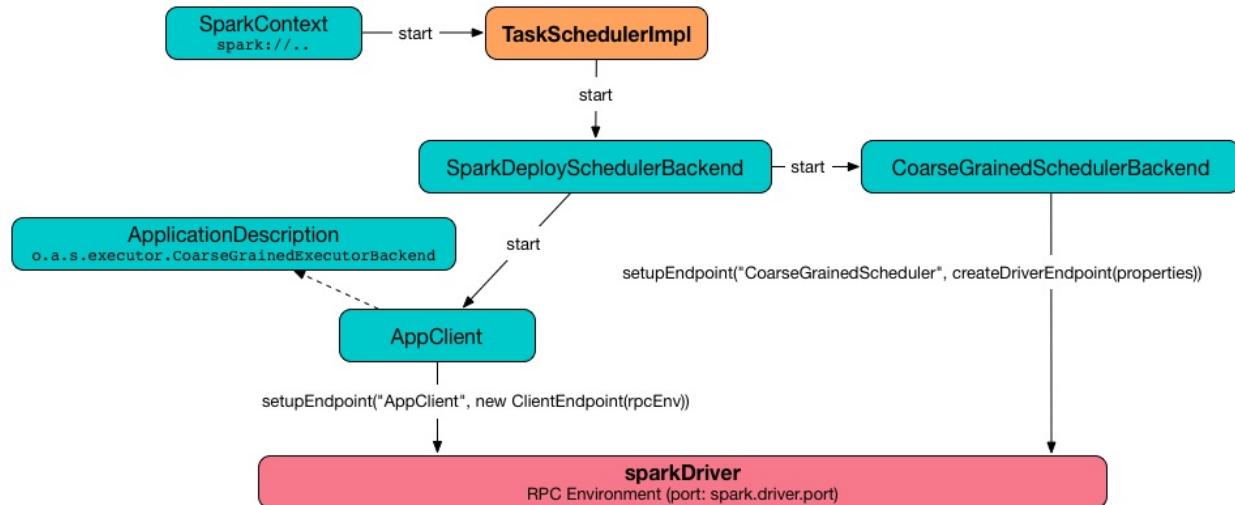


Figure 3. Starting `TaskSchedulerImpl` in Spark Standalone

`start` also starts [task-scheduler-speculation executor service](#).

Requesting TaskResultGetter to Enqueue Task — statusUpdate Method

```
statusUpdate(tid: Long, state: TaskState, serializedData: ByteBuffer): Unit
```

`statusUpdate` finds `TaskSetManager` for the input `tid` task (in `taskIdToTaskSetManager`).

When `state` is `LOST`, `statusUpdate` ...[FIXME](#)

Note

`TaskState.LOST` is only used by the deprecated Mesos fine-grained scheduling mode.

When `state` is one of the [finished states](#), i.e. `FINISHED`, `FAILED`, `KILLED` or `LOST`, `statusUpdate` [cleanupTaskState](#) for the input `tid`.

`statusUpdate` requests `TaskSetManager` to unregister `tid` from running tasks.

`statusUpdate` requests `TaskResultGetter` to schedule an asynchronous task to deserialize the task result (and notify `TaskSchedulerImpl back`) for `tid` in `FINISHED` state and schedule an asynchronous task to deserialize `TaskFailedReason` (and notify `TaskSchedulerImpl back`) for `tid` in the other finished states (i.e. `FAILED`, `KILLED`, `LOST`).

If a task is in `LOST` state, `statusUpdate` notifies `DAGScheduler` that the executor was lost (with `SlaveLost` and the reason `Task [tid] was lost, so marking the executor as lost as well.`) and requests `SchedulerBackend` to revive offers.

In case the `TaskSetManager` for `tid` could not be found (in `taskIdToTaskSetManager` registry), you should see the following ERROR message 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)
```

Any exception is caught and reported as ERROR message in the logs:

```
ERROR Exception in statusUpdate
```

Caution	<code>FIXME</code> image with scheduler backends calling <code>TaskSchedulerImpl.statusUpdate</code> .
---------	--

Note	<code>statusUpdate</code> is used when <code>SchedulerBackends</code> , i.e. <code>CoarseGrainedSchedulerBackend</code> , <code>LocalSchedulerBackend</code> and <code>MesosFineGrainedSchedulerBackend</code> , inform about changes in task states.
------	---

task-scheduler-speculation Scheduled Executor Service — speculationScheduler Internal Attribute

`speculationScheduler` is a `java.util.concurrent.ScheduledExecutorService` with the name **task-scheduler-speculation** for speculative execution of tasks.

When `TaskSchedulerImpl` starts (in non-local run mode) with `spark.speculation` enabled, `speculationScheduler` is used to schedule `checkSpeculatableTasks` to execute periodically every `spark.speculation.interval` after the initial `spark.speculation.interval` passes.

`speculationScheduler` is shut down when `TaskSchedulerImpl` stops.

Checking for Speculatable Tasks — checkSpeculatableTasks Method

```
checkSpeculatableTasks(): Unit
```

`checkSpeculatableTasks` requests `rootPool` to check for speculatable tasks (if they ran for more than `100` ms) and, if there any, requests `SchedulerBackend` to revive offers.

Note

`checkSpeculatableTasks` is executed periodically as part of speculative execution of tasks.

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 submitting tasks through `TaskSetManager`.

Cleaning up After Removing Executor

— removeExecutor Internal Method

```
removeExecutor(executorId: String, reason: ExecutorLossReason): Unit
```

`removeExecutor` removes the `executorId` executor from the following 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 `removeExecutor` is called as part of `statusUpdate` and `executorLost`.

Intercepting Nearly-Completed SparkContext Initialization

— postStartHook Callback

`postStartHook` is a custom implementation of `postStartHook` from the `TaskSchedulerContract` 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`.

Stopping TaskSchedulerImpl — `stop` Method

```
stop(): Unit
```

`stop()` stops all the internal services, i.e. `task-scheduler-speculation` executor service, `SchedulerBackend`, `TaskResultGetter`, and `starvationTimer` timer.

Finding Default Level of Parallelism — `defaultParallelism` Method

```
defaultParallelism(): Int
```

Note

`defaultParallelism` is a part of `TaskScheduler contract` as a hint for sizing jobs.

`defaultParallelism` simply requests `SchedulerBackend` for the default level of parallelism.

Note

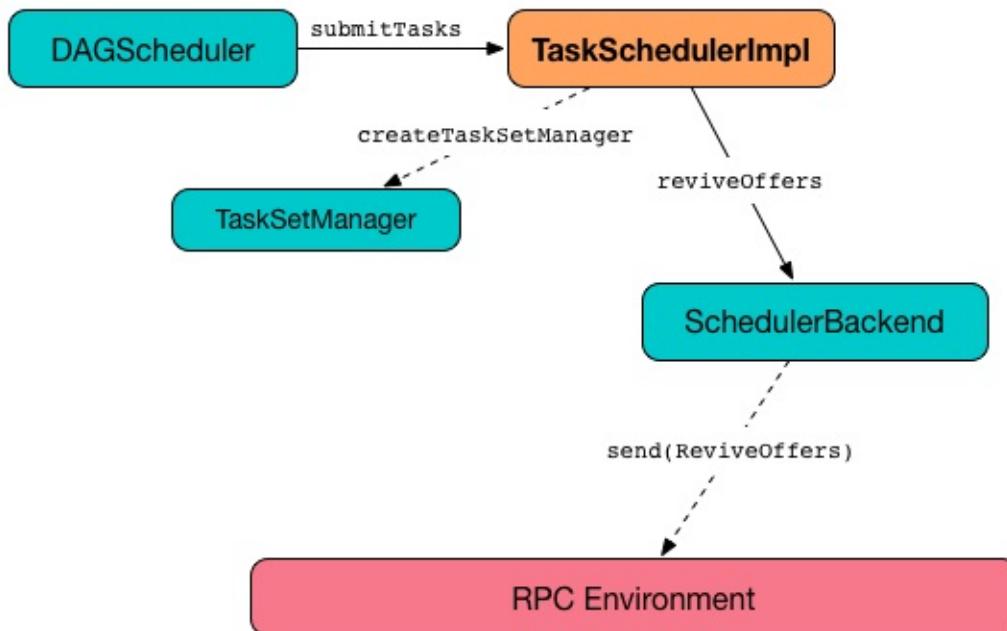
Default level of parallelism is a hint for sizing jobs that `SparkContext` uses to create RDDs with the right number of partitions when not specified explicitly.

Submitting Tasks for Execution (from TaskSet for Stage) — `submitTasks` Method

```
submitTasks(taskSet: TaskSet): Unit
```

Note

`submitTasks` is a part of `TaskScheduler Contract`.

Figure 4. `TaskSchedulerImpl.submitTasks`

When executed, you should see the following INFO message in the logs:

```
INFO TaskSchedulerImpl: Adding task set [id] with [count] tasks
```

`submitTasks` creates a `TaskSetManager` (for the input `taskSet` and acceptable number of task failures).

Note	<code>submitTasks</code> uses acceptable number of task failures that is defined when <code>TaskSchedulerImpl</code> is created.
------	--

`submitTasks` registers the `TaskSetManager` per stage and stage attempt id (in `taskSetsByStageIdAndAttempt`).

Note	The stage and the stage attempt id are attributes of a <code>TaskSet</code> .
------	---

Note	<code>submitTasks</code> assumes that only one <code>TaskSet</code> can be active for a <code>Stage</code> .
------	--

If there is more than one active `TaskSetManager` for the stage, `submitTasks` reports a `IllegalStateException` with the message:

```
more than one active taskSet for stage [stage]: [TaskSet ids]
```

Note	<code>TaskSetManager</code> is considered active when it is not a zombie . <code>submitTasks</code> adds the <code>TaskSetManager</code> to the <code>schedulable</code> root pool (available as <code>schedulableBuilder</code>).
------	---

Note

The `root pool` can be a single flat linked queue (in [FIFO scheduling mode](#)) or a hierarchy of pools of `Schedulables` (in [FAIR scheduling mode](#)).

`submitTasks` makes sure that the requested resources, i.e. CPU and memory, are assigned to the Spark application for a [non-local environment](#).

When `submitTasks` 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 `spark.starvation.timeout` passes, the internal `hasReceivedTask` flag becomes `true`.

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.

In the end, `submitTasks` requests the current `SchedulerBackend` to revive offers (available as `backend`).

Tip

Use `dag-scheduler-event-loop` thread to step through the code in a debugger.

Creating TaskSetManager — `createTaskSetManager` Method

```
createTaskSetManager(taskSet: TaskSet, maxTaskFailures: Int): TaskSetManager
```

`createTaskSetManager` creates a `TaskSetManager` (passing on the reference to `TaskSchedulerImpl`, the input `taskSet` and `maxTaskFailures`, and optional `BlacklistTracker`).

Note

`createTaskSetManager` uses the optional `BlacklistTracker` that is specified when `TaskSchedulerImpl` is created.

Note

`createTaskSetManager` is used exclusively when `TaskSchedulerImpl` submits tasks (for a given `TaskSet`).

Notifying TaskSetManager that Task Failed

— handleFailedTask Method

```
handleFailedTask(
    taskSetManager: TaskSetManager,
    tid: Long,
    taskState: TaskState,
    reason: TaskFailedReason): Unit
```

`handleFailedTask` notifies `taskSetManager` that `tid` task has failed and, only when `taskSetManager` is not in zombie state and `tid` is not in `KILLED` state, requests `SchedulerBackend` to revive offers.

Note

`handleFailedTask` is called when `TaskResultGetter` deserializes a `TaskFailedReason` for a failed task.

taskSetFinished Method

```
taskSetFinished(manager: TaskSetManager): Unit
```

`taskSetFinished` looks all `TaskSets` up by the stage id (in `taskSetsByStageIdAndAttempt` registry) and removes the stage attempt from them, possibly with removing the entire stage record from `taskSetsByStageIdAndAttempt` registry completely (if there are no other attempts registered).

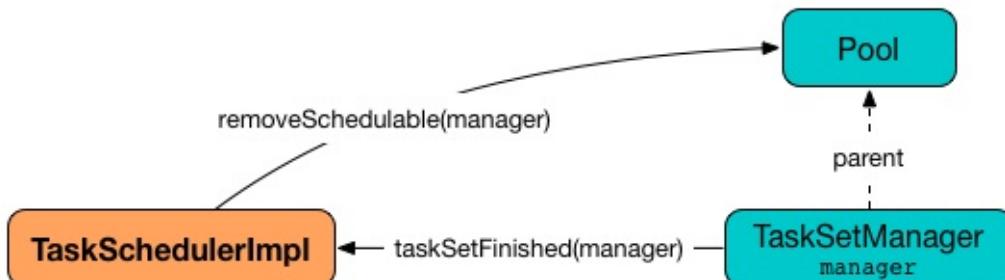


Figure 5. `TaskSchedulerImpl.taskSetFinished` is called when all tasks are finished

Note

A `TaskSetManager` manages a `TaskSet` for a stage.

`taskSetFinished` then removes `manager` from the parent's schedulable pool.

You should see the following INFO message in the logs:

```
INFO Removed TaskSet [id], whose tasks have all completed, from pool [name]
```

Note

`taskSetFinished` method is called when `TaskSetManager` has received the results of all the tasks in a `TaskSet`.

Notifying DAGScheduler About New Executor — `executorAdded` Method

```
executorAdded(execId: String, host: String)
```

`executorAdded` just notifies `DAGScheduler` that an executor was added.

Caution

`FIXME` Image with a call from `TaskSchedulerImpl` to `DAGScheduler`, please.

Note

`executorAdded` uses `DAGScheduler` that was given when `setDAGScheduler`.

Waiting Until SchedulerBackend is Ready — `waitBackendReady` Internal Method

```
waitBackendReady(): Unit
```

`waitBackendReady` waits until a `SchedulerBackend` is ready.

Note

`SchedulerBackend` is ready by default.

`waitBackendReady` keeps checking the status every `100` milliseconds until `SchedulerBackend` is ready or the `SparkContext` is stopped.

If the `SparkContext` happens to be stopped while waiting, `waitBackendReady` reports a `IllegalStateException`:

```
Spark context stopped while waiting for backend
```

Note

`waitBackendReady` is used when `TaskSchedulerImpl` is notified that `SparkContext` is near to get fully initialized.

Creating TaskDescriptions For Available Executor Resource Offers (with CPU Cores) — `resourceOffers` Method

```
resourceOffers(offers: Seq[WorkerOffer]): Seq[Seq[TaskDescription]]
```

`resourceOffers` takes the resources `offers` (as `WorkerOffers`) and generates a collection of tasks (as `TaskDescription`) to launch (given the resources available).

Note [WorkerOffer](#) represents a resource offer with CPU cores free to use on an executor.

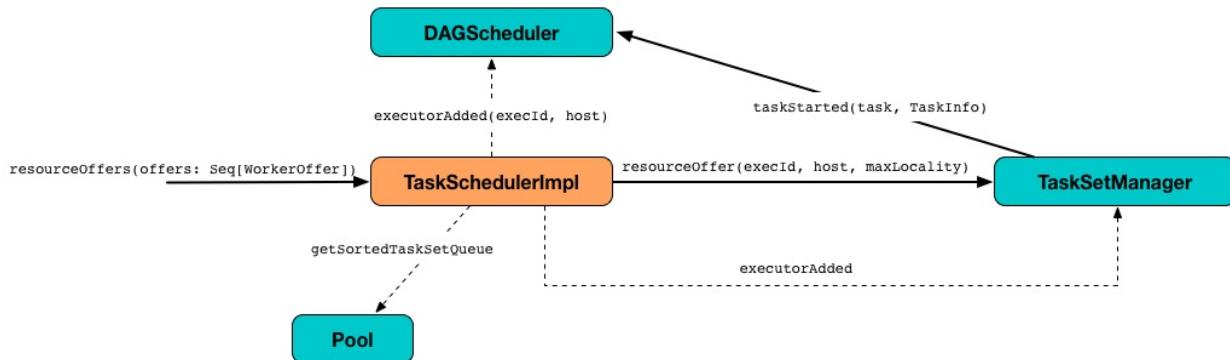


Figure 6. Processing Executor Resource Offers

Internally, `resourceOffers` first updates `hostToExecutors` and `executorIdToHost` lookup tables to record new hosts and executors (given the input `offers`).

For new executors (not in `executorIdToRunningTaskIds`) `resourceOffers` notifies `DAGScheduler` that an executor was added.

Note	TaskSchedulerImpl uses resourceOffers to track active executors.
Caution	FIXME a picture with executorAdded call from TaskSchedulerImpl to DAGScheduler.

`resourceOffers` requests `BlacklistTracker` to `applyBlacklistTimeout` and filters out offers on blacklisted nodes and executors.

Note	<code>resourceOffers</code> uses the optional <code>BlacklistTracker</code> that was given when <code>TaskSchedulerImpl</code> was created.
Caution	<code>FIXME</code> Expand on blacklisting

`resourceOffers` then randomly shuffles offers (to evenly distribute tasks across executors and avoid over-utilizing some executors) and initializes the local data structures `tasks` and `availableCpus` (as shown in the figure below).

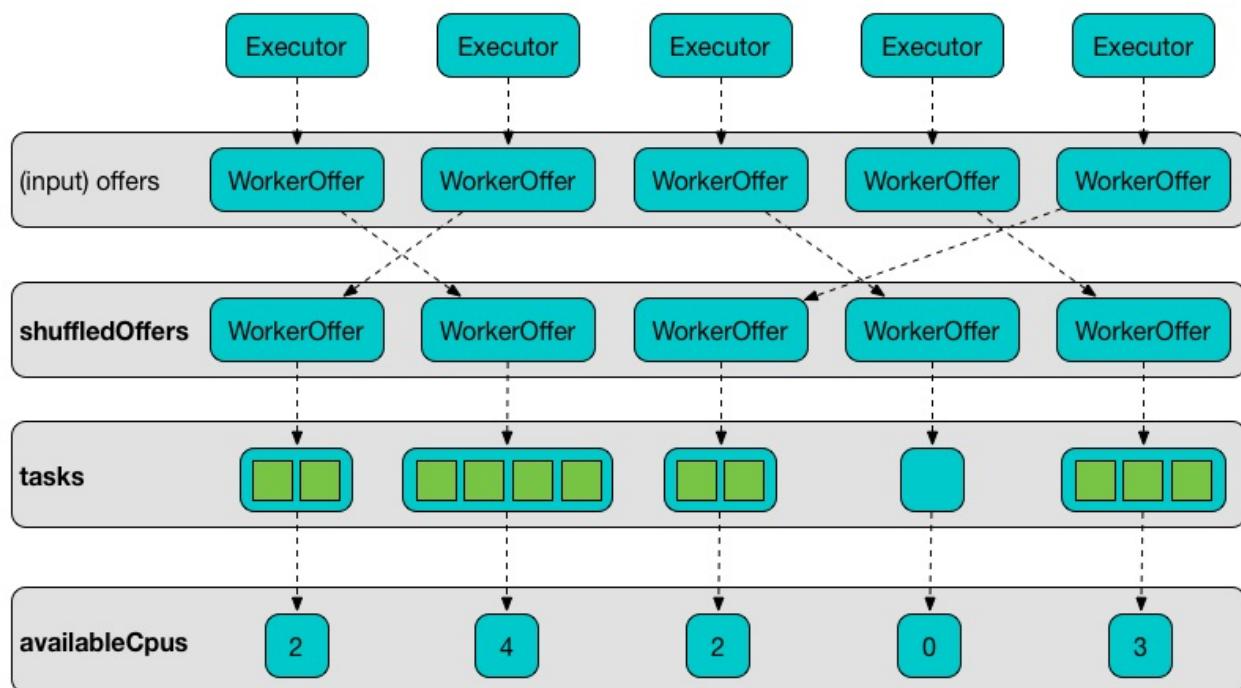


Figure 7. Internal Structures of `resourceOffers` with 5 `WorkerOffer`s (with 4, 2, 0, 3, 2 free cores)

`resourceOffers` takes `TaskSets` in scheduling order from top-level Schedulable Pool.

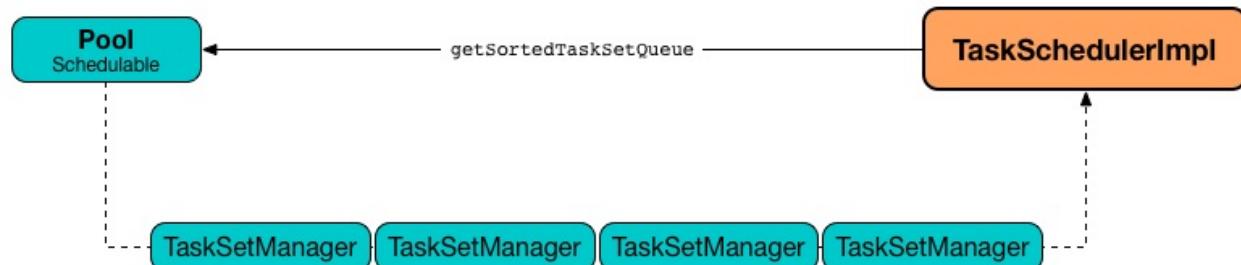


Figure 8. `TaskSchedulerImpl` Requesting `TaskSets` (as `TaskSetManagers`) from Root Pool

Note	<code>rootPool</code> is configured when <code>TaskSchedulerImpl</code> is initialized.
	<code>rootPool</code> is a part of the <code>TaskScheduler Contract</code> and exclusively managed by <code>SchedulableBuilders</code> , i.e. <code>FIFOSchedulableBuilder</code> and <code>FairSchedulableBuilder</code> (that manage registering <code>TaskSetManagers</code> with the root pool).
	<code>TaskSetManager</code> manages execution of the tasks in a single <code>TaskSet</code> that represents a single <code>Stage</code> .

For every `TaskSetManager` (in scheduling order), you should see the following DEBUG message in the logs:

```
DEBUG TaskSchedulerImpl: parentName: [name], name: [name], runningTasks: [count]
```

Only if a new executor was added, `resourceOffers` notifies every `TaskSetManager` about the change (to recompute locality preferences).

`resourceOffers` then takes every `TaskSetManager` (in scheduling order) and offers them each node in increasing order of locality levels (per `TaskSetManager's valid locality levels`).

Note	A <code>TaskSetManager</code> computes locality levels of the tasks it manages.
------	---

For every `TaskSetManager` and the `TaskSetManager`'s valid locality level, `resourceOffers` tries to [find tasks to schedule \(on executors\)](#) as long as the `TaskSetManager` manages to launch a task (given the locality level).

If `resourceOffers` did not manage to offer resources to a `TaskSetManager` so it could launch any task, `resourceOffers` [requests the `TaskSetManager` to abort the `TaskSet`](#) if completely blacklisted.

When `resourceOffers` managed to launch a task, the internal `hasLaunchedTask` flag gets enabled (that effectively means what the name says "*there were executors and I managed to launch a task*").

Note	<code>resourceOffers</code> is used when: <ul style="list-style-type: none"> • <code>CoarseGrainedSchedulerBackend</code> (via RPC endpoint) makes executor resource offers • <code>LocalEndpoint</code> revives resource offers • Spark on Mesos' <code>MesosFineGrainedSchedulerBackend</code> does <code>resourceOffers</code>
------	--

Finding Tasks from TaskSetManager to Schedule on Executors — `resourceOfferSingleTaskSet` Internal Method

```
resourceOfferSingleTaskSet(
  taskSet: TaskSetManager,
  maxLocality: TaskLocality,
  shuffledOffers: Seq[WorkerOffer],
  availableCpus: Array[Int],
  tasks: Seq[ArrayBuffer[TaskDescription]]): Boolean
```

`resourceOfferSingleTaskSet` takes every `WorkerOffer` (from the input `shuffledOffers`) and (only if the number of available CPU cores (using the input `availableCpus`) is at least `spark.task.cpus`) requests `TaskSetManager` (as the input `taskSet`) to find a `Task` to execute (given the resource offer) (as an executor, a host, and the input `maxLocality`).

`resourceOfferSingleTaskSet` adds the task to the input `tasks` collection.

`resourceOfferSingleTaskSet` records the task id and `TaskSetManager` in the following registries:

- `taskIdToTaskSetManager`
- `taskIdToExecutorId`
- `executorIdToRunningTaskIds`

`resourceOfferSingleTaskSet` decreases `spark.task.cpus` from the input `availableCpus` (for the `workerOffer`).

Note	<code>resourceOfferSingleTaskSet</code> makes sure that the number of available CPU cores (in the input <code>availableCpus</code> per <code>workerOffer</code>) is at least <code>0</code> .
------	--

If there is a `TaskNotSerializableException`, you should see the following ERROR in the logs:

ERROR Resource offer failed, task set [name] was not serializable

`resourceOfferSingleTaskSet` returns whether a task was launched or not.

Note	<code>resourceOfferSingleTaskSet</code> is used when <code>TaskSchedulerImpl</code> creates <code>TaskDescriptions</code> for available executor resource offers (with CPU cores).
------	--

TaskLocality — Task Locality Preference

`TaskLocality` represents a task locality preference and can be one of the following (from most localized to the widest):

1. `PROCESS_LOCAL`
2. `NODE_LOCAL`
3. `NO_PREF`
4. `RACK_LOCAL`
5. `ANY`

WorkerOffer — Free CPU Cores on Executor

<code>WorkerOffer(executorId: String, host: String, cores: Int)</code>
--

`WorkerOffer` represents a resource offer with free CPU `cores` available on an `executorId` executor on a `host`.

Settings

Table 2. Spark Properties

Spark Property	Default Value	Description
<code>spark.task.maxFailures</code>	<ul style="list-style-type: none"> 4 in cluster mode 1 in local <code>maxFailures</code> in local-with-retries 	The number of individual task failures before giving up on the entire TaskSet and the job afterwards.
<code>spark.task.cpus</code>	1	The number of CPU cores per task.
<code>spark.starvation.timeout</code>	15s	Threshold above which Spark warns a user that an initial TaskSet may be starved.
<code>spark.scheduler.mode</code>	FIFO	<p>A case-insensitive name of the scheduling mode — FAIR , FIFO , or NONE .</p> <p>NOTE: Only FAIR and FIFO are supported by <code>TaskSchedulerImpl</code> . See schedulableBuilder.</p>

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-submitted to 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 the initial `spark.speculation.interval` passes.

When enabled, you should see the following INFO message in the logs:

```
INFO TaskSchedulerImpl: Starting speculative execution thread
```

It works as `task-scheduler-speculation` **daemon thread pool** using `j.u.c.ScheduledThreadPoolExecutor` with core pool size `1`.

The job with speculatable tasks should finish while speculative tasks are running, and it will leave these tasks running - no KILL command yet.

It uses `checkSpeculatableTasks` method that asks `rootPool` to check for speculatable tasks. If there are any, `SchedulerBackend` is called for `reviveOffers`.

Caution

FIXME How does Spark handle repeated results of speculative tasks since there are copies launched?

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.speculation</code>	<code>false</code>	Enables (<code>true</code>) or disables (<code>false</code>) speculative execution of tasks (by means of <code>task-scheduler-speculation</code> Scheduled Executor Service).
<code>spark.speculation.interval</code>	<code>100ms</code>	The time interval to use before checking for speculative tasks.
<code>spark.speculation.multiplier</code>	<code>1.5</code>	
<code>spark.speculation.quantile</code>	<code>0.75</code>	The percentage of tasks that has not finished yet at which to start speculation.

TaskResultGetter

`TaskResultGetter` is a helper class of `TaskSchedulerImpl` for *asynchronous* deserialization of `task results of tasks that have finished successfully` (possibly fetching remote blocks) or `the failures for failed tasks`.

Caution

`FIXME` Image with the dependencies

Tip

Consult [Task States](#) in Tasks to learn about the different task states.

Note

The only instance of `TaskResultGetter` is created while `TaskSchedulerImpl` is created.

`TaskResultGetter` requires a `SparkEnv` and `TaskSchedulerImpl` to be created and is stopped when `TaskSchedulerImpl` stops.

`TaskResultGetter` uses `task-result-getter` asynchronous task executor for operation.

Enable `DEBUG` logging level for `org.apache.spark.scheduler.TaskResultGetter` logger to see what happens inside.

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

Tip

```
log4j.logger.org.apache.spark.scheduler.TaskResultGetter=DEBUG
```

Refer to [Logging](#).

task-result-getter Asynchronous Task Executor

```
getTaskResultExecutor: ExecutorService
```

`getTaskResultExecutor` creates a daemon thread pool with `spark.resultGetter.threads` threads and `task-result-getter` prefix.

Tip

Read up on [java.util.concurrent.ThreadPoolExecutor](#) that `getTaskResultExecutor` uses under the covers.

stop Method

```
stop(): Unit
```

`stop` stops the internal `task-result-getter` asynchronous task executor.

serializer Attribute

```
serializer: ThreadLocal[SerializerInstance]
```

`serializer` is a thread-local `SerializerInstance` that `TaskResultGetter` uses to deserialize byte buffers (with `TaskResult`s or a `TaskEndReason`).

When created for a new thread, `serializer` is initialized with a new instance of `Serializer` (using `SparkEnv.closureSerializer`).

Note

`TaskResultGetter` uses `java.lang.ThreadLocal` for the thread-local `SerializerInstance` variable.

taskResultSerializer Attribute

```
taskResultSerializer: ThreadLocal[SerializerInstance]
```

`taskResultSerializer` is a thread-local `SerializerInstance` that `TaskResultGetter` uses to...

When created for a new thread, `taskResultSerializer` is initialized with a new instance of `Serializer` (using `SparkEnv.serializer`).

Note

`TaskResultGetter` uses `java.lang.ThreadLocal` for the thread-local `SerializerInstance` variable.

Deserializing Task Result and Notifying TaskSchedulerImpl — enqueueSuccessfulTask Method

```
enqueueSuccessfulTask(  
    taskSetManager: TaskSetManager,  
    tid: Long,  
    serializedData: ByteBuffer): Unit
```

`enqueueSuccessfulTask` submits an asynchronous task (to `task-result-getter` asynchronous task executor) that first deserializes `serializedData` to a `DirectTaskResult`, then updates the internal accumulator (with the size of the `DirectTaskResult`) and ultimately notifies the `TaskSchedulerImpl` that the `tid` task was completed and the task result was received successfully or not.

Note

`enqueueSuccessfulTask` is just the asynchronous task enqueued for execution by `task-result-getter` `asynchronous task executor` at some point in the future.

Internally, the enqueued task first deserializes `serializedData` to a `TaskResult` (using the internal thread-local `serializer`).

The `TaskResult` could be a `DirectTaskResult` or a `IndirectTaskResult`.

For a `DirectTaskResult`, the task checks the available memory for the task result and, when the size overflows `spark.driver.maxResultSize`, it simply returns.

Note

`enqueueSuccessfulTask` is a mere thread so returning from a thread is to do nothing else. That is why the `check for quota does abort` when there is not enough memory.

Otherwise, when there *is* enough memory to hold the task result, it deserializes the `DirectTaskResult` (using the internal thread-local `taskResultSerializer`).

For a `IndirectTaskResult`, the task checks the available memory for the task result and, when the size could overflow the maximum result size, it removes the block and simply returns.

Otherwise, when there *is* enough memory to hold the task result, you should see the following DEBUG message in the logs:

```
DEBUG Fetching indirect task result for TID [tid]
```

The task notifies `TaskSchedulerImpl` that it is about to fetch a remote block for a task result. It then gets the block from remote block managers (as serialized bytes).

When the block could not be fetched, `TaskSchedulerImpl` is informed (with `TaskResultLost` task failure reason) and the task simply returns.

Note

`enqueueSuccessfulTask` is a mere thread so returning from a thread is to do nothing else and so the real handling is when `TaskSchedulerImpl` is informed.

The task result (as a serialized byte buffer) is then deserialized to a `DirectTaskResult` (using the internal thread-local `serializer`) and deserialized again using the internal thread-local `taskResultSerializer` (just like for the `DirectTaskResult` case). The block is removed from `BlockManagerMaster` and simply returns.

Note

A `IndirectTaskResult` is deserialized twice to become the final deserialized task result (using `serializer` for a `DirectTaskResult`). Compare it to a `DirectTaskResult` task result that is deserialized once only.

With no exceptions thrown, `enqueueSuccessfulTask` informs the `TaskSchedulerImpl` that the `tid` task was completed and the task result was received.

A `ClassNotFoundException` leads to aborting the `TaskSet` (with `ClassNotFoundException` with `classloader: [loader]` error message) while any non-fatal exception shows the following ERROR message in the logs followed by aborting the `TaskSet`.

```
ERROR Exception while getting task result
```

Note `enqueueSuccessfulTask` is called when `TaskSchedulerImpl` is notified about a task that has finished successfully (i.e. in `FINISHED` state).

Deserializing TaskFailedReason and Notifying TaskSchedulerImpl — `enqueueFailedTask` Method

```
enqueueFailedTask(  
    taskSetManager: TaskSetManager,  
    tid: Long,  
    taskState: TaskState.TaskState,  
    serializedData: ByteBuffer): Unit
```

`enqueueFailedTask` submits an asynchronous task (to `task-result-getter` asynchronous task executor) that first attempts to deserialize a `TaskFailedReason` from `serializedData` (using the internal thread-local `serializer`) and then notifies `TaskSchedulerImpl` that the task has failed.

Any `ClassNotFoundException` leads to the following ERROR message in the logs (without breaking the flow of `enqueueFailedTask`):

```
ERROR Could not deserialize TaskEndReason: ClassNotFoundException with classloader [loader]
```

Note `enqueueFailedTask` is called when `TaskSchedulerImpl` is notified about a task that has failed (and is in `FAILED`, `KILLED` or `LOST` state).

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.resultGetter.threads</code>	4	The number of threads for <code>TaskResultGetter</code> .

TaskContext

`TaskContext` is the [contract](#) for contextual information about a [Task](#) in Spark that allows for [registering task listeners](#).

You can access the active `TaskContext` instance using [TaskContext.get](#) method.

```
import org.apache.spark.TaskContext
val ctx = TaskContext.get
```

Using `TaskContext` you can [access local properties](#) that were set by the driver.

Note

`TaskContext` is serializable.

TaskContext Contract

```
trait TaskContext {
  def taskSucceeded(index: Int, result: Any)
  def jobFailed(exception: Exception)
}
```

Table 1. TaskContext Contract

Method	Description
<code>stageId</code>	<p>Id of the Stage the task belongs to.</p> <p>Used when...</p>
<code>partitionId</code>	<p>Id of the Partition computed by the task.</p> <p>Used when...</p>
<code>attemptNumber</code>	<p>Specifies how many times the task has been attempted (starting from 0).</p> <p>Used when...</p>
<code>taskAttemptId</code>	<p>Id of the attempt of the task.</p> <p>Used when...</p>
<code>getMetricsSources</code>	<p>Gives all the metrics sources by <code>sourceName</code> which are associated with the instance that runs the task.</p>
<code>getLocalProperty</code>	<p>Used when...</p> <p>Accesses local properties set by the driver using SparkContext.setLocalProperty.</p>
<code>taskMetrics</code>	<p>TaskMetrics of the active Task.</p> <p>Used when...</p>
<code>taskMemoryManager</code>	Used when...
<code>registerAccumulator</code>	Used when...
<code>isCompleted</code>	Used when...
<code>isInterrupted</code>	<p>A flag that is enabled when a task was killed.</p> <p>Used when...</p>
<code>addTaskCompletionListener</code>	<p>Registers a <code>TaskCompletionListener</code></p> <p>Used when...</p>
<code>addTaskFailureListener</code>	<p>Registers a <code>TaskFailureListener</code></p> <p>Used when...</p>

unset Method

Caution

FIXME

setTaskContext Method

Caution

FIXME

Accessing Active TaskContext— get Method

```
get(): TaskContext
```

`get` method returns the `TaskContext` instance for an active task (as a `TaskContextImpl`). 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.

Registering Task Listeners

Using `TaskContext` object you can register task listeners for [task completion](#) regardless of the final state and [task failures only](#).

addTaskCompletionListener Method

```
addTaskCompletionListener(listener: TaskCompletionListener): TaskContext
addTaskCompletionListener(f: (TaskContext) => Unit): TaskContext
```

`addTaskCompletionListener` methods register a `TaskCompletionListener` listener to 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 Method

```
addTaskFailureListener(listener: TaskFailureListener): TaskContext
addTaskFailureListener(f: (TaskContext, Throwable) => Unit): TaskContext
```

`addTaskFailureListener` methods register a `TaskFailureListener` listener to be executed on task failure only. 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

```

(Unused) Accessing Partition Id — `getPartitionId` Method

`getPartitionId(): Int`

`getPartitionId` gets the active `TaskContext` and returns `partitionId` or `0` (if `TaskContext` not available).

Note

`getPartitionId` is not used.

TaskContextImpl

`TaskContextImpl` is the one and only [TaskContext](#).

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

- stage
- partition
- task attempt
- attempt number
- runningLocally = false
- [taskMemoryManager](#)

Caution	FIXME Where and how is <code>TaskMemoryManager</code> used?
---------	---

taskMetrics Property

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

markTaskCompleted Method

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

markTaskFailed Method

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

Creating TaskContextImpl Instance

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

markInterrupted Method

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

TaskResults — DirectTaskResult and IndirectTaskResult

`TaskResult` models a task result. It has exactly two concrete implementations:

1. `DirectTaskResult` is the `TaskResult` to be serialized and sent over the wire to the driver together with the result bytes and accumulators.
2. `IndirectTaskResult` is the `TaskResult` that is just a pointer to a task result in a `BlockManager`.

The decision of the concrete `TaskResult` is made when a `TaskRunner` finishes running a task and checks the size of the result.

Note	The types are <code>private[spark]</code> .
------	---

DirectTaskResult Task Result

```
DirectTaskResult[T](
  var valueBytes: ByteBuffer,
  var accumUpdates: Seq[AccumulatorV2[_, _]])
extends TaskResult[T] with Externalizable
```

`DirectTaskResult` is the `TaskResult` of running a task (that is later returned serialized to the driver) when the size of the task's result is smaller than `spark.driver.maxResultSize` and `spark.task.maxDirectResultSize` (or `spark.rpc.message.maxSize` whatever is smaller).

Note	<code>DirectTaskResult</code> is Java's <code>java.io.Externalizable</code> .
------	---

IndirectTaskResult Task Result

```
IndirectTaskResult[T](blockId: BlockId, size: Int)
extends TaskResult[T] with Serializable
```

`IndirectTaskResult` is a `TaskResult` that...

Note	<code>IndirectTaskResult</code> is Java's <code>java.io.Serializable</code> .
------	---

TaskMemoryManager

TaskMemoryManager manages the memory allocated to an [individual task](#).

TaskMemoryManager 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`)

Table 1. TaskMemoryManager Internal Registries

Name	Description
<code>pageTable</code>	The array of size <code>PAGE_TABLE_SIZE</code> with indices being <code>MemoryBlock</code> objects. When allocating a <code>MemoryBlock</code> page for Tungsten consumers , the index corresponds to <code>pageNumber</code> that points to the <code>MemoryBlock</code> page allocated.
<code>allocatedPages</code>	Collection of flags (<code>true</code> or <code>false</code> values) of size <code>PAGE_TABLE_SIZE</code> with all bits initially disabled (i.e. <code>false</code>). TIP: <code>allocatedPages</code> is java.util.BitSet . 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 <code>true</code> .
<code>consumers</code>	Set of MemoryConsumers
<code>acquiredButNotUsed</code>	The size of memory allocated but not used.

Note

TaskMemoryManager is used to [create a `TaskContextImpl`](#).

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

[FIXME](#) How to trigger the messages in the logs? What to execute to have them printed out to the logs?

cleanUpAllAllocatedMemory Method

`cleanUpAllAllocatedMemory` clears [page table](#).

Caution

[FIXME](#)

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`.

pageSizeBytes Method

Caution

[FIXME](#)

releaseExecutionMemory Method

Caution

[FIXME](#)

showMemoryUsage Method

Caution

FIXME

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 `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.

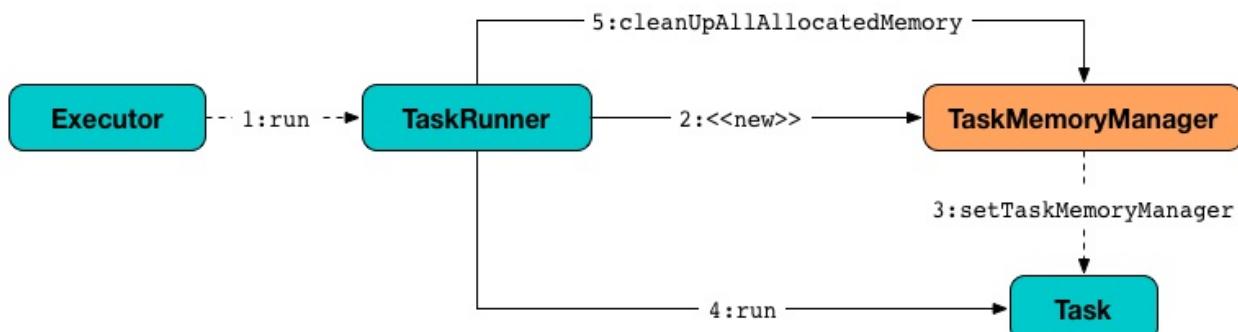


Figure 1. Creating TaskMemoryManager for Task

Acquiring 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, `acquireExecutionMemory` returns the allocated memory.

Note

`acquireExecutionMemory` synchronizes on itself, and so no other calls on the object could be completed.

Note

`MemoryConsumer` knows its mode — on- or off-heap.

`acquireExecutionMemory` first calls `memoryManager.acquireExecutionMemory(required, taskAttemptId, mode)`.

Tip

`TaskMemoryManager` is a mere wrapper of `MemoryManager` to track `consumers`?

Caution

`FIXME`

When the memory obtained is less than requested (by `required`), `acquireExecutionMemory` requests all `consumers` to release memory (by spilling it to disk).

Note

`acquireExecutionMemory` 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 memory to be acquired, `acquireExecutionMemory` requests the input `consumer` to spill memory to disk (that in fact 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**

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?

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, <code>pageSize</code> defaults to <code>TaskMemoryManager.pageSizeBytes</code> and <code>ON_HEAP</code> memory mode.
------	--

spill Method

```
abstract long spill(long size, MemoryConsumer trigger)
throws IOException
```

Caution	FIXME
Note	<code>spill</code> is used when <code>TaskMemoryManager</code> forces <code>MemoryConsumers</code> to release memory when requested to acquire execution memory

Memory Allocated — used Registry

`used` is the amount of memory in use (i.e. allocated) by the `MemoryConsumer`.

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

`TaskMetrics` is a [collection of metrics](#) tracked during execution of a [Task](#).

`TaskMetrics` uses [accumulators](#) to represent the metrics and offers "increment" methods to increment them.

Note	The local values of the accumulators for a task (as accumulated while the task runs) are sent from the executor to the driver when the task completes (and DAGScheduler re-creates TaskMetrics).
------	--

Table 1. Metrics

Property	Name	Type
<code>_memoryBytesSpilled</code>	<code>internal.metrics.memoryBytesSpilled</code>	<code>LongAccumulator</code>
<code>_updatedBlockStatuses</code>	<code>internal.metrics.updatedBlockStatuses</code>	<code>CollectionAccumulator<BlockStatus>]</code>

Table 2. TaskMetrics's Internal Registries and Counters

Name	Description
<code>nameToAccums</code>	Internal accumulators indexed by their names. Used when <code>TaskMetrics</code> re-creates <code>TaskMetrics</code> from <code>AccumulatorV2s</code> , ... FIXME NOTE: <code>nameToAccums</code> is a <code>transient</code> and <code>lazy</code> value.
<code>internalAccums</code>	Collection of internal AccumulatorV2 objects. Used when... FIXME NOTE: <code>internalAccums</code> is a <code>transient</code> and <code>lazy</code> value.
<code>externalAccums</code>	Collection of external AccumulatorV2 objects. Used when <code>TaskMetrics</code> re-creates <code>TaskMetrics</code> from <code>AccumulatorV2s</code> , ... FIXME NOTE: <code>externalAccums</code> is a <code>transient</code> and <code>lazy</code> value.

accumulators Method

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

mergeShuffleReadMetrics Method

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

memoryBytesSpilled Method

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

updatedBlockStatuses Method

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

setExecutorCpuTime Method

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

setResultSerializationTime Method

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

setJvmGCTime Method

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

setExecutorRunTime Method

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

setExecutorDeserializeCpuTime Method

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

setExecutorDeserializeTime Method

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

setUpdatedBlockStatuses Method

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

Re-Creating TaskMetrics From AccumulatorV2s — fromAccumulators Method

```
fromAccumulators(accums: Seq[AccumulatorV2[_, _]]): TaskMetrics
```

`fromAccumulators` creates a new `TaskMetrics` and registers `accums` as internal and external task metrics (using `nameToAccums` internal registry).

Internally, `fromAccumulators` creates a new `TaskMetrics`. It then splits `accums` into internal and external task metrics collections (using `nameToAccums` internal registry).

For every internal task metrics, `fromAccumulators` finds the metrics in `nameToAccums` internal registry (of the new `TaskMetrics` instance), copies `metadata`, and merges `state`.

In the end, `fromAccumulators` adds the external accumulators to the new `TaskMetrics` instance.

Note	<code>fromAccumulators</code> is used exclusively when <code>DAGScheduler</code> gets notified that a task has finished (and re-creates <code>TaskMetrics</code>).
------	---

Recording Memory Bytes Spilled — incMemoryBytesSpilled Method

```
incMemoryBytesSpilled(v: Long): Unit
```

`incMemoryBytesSpilled` adds `v` to `_memoryBytesSpilled` task metrics.

Note

- `incMemoryBytesSpilled` is used when:
1. `Aggregator` updates task metrics
 2. `CoGroupedRDD` computes a `Partition`
 3. `BlockStoreShuffleReader` reads combined key-value records for a reduce task
 4. `ShuffleExternalSorter` frees execution memory by spilling to disk
 5. `ExternalSorter` writes the records into a temporary partitioned file in the disk store
 6. `UnsafeExternalSorter` spills current records due to memory pressure
 7. `SpillableIterator` spills records to disk
 8. `JsonProtocol` creates `TaskMetrics` from JSON

Recording Updated BlockStatus For Block — `incUpdatedBlockStatuses` Method

```
incUpdatedBlockStatuses(v: (BlockId, BlockStatus)): Unit
```

`incUpdatedBlockStatuses` adds `v` in `_updatedBlockStatuses` internal registry.

Note

`incUpdatedBlockStatuses` is used exclusively when `BlockManager` does `addUpdatedBlockStatusToTaskMetrics`.

Registering Internal Accumulators — `register` Method

```
register(sc: SparkContext): Unit
```

`register` registers the internal accumulators (from `nameToAccums` internal registry) with `countFailedValues` enabled (`true`).

Note

`register` is used exclusively when `Stage` is requested for its new attempt.

ShuffleWriteMetrics

`ShuffleWriteMetrics` is a [collection of accumulators](#) that represents task metrics about writing shuffle data.

`ShuffleWriteMetrics` tracks the following task metrics:

1. [Shuffle Bytes Written](#)
2. [Shuffle Write Time](#)
3. [Shuffle Records Written](#)

Note

[Accumulators](#) allow tasks (running on executors) to communicate with the driver.

Table 1. ShuffleWriteMetrics's Accumulators

Name	Description
<code>_bytesWritten</code>	<p>Accumulator to track how many shuffle bytes were written in a shuffle task.</p> <p>Used when <code>ShuffleWriteMetrics</code> is requested the shuffle bytes written and to increment or decrement it.</p> <p>NOTE: <code>_bytesWritten</code> is available as <code>internal.metrics.shuffle.write.bytesWritten</code> (internally <code>shuffleWrite.BYTES_WRITTEN</code>) in TaskMetrics.</p>
<code>_writeTime</code>	<p>Accumulator to track shuffle write time (as 64-bit integer) of a shuffle task.</p> <p>Used when <code>ShuffleWriteMetrics</code> is requested the shuffle write time and to increment it.</p> <p>NOTE: <code>_writeTime</code> is available as <code>internal.metrics.shuffle.write.writeTime</code> (internally <code>shuffleWrite.WRITE_TIME</code>) in TaskMetrics.</p>
<code>_recordsWritten</code>	<p>Accumulator to track how many shuffle records were written in a shuffle task.</p> <p>Used when <code>ShuffleWriteMetrics</code> is requested the shuffle records written and to increment or decrement it.</p> <p>NOTE: <code>_recordsWritten</code> is available as <code>internal.metrics.shuffle.write.recordsWritten</code> (internally <code>shuffleWrite.RECORDS_WRITTEN</code>) in TaskMetrics.</p>

decRecordsWritten Method

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

decBytesWritten Method

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

writeTime Method

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

recordsWritten Method

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

Returning Number of Shuffle Bytes Written — bytesWritten Method

`bytesWritten: Long`

`bytesWritten` represents the **shuffle bytes written** metrics of a shuffle task.

Internally, `bytesWritten` returns the sum of [`_bytesWritten`](#) internal accumulator.

Note	<p><code>bytesWritten</code> is used when:</p> <ol style="list-style-type: none"> 1. <code>ShuffleWriteMetricsUIData</code> is created 2. In <code>decBytesWritten</code> 3. <code>StatsReportListener</code> intercepts stage completed events to show shuffle bytes written 4. <code>ShuffleExternalSorter</code> does <code>writeSortedFile</code> (to <code>incDiskBytesSpilled</code>) 5. <code>JsonProtocol</code> converts <code>ShuffleWriteMetrics</code> to JSON 6. <code>ExecutorsListener</code> intercepts task end events to update executor metrics 7. <code>JobProgressListener</code> updates stage and executor metrics
------	--

Incrementing Shuffle Bytes Written Metrics

— incBytesWritten Method

```
incBytesWritten(v: Long): Unit
```

`incBytesWritten` simply adds `v` to `_bytesWritten` internal accumulator.

Note

- `incBytesWritten` is used when:
1. `UnsafeShuffleWriter` does `mergeSpills`
 2. `DiskBlockObjectWriter` does `updateBytesWritten`
 3. `JsonProtocol` creates `TaskMetrics` from JSON

Incrementing Shuffle Write Time Metrics

— incWriteTime Method

```
incWriteTime(v: Long): Unit
```

`incWriteTime` simply adds `v` to `_writeTime` internal accumulator.

Note

- `incWriteTime` is used when:
1. `SortShuffleWriter` stops.
 2. `BypassMergeSortShuffleWriter` writes records (i.e. when it initializes `DiskBlockObjectWriter` partition writers) and later when concatenates per-partition files into a single file.
 3. `UnsafeShuffleWriter` does `mergeSpillsWithTransferTo`.
 4. `DiskBlockObjectWriter` does `commitAndGet` (but only when `syncwrites` flag is enabled that forces outstanding writes to disk).
 5. `JsonProtocol` creates `TaskMetrics` from JSON
 6. `TimeTrackingOutputStream` does its operation (after all it is an output stream to track shuffle write time).

Incrementing Shuffle Records Written Metrics

— incRecordsWritten Method

```
incRecordsWritten(v: Long): Unit
```

`incRecordsWritten` simply adds `v` to `_recordsWritten` internal accumulator.

Note

`incRecordsWritten` is used when:

1. `shuffleExternalSorter` does `writeSortedFile`
2. `DiskBlockObjectWriter` does `recordWritten`
3. `JsonProtocol` creates `TaskMetrics` from JSON

TaskSetBlacklist — Blacklisting Executors and Nodes For TaskSet

Caution

[FIXME](#)

updateBlacklistForFailedTask Method

Caution

[FIXME](#)

isExecutorBlacklistedForTaskSet Method

Caution

[FIXME](#)

isNodeBlacklistedForTaskSet Method

Caution

[FIXME](#)

SchedulerBackend — Pluggable Scheduler Backends

`SchedulerBackend` is a pluggable [interface](#) to support various cluster managers, e.g. [Apache Mesos](#), [Hadoop YARN](#) or Spark's own [Spark Standalone](#) and [Spark local](#).

As the cluster managers differ by their custom task scheduling modes and resource offers mechanisms Spark abstracts the differences in [SchedulerBackend contract](#).

Table 1. Built-In (Direct and Indirect) SchedulerBackends per Cluster Environment

Cluster Environment	SchedulerBackends
Local mode	LocalSchedulerBackend
(base for custom SchedulerBackends)	CoarseGrainedSchedulerBackend
Spark Standalone	StandaloneSchedulerBackend
Spark on YARN	YarnSchedulerBackend : <ul style="list-style-type: none"> • YarnClientSchedulerBackend (for client deploy mode) • YarnClusterSchedulerBackend (for cluster deploy mode)
Spark on Mesos	<ul style="list-style-type: none"> • MesosCoarseGrainedSchedulerBackend • MesosFineGrainedSchedulerBackend

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 <code>SparkContext</code> in play here or in <code>SparkContext</code> 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

```
trait SchedulerBackend {
    def applicationId(): String
    def applicationAttemptId(): Option[String]
    def defaultParallelism(): Int
    def getDriverLogUrls: Option[Map[String, String]]
    def isReady(): Boolean
    def killTask(taskId: Long, executorId: String, interruptThread: Boolean): Unit
    def reviveOffers(): Unit
    def start(): Unit
    def stop(): Unit
}
```

Note

`org.apache.spark.scheduler.SchedulerBackend` is a `private[spark]` Scala trait in Spark.

Table 2. SchedulerBackend Contract

Method	Description
<code>applicationId</code>	Unique identifier of Spark Application Used when <code>TaskSchedulerImpl</code> is asked for the unique identifier of a Spark application (that is actually a part of TaskScheduler contract).
<code>applicationAttemptId</code>	Attempt id of a Spark application Only supported by YARN cluster scheduler backend as the YARN cluster manager supports multiple application attempts. Used when... NOTE: <code>applicationAttemptId</code> is also a part of TaskScheduler contract and <code>TaskSchedulerImpl</code> directly calls the <code>SchedulerBackend</code> 's <code>applicationAttemptId</code> .
<code>defaultParallelism</code>	Used when <code>TaskSchedulerImpl</code> finds the default level of parallelism (as a hint for sizing jobs).
<code>getDriverLogUrls</code>	Returns no URLs by default and only supported by YarnClusterSchedulerBackend

<code>isReady</code>	<p>Controls whether <code>SchedulerBackend</code> is ready (i.e. <code>true</code>) or not (i.e. <code>false</code>). Enabled by default.</p> <p>Used when <code>TaskSchedulerImpl</code> waits until <code>SchedulerBackend</code> is ready (which happens just before <code>SparkContext</code> is fully initialized).</p>
<code>killTask</code>	<p>Reports a <code>UnsupportedOperationException</code> by default.</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>TaskSchedulerImpl</code> cancels the tasks for a stage • <code>TaskSetManager</code> is notified about successful task attempt.
<code>reviveOffers</code>	<p>Used when <code>TaskSchedulerImpl</code>:</p> <ul style="list-style-type: none"> • Submits tasks (from <code>TaskSet</code>) • Receives task status updates • Notifies <code>TaskSetManager</code> that a task has failed • Checks for speculatable tasks • Gets notified about executor being lost
<code>start</code>	<p>Starts <code>SchedulerBackend</code>.</p> <p>Used when <code>TaskSchedulerImpl</code> is started.</p>
<code>stop</code>	<p>Stops <code>SchedulerBackend</code>.</p> <p>Used when <code>TaskSchedulerImpl</code> is stopped.</p>

CoarseGrainedSchedulerBackend

`CoarseGrainedSchedulerBackend` is a [SchedulerBackend](#).

`CoarseGrainedSchedulerBackend` is an [ExecutorAllocationClient](#).

`CoarseGrainedSchedulerBackend` is responsible for requesting resources from a cluster manager for executors that it in turn uses to [launch tasks](#) (on [coarse-grained executors](#)).

`CoarseGrainedSchedulerBackend` 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.

Caution	FIXME Picture with dependencies
---------	---

`CoarseGrainedSchedulerBackend` registers [CoarseGrainedScheduler RPC Endpoint](#) that executors use for RPC communication.

Note	Active executors are executors that are not pending to be removed or lost .
------	--

Table 1. Built-In CoarseGrainedSchedulerBackends per Cluster Environment

Cluster Environment	CoarseGrainedSchedulerBackend
Spark Standalone	StandaloneSchedulerBackend
Spark on YARN	YarnSchedulerBackend
Spark on Mesos	MesosCoarseGrainedSchedulerBackend

Note	<code>CoarseGrainedSchedulerBackend</code> is only created indirectly through built-in implementations per cluster environment .
------	--

Table 2. CoarseGrainedSchedulerBackend's

Name	Initial Value
<code>currentExecutorIdCounter</code>	
<code>createTime</code>	Current time
<code>defaultAskTimeout</code>	spark.rpc.askTimeout or spark.network.timeout or 120s

driverEndpoint	(uninitialized)
executorDataMap	empty
executorsPendingToRemove	empty
hostToLocalTaskCount	empty
localityAwareTasks	0
maxRegisteredWaitingTimeMs	spark.scheduler.maxRegisteredResourcesWaitingTime
maxRpcMessageSize	spark.rpc.message.maxSize but not greater than 2047
_minRegisteredRatio	spark.scheduler.minRegisteredResourcesRatio
numPendingExecutors	0

totalCoreCount	0
totalRegisteredExecutors	0

Tip	Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend</code> logger to see what happens inside.
	Add the following line to <code>conf/log4j.properties</code> :
	<code>log4j.logger.org.apache.spark.scheduler.cluster.CoarseGrainedSchedulerBackend=DEBUG</code>
	Refer to Logging .

Killing All Executors on Node — `killExecutorsOnHost` Method

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

Making Fake Resource Offers on Executors — `makeOffers` Internal Methods

```
makeOffers(): Unit
makeOffers(executorId: String): Unit
```

`makeOffers` 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.

Creating CoarseGrainedSchedulerBackend Instance

`CoarseGrainedSchedulerBackend` takes the following when created:

1. `TaskSchedulerImpl`
2. `RpcEnv`

`CoarseGrainedSchedulerBackend` initializes the internal registries and counters.

Getting Executor Ids — `getExecutorIds` Method

When called, `getExecutorIds` simply returns executor ids from the internal `executorDataMap` registry.

Note	It is called when <code>SparkContext</code> calculates executor ids.
------	--

CoarseGrainedSchedulerBackend Contract

```
class CoarseGrainedSchedulerBackend {
  def minRegisteredRatio: Double
  def createDriverEndpoint(properties: Seq[(String, String)]): DriverEndpoint
  def reset(): Unit
  def sufficientResourcesRegistered(): Boolean
  def doRequestTotalExecutors(requestedTotal: Int): Future[Boolean]
  def doKillExecutors(executorIds: Seq[String]): Future[Boolean]
}
```

Note	<code>CoarseGrainedSchedulerBackend</code> is a <code>private[spark]</code> contract.
------	---

Table 3. [FIXME](#) Contract

Method	Description
<code>minRegisteredRatio</code>	Ratio between <code>0</code> and <code>1</code> (inclusive). Controlled by <code>spark.scheduler.minRegisteredResourcesRatio</code> .
<code>reset</code>	FIXME
<code>doRequestTotalExecutors</code>	FIXME
<code>doKillExecutors</code>	FIXME
<code>sufficientResourcesRegistered</code>	Always positive, i.e. <code>true</code> , that means that sufficient resources are available. Used when <code>CoarseGrainedSchedulerBackend</code> checks if sufficient compute resources are available.

- It can [reset](#) a current internal state to the initial state.

numExistingExecutors Method

Caution

FIXME

killExecutors Methods

Caution

FIXME

getDriverLogUrls Method

Caution

FIXME

applicationAttemptId Method

Caution

FIXME

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

`requestExecutors` method is a part of [ExecutorAllocationClient Contract](#) that [SparkContext 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]
```

`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 `requestTotalExecutors` is a part of [ExecutorAllocationClient Contract](#) that [SparkContext](#) 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.

Finding Default Level of Parallelism

— `defaultParallelism` Method

```
defaultParallelism(): Int
```

Note

`defaultParallelism` is a part of the [SchedulerBackend Contract](#).

`defaultParallelism` is [spark.default.parallelism](#) Spark property if set.

Otherwise, `defaultParallelism` is the maximum of `totalCoreCount` or `2`.

Killing Task — `killTask` Method

```
killTask(taskId: Long, executorId: String, interruptThread: Boolean): Unit
```

Note

`killTask` is part of the [SchedulerBackend contract](#).

`killTask` simply sends a [KillTask](#) message to [driverEndpoint](#).

Caution

[FIXME Image](#)

Stopping All Executors — `stopExecutors` Method

`stopExecutors` sends a blocking [StopExecutors](#) message to [driverEndpoint](#) (if already initialized).

Note

It is called exclusively while `CoarseGrainedSchedulerBackend` is [being stopped](#).

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 [SparkDeploySchedulerBackend](#), [CoarseMesosSchedulerBackend](#), and [YarnSchedulerBackend](#).

CoarseGrainedScheduler RPC Endpoint — `driverEndpoint`

When [CoarseGrainedSchedulerBackend](#) starts, it registers **CoarseGrainedScheduler** RPC endpoint to be the driver's communication endpoint.

`driverEndpoint` is a [DriverEndpoint](#).

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:

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.

Starting CoarseGrainedSchedulerBackend (and Registering CoarseGrainedScheduler RPC Endpoint) — `start` Method

```
start(): Unit
```

Note

`start` is a part of the [SchedulerBackend](#) contract.

`start` takes all `spark.`-prefixed properties and registers the [CoarseGrainedScheduler](#) RPC endpoint (backed by [DriverEndpoint ThreadSafeRpcEndpoint](#)).

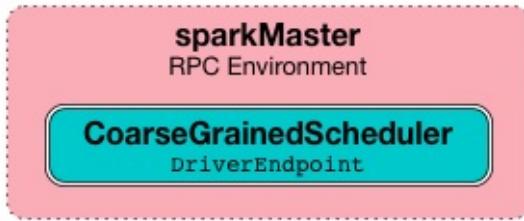


Figure 1. CoarseGrainedScheduler Endpoint

Note	<code>start</code> uses <code>TaskSchedulerImpl</code> to access the current <code>SparkContext</code> and in turn <code>SparkConf</code> .
------	---

Note	<code>start</code> uses <code>RpcEnv</code> that was given when <code>CoarseGrainedSchedulerBackend</code> was created.
------	---

Checking If Sufficient Compute Resources Available Or Waiting Time Passed — `isReady` Method

```
isReady(): Boolean
```

Note	<code>isReady</code> is a part of the <code>SchedulerBackend</code> contract.
------	---

`isReady` allows to delay task launching until sufficient resources are available or `spark.scheduler.maxRegisteredResourcesWaitingTime` passes.

Internally, `isReady` checks whether there are sufficient resources available.

Note	<code>sufficientResourcesRegistered</code> by default responds that sufficient resources are available.
------	---

If the resources are available, you should see the following INFO message in the logs and `isReady` is positive.

```
INFO SchedulerBackend is ready for scheduling beginning after reached minRegisteredResourcesRatio: [minRegisteredRatio]
```

Note	<code>minRegisteredRatio</code> 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.
------	--

If there are no sufficient resources available yet (the above requirement does not hold), `isReady` checks whether the time since `startup` passed `spark.scheduler.maxRegisteredResourcesWaitingTime` to give a way to launch tasks (even when `minRegisteredRatio` not being reached yet).

You should see the following INFO message in the logs and `isReady` is positive.

```
INFO SchedulerBackend is ready for scheduling beginning after
waiting maxRegisteredResourcesWaitingTime:
[maxRegisteredWaitingTimeMs] (ms)
```

Otherwise, when `no` sufficient resources are available and `spark.scheduler.maxRegisteredResourcesWaitingTime` has not elapsed, `isReady` is negative.

Reviving Resource Offers (by Posting ReviveOffers to CoarseGrainedSchedulerBackend RPC Endpoint) — `reviveOffers` Method

```
reviveOffers(): Unit
```

Note

`reviveOffers` is a part of the `SchedulerBackend` contract.

`reviveOffers` simply sends a `ReviveOffers` message to `CoarseGrainedSchedulerBackend` RPC endpoint.

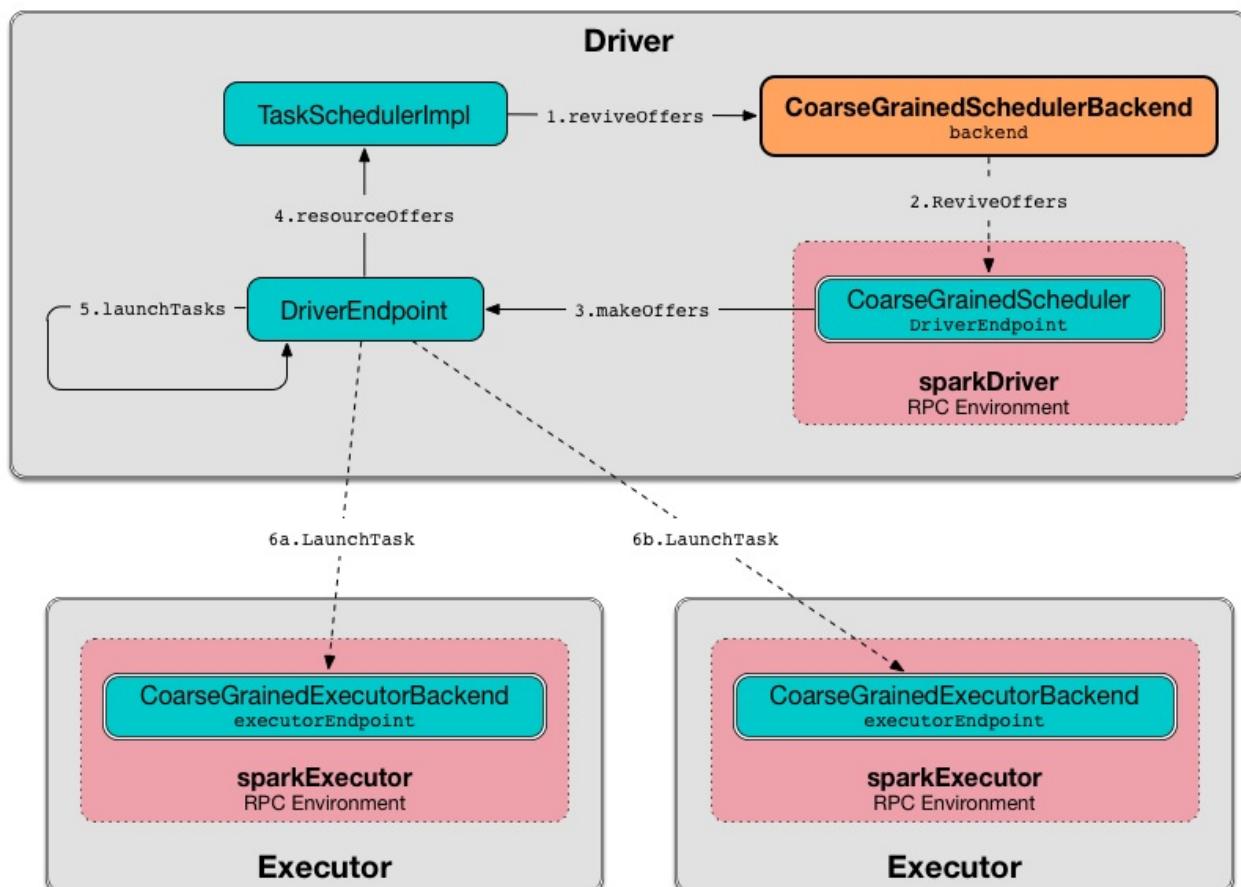


Figure 2. CoarseGrainedExecutorBackend Revives Offers

Stopping CoarseGrainedSchedulerBackend (and Stopping Executors) — stop Method

```
stop(): Unit
```

Note stop is a part of the SchedulerBackend contract.

stop stops all executors and coarseGrainedScheduler RPC endpoint (by sending a blocking StopDriver message).

In case of any Exception , stop reports a sparkException with the message:

```
Error stopping standalone scheduler's driver endpoint
```

createDriverEndpointRef Method

```
createDriverEndpointRef(properties: ArrayBuffer[(String, String)]): RpcEndpointRef
```

createDriverEndpointRef creates DriverEndpoint and registers it as CoarseGrainedScheduler.

Note createDriverEndpointRef is used when CoarseGrainedSchedulerBackend starts.

Creating DriverEndpoint — createDriverEndpoint Method

```
createDriverEndpoint(properties: Seq[(String, String)]): DriverEndpoint
```

createDriverEndpoint simply creates a DriverEndpoint.

Note DriverEndpoint is the RPC endpoint of CoarseGrainedSchedulerBackend .

Note The purpose of createDriverEndpoint is to allow YARN to use the custom YarnDriverEndpoint .

Note createDriverEndpoint is used when CoarseGrainedSchedulerBackend createDriverEndpointRef .

Settings

Table 4. Spark Properties

Property	Default Value	Description
spark.scheduler.revive.interval	1s	Time (in milliseconds) between resource offers revives.
spark.rpc.message.maxSize	128	<p>Maximum message size to allow in RPC communication. In MB when the unit is not given.</p> <p>Generally only applies to map output size (serialized) information sent between executors and the driver.</p> <p>Increase this if you are running jobs with many thousands of map and reduce tasks and see messages about the RPC message size.</p>
spark.scheduler.minRegisteredResourcesRatio	0	<p>Double number between 0 and 1 (including) that controls the minimum ratio of (registered resources / total expected resources) before submitting tasks.</p> <p>See isReady in this document.</p>
spark.scheduler.maxRegisteredResourcesWaitingTime	30s	Time to wait for sufficient resources available.

		See isReady in this document.
--	--	---

DriverEndpoint — CoarseGrainedSchedulerBackend RPC Endpoint

`DriverEndpoint` is a `ThreadSafeRpcEndpoint` that acts as a message handler for `CoarseGrainedSchedulerBackend` to communicate with `CoarseGrainedExecutorBackend`.

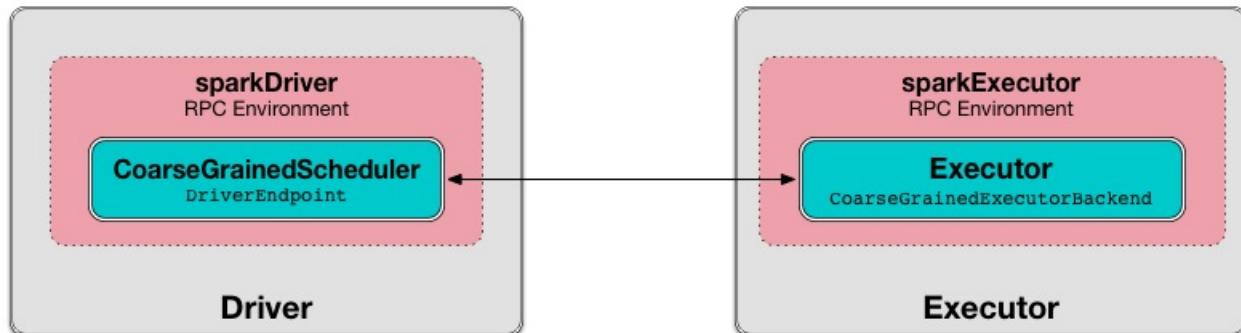


Figure 1. `CoarseGrainedSchedulerBackend` uses `DriverEndpoint` for communication with `CoarseGrainedExecutorBackend`

`DriverEndpoint` is created when `CoarseGrainedSchedulerBackend` starts.

`DriverEndpoint` uses `executorDataMap` internal registry of all the executors that registered with the driver. An executor sends a `RegisterExecutor` message to inform that it wants to register.

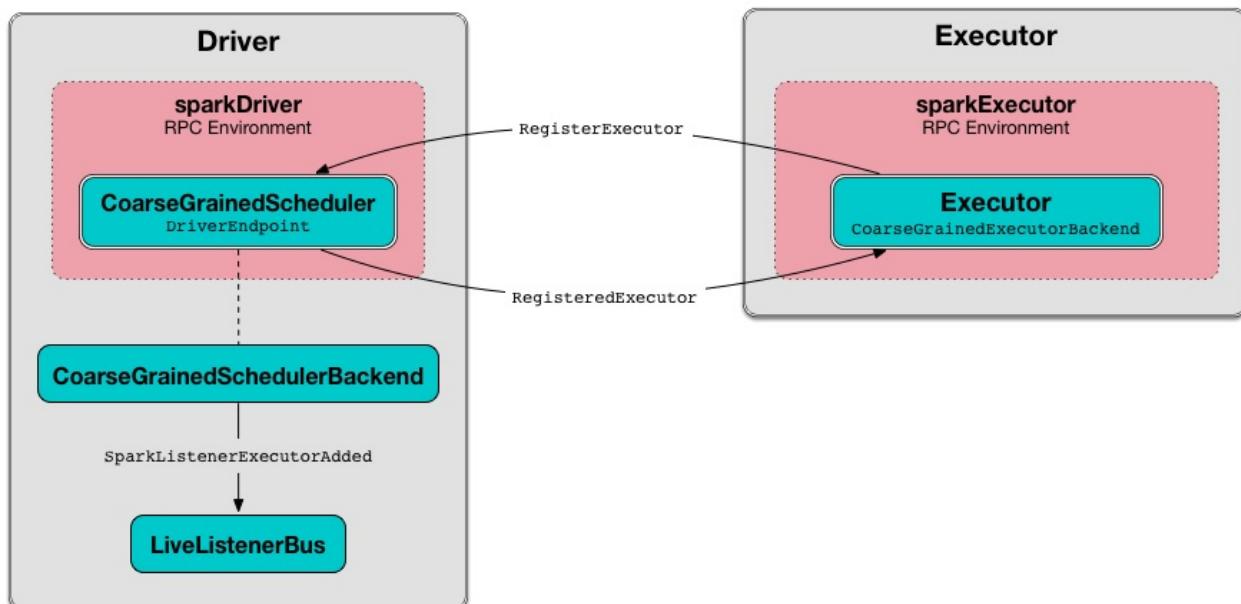


Figure 2. Executor registration (RegisterExecutor RPC message flow)

`DriverEndpoint` uses a single thread executor called `driver-revive-thread` to make executor resource offers (for launching tasks) (by emitting `ReviveOffers` message every `spark.scheduler.revive.interval`).

Table 1. CoarseGrainedClusterMessages and Their Handlers (in alphabetical order)

CoarseGrainedClusterMessage	Event Handler	When emitted?
KillExecutorsOnHost	KillExecutorsOnHost handler	CoarseGrainedSchedulerBackend requested to kill all executors on node.
KillTask	KillTask handler	CoarseGrainedSchedulerBackend requested to kill a task.
ReviveOffers	makeOffers	<ul style="list-style-type: none"> Periodically (every <code>spark.scheduler.revive.interval</code>) soon after DriverEndpoint starts accepting messages. CoarseGrainedSchedulerBackend is requested to revive received offers.
RegisterExecutor	RegisterExecutor handler	CoarseGrainedExecutorBackend registers with the driver.
StatusUpdate	StatusUpdate handler	CoarseGrainedExecutorBackend sends task status updates to driver.

Table 2. DriverEndpoint's Internal Properties

Name	Initial Value	Description
<code>addressToExecutorId</code>		Executor addresses (host and port) for executors. Set when an executor connects to register itself. See RegisterExecutor RPC message.
<code>executorsPendingLossReason</code>		
<code>reviveThread</code>		

disableExecutor Internal Method

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

KillExecutorsOnHost Handler

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

executorIsAlive Internal Method

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

onStop Callback

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

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	<code>onDisconnected</code> is called when a remote host is lost.
------	---

RemoveExecutor

RetrieveSparkProps

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`).

Scheduling Sending ReviveOffers Periodically

— onStart Callback

```
onStart(): Unit
```

Note

`onStart` is a part of [RpcEndpoint](#) contract that is executed before a RPC endpoint starts accepting messages.

`onStart` schedules a periodic action to send [ReviveOffers](#) immediately every [spark.scheduler.revive.interval](#).

Note

`spark.scheduler.revive.interval` defaults to `1s`.

Making Executor Resource Offers (for Launching Tasks)

— makeOffers Internal Method

```
makeOffers(): Unit
```

`makeOffers` first creates `WorkerOffers` for all active executors (registered in the internal `executorDataMap` cache).

Note

`WorkerOffer` represents a resource offer with CPU cores available on an executor.

`makeOffers` then requests `TaskSchedulerImpl` to generate tasks for the available `WorkerOffers` followed by launching the tasks on respective executors.

Note

`makeOffers` uses `TaskSchedulerImpl` that was given when `CoarseGrainedSchedulerBackend` was created.

Note

Tasks are described using `TaskDescription` that holds...[FIXME](#)

Note

`makeOffers` is used when `CoarseGrainedSchedulerBackend` RPC endpoint (`DriverEndpoint`) handles [ReviveOffers](#) or [RegisterExecutor](#) messages.

Making Executor Resource Offer on Single Executor (for Launching Tasks) — makeOffers Internal Method

```
makeOffers(executorId: String): Unit
```

`makeOffers` makes sure that the input `executorId` is alive.

Note

`makeOffers` does nothing when the input `executorId` is registered as pending to be removed or got lost.

`makeOffers` finds the executor data (in `executorDataMap` registry) and creates a [WorkerOffer](#).

Note

`WorkerOffer` represents a resource offer with CPU cores available on an executor.

`makeOffers` then requests `TaskSchedulerImpl` to generate tasks for the `WorkerOffer` followed by [launching the tasks](#) (on the executor).

Note

`makeOffers` is used when `CoarseGrainedSchedulerBackend` RPC endpoint (`DriverEndpoint`) handles [StatusUpdate](#) messages.

Launching Tasks on Executors — `launchTasks` Method

```
launchTasks(tasks: Seq[Seq[TaskDescription]]): Unit
```

`launchTasks` flattens (and hence "destroys" the structure of) the input `tasks` collection and takes one task at a time. Tasks are described using [TaskDescription](#).

Note

The input `tasks` collection contains one or more [TaskDescriptions](#) per executor (and the "task partitioning" per executor is of no use in `launchTasks` so it simply flattens the input data structure).

`launchTasks` encodes the `TaskDescription` and makes sure that the encoded task's size is below the [maximum RPC message size](#).

Note

The [maximum RPC message size](#) is calculated when `coarseGrainedSchedulerBackend` is created and corresponds to `spark.rpc.message.maxSize` Spark property (with maximum of `2047` MB).

If the size of the encoded task is acceptable, `launchTasks` finds the `ExecutorData` of the executor that has been assigned to execute the task (in `executorDataMap` internal registry) and decreases the executor's [available number of cores](#).

Note

`ExecutorData` tracks the number of free cores of an executor (as `freeCores`).

Note

The default task scheduler in Spark — `TaskSchedulerImpl` — uses `spark.task.cpus` Spark property to control the number of tasks that can be scheduled per executor.

You should see the following DEBUG message in the logs:

```
DEBUG DriverEndpoint: Launching task [taskId] on executor id: [executorId] hostname: [executorHost].
```

In the end, `launchTasks` sends the (serialized) task to associated executor to launch the task (by sending a [LaunchTask](#) message to the executor's RPC endpoint with the serialized task in size `SerializableBuffer`).

Note

`ExecutorData` tracks the [RpcEndpointRef](#) of executors to send serialized tasks to (as `executorEndpoint`).

Important

This is the moment in a task's lifecycle when the driver sends the serialized task to an assigned executor.

In case the size of a serialized `TaskDescription` equals or exceeds the [maximum RPC message size](#), `launchTasks` finds the [TaskSetManager](#) (associated with the `TaskDescription`) and [aborts it](#) with the following message:

Serialized task [id]:[index] was [limit] bytes, which exceeds max allowed: `spark.rpc.message.maxSize` ([`maxRpcMessageSize`] bytes). Consider increasing `spark.rpc.message.maxSize` or using broadcast variables for large values.

Note

`launchTasks` uses the [registry of active TaskSetManagers](#) per task id from [TaskSchedulerImpl](#) that was given when [coarseGrainedSchedulerBackend](#) was created.

Note

Scheduling in Spark relies on cores only (not memory), i.e. the number of tasks Spark can run on an executor is limited by the number of cores available only. When submitting a Spark application for execution both executor resources — memory and cores — can however be specified explicitly. It is the job of a cluster manager to monitor the memory and take action when its use exceeds what was assigned.

Note

`launchTasks` is used when [CoarseGrainedSchedulerBackend](#) makes resource offers on [single](#) or [all](#) executors in a cluster.

Creating DriverEndpoint Instance

`DriverEndpoint` takes the following when created:

- [RpcEnv](#)
- Collection of Spark properties and their values

`DriverEndpoint` initializes the internal registries and counters.

RegisterExecutor Handler

```
RegisterExecutor(  
    executorId: String,  
    executorRef: RpcEndpointRef,  
    hostname: String,  
    cores: Int,  
    logUrls: Map[String, String])  
extends CoarseGrainedClusterMessage
```

Note

`RegisterExecutor` is sent when `coarseGrainedExecutorBackend` (RPC Endpoint) is started.

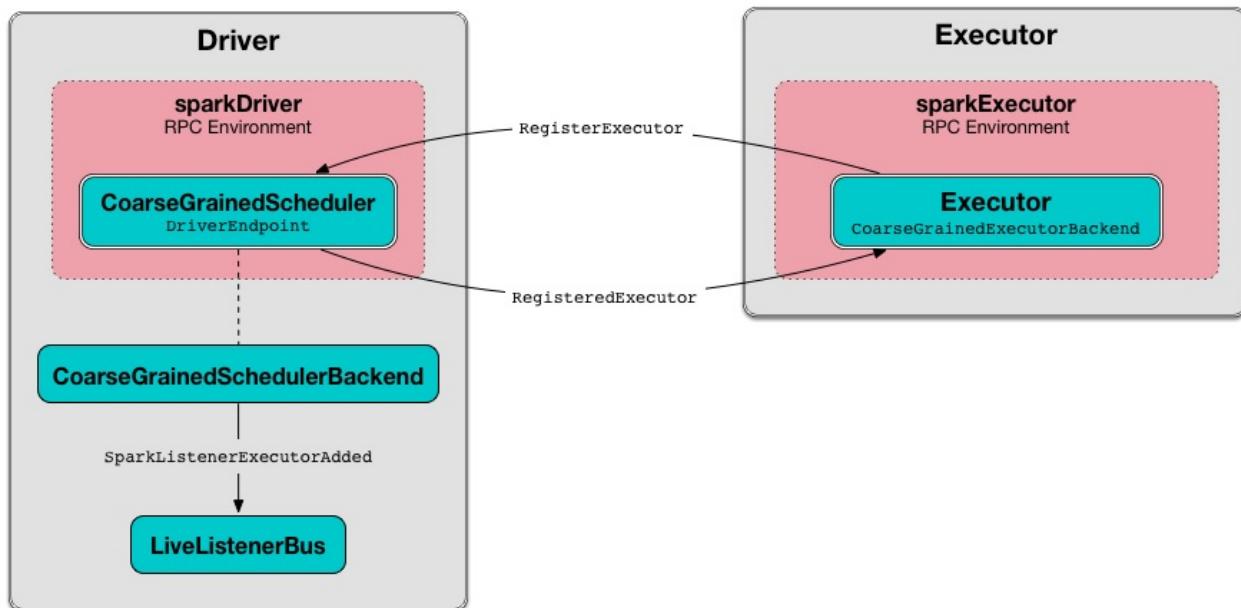


Figure 3. Executor registration (RegisterExecutor RPC message flow)

When received, `DriverEndpoint` makes sure that no other executors were registered under the input `executorId` and that the input `hostname` is not blacklisted.

Note

`DriverEndpoint` uses `TaskSchedulerImpl` (for the list of blacklisted nodes) that was specified when `CoarseGrainedSchedulerBackend` was created.

If the requirements hold, you should see the following INFO message in the logs:

```
INFO Registered executor [executorRef] ([address]) with ID [executorId]
```

`DriverEndpoint` does the bookkeeping:

- Registers `executorId` (in `addressToExecutorId`)

- Adds `cores` (in `totalCoreCount`)
- Increments `totalRegisteredExecutors`
- Creates and registers `ExecutorData` for `executorId` (in `executorDataMap`)
- Updates `currentExecutorIdCounter` if the input `executorId` is greater than the current value.

If `numPendingExecutors` is greater than `0`, you should see the following DEBUG message in the logs and `DriverEndpoint` decrements `numPendingExecutors`.

```
DEBUG Decrementing number of pending executors ([numPendingExecutors] left)
```

`DriverEndpoint` sends `RegisteredExecutor` message back (that is to confirm that the executor was registered successfully).

Note

`DriverEndpoint` uses the input `executorRef` as the executor's `RpcEndpointRef`.

`DriverEndpoint` replies `true` (to acknowledge the message).

`DriverEndpoint` then announces the new executor by posting `SparkListenerExecutorAdded` to `LiveListenerBus` (with the current time, executor id, and `ExecutorData`).

In the end, `DriverEndpoint` makes executor resource offers (for launching tasks).

If however there was already another executor registered under the input `executorId`, `DriverEndpoint` sends `RegisterExecutorFailed` message back with the reason:

```
Duplicate executor ID: [executorId]
```

If however the input `hostname` is `blacklisted`, you should see the following INFO message in the logs:

```
INFO Rejecting [executorId] as it has been blacklisted.
```

`DriverEndpoint` sends `RegisterExecutorFailed` message back with the reason:

```
Executor is blacklisted: [executorId]
```

StatusUpdate Handler

```

StatusUpdate(
    executorId: String,
    taskId: Long,
    state: TaskState,
    data: SerializableBuffer)
extends CoarseGrainedClusterMessage

```

Note

`StatusUpdate` is sent when `CoarseGrainedExecutorBackend` sends task status updates to the driver.

When `StatusUpdate` is received, `DriverEndpoint` passes the task's status update to `TaskSchedulerImpl`.

Note

`TaskSchedulerImpl` is specified when `CoarseGrainedSchedulerBackend` is created.

If the task has finished, `DriverEndpoint` updates the number of cores available for work on the corresponding executor (registered in `executorDataMap`).

Note

`DriverEndpoint` uses `TaskSchedulerImpl`'s `spark.task.cpus` as the number of cores that became available after the task has finished.

`DriverEndpoint` makes an executor resource offer on the single executor.

When `DriverEndpoint` found no executor (in `executorDataMap`), you should see the following WARN message in the logs:

```
WARN Ignored task status update ([taskId] state [state]) from unknown executor with ID [executorId]
```

KillTask Handler

```

KillTask(
    taskId: Long,
    executor: String,
    interruptThread: Boolean)
extends CoarseGrainedClusterMessage

```

Note

`KillTask` is sent when `CoarseGrainedSchedulerBackend` kills a task.

When `KillTask` is received, `DriverEndpoint` finds `executor` (in `executorDataMap` registry).

If found, `DriverEndpoint` [passes the message on to the executor](#) (using its registered RPC endpoint for `CoarseGrainedExecutorBackend`).

Otherwise, you should see the following WARN in the logs:

```
WARN Attempted to kill task [taskId] for unknown executor [executor].
```

Removing Executor from Internal Registries (and Notifying TaskSchedulerImpl and Posting SparkListenerExecutorRemoved) — `removeExecutor` Internal Method

```
removeExecutor(executorId: String, reason: ExecutorLossReason): Unit
```

When `removeExecutor` is executed, you should see the following DEBUG message in the logs:

```
DEBUG Asked to remove executor [executorId] with reason [reason]
```

`removeExecutor` then tries to find the `executorId` executor (in `executorDataMap` internal registry).

If the `executorId` executor was found, `removeExecutor` removes the executor from the following registries:

- [addressToExecutorId](#)
- [executorDataMap](#)
- [executorsPendingLossReason](#)
- [executorsPendingToRemove](#)

`removeExecutor` decrements:

- [totalCoreCount](#) by the executor's `totalCores`
- [totalRegisteredExecutors](#)

In the end, `removeExecutor` notifies `TaskSchedulerImpl` that an [executor was lost](#).

Note

`removeExecutor` uses [TaskSchedulerImpl](#) that is specified when `CoarseGrainedSchedulerBackend` is created.

`removeExecutor` posts [SparkListenerExecutorRemoved](#) to [LiveListenerBus](#) (with the `executorId` `executor`).

If however the `executorId` `executor` could not be found, `removeExecutor` requests [BlockManagerMaster](#) to remove the executor asynchronously.

Note	<code>removeExecutor</code> uses SparkEnv to access the current BlockManager and then BlockManagerMaster .
------	--

You should see the following INFO message in the logs:

```
INFO Asked to remove non-existent executor [executorId]
```

Note	<code>removeExecutor</code> is used when DriverEndpoint handles RemoveExecutor message and gets disassociated with a remote RPC endpoint of an executor.
------	--

ExecutorBackend — Pluggable Executor Backends

`ExecutorBackend` is a [pluggable interface](#) that [TaskRunners](#) use to [send task status updates](#) to a scheduler.

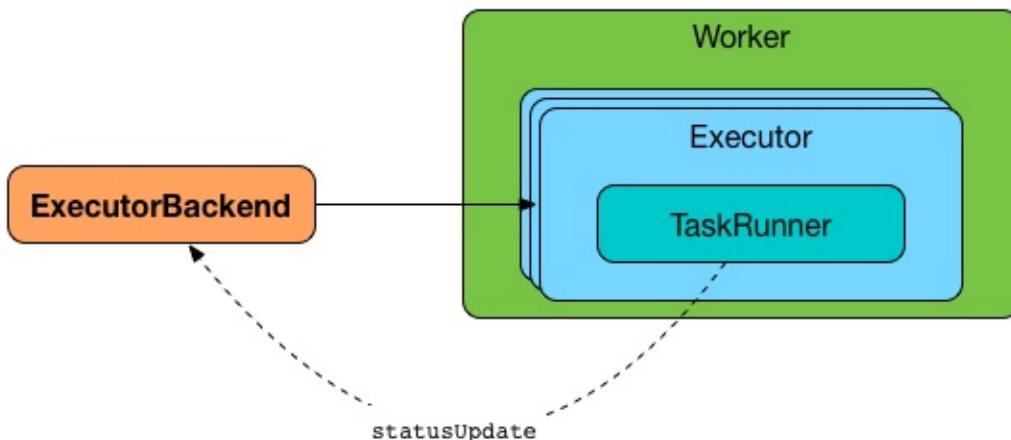


Figure 1. ExecutorBackend receives notifications from TaskRunners

Note	TaskRunner manages a single individual task and is managed by an Executor to launch a task.
------	---

Caution	FIXME What is "a scheduler" in this context?
---------	--

It is effectively a bridge between the driver and an executor, i.e. there are two endpoints running.

There are three concrete executor backends:

1. [CoarseGrainedExecutorBackend](#)
2. [LocalSchedulerBackend](#) (for [local run mode](#))
3. [MesosExecutorBackend](#)

ExecutorBackend Contract

```

trait ExecutorBackend {
  def statusUpdate(taskId: Long, state: TaskState, data: ByteBuffer): Unit
}
  
```

Note	<code>ExecutorBackend</code> is a <code>private[spark]</code> contract.
------	---

Table 1. ExecutorBackend Contract

Method	Description
statusUpdate	Used when TaskRunner runs a task (to send task status updates).

CoarseGrainedExecutorBackend

`CoarseGrainedExecutorBackend` is a [standalone application](#) that is started in a resource container when:

1. Spark Standalone's `StandaloneSchedulerBackend` starts
2. Spark on YARN's `ExecutorRunnable` is started.
3. Spark on Mesos's `MesosCoarseGrainedSchedulerBackend` launches Spark executors

When [started](#), `CoarseGrainedExecutorBackend` registers the Executor RPC endpoint to communicate with the driver (i.e. with `CoarseGrainedScheduler` RPC endpoint).

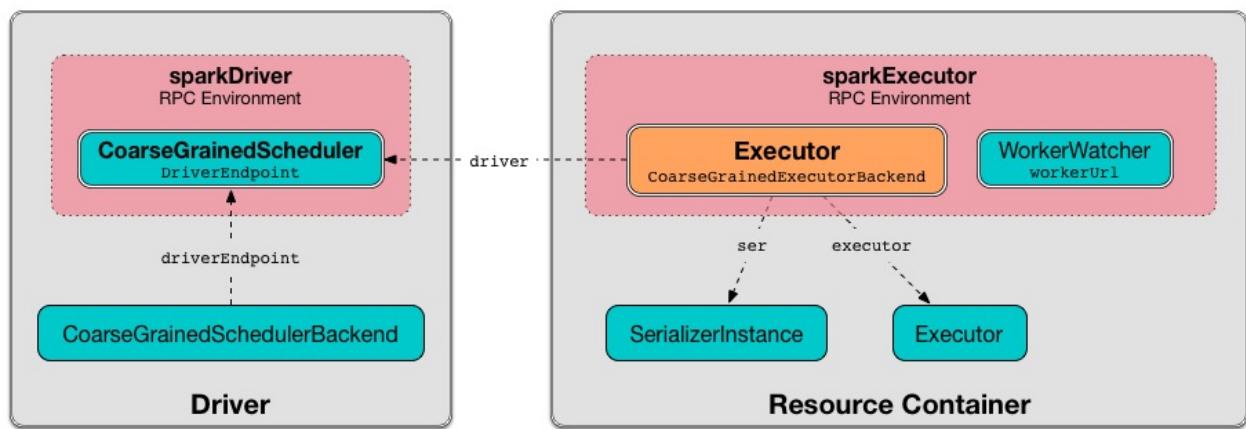


Figure 1. CoarseGrainedExecutorBackend Communicates with Driver's CoarseGrainedSchedulerBackend Endpoint

When [launched](#), `CoarseGrainedExecutorBackend` immediately connects to the owning `CoarseGrainedSchedulerBackend` to inform that it is ready to launch tasks.

`CoarseGrainedExecutorBackend` is an [ExecutorBackend](#) that controls the lifecycle of a single [executor](#) and sends [the executor's status updates](#) to the driver.

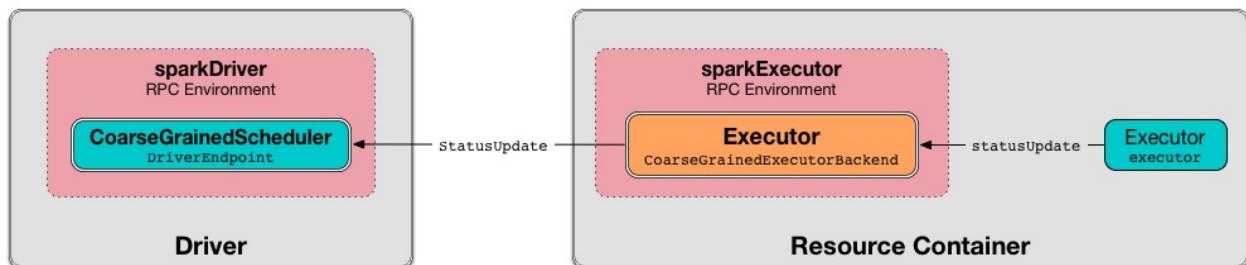


Figure 2. CoarseGrainedExecutorBackend Sending Task Status Updates to Driver's CoarseGrainedScheduler Endpoint

`CoarseGrainedExecutorBackend` is a [ThreadSafeRpcEndpoint](#) that [connects to the driver](#) (before accepting [messages](#)) and [shuts down when the driver disconnects](#).

Table 1. CoarseGrainedExecutorBackend's Executor RPC Endpoint Messages (in alphabetical order)

Message	Description
KillTask	
LaunchTask	Forwards launch task requests from the driver to the single managed coarse-grained <code>executor</code> .
RegisteredExecutor	Creates the single managed <code>Executor</code> . Sent exclusively when <code>CoarseGrainedSchedulerBackend</code> receives <code>RegisterExecutor</code> .
RegisterExecutorFailed	
StopExecutor	
Shutdown	

Table 2. CoarseGrainedExecutorBackend's Internal Properties

Name	Initial Value	Description
ser	SerializerInstance	Initialized when <code>CoarseGrainedExecutorBackend</code> is created. NOTE: <code>CoarseGrainedExecutorBackend</code> uses the input <code>env</code> to access <code>closureSerializer</code> .
driver	(empty)	RpcEndpointRef of the driver FIXME
stopping	false	Enabled when <code>CoarseGrainedExecutorBackend</code> gets notified to stop itself or shut down the managed executor. Used when <code>CoarseGrainedExecutorBackend</code> RPC Endpoint gets notified that a remote RPC endpoint disconnected.
executor	(uninitialized)	Single managed coarse-grained Executor managed exclusively by the <code>CoarseGrainedExecutorBackend</code> to forward launch and kill task requests to from the driver. Initialized after <code>CoarseGrainedExecutorBackend</code> has registered with <code>CoarseGrainedSchedulerBackend</code> and stopped when <code>CoarseGrainedExecutorBackend</code> gets requested to shut down.
Tip	<p>Enable <code>INFO</code> logging level for <code>org.apache.spark.executor.CoarseGrainedExecutorBackend</code> logger to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.executor.CoarseGrainedExecutorBackend=INFO</pre>	

Forwarding Launch Task Request to Executor (from Driver) — LaunchTask Message Handler

```
LaunchTask(data: SerializableBuffer) extends CoarseGrainedClusterMessage
```

Note

`CoarseGrainedExecutorBackend` acts as a proxy between the driver and the managed single `executor` and merely re-packages `LaunchTask` payload (as serialized `data`) to pass it along for execution.

`LaunchTask` first decodes `TaskDescription` from `data`. You should see the following INFO message in the logs:

```
INFO CoarseGrainedExecutorBackend: Got assigned task [id]
```

`LaunchTask` then launches the task on the executor (passing itself as the owning `ExecutorBackend` and decoded `TaskDescription`).

If `executor` is not available, `LaunchTask` terminates `CoarseGrainedExecutorBackend` with the error code `1` and `ExecutorLossReason` with the following message:

```
Received LaunchTask command but executor was null
```

Note

`LaunchTask` is sent when `CoarseGrainedSchedulerBackend` launches tasks (one per task).

Sending Task Status Updates to Driver— `statusUpdate` Method

Note

`statusUpdate` is a part of `ExecutorBackend` contract to send task status updates to a scheduler (on the driver).

```
statusUpdate(taskId: Long, state: TaskState, data: ByteBuffer): Unit
```

`statusUpdate` creates a `StatusUpdate` (with the input `taskId`, `state`, and `data` together with the `executor id`) and sends it to the `driver` (if already defined).

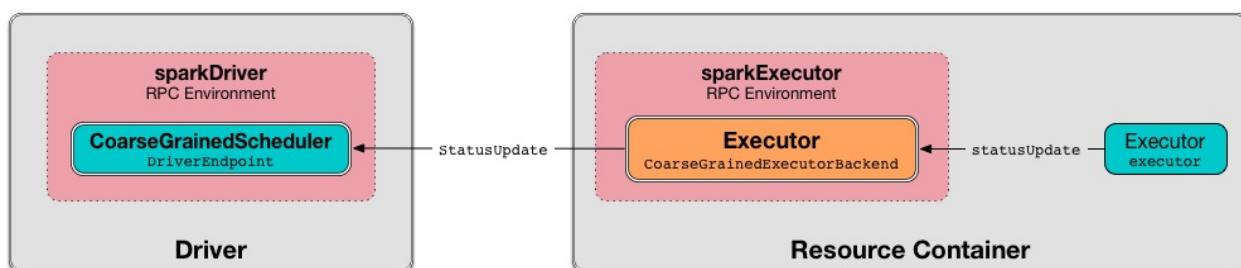


Figure 3. CoarseGrainedExecutorBackend Sending Task Status Updates to Driver's CoarseGrainedScheduler Endpoint

When no `driver` is available, you should see the following WARN message in the logs:

```
WARN Drop [msg] because has not yet connected to driver
```

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`.

Launching CoarseGrainedExecutorBackend Standalone Application (in Resource Container) — `main` Method

`CoarseGrainedExecutorBackend` is a standalone application (i.e. comes with `main` entry method) that parses [command-line arguments](#) and runs `CoarseGrainedExecutorBackend`'s [Executor RPC endpoint](#) to communicate with the driver.

Table 3. `CoarseGrainedExecutorBackend` Command-Line Arguments

Argument	Required?	Description
<code>--driver-url</code>	yes	Driver's URL. See driver's URL
<code>--executor-id</code>	yes	Executor id
<code>--hostname</code>	yes	Host name
<code>--cores</code>	yes	Number of cores (that must be greater than <code>0</code>).
<code>--app-id</code>	yes	Application id
<code>--worker-url</code>	no	Worker's URL, e.g. <code>spark://Worker@192.168.1.6:64557</code> NOTE: <code>--worker-url</code> is only used in Spark Standalone to enforce fate-sharing with the worker.
<code>--user-class-path</code>	no	User-defined class path entry which can be an URL or path to a resource (often a jar file) to be added to CLASSPATH; can be specified multiple times.

When executed with unrecognized command-line arguments or required arguments are missing, `main` shows the usage help and exits (with exit status `1`).

```
$ ./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>
```

	<code>main</code> is used when:
Note	<ul style="list-style-type: none"> • Spark Standalone's <code>StandaloneSchedulerBackend</code> starts. • Spark on YARN's <code>ExecutorRunnable</code> is started (in a YARN resource container). • Spark on Mesos's <code>MesosCoarseGrainedSchedulerBackend</code> launches Spark executors

Running CoarseGrainedExecutorBackend (and Registering Executor RPC Endpoint) — `run` Internal Method

```
run(
  driverUrl: String,
  executorId: String,
  hostname: String,
  cores: Int,
  appId: String,
  workerUrl: Option[String],
  userClassPath: scala.Seq[URL]): Unit
```

When executed, `run` executes `Utils.initDaemon(log)`.

Caution	<code>FIXME</code> What does <code>initDaemon</code> do?
Note	<code>run</code> runs itself with a Hadoop <code>UserGroupInformation</code> (as a thread local variable distributed to child threads for authenticating HDFS and YARN calls).
Note	<code>run</code> expects a clear <code>hostname</code> with no <code>:</code> included (for a port perhaps).

`run` uses `spark.executor.port` Spark property (or `0` if not set) for the port to [create a `RpcEnv`](#) called **driverPropsFetcher** (together with the input `hostname` and `clientMode` enabled).

`run` [resolves `RpcEndpointRef`](#) for the input `driverUrl` and requests `SparkAppConfig` (by posting a blocking `RetrieveSparkAppConfig`).

Important	This is the first moment when <code>CoarseGrainedExecutorBackend</code> initiates communication with the driver available at <code>driverUrl</code> through <code>RpcEnv</code> .
-----------	---

`run` uses `SparkAppConfig` to get the driver's `sparkProperties` and adds `spark.app.id` Spark property with the value of the input `appId` .

`run` [shuts `driverPropsFetcher` RPC Endpoint down](#).

`run` creates a **SparkConf** using the Spark properties fetched from the driver, i.e. with the executor-related Spark settings if they [were missing](#) and the [rest unconditionally](#).

If `spark.yarn.credentials.file` Spark property is defined in `SparkConf`, you should see the following INFO message in the logs:

```
INFO Will periodically update credentials from: [spark.yarn.credentials.file]
```

`run` [requests the current `SparkHadoopUtil` to start start the credential updater](#).

Note	<code>run</code> uses <code>SparkHadoopUtil.get</code> to access the current <code>SparkHadoopUtil</code> .
------	---

`run` [creates `SparkEnv`](#) for executors (with the input `executorId`, `hostname` and `cores`, and `isLocal` disabled).

Important	This is the moment when <code>SparkEnv</code> gets created with all the executor services.
-----------	--

`run` [sets up an RPC endpoint with the name **Executor** and `CoarseGrainedExecutorBackend` as the endpoint](#).

(only in Spark Standalone) If the optional input `workerUrl` was defined, `run` sets up an RPC endpoint with the name **WorkerWatcher** and `workerwatcher` RPC endpoint.

Note	The optional input <code>workerUrl</code> is defined only when --worker-url command-line argument was used to launch <code>CoarseGrainedExecutorBackend</code> standalone application. --worker-url is only used in Spark Standalone .
------	---

`run`'s main thread is blocked until `RpcEnv` terminates and only the RPC endpoints process RPC messages.

Once `RpcEnv` has terminated, `run` stops the credential updater.

Caution	<code>FIXME</code> Think of the place for <code>utils.initDaemon</code> , <code>Utils.getProcessName</code> et al.
---------	--

Note	<code>run</code> is used exclusively when <code>CoarseGrainedExecutorBackend</code> standalone application is launched.
------	---

Creating CoarseGrainedExecutorBackend Instance

`CoarseGrainedExecutorBackend` takes the following when created:

1. `RpcEnv`
2. `driverUrl`
3. `executorId`
4. `hostname`
5. `cores`
6. `userClassPath`
7. `SparkEnv`

Note	<code>driverUrl</code> , <code>executorId</code> , <code>hostname</code> , <code>cores</code> and <code>userClassPath</code> correspond to <code>coarseGrainedExecutorBackend</code> standalone application's command-line arguments.
------	---

`CoarseGrainedExecutorBackend` initializes the internal properties.

Note	<code>CoarseGrainedExecutorBackend</code> is created (to act as an RPC endpoint) when Executor RPC endpoint is registered.
------	--

Registering with Driver — `onStart` Method

<pre>onStart(): Unit</pre>

Note	<code>onStart</code> is a part of <code>RpcEndpoint</code> contract that is executed before a RPC endpoint starts accepting messages.
------	---

When executed, you should see the following INFO message in the logs:

```
INFO CoarseGrainedExecutorBackend: Connecting to driver: [driverUrl]
```

Note `driverUrl` is given when `CoarseGrainedExecutorBackend` is created.

`onStart` then takes the `RpcEndpointRef` of the driver asynchronously and initializes the internal `driver` property. `onStart` sends a blocking `RegisterExecutor` message immediately (with `executorId`, `RpcEndpointRef` to itself, `hostname`, `cores` and `log URLs`).

In case of failures, `onStart` terminates `CoarseGrainedExecutorBackend` with the error code `1` and the reason (and no notification to the driver):

```
Cannot register with driver: [driverUrl]
```

Creating Single Managed Executor — RegisteredExecutor Message Handler

```
RegisteredExecutor
extends CoarseGrainedClusterMessage with RegisterExecutorResponse
```

When `RegisteredExecutor` is received, you should see the following INFO in the logs:

```
INFO CoarseGrainedExecutorBackend: Successfully registered with driver
```

`CoarseGrainedExecutorBackend` creates a `Executor` (with `isLocal` disabled) that becomes the single managed `Executor`.

Note `CoarseGrainedExecutorBackend` uses `executorId`, `hostname`, `env`, `userClassPath` to create the `Executor` that are specified when `CoarseGrainedExecutorBackend` is created.

If creating the `Executor` fails with a non-fatal exception, `RegisteredExecutor` terminates `CoarseGrainedExecutorBackend` with the reason:

```
Unable to create executor due to [message]
```

Note `RegisteredExecutor` is sent exclusively when `CoarseGrainedSchedulerBackend` RPC Endpoint receives a `RegisterExecutor` (that is sent right before `CoarseGrainedExecutorBackend` RPC Endpoint starts accepting messages which happens when `CoarseGrainedExecutorBackend` is started).

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`.

Killing Tasks — `KillTask` Message Handler

`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 Handler

```
case object StopExecutor
  extends CoarseGrainedClusterMessage
```

When `StopExecutor` is received, the handler turns `stopping` internal flag on. You should see the following INFO message in the logs:

```
INFO CoarseGrainedExecutorBackend: Driver commanded a shutdown
```

In the end, the handler sends a `Shutdown` message to itself.

Note	<code>StopExecutor</code> message is sent when <code>CoarseGrainedSchedulerBackend</code> RPC Endpoint (aka <code>DriverEndpoint</code>) processes <code>StopExecutors</code> or <code>RemoveExecutor</code> messages.
------	---

Shutdown Handler

```
case object Shutdown
  extends CoarseGrainedClusterMessage
```

`Shutdown` turns `stopping` internal flag on and starts the `CoarseGrainedExecutorBackend-stop-executor` thread that stops the owned `Executor` (using `executor` reference).

Note

`shutdown` message is sent exclusively when `CoarseGrainedExecutorBackend receives StopExecutor`.

Terminating CoarseGrainedExecutorBackend (and Notifying Driver with RemoveExecutor) — `exitExecutor` Method

```
exitExecutor(  
    code: Int,  
    reason: String,  
    throwable: Throwable = null,  
    notifyDriver: Boolean = true): Unit
```

When `exitExecutor` is executed, you should see the following ERROR message in the logs (followed by `throwable` if available):

```
ERROR Executor self-exiting due to : [reason]
```

If `notifyDriver` is enabled (it is by default) `exitExecutor` informs the `driver` that the executor should be removed (by sending a blocking `RemoveExecutor` message with `executor id` and a `ExecutorLossReason` with the input `reason`).

You may see the following WARN message in the logs when the notification fails.

```
Unable to notify the driver due to [message]
```

In the end, `exitExecutor` terminates the `CoarseGrainedExecutorBackend` JVM process with the status `code`.

Note

`exitExecutor` uses Java's `System.exit` and initiates JVM's shutdown sequence (and executing all registered shutdown hooks).

Note

- `exitExecutor` is used when:
 - `CoarseGrainedExecutorBackend` fails to associate with the `driver`, create a managed executor or register with the driver
 - no `executor` has been created before `launch` or `kill` task requests
 - `driver` has disconnected.

onDisconnected Callback

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

start Method

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

stop Method

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

requestTotalExecutors

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

Extracting Log URLs — extractLogUrls Method

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

MesosExecutorBackend

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

registered Method

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

BlockManager — Key-Value Store for Blocks

`BlockManager` is a key-value store for blocks of data (simply *blocks*) in Spark. `BlockManager` acts as a local cache that runs on every "node" in a Spark application, i.e. the `driver` and `executors` (and is created when `SparkEnv` is created).

`BlockManager` provides interface for uploading and fetching blocks both locally and remotely using various stores, i.e. `memory`, `disk`, and `off-heap`.

When `BlockManager` is created, it creates its own private instances of `DiskBlockManager`, `BlockInfoManager`, `MemoryStore` and `DiskStore` (that it immediately wires together, i.e. `BlockInfoManager` with `MemoryStore` and `DiskStore` with `DiskBlockManager`).

The common idiom in Spark to access a `BlockManager` regardless of a location, i.e. the driver or executors, is through `SparkEnv`:

```
SparkEnv.get.blockManager
```

`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.

Tip	Use Web UI , esp. Storage and Executors tabs, to monitor the memory used.
-----	---

Tip	Use <code>spark-submit</code> 's command-line options, i.e. <code>--driver-memory</code> for the driver and <code>--executor-memory</code> for executors or their equivalents as Spark properties, i.e. <code>spark.executor.memory</code> and <code>spark.driver.memory</code> , to control the memory for storage memory.
-----	---

A `BlockManager` is created when a [Spark application starts](#) and must be [initialized](#) before it is fully operable.

When [External Shuffle Service is enabled](#), `BlockManager` uses [ExternalShuffleClient](#) to read other executors' shuffle files.

`BlockManager` uses [BlockManagerSource](#) to report metrics under the name **BlockManager**.

Table 1. BlockManager's Internal Properties

Name	Initial Value	Description
diskBlockManager	FIXME	DiskBlockManager for...FIXME
maxMemory	Total available on-heap and off-heap memory for storage (in bytes)	Total maximum value that BlockManager can ever possibly use (that depends on MemoryManager and may vary over time).

Tip	Enable <code>INFO</code> , <code>DEBUG</code> or <code>TRACE</code> logging level for <code>org.apache.spark.storage.BlockManager</code> logger to see what happens inside. Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.storage.BlockManager=TRACE</code>
	Refer to Logging .

Tip	You may want to shut off <code>WARN</code> messages being printed out about the current state of blocks using the following line to cut the noise: <code>log4j.logger.org.apache.spark.storage.BlockManager=OFF</code>
-----	---

getLocations Method

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

blockIdsToHosts Method

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

getLocationBlockIds Method

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

getPeers Method

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

releaseAllLocksForTask Method

Caution

FIXME

memoryStore Property

Caution

FIXME

stop Method

Caution

FIXME

putSingle Method

Caution

FIXME

Note

`putSingle` is used when `TorrentBroadcast` reads the blocks of a broadcast variable and stores them in a local `BlockManager`.

Getting Ids of Existing Blocks (For a Given Filter) — getMatchingBlockIds Method

Caution

FIXME

Note

`getMatchingBlockIds` is used to handle `GetMatchingBlockIds` messages.

getLocalValues Method

```
getLocalValues(blockId: BlockId): Option[BlockResult]
```

```
getLocalValues ...FIXME
```

Internally, when `getLocalValues` is executed, you should see the following DEBUG message in the logs:

```
DEBUG BlockManager: Getting local block [blockId]
```

```
getLocalValues obtains a read lock for blockId .
```

When no `blockId` block was found, you should see the following DEBUG message in the logs and `getLocalValues` returns "nothing" (i.e. `NONE`).

```
DEBUG Block [blockId] was not found
```

When the `blockId` block was found, you should see the following DEBUG message in the logs:

```
DEBUG Level for block [blockId] is [level]
```

If `blockId` block has memory level and is registered in `MemoryStore`, `getLocalValues` returns a `BlockResult` as `Memory` read method and with a `CompletionIterator` for an iterator:

1. Values iterator from `MemoryStore` for `blockId` for "deserialized" persistence levels.
2. Iterator from `SerializerManager` after the data stream has been deserialized for the `blockId` block and the bytes for `blockId` block for "serialized" persistence levels.

Note

`getLocalValues` is used when `TorrentBroadcast` reads the blocks of a broadcast variable and stores them in a local `BlockManager`.

Caution

[FIXME](#)

getRemoteValues Internal Method

```
getRemoteValues[T: ClassTag](blockId: BlockId): Option[BlockResult]
```

`getRemoteValues` ...[FIXME](#)

Retrieving Block from Local or Remote Block Managers — get Method

```
get[T](blockId: BlockId): Option[BlockResult]
```

`get` attempts to get the `blockId` block from a local block manager first before querying remote block managers.

Internally, `get` tries to get `blockId` block from the local `BlockManager`. If the `blockId` block was found, you should see the following INFO message in the logs and `get` returns the local `BlockResult`.

```
INFO Found block [blockId] locally
```

If however the `blockId` block was not found locally, `get` tries to [get the block from remote BlockManager](#)s. If the `blockId` block was retrieved from a remote `BlockManager`, you should see the following INFO message in the logs and `get` returns the remote [BlockResult](#).

```
INFO Found block [blockId] remotely
```

In the end, `get` returns "nothing" (i.e. `NONE`) when the `blockId` block was not found either in the local `BlockManager` or any remote `BlockManager`.

Note

`get` is used when `BlockManager` is requested to [getOrElseUpdate a block](#), [getSingle](#) and to [compute a BlockRDD](#).

getSingle Method

Caution[FIXME](#)

getOrElseUpdate Method

Caution[FIXME](#)

```
getOrElseUpdate[T](
  blockId: BlockId,
  level: StorageLevel,
  classTag: ClassTag[T],
  makeIterator: () => Iterator[T]): Either[BlockResult, Iterator[T]]
```

```
getOrElseUpdate ...FIXME
```

Getting Local Block Data As Bytes — getLocalBytes Method

Caution[FIXME](#)

getRemoteBytes Method

Caution[FIXME](#)

Finding Shuffle Block Data — `getBlockData` Method

Caution

FIXME

`removeBlockInternal` Method

Caution

FIXME

Is External Shuffle Service Enabled?

— `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?

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 Data Locally — `putBlockData` Method

```
putBlockData(
    blockId: BlockId,
    data: ManagedBuffer,
    level: StorageLevel,
    classTag: ClassTag[_]): Boolean
```

`putBlockData` simply stores `blockId` locally (given the given storage `level`).

Note

`putBlockData` is a part of `BlockDataManager` contract.

Internally, `putBlockData` wraps `ChunkedByteBuffer` around `data` buffer's NIO `ByteBuffer` and calls `putBytes`.

Note

`putBlockData` is used when `NettyBlockRpcServer` handles a `UploadBlock` message.

Storing Block Bytes Locally — `putBytes` Method

```
putBytes(  
    blockId: BlockId,  
    bytes: ChunkedByteBuffer,  
    level: StorageLevel,  
    tellMaster: Boolean = true): Boolean
```

`putBytes` stores the `blockId` block (with `bytes` bytes and `level` storage level).

`putBytes` simply passes the call on to the internal `doPutBytes`.

Note

`putBytes` is executed when `TaskRunner` sends a task result via `BlockManager`, `BlockManager` puts a block locally and in `TorrentBroadcast`.

`doPutBytes` Internal Method

```
def doPutBytes[T](  
    blockId: BlockId,  
    bytes: ChunkedByteBuffer,  
    level: StorageLevel,  
    classTag: ClassTag[T],  
    tellMaster: Boolean = true,  
    keepReadLock: Boolean = false): Boolean
```

`doPutBytes` calls the internal helper `doPut` with a function that accepts a `BlockInfo` and does the uploading.

Inside the function, if the `storage level`'s replication is greater than 1, it immediately starts `replication` of the `blockId` block on a separate thread (from `futureExecutionContext` thread pool). The replication uses the input `bytes` and `level` storage level.

For a memory storage level, the function checks whether the storage `level` is deserialized or not. For a serialized storage `level`, `BlockManager`'s `SerializerManager` `deserializes bytes` into an iterator of values that `MemoryStore` stores. If however the storage `level` is not serialized, the function requests `MemoryStore` to store the bytes

If the put did not succeed and the storage level is to use disk, you should see the following WARN message in the logs:

```
WARN BlockManager: Persisting block [blockId] to disk instead.
```

And `DiskStore` stores the bytes.

Note

`DiskStore` is requested to store the bytes of a block with memory and disk storage level only when `MemoryStore` has failed.

If the storage level is to use disk only, `DiskStore` stores the bytes.

`doPutBytes` requests `current block status` and if the block was successfully stored, and the driver should know about it (`tellMaster`), the function `reports the current storage status of the block to the driver`. The `current TaskContext metrics` are updated with the updated block status (only when executed inside a task where `TaskContext` is available).

You should see the following DEBUG message in the logs:

```
DEBUG BlockManager: Put block [blockId] locally took [time] ms
```

The function waits till the earlier asynchronous replication finishes for a block with replication level greater than `1`.

The final result of `doPutBytes` is the result of storing the block successful or not (as computed earlier).

Note

`doPutBytes` is called exclusively from `putBytes` method.

replicate Internal Method

Caution

[FIXME](#)

maybeCacheDiskValuesInMemory Method

Caution

[FIXME](#)

doPutIterator Method

Caution

[FIXME](#)

doPut Internal Method

```
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 (with `level` storage level) that `BlockInfoManager` managed to create a write lock for.

If the block has already been created (and `BlockInfoManager` did not manage to create a write lock for), the following WARN message is printed out to the logs:

```
WARN Block [blockId] already exists on this machine; not re-adding it
```

`doPut` releases the read lock for the block when `keepReadLock` flag is disabled and returns `None` immediately.

If however the write lock has been given, `doPut` executes `putBody`.

If the result of `putBody` is `None` the block is considered saved successfully.

For successful save and `keepReadLock` enabled, `BlockInfoManager` is requested to downgrade an exclusive write lock for `blockId` to a shared read lock.

For successful save and `keepReadLock` disabled, `BlockInfoManager` is requested to release lock on `blockId`.

For unsuccessful save, the block is removed from memory and disk stores 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 <code>BlockManagerSlaveEndpoint</code> while handling <code>RemoveRdd</code> messages.
------	--

Removing Broadcast Blocks — `removeBroadcast` Method

```
removeBroadcast(broadcastId: Long, tellMaster: Boolean): Int
```

`removeBroadcast` removes all the blocks of the input `broadcastId` broadcast.

Internally, it starts by printing out the following DEBUG message to the logs:

```
DEBUG Removing broadcast [broadcastId]
```

It then requests all the `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 <code>BlockManagerSlaveEndpoint</code> while handling <code>RemoveBroadcast</code> messages.
------	--

Getting Block Status — `getStatus` Method

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

Creating BlockManager Instance

`BlockManager` takes the following when 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 created, `BlockManager` sets [externalShuffleServiceEnabled](#) internal flag per `spark.shuffle.service.enabled` Spark property.

`BlockManager` then creates an instance of [DiskBlockManager](#) (requesting `deleteFilesOnStop` when an external shuffle service is not in use).

`BlockManager` creates an instance of [BlockInfoManager](#) (as `blockInfoManager`).

`BlockManager` creates **block-manager-future** daemon cached thread pool with 128 threads maximum (as `futureExecutionContext`).

`BlockManager` creates a [MemoryStore](#) and [DiskStore](#).

[MemoryManager](#) gets the [MemoryStore](#) object assigned.

`BlockManager` calculates the maximum memory to use (as `maxMemory`) by requesting the maximum [on-heap](#) and [off-heap](#) storage memory from the assigned `MemoryManager`.

Note

[UnifiedMemoryManager](#) is the default `MemoryManager` (as of Spark 1.6).

`BlockManager` 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.

`BlockManager` 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.

`BlockManager` sets the maximum number of failures before this block manager refreshes the block locations from the driver (as `maxFailuresBeforeLocationRefresh`).

`BlockManager` registers [BlockManagerSlaveEndpoint](#) with the input [RpcEnv](#), itself, and [MapOutputTracker](#) (as `slaveEndpoint`).

shuffleClient

Caution

[FIXME](#)

(that is assumed to be a [ExternalShuffleClient](#))

shuffleServerId

Caution

[FIXME](#)

Initializing BlockManager— `initialize` Method

```
initialize(appId: String): Unit
```

`initialize` initializes a `BlockManager` 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.

`initialize` does the following in order:

1. Initializes [BlockTransferService](#)
2. Initializes the internal shuffle client, be it [ExternalShuffleClient](#) or [BlockTransferService](#).
3. Registers itself with the driver's `BlockManagerMaster` (using the `id`, `maxMemory` and its `slaveEndpoint`).

The `BlockManagerMaster` reference is passed in when the `BlockManager` is created on the driver and executors.

4. Sets `shuffleServerId` to an instance of `BlockManagerId` given an executor id, host name and port for [BlockTransferService](#).
5. It creates the address of the server that serves this executor's shuffle files (using `shuffleServerId`)

Caution

[FIXME](#) Review the initialize procedure again

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

`initialize` is called when the driver is launched (and `SparkContext` is created) and when an `Executor` is created (for `CoarseGrainedExecutorBackend` and `MesosExecutorBackend`).

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 BlockManager with Driver and Reporting Blocks — `reregister` Method

```
reregister(): Unit
```

When executed, `reregister` prints the following INFO message to the logs:

```
INFO BlockManager: BlockManager [blockManagerId] re-registering with master
```

`reregister` then 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`.

`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 when a `Executor` was informed 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` and `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.

reportAllBlocks Method

Caution

FIXME

Note

`reportAllBlocks` is called when `BlockManager` is requested to [re-register all blocks to the driver](#).

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 `getBlockData`, `doPutBytes`, `doPutIterator`, `dropFromMemory` and `removeBlockInternal`.

Reporting Block Status Update to Driver — `tryToReportBlockStatus` Internal Method

```
def tryToReportBlockStatus(  
    blockId: BlockId,  
    info: BlockInfo,  
    status: BlockStatus,  
    droppedMemorySize: Long = 0L): Boolean
```

`tryToReportBlockStatus` reports block status update to `BlockManagerMaster` and returns its response.

Note

`tryToReportBlockStatus` is used when `BlockManager` `reportAllBlocks` or `reportBlockStatus`.

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.

Broadcast Values

When a new broadcast value is created, `TorrentBroadcast` blocks are put in the block manager.

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](#)

Execution Context

block-manager-future is the execution context for...[FIXME](#)

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 using [spark.app.id](#) Spark property for the id.

BlockResult

`BlockResult` is a description of a fetched block with the `readMethod` and `bytes`.

Registering Task with BlockInfoManager — `registerTask` Method

```
registerTask(taskAttemptId: Long): Unit
```

`registerTask` registers the input `taskAttemptId` with `BlockInfoManager`.

Note	<code>registerTask</code> is used exclusively when <code>Task</code> runs.
------	--

Offering DiskBlockObjectWriter To Write Blocks To Disk (For Current BlockManager) — `getDiskWriter` Method

```
getDiskWriter(  
    blockId: BlockId,  
    file: File,  
    serializerInstance: SerializerInstance,  
    bufferSize: Int,  
    writeMetrics: ShuffleWriteMetrics): DiskBlockObjectWriter
```

`getDiskWriter` creates a `DiskBlockObjectWriter` with `spark.shuffle.sync` Spark property for syncwrites .

Note	<code>getDiskWriter</code> uses the same <code>serializerManager</code> that was used to create a <code>BlockManager</code> .
------	---

Note	<code>getDiskWriter</code> is used when <code>BypassMergeSortShuffleWriter</code> writes records into one single shuffle block data file, in <code>ShuffleExternalSorter</code> , <code>UnsafeSorterSpillWriter</code> , <code>ExternalSorter</code> , and <code>ExternalAppendOnlyMap</code> .
------	---

Recording Updated BlockStatus In Current Task's TaskMetrics

— `addUpdatedBlockStatusToTaskMetrics` Internal Method

```
addUpdatedBlockStatusToTaskMetrics(blockId: BlockId, status: BlockStatus): Unit
```

`addUpdatedBlockStatusToTaskMetrics` takes an active `TaskContext` (if available) and records updated `BlockStatus` for `Block` (in the task's `TaskMetrics`).

Note	<code>addUpdatedBlockStatusToTaskMetrics</code> is used when <code>BlockManager</code> <code>doPutBytes</code> (for a block that was successfully stored), <code>doPut</code> , <code>doPutIterator</code> , removes blocks from memory (possibly spilling it to disk) and removes block from memory and disk.
------	--

Settings

Table 2. Spark Properties

Spark Property	Default Value	Description
<code>spark.blockManager.port</code>	0	Port to use for the block manager when a more specific setting for the driver or executors is not provided.
<code>spark.shuffle.sync</code>	false	Controls whether <code>DiskBlockObjectWriter</code> should force outstanding writes to disk when committing a single atomic block, i.e. all operating system buffers should synchronize with the disk to ensure that all changes to a file are in fact recorded in the storage.

MemoryStore

Memory store (`MemoryStore`) manages blocks.

`MemoryStore` requires [SparkConf](#), [BlockInfoManager](#), [SerializerManager](#), [MemoryManager](#) and [BlockEvictionHandler](#).

Table 1. `MemoryStore` Internal Registries

Name	Description
<code>entries</code>	<p>Collection of ...FIXME</p> <p><code>entries</code> is Java's <code>LinkedHashMap</code> with the initial capacity of <code>32</code>, the load factor of <code>0.75</code> and <code>access-order</code> ordering mode (i.e. iteration is in the order in which its entries were last accessed, from least-recently accessed to most-recently).</p> <p>NOTE: <code>entries</code> is Java's java.util.LinkedHashMap.</p>

Caution [FIXME](#) Where are these dependencies used?

Caution [FIXME](#) Where is the `MemoryStore` created? What params provided?

Note `MemoryStore` is a `private[spark]` class.

Tip Enable `INFO` or `DEBUG` logging level for `org.apache.spark.storage.memory.MemoryStore` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.storage.memory.MemoryStore=DEBUG
```

Refer to [Logging](#).

releaseUnrollMemoryForThisTask Method

Caution

[FIXME](#)

getValues Method

```
getValues(blockId: BlockId): Option[Iterator[_]]
```

`getValues` does...[FIXME](#)

getBytes Method

```
getBytes(blockId: BlockId): Option[ChunkedByteBuffer]
```

`getBytes` does...[FIXME](#)

Is Block Available?— contains Method

```
contains(blockId: BlockId): Boolean
```

`contains` returns `true` when the internal `entries` registry contains `blockId`.

putIteratorAsBytes Method

```
putIteratorAsBytes[T](
  blockId: BlockId,
  values: Iterator[T],
  classTag: ClassTag[T],
  memoryMode: MemoryMode): Either[PartiallySerializedBlock[T], Long]
```

`putIteratorAsBytes` tries to put the `blockId` block in memory store as bytes.

Caution

[FIXME](#)

putIteratorAsValues Method

```
putIteratorAsValues[T](
  blockId: BlockId,
  values: Iterator[T],
  classTag: ClassTag[T]): Either[PartiallyUnrolledIterator[T], Long]
```

`putIteratorAsValues` tries to put the `blockId` block in memory store as `values`.

Note

`putIteratorAsValues` is a `private[storage]` method.

Note

is called when `BlockManager` stores bytes of a block or iterator of values of a block or when attempting to cache spilled values read from disk.

Evicting Blocks to Free Space

Caution

FIXME

Removing Block

Caution

FIXME

Acquiring Storage Memory for Blocks — `putBytes` Method

```
putBytes[T](
  blockId: BlockId,
  size: Long,
  memoryMode: MemoryMode,
  _bytes: () => ChunkedByteBuffer): Boolean
```

`putBytes` requests storage memory for `blockId` from `MemoryManager` and registers the block in `entries` internal registry.

Internally, `putBytes` first makes sure that `blockId` block has not been registered already in `entries` internal registry.

`putBytes` then requests `size` memory for the `blockId` block in a given `memoryMode` from the current `MemoryManager`.

Note

`memoryMode` can be `ON_HEAP` or `OFF_HEAP` and is a property of a [StorageLevel](#).

```
import org.apache.spark.storage.StorageLevel._
scala> MEMORY_AND_DISK.useOffHeap
res0: Boolean = false

scala> OFF_HEAP.useOffHeap
res1: Boolean = true
```

If successful, `putBytes` "materializes" `_bytes` byte buffer and makes sure that the size is exactly `size`. It then registers a `SerializedMemoryEntry` (for the 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` only after `blockId` was successfully registered in the internal [entries](#) registry.

Settings

Table 2. Spark Properties

Spark Property	Default Value	Description
<code>spark.storage.unrollMemoryThreshold</code>	<code>1k</code>	

DiskStore

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

putBytes

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

Removing Block

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

BlockDataManager — Block Storage Management API

`BlockDataManager` is a pluggable [interface](#) to manage storage for blocks of data (aka *block storage management API*). Blocks are identified by `BlockId` that has a globally unique identifier (`name`) and stored as [ManagedBuffer](#).

Table 1. Types of BlockIds

Name	Description
RDDBlockId	Described by <code>rddId</code> and <code>splitIndex</code> Created when a <code>RDD</code> is requested to getOrCompute a <code>partition</code> (identified by <code>splitIndex</code>).
ShuffleBlockId	Described by <code>shuffleId</code> , <code>mapId</code> and <code>reduceId</code>
ShuffleDataBlockId	Described by <code>shuffleId</code> , <code>mapId</code> and <code>reduceId</code>
ShuffleIndexBlockId	Described by <code>shuffleId</code> , <code>mapId</code> and <code>reduceId</code>
BroadcastBlockId	Described by <code>broadcastId</code> identifier and optional <code>field</code>
TaskResultBlockId	Described by <code>taskId</code>
StreamBlockId	Described by <code>streamId</code> and <code>uniqueId</code>

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
```

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 — Pluggable Block Transfers

`BlockTransferService` is a contract for `ShuffleClients` that can fetch and upload blocks synchronously or asynchronously.

Note `BlockTransferService` is a `private[spark]` abstract class .

Note `NettyBlockTransferService` is the only available implementation of `BlockTransferService` Contract.

Note `BlockTransferService` was introduced in [SPARK-3019 Pluggable block transfer interface \(BlockTransferService\)](#) and is available since Spark 1.2.0.

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 a `BlockTransferService` 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.

uploadBlockSync Method

```
uploadBlockSync(  
    hostname: String,  
    port: Int,  
    execId: String,  
    blockId: BlockId,  
    blockData: ManagedBuffer,  
    level: StorageLevel,  
    classTag: ClassTag[_]): Unit
```

`uploadBlockSync` is a mere blocking wrapper around [uploadBlock](#) that waits until the upload finishes.

Note	<code>uploadBlockSync</code> is only executed when BlockManager replicates a block to another node(s) (i.e. when a replication level is greater than 1).
------	--

NettyBlockTransferService — Netty-Based BlockTransferService

`NettyBlockTransferService` is a [BlockTransferService](#) that uses Netty for block transport (when [uploading](#) or [fetching](#) blocks of data).

Note

`NettyBlockTransferService` is created when [SparkEnv](#) is created (and later passed on to create a [BlockManager](#) for the driver and executors).

Tip

Enable `INFO` or `TRACE` logging level for `org.apache.spark.network.netty.NettyBlockTransferService` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.network.netty.NettyBlockTransferService=TRACE
```

Refer to [Logging](#).

Creating NettyBlockTransferService Instance

Caution

FIXME

fetchBlocks Method

```
fetchBlocks(
    host: String,
    port: Int,
    execId: String,
    blockIds: Array[String],
    listener: BlockFetchingListener): Unit
```

`fetchBlocks` ...[FIXME](#)

When executed, `fetchBlocks` prints out the following TRACE message in the logs:

```
TRACE Fetch blocks from [host]:[port] (executor id [execId])
```

`fetchBlocks` then creates a `RetryingBlockFetcher.BlockFetchStarter` where `createAndStart` method...[FIXME](#)

Depending on the maximum number of acceptable IO exceptions (such as connection timeouts) per request, if the number is greater than `0`, `fetchBlocks` creates `RetryingBlockFetcher` and starts it immediately.

Note

`RetryingBlockFetcher` is created with the `RetryingBlockFetcher.BlockFetchStarter` created earlier, the input `blockIds` and `listener`.

If however the number of retries is not greater than `0` (it could be `0` or less), the `RetryingBlockFetcher.BlockFetchStarter` created earlier is started (with the input `blockIds` and `listener`).

In case of any `Exception`, you should see the following ERROR message in the logs and the input `BlockFetchingListener` gets notified (using `onBlockFetchFailure` for every block id).

```
ERROR Exception while beginning fetchBlocks
```

Note

`fetchBlocks` is called when `BlockTransferService` fetches one block synchronously and `ShuffleBlockFetcherIterator` sends a request for blocks (using `sendRequest`).

Application Id — `appId` Property

Caution
[FIXME](#)

Initializing NettyBlockTransferService — `init` Method

```
init(blockDataManager: BlockDataManager): Unit
```

Note

`init` is a part of the `BlockTransferService` contract.

`init` starts a server for...[FIXME](#)

Internally, `init` creates a `NettyBlockRpcServer` (using the application id, a `JavaSerializer` and the input `blockDataManager`).

Caution

[FIXME](#) Describe security when `authEnabled` is enabled.

`init` creates a `TransportContext` with the `NettyBlockRpcServer` created earlier.

Caution

[FIXME](#) Describe `transportConf` and `TransportContext`.

`init` creates the internal `clientFactory` and a server.

Caution

[FIXME](#) What's the "a server"?

In the end, you should see the INFO message in the logs:

```
INFO NettyBlockTransferService: Server created on [hostName]:[port]
```

Note

`hostname` is given when `NettyBlockTransferService` is created and is controlled by `spark.driver.host` [Spark property](#) for the driver and differs per deployment environment for executors (as controlled by `--hostname` for `CoarseGrainedExecutorBackend`).

Uploading Block — `uploadBlock` Method

```
uploadBlock(  
    hostname: String,  
    port: Int,  
    execId: String,  
    blockId: BlockId,  
    blockData: ManagedBuffer,  
    level: StorageLevel,  
    classTag: ClassTag[_]): Future[Unit]
```

Note

`uploadBlock` is a part of the [BlockTransferService contract](#).

Internally, `uploadBlock` creates a `TransportClient` client to send a `UploadBlock` message (to the input `hostname` and `port`).

Note

`UploadBlock` message is processed by [NettyBlockRpcServer](#).

The `UploadBlock` message holds the [application id](#), the input `execId` and `blockId`. It also holds the serialized bytes for block metadata with `level` and `classTag` serialized (using the internal `JavaSerializer`) as well as the serialized bytes for the input `blockData` itself (this time however the serialization uses `ManagedBuffer.nioByteBuffer` method).

The entire `UploadBlock` message is further serialized before sending (using `TransportClient.sendRpc`).

Caution

[FIXME](#) Describe `TransportClient` and `clientFactory.createClient` .

When `blockId` block was successfully uploaded, you should see the following TRACE message in the logs:

```
TRACE NettyBlockTransferService: Successfully uploaded block [blockId]
```

When an upload failed, you should see the following ERROR message in the logs:

```
ERROR Error while uploading block [blockId]
```

Note

`uploadBlock` is executed when `BlockTransferService` does block upload in a blocking fashion.

UploadBlock Message

`UploadBlock` is a `BlockTransferMessage` that describes a block being uploaded, i.e. send over the wire from a [NettyBlockTransferService](#) to a [NettyBlockRpcServer](#).

Table 1. `UploadBlock` Attributes

Attribute	Description
<code>appId</code>	The application id (the block belongs to)
<code>execId</code>	The executor id
<code>blockId</code>	The block id
<code>metadata</code>	
<code>blockData</code>	The block data as an array of bytes

As an `Encodable`, `UploadBlock` can calculate the encoded size and do encoding and decoding itself to or from a `ByteBuf`, respectively.

NettyBlockRpcServer

`NettyBlockRpcServer` is a `RpcHandler` (i.e. a handler for `sendRPC()` messages sent by `TransportClient`s) that handles `BlockTransferMessage` messages for [NettyBlockTransferService](#).

`NettyBlockRpcServer` uses [OneForOneStreamManager](#) as the internal `streamManager`.

Table 1. `NettyBlockRpcServer` Messages

Message	Behaviour
<code>OpenBlocks</code>	Obtaining local blocks and registering them with the internal OneForOneStreamManager .
<code>UploadBlock</code>	Deserializes a block and stores it in BlockDataManager .

Tip Enable `TRACE` logging level to see received messages in the logs.

Tip Enable `TRACE` logging level for `org.apache.spark.network.netty.NettyBlockRpcServer` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.network.netty.NettyBlockRpcServer=TRACE
```

Refer to [Logging](#).

Creating NettyBlockRpcServer Instance

```
class NettyBlockRpcServer(
    appId: String,
    serializer: Serializer,
    blockManager: BlockDataManager)
extends RpcHandler
```

When created, `NettyBlockRpcServer` gets the application id (`appId`) and a `Serializer` and a `BlockDataManager`.

Note `NettyBlockRpcServer` is created when [NettyBlockTransferService](#) is initialized.

`NettyBlockRpcServer` merely creates the internal instance of [OneForOneStreamManager](#).

Note

As a `RpcHandler`, `NettyBlockRpcServer` uses the `OneForOneStreamManager` for `getStreamManager` (which is a part of the `RpcHandler` contract).

Obtaining Local Blocks and Registering with Internal OneForOneStreamManager — OpenBlocks Message Handler

When `OpenBlocks` arrives, `NettyBlockRpcServer` requests block data (from `BlockDataManager`) for every block id in the message. The block data is a collection of `ManagedBuffer` for every block id in the incoming message.

Note

`BlockDataManager` is given when `NettyBlockRpcServer` is created.

`NettyBlockRpcServer` then registers a stream of `ManagedBuffer`s (for the blocks) with the internal `StreamManager` under `streamId`.

Note

The internal `StreamManager` is `OneForOneStreamManager` and is created when `NettyBlockRpcServer` is created.

You should see the following TRACE message in the logs:

```
TRACE NettyBlockRpcServer: Registered streamId [streamId] with [size] buffers
```

In the end, `NettyBlockRpcServer` responds with a `StreamHandle` (with the `streamId` and the number of blocks). The response is serialized as a `ByteBuffer`.

Deserializing Block and Storing in BlockDataManager — UploadBlock Message Handler

When `UploadBlock` arrives, `NettyBlockRpcServer` deserializes the `metadata` of the input message to get the `StorageLevel` and `ClassTag` of the block being uploaded.

Note

`metadata` is serialized before `NettyBlockTransferService` sends a `UploadBlock` message (using the internal `JavaSerializer`) that is given as `serializer` when `NettyBlockRpcServer` is created.

`NettyBlockRpcServer` creates a `BlockID` for the block id and requests the `BlockDataManager` to store the block.

Note

The `BlockDataManager` is passed in when `NettyBlockRpcServer` is created.

In the end, `NettyBlockRpcServer` responds with a 0 -capacity `ByteBuffer`.

Note	<code>UploadBlock</code> is sent when <code>NettyBlockTransferService</code> uploads a block.
------	---

BlockManagerMaster — BlockManager for Driver

`BlockManagerMaster` runs on the driver.

`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	<code>BlockManagerMaster</code> is created in <code>SparkEnv</code> (for the driver and executors), and immediately used to create their <code>BlockManagers</code> .
------	---

Tip	<p>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 <code>conf/log4j.properties</code> :</p>
-----	--

```
log4j.logger.org.apache.spark.storage.BlockManagerMaster=INFO
```

Refer to [Logging](#).

removeExecutorAsync Method

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

contains Method

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

Creating BlockManagerMaster Instance

`BlockManagerMaster` takes the following when created:

- `RpcEndpointRef` to...[FIXME](#)
- `SparkConf`
- Flag whether `BlockManagerMaster` is created for the driver or executors.

`BlockManagerMaster` initializes the [internal registries and counters](#).

Removing Executor— `removeExecutor` Method

```
removeExecutor(execId: String): Unit
```

`removeExecutor` posts `RemoveExecutor` to `BlockManagerMaster` RPC endpoint and waits for a response.

If `false` in response comes in, a `SparkException` is thrown with the following message:

```
BlockManagerMasterEndpoint returned false, expected true.
```

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

```
INFO BlockManagerMaster: Removed executor [execId]
```

Note

`removeExecutor` is executed when `DAGScheduler` processes `ExecutorLostEvent`.

Removing Block— `removeBlock` Method

```
removeBlock(blockId: BlockId): Unit
```

`removeBlock` simply posts a `RemoveBlock` blocking message to `BlockManagerMaster` RPC endpoint (and ultimately disregards the response).

Removing RDD Blocks— `removeRdd` Method

```
removeRdd(rddId: Int, blocking: Boolean)
```

`removeRdd` removes all the blocks of `rddId` RDD, possibly in `blocking` fashion.

Internally, `removeRdd` 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.

Note `removeShuffle` is used exclusively when `ContextCleaner` removes a shuffle.

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 with Driver — `registerBlockManager` Method

```
registerBlockManager(  
    blockManagerId: BlockManagerId,  
    maxMemSize: Long,  
    slaveEndpoint: RpcEndpointRef): BlockManagerId
```

`registerBlockManager` prints the following INFO message to the logs:

```
INFO BlockManagerMaster: Registering BlockManager [blockManagerId]
```

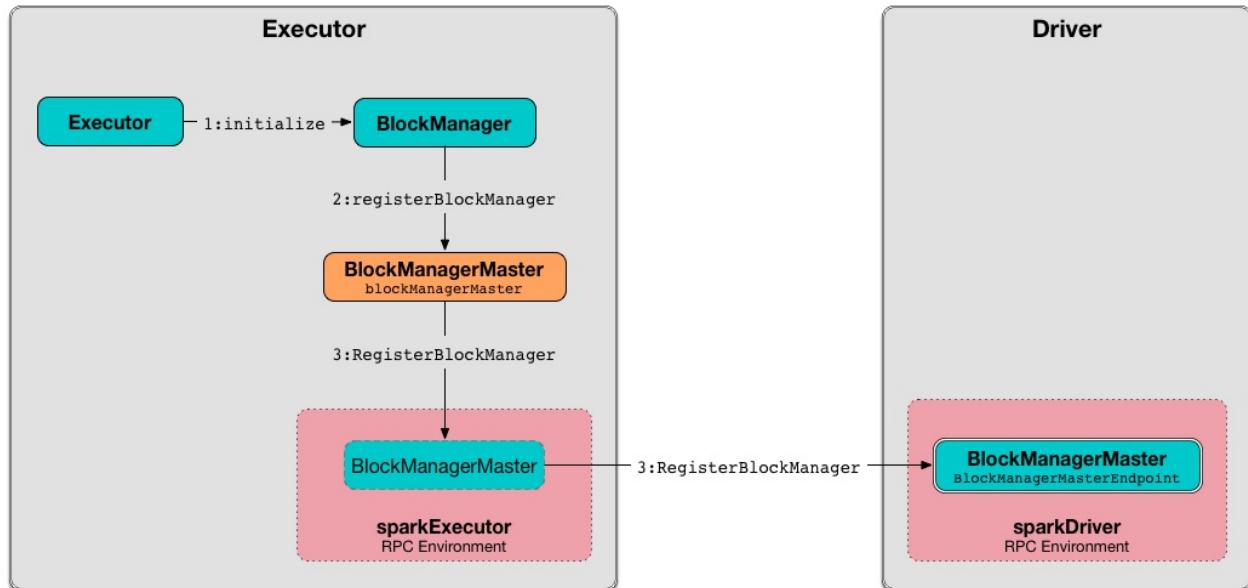


Figure 1. Registering BlockManager with the Driver

`registerBlockManager` then notifies the driver that the `blockManagerId` `BlockManager` tries to register. `registerBlockManager` posts a `blocking` `RegisterBlockManager` message to `BlockManagerMaster` RPC endpoint.

Note

The input `maxMemSize` is the total available on-heap and off-heap memory for storage on a `BlockManager`.

`registerBlockManager` waits until a confirmation comes (as `BlockManagerId`) that becomes the return value.

You should see the following INFO message in the logs:

```
INFO BlockManagerMaster: Registered BlockManager [updatedId]
```

Note

`registerBlockManager` is used when `BlockManager` is initialized or re-registers itself with the driver (and reports the blocks).

Relaying Block Status Update From BlockManager to Driver (by Sending Blocking `UpdateBlockInfo` to `BlockManagerMaster` RPC endpoint) — `updateBlockInfo` Method

```
updateBlockInfo(
    blockManagerId: BlockManagerId,
    blockId: BlockId,
    storageLevel: StorageLevel,
    memSize: Long,
    diskSize: Long): Boolean
```

`updateBlockInfo` sends a `blocking 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]
```

`updateBlockInfo` returns the response from the `BlockManagerMaster` RPC endpoint.

Note	<code>updateBlockInfo</code> is used when <code>BlockManager</code> reports a block status update to the driver.
------	--

Get Block Locations of One Block — `getLocations` Method

```
getLocations(blockId: BlockId): Seq[BlockManagerId]
```

`getLocations` posts a blocking `GetLocations` message to `BlockManagerMaster` RPC endpoint and returns the response.

Note	<code>getLocations</code> is used when <code>BlockManagerMaster</code> checks if a block was registered and <code>BlockManager</code> <code>getLocations</code> .
------	---

Get Block Locations for Multiple Blocks — `getLocations` Method

```
getLocations(blockIds: Array[BlockId]): IndexedSeq[Seq[BlockManagerId]]
```

`getLocations` posts a blocking `GetLocationsMultipleBlockIds` message to `BlockManagerMaster` RPC endpoint and returns the response.

Note	<code>getLocations</code> is used when <code>DAGScheduler</code> finds <code>BlockManagers</code> (and so <code>executors</code>) for cached <code>RDD partitions</code> and when <code>BlockManager</code> <code>getLocationBlockIds</code> and <code>blockIdsToHosts</code> .
------	--

Finding Peers of BlockManager — `getPeers` Internal Method

```
getPeers(blockManagerId: BlockManagerId): Seq[BlockManagerId]
```

`getPeers` posts a blocking `GetPeers` message to `BlockManagerMaster` RPC endpoint and returns the response.

Note

Peers of a `BlockManager` are the other `BlockManagers` in a cluster (except the driver's `BlockManager`). Peers are used to know the available executors in a Spark application.

Note

`getPeers` is used when `BlockManager` finds the peers of a `BlockManager`, Structured Streaming's `KafkaSource` and Spark Streaming's `KafkaRDD`.

getExecutorEndpointRef Method

```
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 Method

```
getMemoryStatus: Map[BlockManagerId, (Long, Long)]
```

`getMemoryStatus` posts a `GetMemoryStatus` message `BlockManagerMaster` RPC endpoint and waits for a response which becomes the return value.

Storage Status (Posting GetStorageStatus to BlockManagerMaster RPC endpoint)

— getStorageStatus Method

```
getStorageStatus: Array[StorageStatus]
```

`getStorageStatus` posts a `GetStorageStatus` message to `BlockManagerMaster` RPC endpoint and waits for a response which becomes the return value.

getBlockStatus Method

```
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 Method

```
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 Method

```
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 [ThreadSafeRpcEndpoint](#) for [BlockManagerMaster](#) under **BlockManagerMaster** name.

`BlockManagerMasterEndpoint` tracks status of the [BlockManagers](#) (on the executors) in a Spark application.

`BlockManagerMasterEndpoint` is created when `SparkEnv` is created (for the driver and executors).

Table 1. BlockManagerMaster RPC Endpoint's Messages (in alphabetical order)

Message	When posted?
<code>RegisterBlockManager</code>	Posted when <code>BlockManagerMaster</code> registers a <code>BlockManager</code> .
<code>UpdateBlockInfo</code>	Posted when <code>BlockManagerMaster</code> receives a block status update (from <code>BlockManager</code> on an executor).

Table 2. BlockManagerMasterEndpoint's Internal Registries and Counters

Name	Description
<code>blockManagerIdByExecutor</code>	FIXME
<code>blockManagerInfo</code>	Lookup table of <code>BlockManagerInfo</code> per <code>BlockManagerId</code> . Updated when <code>BlockManagerMasterEndpoint</code> registers a new <code>BlockManager</code> or removes a <code>BlockManager</code> .
<code>blockLocations</code>	Collection of <code>BlockId</code> s and their locations (as <code>BlockManagerId</code>). Used in <code>removeRdd</code> to remove blocks for a RDD, <code>removeBlockManager</code> to remove blocks after a <code>BlockManager</code> gets removed, <code>removeBlockFromWorkers</code> , <code>updateBlockInfo</code> , and <code>getLocations</code> .

Enable `INFO` logging level for `org.apache.spark.storage.BlockManagerMasterEndpoint` logger to see what happens inside.

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

Tip

```
log4j.logger.org.apache.spark.storage.BlockManagerMasterEndpoint=INFO
```

Refer to [Logging](#).

storageStatus Internal Method

Caution

[FIXME](#)

getLocationsMultipleBlockIds Method

Caution

[FIXME](#)

Removing Shuffle Blocks — removeShuffle Internal Method

Caution

[FIXME](#)

UpdateBlockInfo

```
class UpdateBlockInfo(
    var blockManagerId: BlockManagerId,
    var blockId: BlockId,
    var storageLevel: StorageLevel,
    var memSize: Long,
    var diskSize: Long)
```

When `RegisterBlockManager` arrives, `BlockManagerMasterEndpoint` ...[FIXME](#)

Caution

[FIXME](#)

RemoveExecutor

```
RemoveExecutor(execId: String)
```

When `RemoveExecutor` is received, `executor execId` is removed and the response `true` sent back.

Note

`RemoveExecutor` is posted when `blockManagerMaster` removes an executor.

Finding Peers of BlockManager — `getPeers` Internal Method

```
getPeers(blockManagerId: BlockManagerId): Seq[BlockManagerId]
```

`getPeers` finds all the registered `BlockManagers` (using `blockManagerInfo` internal registry) and checks if the input `blockManagerId` is amongst them.

If the input `blockManagerId` is registered, `getPeers` returns all the registered `BlockManagers` but the one on the driver and `blockManagerId`.

Otherwise, `getPeers` returns no `BlockManagers`.

Note

Peers of a `BlockManager` are the other `BlockManagers` in a cluster (except the driver's `BlockManager`). Peers are used to know the available executors in a Spark application.

Note

`getPeers` is used exclusively when `BlockManagerMasterEndpoint` handles `GetPeers` message.

Finding Peers of BlockManager — `GetPeers` Message

```
GetPeers(blockManagerId: BlockManagerId)
extends ToBlockManagerMaster
```

`GetPeers` replies with the `peers` of `blockManagerId`.

Note

Peers of a `BlockManager` are the other `BlockManagers` in a cluster (except the driver's `BlockManager`). Peers are used to know the available executors in a Spark application.

Note

`GetPeers` is posted when `BlockManagerMaster` requests the peers of a `BlockManager`.

BlockManagerHeartbeat

Caution**FIXME**

GetLocations Message

```
GetLocations(blockId: BlockId)
extends ToBlockManagerMaster
```

`GetLocations` replies with the [locations](#) of `blockId`.

Note

`GetLocations` is posted when [BlockManagerMaster](#) requests the block locations of a single block.

GetLocationsMultipleBlockIds Message

```
GetLocationsMultipleBlockIds(blockIds: Array[BlockId])
extends ToBlockManagerMaster
```

`GetLocationsMultipleBlockIds` replies with the [getLocationsMultipleBlockIds](#) for the input `blockIds`.

Note

`GetLocationsMultipleBlockIds` is posted when [BlockManagerMaster](#) requests the block locations for multiple blocks.

RegisterBlockManager Event

```
RegisterBlockManager(
  blockManagerId: BlockManagerId,
  maxMemSize: Long,
  sender: RpcEndpointRef)
```

When `RegisterBlockManager` arrives, [BlockManagerMasterEndpoint](#) registers the [BlockManager](#).

Registering BlockManager (on Executor) — register Internal Method

```
register(id: BlockManagerId, maxMemSize: Long, slaveEndpoint: RpcEndpointRef): Unit
```

`register` records the current time and registers [BlockManager](#) (using `BlockManagerId`) unless it has been registered already (in `blockManagerInfo` internal registry).

Note The input `maxMemSize` is the total available on-heap and off-heap memory for storage on a `BlockManager`.

Note `register` is executed when `RegisterBlockManager` has been received.

Note Registering a `BlockManager` can only happen once for an executor (identified by `BlockManagerId.executorId` in `blockManagerIdByExecutor` internal 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](#)
- [blockManagerInfo](#)

Caution [FIXME Why does `blockManagerInfo` require a new `System.currentTimeMillis\(\)` since `time` was already recorded?](#)

In either case, [SparkListenerBlockManagerAdded](#) is posted (to `listenerBus`).

Note The method can only be executed on the driver where `listenerBus` is available.

Caution [FIXME Describe `listenerBus` + omnigraffle it.](#)

Other RPC Messages

- [GetLocationsMultipleBlockIds](#)
- [GetRpcHostPortForExecutor](#)
- [GetMemoryStatus](#)
- [GetStorageStatus](#)

- GetBlockStatus
- GetMatchingBlockIds
- RemoveShuffle
- RemoveBroadcast
- RemoveBlock
- StopBlockManagerMaster
- BlockManagerHeartbeat
- HasCachedBlocks

Removing Executor— `removeExecutor` Internal Method

```
removeExecutor(execId: String)
```

`removeExecutor` prints the following INFO message to the logs:

```
INFO BlockManagerMasterEndpoint: Trying to remove executor [execId] from BlockManagerMaster.
```

If the `execId` executor is registered (in the internal `blockManagerIdByExecutor` internal registry), `removeExecutor` removes the corresponding `BlockManager`.

Note

`removeExecutor` is executed when `BlockManagerMasterEndpoint` receives a `RemoveExecutor` or registers a new `BlockManager` (and another `BlockManager` was already registered that is replaced by the new one).

Removing BlockManager— `removeBlockManager` Internal Method

```
removeBlockManager(blockManagerId: BlockManagerId)
```

`removeBlockManager` looks up `blockManagerId` and removes the executor it was working on from the internal registries:

- `blockManagerIdByExecutor`
- `blockManagerInfo`

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]
```

Note

`removeBlockManager` is used exclusively when `BlockManagerMasterEndpoint` removes an executor.

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.

Creating BlockManagerMasterEndpoint Instance

`BlockManagerMasterEndpoint` takes the following when created:

- `RpcEnv`
- Flag whether `BlockManagerMasterEndpoint` works in local or cluster mode
- `SparkConf`
- `LiveListenerBus`

`BlockManagerMasterEndpoint` initializes the [internal registries and counters](#).

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 [local directories](#).

Note

`DiskBlockManager` is used exclusively by `DiskStore` and created when `BlockManager` is created (and passed to `DiskStore`).

Enable `INFO` or `DEBUG` logging levels for `org.apache.spark.storage.DiskBlockManager` logger to see what happens inside.

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

Tip

```
log4j.logger.org.apache.spark.storage.DiskBlockManager=DEBUG
```

Refer to [Logging](#).

Finding File — `getFile` Method

Caution

FIXME

createTempShuffleBlock Method

```
createTempShuffleBlock(): (TempShuffleBlockId, File)
```

`createTempShuffleBlock` creates a temporary `TempShuffleBlockId` block.

Caution

FIXME

Collection of Locks for Local Directories — `subDirs` Internal Property

```
subDirs: Array[Array[File]]
```

`subDirs` is a collection of locks for every [local directory](#) where `DiskBlockManager` stores block data (with the columns being the number of local directories and the rows as collection of `subDirsPerLocalDir` size).

Note	<code>subDirs(n)</code> is to access <code>n</code> -th local directory.
------	--

getAllFiles Method

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

Creating DiskBlockManager Instance

```
DiskBlockManager(conf: SparkConf, deleteFilesOnStop: Boolean)
```

When created, `DiskBlockManager` uses [spark.diskStore.subDirectories](#) to set `subDirsPerLocalDir`.

`DiskBlockManager` creates one or many local directories to store block data (as `localDirs`). When not successful, you should see the following ERROR message in the logs and `DiskBlockManager` exits with error code `53`.

```
ERROR DiskBlockManager: Failed to create any local dir.
```

`DiskBlockManager` initializes the internal `subDirs` collection of locks for every local directory to store block data with an array of `subDirsPerLocalDir` size for files.

In the end, `DiskBlockManager` registers a shutdown hook to clean up the local directories for blocks.

Registering Shutdown Hook — addShutdownHook Internal Method

```
addShutdownHook(): AnyRef
```

`addShutdownHook` registers a shutdown hook to execute `doStop` at shutdown.

When executed, you should see the following DEBUG message in the logs:

```
DEBUG DiskBlockManager: Adding shutdown hook
```

`addShutdownHook` adds the shutdown hook so it prints the following INFO message and executes `doStop`.

```
INFO DiskBlockManager: Shutdown hook called
```

Removing Local Directories for Blocks — `doStop` Internal Method

```
doStop(): Unit
```

`doStop` deletes the local directories recursively (only when the constructor's `deleteFilesOnStop` is enabled and the parent directories are not registered to be removed at shutdown).

Creating Directories for Blocks — `createLocalDirs` Internal Method

```
createLocalDirs(conf: SparkConf): Array[File]
```

`createLocalDirs` creates `blockmgr-[random UUID]` directory under local directories to store block data.

Internally, `createLocalDirs` reads `local writable directories` and creates a subdirectory `blockmgr-[random UUID]` under every configured parent directory.

If successful, you should see the following INFO message in the logs:

```
INFO DiskBlockManager: Created local directory at [localDir]
```

When failed to create a local directory, you should see the following ERROR message in the logs:

```
ERROR DiskBlockManager: Failed to create local dir in [rootDir]. Ignoring this directory.
```

Getting Local Directories for Spark to Write Files — `Utils.getConfiguredLocalDirs` Internal Method

```
getConfiguredLocalDirs(conf: SparkConf): Array[String]
```

`getConfiguredLocalDirs` returns the local directories where Spark can write files.

Internally, `getConfiguredLocalDirs` uses `conf` [SparkConf](#) to know if [External Shuffle Service](#) is enabled (using `spark.shuffle.service.enabled`).

`getConfiguredLocalDirs` checks if [Spark runs on YARN](#) and if so, returns `LOCAL_DIRS` - controlled local directories.

In non-YARN mode (or for the driver in `yarn-client` mode), `getConfiguredLocalDirs` checks the following environment variables (in the order) and returns the value of the first met:

1. `SPARK_EXECUTOR_DIRS` environment variable
2. `SPARK_LOCAL_DIRS` environment variable
3. `MESOS_DIRECTORY` environment variable (only when External Shuffle Service is not used)

In the end, when no earlier environment variables were found, `getConfiguredLocalDirs` uses `spark.local.dir` [Spark](#) property or eventually `java.io.tmpdir` [System](#) property.

Getting Writable Directories in YARN

— `getYarnLocalDirs` Internal Method

```
getYarnLocalDirs(conf: SparkConf): String
```

`getYarnLocalDirs` uses `conf` [SparkConf](#) to read `LOCAL_DIRS` environment variable with comma-separated local directories (that have already been created and secured so that only the user has access to them).

`getYarnLocalDirs` throws an `Exception` with the message `Yarn Local dirs can't be empty` if `LOCAL_DIRS` environment variable was not set.

Checking If Spark Runs on YARN

— `isRunningInYarnContainer` Internal Method

```
isRunningInYarnContainer(conf: SparkConf): Boolean
```

`isRunningInYarnContainer` uses `conf` [SparkConf](#) to read Hadoop YARN's `CNTAINER_ID` [environment variable](#) to find out if Spark runs in a YARN container.

Note	<code>CONTAINER_ID</code> environment variable is exported by YARN NodeManager.
------	---

getAllBlocks Method

```
getAllBlocks(): Seq[BlockId]
```

`getAllBlocks` lists all the blocks currently stored on disk.

Internally, `getAllBlocks` takes the [block files](#) and returns their names (as `BlockId`).

Note	<code>getAllBlocks</code> is used when BlockManager computes the ids of existing blocks (for a given filter).
------	---

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.diskStore.subDirectories</code>	64	The number of ... FIXME

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.

Table 1. `BlockInfoManager` Internal Registries and Counters

Name	Description
<code>infos</code>	Tracks <code>BlockInfo</code> per block (as <code>BlockId</code>).
<code>readLocksByTask</code>	Tracks tasks (by <code>TaskAttemptId</code>) and the blocks they locked for reading (as <code>BlockId</code>).
<code>writeLocksByTask</code>	Tracks tasks (by <code>TaskAttemptId</code>) and the blocks they locked for writing (as <code>BlockId</code>).

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](#).

registerTask Method

Caution

FIXME

Downgrading Exclusive Write Lock For Block to Shared Read Lock — `downgradeLock` Method

```
downgradeLock(blockId: BlockId): Unit
```

`downgradeLock` ...FIXME

Obtaining Read Lock For Block — `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]
```

`lockForReading` looks up the metadata of the `blockId` block (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. 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`, aka `NON_TASK_WRITER`, 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 for Block — `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, `blockId` block 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 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.

currentTaskAttemptId Method

Caution**FIXME**

Releasing Lock on Block — unlock Method

```
unlock(blockId: BlockId): Unit
```

`unlock` releases...[FIXME](#)

When executed, `unlock` starts by printing out the following TRACE message to the logs:

```
TRACE BlockInfoManager: Task [currentTaskAttemptId] releasing lock for [blockId]
```

`unlock` gets the metadata for `blockId`. It may throw a `IllegalStateException` if the block was not found.

If the [writer task](#) for the block is not `NO_WRITER`, it becomes so and the `blockId` block is removed from the internal [writeLocksByTask](#) registry for the [current task attempt](#).

Otherwise, if the writer task is indeed `NO_WRITER`, it is assumed that the [blockId block is locked for reading](#). The `readerCount` counter is decremented for the `blockId` block and the read lock removed from the internal [readLocksByTask](#) registry for the [current task attempt](#).

In the end, `unlock` wakes up all the threads waiting for the `BlockInfoManager` (using Java's `Object.notifyAll`).

Caution	FIXME What threads could wait?
---------	--

Releasing All Locks Obtained by Task — `releaseAllLocksForTask` Method

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

Removing Memory Block — `removeBlock` Method

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

`assertBlockIsLockedForWriting` Method

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

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 [id of the writer task](#).

`BlockInfo` 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` Counter

`readerCount` counter is the number of readers of the memory block, i.e. the number of read locks. It starts with `0`.

`readerCount` 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`

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 <code>RemoveShuffle</code> 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]
```

Note	<code>RemoveShuffle</code> is posted when BlockManagerMaster and BlockManagerMasterEndpoint remove all blocks for a shuffle.
------	--

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, askSlaves: Boolean = true)
```

`GetMatchingBlockIds` triggers a computation of [the memory and disk blocks matching `filter`](#) and sends it back.

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 (`asyncThreadPool`) 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

`BlockManagerSlaveEndpoint` uses Java's
java.util.concurrent.ThreadPoolExecutor.

DiskBlockObjectWriter

`DiskBlockObjectWriter` is a `java.io.OutputStream` that `BlockManager` offers for writing blocks to disk.

Whenever `DiskBlockObjectWriter` is requested to write a key-value pair, it makes sure that the underlying output streams are open.

`DiskBlockObjectWriter` can be in the following states (that match the state of the underlying output streams):

1. Initialized
2. Open
3. Closed

Table 1. DiskBlockObjectWriter's Internal Registries and Counters

Name	Description
initialized	Internal flag... FIXME Used when... FIXME
hasBeenClosed	Internal flag... FIXME Used when... FIXME
streamOpen	Internal flag... FIXME Used when... FIXME
objOut	FIXME Used when... FIXME
mcs	FIXME Used when... FIXME
bs	FIXME Used when... FIXME
objOut	FIXME Used when... FIXME
blockId	FIXME Used when... FIXME

Note

`DiskBlockObjectWriter` is a `private[spark]` class.

updateBytesWritten Method

Caution

[FIXME](#)

initialize Method

Caution

[FIXME](#)

Writing Bytes (From Byte Array Starting From Offset) — write Method

```
write(kvBytes: Array[Byte], offs: Int, len: Int): Unit
```

`write` ...[FIXME](#)

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

recordWritten Method

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

commitAndGet Method

```
commitAndGet(): FileSegment
```

Note	<code>commitAndGet</code> is used when... FIXME
------	---

close Method

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

Creating DiskBlockObjectWriter Instance

`DiskBlockObjectWriter` takes the following when created:

1. `file`
2. `serializerManager` — [SerializerManager](#)
3. `serializerInstance` — [SerializerInstance](#)
4. `bufferSize`
5. `syncWrites` flag
6. `writeMetrics` — [ShuffleWriteMetrics](#)
7. `blockId` — [BlockId](#)

`DiskBlockObjectWriter` initializes the internal registries and counters.

Writing Key-Value Pair — `write` Method

```
write(key: Any, value: Any): Unit
```

Before writing, `write` opens the stream unless already `open`.

`write` then writes the `key` first followed by writing the `value`.

In the end, `write recordWritten`.

Note	<code>write</code> is used when <code>BypassMergeSortShuffleWriter</code> writes records and in <code>ExternalAppendOnlyMap</code> , <code>ExternalSorter</code> and <code>WritablePartitionedPairCollection</code> .
------	---

Opening DiskBlockObjectWriter — `open` Method

```
open(): DiskBlockObjectWriter
```

`open` opens `DiskBlockObjectWriter`, i.e. initializes and re-sets `bs` and `objOut` internal output streams.

Internally, `open` makes sure that `DiskBlockObjectWriter` is not closed (i.e. `hasBeenClosed` flag is disabled). If it was, `open` throws a `IllegalStateException`:

```
Writer already closed. Cannot be reopened.
```

Unless `DiskBlockObjectWriter` has already been initialized (i.e. `initialized` flag is enabled), `open` initializes it (and turns `initialized` flag on).

Regardless of whether `DiskBlockObjectWriter` was already initialized or not, `open` requests `SerializerManager` to wrap `mcs` output stream for encryption and compression (for `blockId`) and sets it as `bs`.

Note	<code>open</code> uses <code>SerializerManager</code> that was specified when <code>DiskBlockObjectWriter</code> was created
------	--

`open` requests `SerializerInstance` to serialize `bs` output stream and sets it as `objOut`.

Note	<code>open</code> uses <code>SerializerInstance</code> that was specified when <code>DiskBlockObjectWriter</code> was created
------	---

In the end, `open` turns `streamOpen` flag on.

Note	<code>open</code> is used exclusively when <code>DiskBlockObjectWriter</code> writes a key-value pair or bytes from a specified byte array but the stream is not open yet.
------	--

BlockManagerSource — Metrics Source for BlockManager

`BlockManagerSource` is the metrics [Source](#) for `BlockManager`.

`BlockManagerSource` is registered under the name **BlockManager** (when `SparkContext` is created).

Table 1. `BlockManagerSource`'s Metrics

Name	Type	Description
<code>memory.maxMem_MB</code>	long	Requests <code>BlockManagerMaster</code> for storage status (for every <code>BlockManager</code>) and sums up their maximum memory limit.
<code>memory.remainingMem_MB</code>	long	Requests <code>BlockManagerMaster</code> for storage status (for every <code>BlockManager</code>) and sums up their memory remaining.
<code>memory.memUsed_MB</code>	long	Requests <code>BlockManagerMaster</code> for storage status (for every <code>BlockManager</code>) and sums up their memory used.
<code>disk.diskSpaceUsed_MB</code>	long	Requests <code>BlockManagerMaster</code> for storage status (for every <code>BlockManager</code>) and sums up their disk space used.

You can access the `BlockManagerSource` [metrics](#) using the web UI's port (as `spark.ui.port` property).

```
$ http --follow http://localhost:4040/metrics/json \
| jq '.gauges | keys | .[] | select(test(".driver.BlockManager"; "g"))'
"local-1488272192549.driver.BlockManager.disk.diskSpaceUsed_MB"
"local-1488272192549.driver.BlockManager.memory.maxMem_MB"
"local-1488272192549.driver.BlockManager.memory.memUsed_MB"
"local-1488272192549.driver.BlockManager.memory.remainingMem_MB"
```

StorageStatus

`StorageStatus` is a developer API that Spark uses to pass "just enough" information about registered `BlockManagers` in a Spark application between Spark services (mostly for monitoring purposes like [web UI](#) or [SparkListeners](#)).

Note	<p>There are two ways to access <code>StorageStatus</code> about all the known <code>BlockManagers</code> in a Spark application:</p> <ul style="list-style-type: none"> • <code>SparkContext.getExecutorStorageStatus</code> • Being a <code>SparkListener</code> and intercepting <code>onBlockManagerAdded</code> and <code>onBlockManagerRemoved</code> events
------	--

`StorageStatus` is [created](#) when:

- `BlockManagerMasterEndpoint` is requested for storage status (of every `BlockManager` in a Spark application)
- `StorageStatusListener` gets notified about a new `BlockManager` (in a Spark application)

Table 1. StorageStatus's Internal Registries and Counters

Name	Description
<code>_nonRddBlocks</code>	Lookup table of <code>BlockIds</code> per <code>BlockId</code> . Used when... FIXME
<code>_rddBlocks</code>	Lookup table of <code>BlockIds</code> with <code>BlockStatus</code> per RDD id. Used when... FIXME

updateStorageInfo Method

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

Creating StorageStatus Instance

`StorageStatus` takes the following when created:

- `BlockManagerId`
- Maximum memory — total available on-heap and off-heap memory for storage on the `BlockManager`

`StorageStatus` initializes the internal registries and counters.

Getting RDD Blocks For RDD — `rddBlocksById` Method

```
rddBlocksById(rddId: Int): Map[BlockId, BlockStatus]
```

`rddBlocksById` gives the blocks (as `BlockId` with their status as `BlockStatus`) that belong to `rddId` RDD.

Note

`rddBlocksById` is used when:

- `StorageStatusListener` removes the RDD blocks of an unpersisted RDD.
- `AllRDDResource` does `getRDDStorageInfo`
- `StorageUtils` does `getRddBlockLocations`

Removing Block (From Internal Registries) — `removeBlock` Internal Method

```
removeBlock(blockId: BlockId): Option[BlockStatus]
```

`removeBlock` removes `blockId` from `_rddBlocks` registry and returns it.

Internally, `removeBlock` updates block status of `blockId` (to be empty, i.e. removed).

`removeBlock` branches off per the type of `BlockId`, i.e. `RDDBlockId` or not.

For a `RDDBlockId`, `removeBlock` finds the RDD in `_rddBlocks` and removes the `blockId`. `removeBlock` removes the RDD (from `_rddBlocks`) completely, if there are no more blocks registered.

For a non- `RDDBlockId`, `removeBlock` removes `blockId` from `_nonRddBlocks` registry.

Note

`removeBlock` is used when `StorageStatusListener` removes RDD blocks for an unpersisted RDD or updates storage status for an executor.

MapOutputTracker — Shuffle Map Output Registry

`MapOutputTracker` is a Spark service that runs on the driver and executors that tracks the shuffle map outputs (with information about the `BlockManager` and estimated size of the reduce blocks per shuffle).

Note

`MapOutputTracker` is registered as the **MapOutputTracker** RPC Endpoint in the RPC Environment when `sparkEnv` is created.

There are two concrete `MapOutputTrackers`, i.e. one for the driver and another for executors:

- `MapOutputTrackerMaster` for the driver
- `MapOutputTrackerWorker` for executors

Given the different runtime environments of the driver and executors, accessing the current `MapOutputTracker` is possible using `SparkEnv`.

```
SparkEnv.get.mapOutputTracker
```

Table 1. `MapOutputTracker` Internal Registries and Counters

Name	Description
<code>mapStatuses</code>	Internal cache with <code>MapStatus</code> array (indexed by partition id) per <code>shuffle id</code> . Used when <code>MapOutputTracker</code> finds map outputs for a <code>ShuffleDependency</code> , updates epoch and unregisters a shuffle.
<code>epoch</code>	Tracks the epoch in a Spark application. Starts from <code>0</code> when <code>MapOutputTracker</code> is created. Can be updated (on <code>MapOutputTrackerWorkers</code>) or incremented (on the driver's <code>MapOutputTrackerMaster</code>).
<code>epochLock</code>	FIXME

`MapOutputTracker` is also used for `mapOutputTracker.containsShuffle` and `MapOutputTrackerMaster.registerShuffle` when a new `ShuffleMapStage` is created.

`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.

Note

`MapOutputTracker` is used in `BlockStoreShuffleReader` and when creating `BlockManager` and `BlockManagerSlaveEndpoint`.

trackerEndpoint Property

`trackerEndpoint` is a `RpcEndpointRef` that `MapOutputTracker` uses to send RPC messages.

`trackerEndpoint` is initialized when `SparkEnv` is created for the driver and executors and cleared when `MapOutputTrackerMaster` is stopped.

Creating MapOutputTracker Instance

Caution**FIXME**

deserializeMapStatuses Method

Caution**FIXME**

sendTracker Method

Caution**FIXME**

serializeMapStatuses Method

Caution**FIXME**

Computing Statistics for ShuffleDependency — getStatistics Method

```
getStatistics(dep: ShuffleDependency[_, _, _]): MapOutputStatistics
```

`getStatistics` returns a `MapOutputStatistics` which is simply a pair of the `shuffle id` (of the input `ShuffleDependency`) and the total sums of estimated sizes of the reduce shuffle blocks from all the `BlockManagers`.

Internally, `getStatistics` finds map outputs for the input `ShuffleDependency` and calculates the total sizes for the estimated sizes of the reduce block (in bytes) for every `MapStatus` and partition.

Note	The internal <code>totalsizes</code> array has the number of elements as specified by the number of partitions of the <code>Partitioner</code> of the input <code>ShuffleDependency</code> . <code>totalsizes</code> contains elements as a sum of the estimated size of the block for partition in a <code>BlockManager</code> (for a <code>MapStatus</code>).
Note	<code>getStatistics</code> is used when <code>DAGScheduler</code> accepts a <code>ShuffleDependency</code> for execution (and the corresponding <code>ShuffleMapStage</code> has already been computed) and gets notified that a <code>ShuffleMapTask</code> has completed (and map-stage jobs waiting for the stage are then marked as finished).

Computing BlockManagerIds with Their Blocks and Sizes — `getMapSizesByExecutorId` Methods

```
getMapSizesByExecutorId(shuffleId: Int, startPartition: Int, endPartition: Int)
: Seq[(BlockManagerId, Seq[(BlockId, Long)])]

getMapSizesByExecutorId(shuffleId: Int, reduceId: Int)
: Seq[(BlockManagerId, Seq[(BlockId, Long)])] (1)
```

1. Calls the other `getMapSizesByExecutorId` with `endPartition` as `reduceId + 1` and is used exclusively in tests.

Caution	<code>FIXME</code> How do the start and end partitions influence the return value?
---------	--

`getMapSizesByExecutorId` returns a collection of `BlockManagerIds` with their blocks and sizes.

When executed, you should see the following DEBUG message in the logs:

```
DEBUG Fetching outputs for shuffle [id], partitions [startPartition]-[endPartition]
```

`getMapSizesByExecutorId` finds map outputs for the input `shuffleId`.

Note	<code>getMapSizesByExecutorId</code> gets the map outputs for all the partitions (despite the method's signature).
------	--

In the end, `getMapSizesByExecutorId` converts shuffle map outputs (as `MapStatuses`) into the collection of `BlockManagerIds` with their blocks and sizes.

Note

`getMapSizesByExecutorId` is exclusively used when `BlockStoreShuffleReader` reads combined records for a reduce task.

Returning Current Epoch — `getEpoch` Method

```
getEpoch: Long
```

`getEpoch` returns the current epoch.

Note

`getEpoch` is used when `DAGScheduler` is notified that an executor was lost and when `TaskSetManager` is created (and sets the epoch for the tasks in a `TaskSet`).

Updating Epoch — `updateEpoch` Method

```
updateEpoch(newEpoch: Long): Unit
```

`updateEpoch` updates epoch when the input `newEpoch` is greater (and hence more recent) and clears the `mapStatuses` internal cache.

You should see the following INFO message in the logs:

```
INFO MapOutputTrackerWorker: Updating epoch to [newEpoch] and clearing cache
```

Note

`updateEpoch` is exclusively used when `TaskRunner` runs (for a task).

Unregistering Shuffle — `unregisterShuffle` Method

```
unregisterShuffle(shuffleId: Int): Unit
```

`unregisterShuffle` unregisters `shuffleId`, i.e. removes `shuffleId` entry from the `mapStatuses` internal cache.

Note

`unregisterShuffle` is used when `ContextCleaner` removes a shuffle (blocks) from `MapOutputTrackerMaster` and `BlockManagerMaster` (aka shuffle cleanup) and when `BlockManagerSlaveEndpoint` handles `RemoveShuffle` message.

stop Method

```
stop(): Unit
```

`stop` does nothing at all.

Note	<code>stop</code> is used exclusively when <code>SparkEnv</code> stops (and stops all the services, <code>MapOutputTracker</code> including).
------	---

Note	<code>stop</code> is overridden by <code>MapOutputTrackerMaster</code> .
------	--

Finding Map Outputs For `ShuffleDependency` in Cache or Fetching Remotely — `getstatuses` Internal Method

```
getstatuses(shuffleId: Int): Array[MapStatus]
```

`getstatuses` finds `MapStatuses` for the input `shuffleId` in the `mapStatuses` internal cache and, when not available, fetches them from a remote `MapOutputTrackerMaster` (using RPC).

Internally, `getstatuses` first queries the `mapStatuses` internal cache and returns the map outputs if found.

If not found (in the `mapStatuses` internal cache), you should see the following INFO message in the logs:

```
INFO Don't have map outputs for shuffle [id], fetching them
```

If some other process fetches the map outputs for the `shuffleId` (as recorded in `fetching` internal registry), `getstatuses` waits until it is done.

When no other process fetches the map outputs, `getstatuses` registers the input `shuffleId` in `fetching` internal registry (of shuffle map outputs being fetched).

You should see the following INFO message in the logs:

```
INFO Doing the fetch; tracker endpoint = [trackerEndpoint]
```

`getstatuses` sends a `GetMapOutputStatuses` RPC remote message for the input `shuffleId` to the `trackerEndpoint` expecting a `Array[Byte]`.

Note	<code>getstatuses</code> requests shuffle map outputs remotely within a timeout and with retries. Refer to RpcEndpointRef .
------	---

`getstatuses` [deserializes the map output statuses](#) and records the result in the `mapStatuses` [internal cache](#).

You should see the following INFO message in the logs:

```
INFO Got the output locations
```

`getstatuses` removes the input `shuffleId` from `fetching` internal registry.

You should see the following DEBUG message in the logs:

```
DEBUG Fetching map output statuses for shuffle [id] took [time] ms
```

If `getstatuses` could not find the map output locations for the input `shuffleId` (locally and remotely), you should see the following ERROR message in the logs and throws a `MetadataFetchFailedException`.

```
ERROR Missing all output locations for shuffle [id]
```

Note

`getstatuses` is used when `MapOutputTracker` [getMapSizesByExecutorId](#) and [computes statistics for](#) `shuffleDependency`.

Converting MapStatuses To BlockManagerIds with ShuffleBlockIds and Their Sizes

— `convertMapStatuses` Internal Method

```
convertMapStatuses(  
    shuffleId: Int,  
    startPartition: Int,  
    endPartition: Int,  
    statuses: Array[MapStatus]): Seq[(BlockManagerId, Seq[(BlockId, Long)])]
```

`convertMapStatuses` iterates over the input `statuses` array (of `MapStatus` entries indexed by map id) and creates a collection of `BlockManagerId` (for each `MapStatus` entry) with a `ShuffleBlockId` (with the input `shuffleId`, a `mapId`, and `partition` ranging from the input `startPartition` and `endPartition`) and [estimated size for the reduce block](#) for every status and partitions.

For any empty `MapStatus`, you should see the following ERROR message in the logs:

```
ERROR Missing an output location for shuffle [id]
```

And `convertMapStatuses` throws a `MetadataFetchFailedException` (with `shuffleId`, `startPartition`, and the above error message).

Note

`convertMapStatuses` is exclusively used when `MapOutputTracker` computes `BlockManagerId`s with their `ShuffleBlockId`s and sizes.

Sending Blocking Messages To trackerEndpoint RpcEndpointRef — askTracker Method

```
askTracker[T](message: Any): T
```

`askTracker` sends the `message` to `trackerEndpoint` `RpcEndpointRef` and waits for a result.

When an exception happens, you should see the following ERROR message in the logs and `askTracker` throws a `SparkException`.

```
ERROR Error communicating with MapOutputTracker
```

Note

`askTracker` is used when `MapOutputTracker` fetches map outputs for `ShuffleDependency` remotely and sends a one-way message.

MapOutputTrackerMaster — MapOutputTracker For Driver

`MapOutputTrackerMaster` is the [MapOutputTracker](#) for the driver.

A `MapOutputTrackerMaster` is the source of truth for [MapStatus](#) objects (map output locations) per shuffle id (as recorded from [ShuffleMapTasks](#)).

Note

`MapOutputTrackerMaster` uses Java's thread-safe [java.util.concurrent.ConcurrentHashMap](#) for `mapStatuses` internal cache.

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 — `1000` for map and reduce each.

`MapOutputTrackerMaster` uses `MetadataCleaner` with `MetadataCleanerType.MAP_OUTPUT_TRACKER` as `cleanerType` and `cleanup` function to drop entries in `mapStatuses`.

Table 1. MapOutputTrackerMaster Internal Registries and Counters

Name	Description
cachedSerializedBroadcast	Internal registry of... FIXME Used when... FIXME
cachedSerializedStatuses	Internal registry of serialized shuffle map output statuses (as <code>Array[Byte]</code>) per... FIXME Used when... FIXME
cacheEpoch	Internal registry with... FIXME Used when... FIXME
shuffleIdLocks	Internal registry of locks for shuffle ids. Used when... FIXME
mapOutputRequests	Internal queue with <code>GetMapOutputMessage</code> requests for map output statuses. Used when <code>MapOutputTrackerMaster</code> posts <code>GetMapOutputMessage</code> messages to and take one head element off this queue . NOTE: <code>mapOutputRequests</code> uses Java's java.util.concurrent.LinkedBlockingQueue .

Tip	Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.MapOutputTrackerMaster</code> logger to see what happens in <code>MapOutputTrackerMaster</code> . Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.MapOutputTrackerMaster=DEBUG</code>
	Refer to Logging .

removeBroadcast Method

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

clearCachedBroadcast Method

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

post Method

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

stop Method

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

unregisterMapOutput Method

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

cleanup Function for MetadataCleaner

`cleanup(cleanupTime: Long)` method removes old entries in `mapStatuses` and `cachedSerializedStatuses` that have timestamp earlier than `cleanupTime`.

It uses `org.apache.spark.util.TimeStampedHashMap.clearOldValues` method.

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]

Creating MapOutputTrackerMaster Instance

`MapOutputTrackerMaster` takes the following when created:

1. [SparkConf](#)
2. `broadcastManager` — [BroadcastManager](#)
3. `isLocal` — flag to control whether `MapOutputTrackerMaster` runs in local or on a cluster.

`MapOutputTrackerMaster` initializes the [internal registries and counters](#) and [starts map-output-dispatcher threads](#).

Note	<code>MapOutputTrackerMaster</code> is created when <code>SparkEnv</code> is created.
------	---

threadpool Thread Pool with map-output-dispatcher Threads

`threadpool: ThreadPoolExecutor`

`threadpool` is a daemon fixed thread pool registered with **map-output-dispatcher** thread name prefix.

`threadpool` uses `spark.shuffle.mapOutput.dispatcher.numThreads` (default: 8) for the number of `MessageLoop` dispatcher threads to process received `GetMapOutputMessage` messages.

Note	The dispatcher threads are started immediately when <code>MapOutputTrackerMaster</code> is created.
------	---

Note	<code>threadpool</code> is shut down when <code>MapOutputTrackerMaster</code> stops.
------	--

Finding Preferred BlockManagers with Most Shuffle Map Outputs (For ShuffleDependency and Partition) — `getPreferredLocationsForShuffle` Method

```
getPreferredLocationsForShuffle(dep: ShuffleDependency[_, _, _], partitionId: Int): Seq[String]
```

`getPreferredLocationsForShuffle` finds the locations (i.e. `BlockManagers`) with the most map outputs for the input `ShuffleDependency` and `Partition`.

Note	<code>getPreferredLocationsForShuffle</code> is simply <code>getLocationsWithLargestOutputs</code> with a guard condition.
------	--

Internally, `getPreferredLocationsForShuffle` checks whether `spark.shuffle.reduceLocality.enabled` Spark property is enabled (it is by default) with the number of partitions of the RDD of the input `ShuffleDependency` and partitions in the `partitioner of the input ShuffleDependency` both being less than 1000.

Note	The thresholds for the number of partitions in the RDD and of the partitioner when computing the preferred locations are 1000 and are not configurable.
------	---

If the condition holds, `getPreferredLocationsForShuffle` finds locations with the largest number of shuffle map outputs for the input `shuffleDependency` and `partitionId` (with the number of partitions in the partitioner of the input `shuffleDependency` and `0.2`) and returns the hosts of the preferred `BlockManagers`.

Note	<code>0.2</code> is the fraction of total map output that must be at a location to be considered as a preferred location for a reduce task. It is not configurable.
------	---

Note	<code>getPreferredLocationsForShuffle</code> is used when <code>ShuffledRDD</code> and <code>ShuffledRowRDD</code> ask for preferred locations for a partition.
------	---

Incrementing Epoch — `incrementEpoch` Method

```
incrementEpoch(): Unit
```

`incrementEpoch` increments the internal `epoch`.

You should see the following DEBUG message in the logs:

```
DEBUG MapOutputTrackerMaster: Increasing epoch to [epoch]
```

Note	<code>incrementEpoch</code> is used when <code>MapOutputTrackerMaster</code> registers map outputs (with <code>changeEpoch</code> flag enabled — it is disabled by default) and unregisters map outputs (for a shuffle, mapper and block manager), and when <code>DAGScheduler</code> is notified that an executor got lost (with <code>filesLost</code> flag enabled).
------	---

Finding Locations with Largest Number of Shuffle Map Outputs — `getLocationsWithLargestOutputs` Method

```
getLocationsWithLargestOutputs(  
    shuffleId: Int,  
    reducerId: Int,  
    numReducers: Int,  
    fractionThreshold: Double): Option[Array[BlockManagerId]]
```

`getLocationsWithLargestOutputs` returns `BlockManagerId`s with the largest size (of all the shuffle blocks they manage) above the input `fractionThreshold` (given the total size of all the shuffle blocks for the shuffle across all `BlockManagers`).

Note	<code>getLocationsWithLargestOutputs</code> may return no <code>BlockManagerId</code> if their shuffle blocks do not total up above the input <code>fractionThreshold</code> .
------	--

Note	The input <code>numReducers</code> is not used.
------	---

Internally, `getLocationsWithLargestOutputs` queries the `mapStatuses` internal cache for the input `shuffleId`.

Note	One entry in <code>mapStatuses</code> internal cache is a <code>MapStatus</code> array indexed by partition id.
------	---

`MapStatus` includes information about the `BlockManager` (as `BlockManagerId`) and estimated size of the reduce blocks.

`getLocationsWithLargestOutputs` iterates over the `MapStatus` array and builds an interim mapping between `BlockManagerId` and the cumulative sum of shuffle blocks across `BlockManagers`.

Note	<code>getLocationsWithLargestOutputs</code> is used exclusively when <code>MapOutputTrackerMaster</code> finds the preferred locations (BlockManagers and hence executors) for a shuffle.
------	---

Requesting Tracking Status of Shuffle Map Output — `containsShuffle` Method

```
containsShuffle(shuffleId: Int): Boolean
```

`containsShuffle` checks if the input `shuffleId` is registered in the `cachedSerializedStatuses` or `mapStatuses` internal caches.

Note	<code>containsShuffle</code> is used exclusively when <code>DAGScheduler</code> creates a <code>shuffleMapStage</code> (for <code>ShuffleDependency</code> and <code>ActiveJob</code>).
------	--

Registering ShuffleDependency — `registerShuffle` Method

```
registerShuffle(shuffleId: Int, numMaps: Int): Unit
```

`registerShuffle` registers the input `shuffleId` in the `mapStatuses` internal cache.

Note	The number of <code>MapStatus</code> entries in the new array in <code>mapStatuses</code> internal cache is exactly the input <code>numMaps</code> .
------	--

`registerShuffle` adds a lock in the `shuffleIdLocks` internal registry (without using it).

If the `shuffleId` has already been registered, `registerShuffle` throws a `IllegalArgumentException` with the following message:

```
Shuffle ID [id] registered twice
```

Note

`registerShuffle` is used exclusively when `DAGScheduler` creates a `shuffleMapStage` (for `ShuffleDependency` and `ActiveJob`).

Registering Map Outputs for Shuffle (Possibly with Epoch Change) — `registerMapOutputs` Method

```
registerMapOutputs(  
    shuffleId: Int,  
    statuses: Array[MapStatus],  
    changeEpoch: Boolean = false): Unit
```

`registerMapOutputs` registers the input `statuses` (as the shuffle map output) with the input `shuffleId` in the `mapStatuses` internal cache.

`registerMapOutputs` increments epoch if the input `changeEpoch` is enabled (it is not by default).

Note

`registerMapOutputs` is used when `DAGScheduler` handles successful `shuffleMapTask` completion and `executor lost` events.

In both cases, the input `changeEpoch` is enabled.

Finding Serialized Map Output Statuses (And Possibly Broadcasting Them)

— `getSerializedMapOutputStatuses` Method

```
getSerializedMapOutputStatuses(shuffleId: Int): Array[Byte]
```

`getSerializedMapOutputStatuses` finds cached serialized map statuses for the input `shuffleId`.

If found, `getSerializedMapOutputStatuses` returns the cached serialized map statuses.

Otherwise, `getSerializedMapOutputStatuses` acquires the `shuffle lock` for `shuffleId` and finds cached serialized map statuses again since some other thread could not update the `cachedSerializedStatuses` internal cache.

`getSerializedMapOutputStatuses` returns the serialized map statuses if found.

If not, `getSerializedMapOutputStatuses` serializes the local array of `MapStatuses` (from `checkCachedStatuses`).

You should see the following INFO message in the logs:

```
INFO Size of output statuses for shuffle [shuffleId] is [bytes] bytes
```

`getSerializedMapOutputStatuses` saves the serialized map output statuses in `cachedSerializedStatuses` internal cache if the `epoch` has not changed in the meantime. `getSerializedMapOutputStatuses` also saves its broadcast version in `cachedSerializedBroadcast` internal cache.

If the `epoch` has changed in the meantime, the serialized map output statuses and their broadcast version are not saved, and you should see the following INFO message in the logs:

```
INFO Epoch changed, not caching!
```

`getSerializedMapOutputStatuses` removes the broadcast.

`getSerializedMapOutputStatuses` returns the serialized map statuses.

Note

`getSerializedMapOutputStatuses` is used when `MapOutputTrackerMaster` responds to `GetMapOutputMessage` requests and `DAGScheduler` creates `ShuffleMapStage` for `ShuffleDependency` (copying the shuffle map output locations from previous jobs to avoid unnecessarily regenerating data).

Finding Cached Serialized Map Statuses

— `checkCachedStatuses` Internal Method

```
checkCachedStatuses(): Boolean
```

`checkCachedStatuses` is an internal helper method that `getSerializedMapOutputStatuses` uses to do some bookkeeping (when the `epoch` and `cacheEpoch` differ) and set local `statuses`, `retBytes` and `epochGotten` (that `getSerializedMapOutputStatuses` uses).

Internally, `checkCachedStatuses` acquires the `epochLock` lock and checks the status of `epoch` to `cached cacheEpoch`.

If `epoch` is younger (i.e. greater), `checkCachedStatuses` clears `cachedSerializedStatuses` internal cache, `cached broadcasts` and sets `cacheEpoch` to be `epoch`.

`checkCachedStatuses` gets the serialized map output statuses for the `shuffleId` (of the owning [getSerializedMapOutputStatuses](#)).

When the serialized map output status is found, `checkCachedStatuses` saves it in a local `retBytes` and returns `true`.

When not found, you should see the following DEBUG message in the logs:

```
DEBUG cached status not found for : [shuffleId]
```

`checkCachedStatuses` uses [mapStatuses](#) internal cache to get map output statuses for the `shuffleId` (of the owning [getSerializedMapOutputStatuses](#)) or falls back to an empty array and sets it to a local `statuses`. `checkCachedStatuses` sets the local `epochGotten` to the current [epoch](#) and returns `false`.

MessageLoop Dispatcher Thread

`MessageLoop` is a dispatcher thread that, once started, runs indefinitely until [PoisonPill](#) arrives.

`MessageLoop` takes `GetMapOutputMessage` messages off [mapOutputRequests](#) internal queue (waiting if necessary until a message becomes available).

Unless `PoisonPill` is processed, you should see the following DEBUG message in the logs:

```
DEBUG Handling request to send map output locations for shuffle [shuffleId] to [hostPort]
```

`MessageLoop` replies back with [serialized map output statuses](#) for the `shuffleId` (from the incoming `GetMapOutputMessage` message).

Note	<code>MessageLoop</code> is created and executed immediately when MapOutputTrackerMaster is created.
------	--

PoisonPill Message

`PoisonPill` is a `GetMapOutputMessage` (with `-99` as `shuffleId`) that indicates that `MessageLoop` should exit its message loop.

`PoisonPill` is posted when [MapOutputTrackerMaster](#) stops.

Settings

Table 2. Spark Properties

Spark Property	Default Value	Description
<code>spark.shuffle.mapOutput.dispatcher.numThreads</code>	8	FIXME
<code>spark.shuffle.mapOutput.minSizeForBroadcast</code>	512k	FIXME
<code>spark.shuffle.reduceLocality.enabled</code>	true	<p>Controls whether to compute locality preferences for reduce tasks.</p> <p>When enabled (i.e. <code>true</code>), <code>MapOutputTrackerMaster</code> 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.</p>

MapOutputTrackerMasterEndpoint

`MapOutputTrackerMasterEndpoint` is a [RpcEndpoint](#) for `MapOutputTrackerMaster`.

`MapOutputTrackerMasterEndpoint` handles the following messages:

- [GetMapOutputStatuses](#)
- [StopMapOutputTracker](#)

Tip Enable `INFO` or `DEBUG` logging levels for `org.apache.spark.MapOutputTrackerMasterEndpoint` logger to see what happens in `MapOutputTrackerMasterEndpoint`.

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

```
log4j.logger.org.apache.spark.MapOutputTrackerMasterEndpoint=DEBUG
```

Refer to [Logging](#).

Creating MapOutputTrackerMasterEndpoint Instance

`MapOutputTrackerMasterEndpoint` takes the following when created:

1. `rpcEnv` — [RpcEnv](#)
2. `tracker` — [MapOutputTrackerMaster](#)
3. `conf` — [SparkConf](#)

When created, you should see the following DEBUG message in the logs:

```
DEBUG init
```

Note `MapOutputTrackerMasterEndpoint` is created when `SparkEnv` is created for the driver and executors.

GetMapOutputStatuses Message

```
GetMapOutputStatuses(shuffleId: Int)
extends MapOutputTrackerMessage
```

When `GetMapOutputStatuses` arrives, `MapOutputTrackerMasterEndpoint` reads the host and the port of the sender.

You should see the following INFO message in the logs:

```
INFO Asked to send map output locations for shuffle [shuffleId] to [hostPort]
```

`MapOutputTrackerMasterEndpoint` posts a `GetMapOutputMessage` to `MapOutputTrackerMaster` (with `shuffleId` and the current `RpcCallContext`).

Note	<code>GetMapOutputStatuses</code> is posted when <code>MapOutputTracker</code> fetches shuffle map outputs remotely.
------	--

StopMapOutputTracker Message

```
StopMapOutputTracker  
extends MapOutputTrackerMessage
```

When `StopMapOutputTracker` arrives, you should see the following INFO message in the logs:

```
INFO MapOutputTrackerMasterEndpoint stopped!
```

`MapOutputTrackerMasterEndpoint` confirms the request (by replying `true`) and stops itself (and stops accepting messages).

Note	<code>StopMapOutputTracker</code> is posted when <code>MapOutputTrackerMaster</code> stops.
------	---

MapOutputTrackerWorker — MapOutputTracker for Executors

A **MapOutputTrackerWorker** is the `MapOutputTracker` for executors.

`MapOutputTrackerWorker` uses Java's thread-safe `java.util.concurrent.ConcurrentHashMap` for `mapStatuses` [internal cache](#) and any lookup cache 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 <code>mapStatuses</code> internal registry is an instance of the thread-safe <code>java.util.concurrent.ConcurrentHashMap</code> .
------	--

Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.MapOutputTrackerWorker</code> logger to see what happens in <code>MapOutputTrackerWorker</code>.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.MapOutputTrackerWorker=DEBUG</pre> <p>Refer to Logging.</p>
-----	--

ShuffleManager — Pluggable Shuffle Systems

`ShuffleManager` is the pluggable mechanism for **shuffle systems** that track `shuffle dependencies` for `shuffleMapStage` on the driver and executors.

Note	<code>SortShuffleManager</code> (short name: <code>sort</code> or <code>tungsten-sort</code>) is the one and only <code>ShuffleManager</code> in Spark 2.0.
------	--

`spark.shuffle.manager` Spark property sets up the default shuffle manager.

The driver and executor access their `ShuffleManager` instances using `SparkEnv`.

```
val shuffleManager = SparkEnv.get.shuffleManager
```

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.

ShuffleManager Contract

```
trait ShuffleManager {
  def registerShuffle[K, V, C](
    shuffleId: Int,
    numMaps: Int,
    dependency: ShuffleDependency[K, V, C]): ShuffleHandle
  def getWriter[K, V](
    handle: ShuffleHandle,
    mapId: Int,
    context: TaskContext): ShuffleWriter[K, V]
  def getReader[K, C](
    handle: ShuffleHandle,
    startPartition: Int,
    endPartition: Int,
    context: TaskContext): ShuffleReader[K, C]
  def unregisterShuffle(shuffleId: Int): Boolean
  def shuffleBlockResolver: ShuffleBlockResolver
  def stop(): Unit
}
```

Note	<code>ShuffleManager</code> is a <code>private[spark]</code> contract.
------	--

Table 1. ShuffleManager Contract

Method	Description
<code>registerShuffle</code>	Executed when <code>ShuffleDependency</code> is created and registers itself.
<code>getWriter</code>	Used when a <code>ShuffleMapTask</code> runs (and requests a <code>ShuffleWriter</code> to write records for a partition).
<code>getReader</code>	Returns a <code>ShuffleReader</code> for a range of partitions (to read key-value records for a <code>ShuffleDependency</code> dependency). Used when <code>CoGroupedRDD</code> , <code>ShuffledRDD</code> , <code>SubtractedRDD</code> , and <code>ShuffledRowRDD</code> compute their partitions.
<code>unregisterShuffle</code>	Executed when ??? removes the metadata of a shuffle.
<code>shuffleBlockResolver</code>	Used when: <ol style="list-style-type: none"> 1. <code>BlockManager</code> requests a <code>ShuffleBlockResolver</code> capable of retrieving shuffle block data (for a <code>ShuffleBlockId</code>) 2. <code>BlockManager</code> requests a <code>ShuffleBlockResolver</code> for local shuffle block data as bytes.
<code>stop</code>	Used when <code>SparkEnv</code> stops.

Tip

Review [ShuffleManager sources](#).

Settings

Table 2. Spark Properties

Spark Property	Default Value	Description
<code>spark.shuffle.manager</code>	<code>sort</code>	<code>ShuffleManager</code> for a Spark application. You can use a short name or the fully-qualified class name of a custom implementation. The predefined aliases are <code>sort</code> and <code>tungsten-sort</code> with <code>org.apache.spark.shuffle.sort.SortShuffleManager</code> being the one and only <code>ShuffleManager</code> .

Further Reading or Watching

1. (slides) Spark shuffle introduction by [Raymond Liu](#) (aka *colorant*).

SortShuffleManager — The Default (And Only) Sort-Based Shuffle System

`SortShuffleManager` is the one and only [ShuffleManager](#) in Spark with the short name `sort` or `tungsten-sort`.

Note

You can use `spark.shuffle.manager` Spark property to activate your own implementation of [ShuffleManager contract](#).

Caution

[FIXME](#) The internal registries

Table 1. SortShuffleManager's Internal Registries and Counters

Name	Description
<code>numMapsForShuffle</code>	
<code>shuffleBlockResolver</code>	<p>IndexShuffleBlockResolver created when SortShuffleManager is created and used throughout the lifetime of the owning <code>SortShuffleManager</code>.</p> <p>NOTE: <code>shuffleBlockResolver</code> is a part of ShuffleManager contract.</p> <p>Beside the uses due to the contract, <code>shuffleBlockResolver</code> is used in <code>unregisterShuffle</code> and stopped in <code>stop</code>.</p>

Tip

Enable `DEBUG` logging level for `org.apache.spark.shuffle.sort.SortShuffleManager$` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.shuffle.sort.SortShuffleManager$=DEBUG
```

Refer to [Logging](#).

unregisterShuffle Method

Caution

[FIXME](#)

Creating SortShuffleManager Instance

`SortShuffleManager` takes a [SparkConf](#).

`SortShuffleManager` makes sure that `spark.shuffle.spill` Spark property is enabled. If not you should see the following WARN message 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.
```

`SortShuffleManager` initializes the [internal registries and counters](#).

Note

`SortShuffleManager` is created when `SparkEnv` is created (on the driver and executors) which is at the very beginning of a Spark application's lifecycle.

Creating ShuffleHandle (For ShuffleDependency) — `registerShuffle` Method

```
registerShuffle[K, V, C](  
    shuffleId: Int,  
    numMaps: Int,  
    dependency: ShuffleDependency[K, V, C]): ShuffleHandle
```

Note

`registerShuffle` is a part of [ShuffleManager contract](#).

Caution

FIXME Copy the conditions

`registerShuffle` returns a new `ShuffleHandle` that can be one of the following:

1. [BypassMergeSortShuffleHandle](#) (with `shuffleDependency[K, V, V]`) when `shouldBypassMergeSort` condition holds.
2. [SerializedShuffleHandle](#) (with `shuffleDependency[K, V, V]`) when `canUseSerializedShuffle` condition holds.
3. [BaseShuffleHandle](#)

Selecting ShuffleWriter For ShuffleHandle — `getWriter` Method

```
getWriter[K, V](  
    handle: ShuffleHandle,  
    mapId: Int,  
    context: TaskContext): ShuffleWriter[K, V]
```

Note	<code>getWriter</code> is a part of <code>ShuffleManager</code> contract.
------	---

Internally, `getWriter` makes sure that a `ShuffleHandle` is associated with its `numMaps` in `numMapsForShuffle` internal registry.

Caution	<code>FIXME Associated?!</code> What's that?
---------	--

Note	<code>getWriter</code> expects that the input <code>handle</code> is of type <code>BaseShuffleHandle</code> (despite the signature that says that it can work with any <code>ShuffleHandle</code>). Moreover, <code>getWriter</code> further expects that in 2 (out of 3 cases) the input <code>handle</code> is a more specialized <code>IndexShuffleBlockResolver</code> .
------	---

`getWriter` then returns a new `ShuffleWriter` for the input `ShuffleHandle`:

1. `UnsafeShuffleWriter` for `SerializedShuffleHandle`.
2. `BypassMergeSortShuffleWriter` for `BypassMergeSortShuffleHandle`.
3. `SortShuffleWriter` for `BaseShuffleHandle`.

Creating BlockStoreShuffleReader For ShuffleHandle — `getReader` Method

```
getReader[K, C](
    handle: ShuffleHandle,
    startPartition: Int,
    endPartition: Int,
    context: TaskContext): ShuffleReader[K, C]
```

Note	<code>getReader</code> is a part of <code>ShuffleManager</code> contract.
------	---

`getReader` returns a new `BlockStoreShuffleReader` passing all the input parameters on to it.

Note	<code>getReader</code> assumes that the input <code>ShuffleHandle</code> is of type <code>BaseShuffleHandle</code> .
------	--

Stopping SortShuffleManager — `stop` Method

```
stop(): Unit
```

Note	<code>stop</code> is a part of <code>ShuffleManager</code> contract.
------	--

`stop` stops `IndexShuffleBlockResolver` (available as `shuffleBlockResolver` internal reference).

Considering BypassMergeSortShuffleHandle for ShuffleHandle — `shouldBypassMergeSort` Method

```
shouldBypassMergeSort(conf: SparkConf, dep: ShuffleDependency[_, _, _]): Boolean
```

`shouldBypassMergeSort` holds (i.e. is positive) when:

1. The input `ShuffleDependency` has `mapSideCombine` flag enabled and `aggregator` defined.
2. `mapSideCombine` flag is disabled (i.e. `false`) but the number of partitions (of the `Partitioner` of the input `ShuffleDependency`) is at most `spark.shuffle.sort.bypassMergeThreshold` Spark property (which defaults to `200`).

Otherwise, `shouldBypassMergeSort` does not hold (i.e. `false`).

Note

`shouldBypassMergeSort` is exclusively used when `sortShuffleManager` selects a `shuffleHandle` (for a `ShuffleDependency`).

Considering SerializedShuffleHandle for ShuffleHandle — `canUseSerializedShuffle` Method

```
canUseSerializedShuffle(dependency: ShuffleDependency[_, _, _]): Boolean
```

`canUseSerializedShuffle` condition holds (i.e. is positive) when all of the following hold (checked in that order):

1. The `Serializer` of the input `ShuffleDependency` supports relocation of serialized objects.
2. The `Aggregator` of the input `ShuffleDependency` is not defined.
3. The number of shuffle output partitions of the input `ShuffleDependency` is at most the supported maximum number (which is `(1 << 24) - 1`, i.e. `16777215`).

You should see the following DEBUG message in the logs when `canUseSerializedShuffle` holds:

```
DEBUG Can use serialized shuffle for shuffle [id]
```

Otherwise, `canUseSerializedShuffle` does not hold and you should see one of the following DEBUG messages:

```
DEBUG Can't use serialized shuffle for shuffle [id] because the serializer, [name], does not support object relocation
```

```
DEBUG SortShuffleManager: Can't use serialized shuffle for shuffle [id] because an aggregator is defined
```

```
DEBUG Can't use serialized shuffle for shuffle [id] because it has more than [number] partitions
```

Note

`canUseSerializedShuffle` is exclusively used when `SortShuffleManager` selects a `ShuffleHandle` (for a `ShuffleDependency`).

Settings

Table 2. Spark Properties

Spark Property	Default Value	Description
<code>spark.shuffle.sort.bypassMergeThreshold</code>	200	The maximum number of reduce partitions below which <code>SortShuffleManager</code> avoids merge-sorting data if there is no map-side aggregation either.
<code>spark.shuffle.spill</code>	true	No longer in use. When <code>false</code> the following WARN shows in the logs when <code>SortShuffleManager</code> is created: WARN SortShuffleManager: <code>spark.shuffle.spill</code> was set to <code>false</code> , but this configuration is ignored as of Spark 1.6+. Shuffle will continue to spill to disk when necessary.

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	FIXME <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 `TransportClient`s).

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

[FIXME](#) `checkAuth` ?

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

[FIXME](#) `callback.onSuccess` ?

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])"
```

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.shuffle.service.enabled</code>	<code>false</code>	<p>Enables External Shuffle Service. When <code>true</code>, the driver registers itself with the shuffle service.</p> <p>Used to enable for dynamic allocation of executors and in CoarseMesosSchedulerBackend to instantiate MesosExternalShuffleClient.</p> <p>Explicitly disabled for LocalSparkCluster (and <i>any</i> attempts to set it are ignored).</p>
<code>spark.shuffle.service.port</code>	7337	

OneForOneStreamManager

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

registerStream Method

```
long registerStream(String appId, Iterator<ManagedBuffer> buffers)
```

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

ShuffleBlockResolver

`ShuffleBlockResolver` is used to [find shuffle block data](#).

Note	The one and only implementation of ShuffleBlockResolver contract in Spark is IndexShuffleBlockResolver .
------	--

Note	<code>ShuffleBlockResolver</code> is used exclusively in BlockManager to find shuffle block data.
------	---

ShuffleBlockResolver Contract

```
trait ShuffleBlockResolver {
  def getBlockData(blockId: ShuffleBlockId): ManagedBuffer
  def stop(): Unit
}
```

Note	<code>ShuffleBlockResolver</code> is a <code>private[spark]</code> contract.
------	--

Table 1. ShuffleBlockResolver Contract

Method	Description
<code>getBlockData</code>	Used when <code>BlockManager</code> is requested to find shuffle block data and later (duplicate?) for local shuffle block data as serialized bytes .
<code>stop</code>	Used when <code>SortShuffleManager</code> stops .

IndexShuffleBlockResolver

`IndexShuffleBlockResolver` is the one and only `ShuffleBlockResolver` in Spark.

`IndexShuffleBlockResolver` manages shuffle block data and uses **shuffle index files** for faster shuffle data access. `IndexShuffleBlockResolver` can [write a shuffle block index and data file](#), [find](#) and [remove](#) shuffle index and data files per shuffle and map.

Note	Shuffle block data files are more often referred as map outputs files .
------	--

`IndexShuffleBlockResolver` is managed exclusively by `SortShuffleManager` (so `BlockManager` can access shuffle block data).

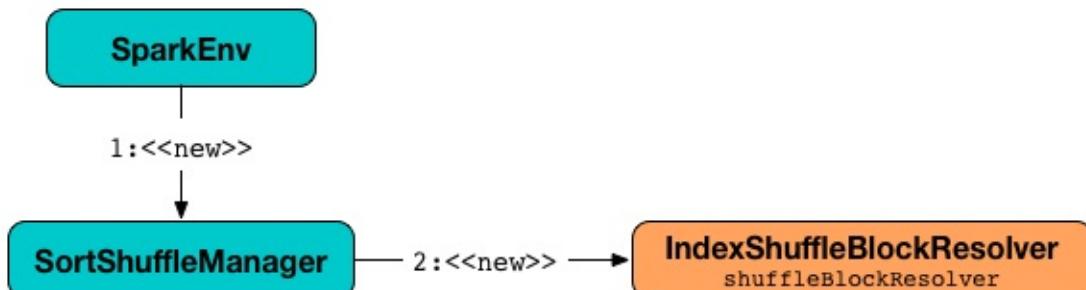


Figure 1. SortShuffleManager creates IndexShuffleBlockResolver

`IndexShuffleBlockResolver` is later passed in when `sortShuffleManager` creates a `ShuffleWriter` for `shuffleHandle`.

Table 1. IndexShuffleBlockResolver's Internal Properties

Name	Initial Value	Description
<code>transportConf</code>	<code>TransportConf</code> for <code>shuffle</code> module	Used when <code>IndexShuffleBlockResolver</code> creates a <code>ManagedBuffer</code> for a <code>ShuffleBlockId</code> .

Creating IndexShuffleBlockResolver Instance

`IndexShuffleBlockResolver` takes the following when created:

1. `SparkConf`,
2. `BlockManager` (default: unspecified and `SparkEnv` is used to access one)

`IndexShuffleBlockResolver` initializes the internal properties.

Note	<code>IndexShuffleBlockResolver</code> is created exclusively when <code>SortShuffleManager</code> is created.
------	--

Writing Shuffle Index and Data Files

— `writeIndexFileAndCommit` Method

```
writeIndexFileAndCommit(
    shuffleId: Int,
    mapId: Int,
    lengths: Array[Long],
    dataTmp: File): Unit
```

Internally, `writeIndexFileAndCommit` first finds the index file for the input `shuffleId` and `mapId`.

`writeIndexFileAndCommit` creates a temporary file for the index file (in the same directory) and writes offsets (as the moving sum of the input `lengths` starting from 0 to the final offset at the end for the end of the output file).

Note

The offsets are the sizes in the input `lengths` exactly.

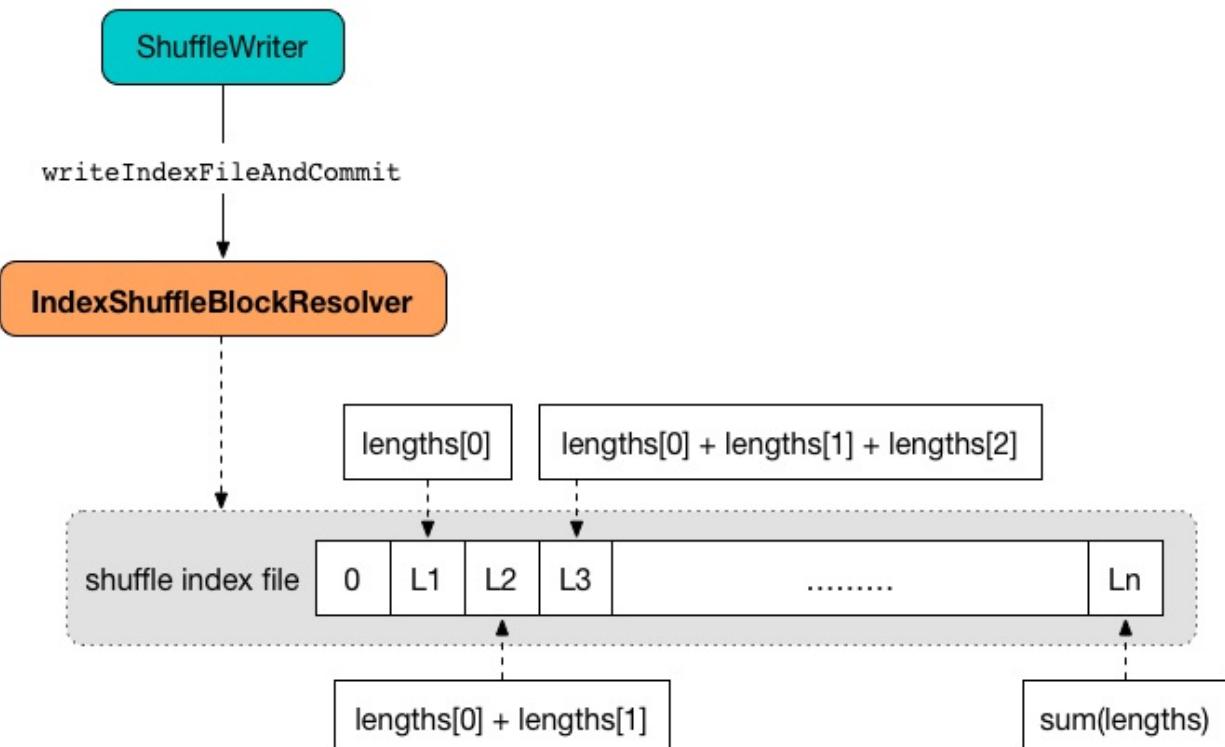


Figure 2. `writeIndexFileAndCommit` and offsets in a shuffle index file

`writeIndexFileAndCommit` requests a shuffle block data file for the input `shuffleId` and `mapId`.

`writeIndexFileAndCommit` checks if the given index and data files match each other (aka *consistency check*).

If the consistency check fails, it means that another attempt for the same task has already written the map outputs successfully and so the input `dataTmp` and temporary index files are deleted (as no longer correct).

If the consistency check succeeds, the existing index and data files are deleted (if they exist) and the temporary index and data files become "official", i.e. renamed to their final names.

In case of any IO-related exception, `writeIndexFileAndCommit` throws a `IOException` with the messages:

```
fail to rename file [indexTmp] to [indexFile]
```

or

```
fail to rename file [dataTmp] to [dataFile]
```

Note	<code>writeIndexFileAndCommit</code> is used when ShuffleWriter is requested to write records to shuffle system, i.e. SortShuffleWriter , BypassMergeSortShuffleWriter , and UnsafeShuffleWriter .
------	--

Creating ManagedBuffer to Read Shuffle Block Data File

— `getBlockData` Method

```
getBlockData(blockId: ShuffleBlockId): ManagedBuffer
```

Note	<code>getBlockData</code> is a part of ShuffleBlockResolver contract .
------	--

Internally, `getBlockData` finds the index file for the input shuffle `blockId`.

Note	<code>ShuffleBlockId</code> knows <code>shuffleId</code> and <code>mapId</code> .
------	---

`getBlockData` discards `blockId.reduceId` bytes of data from the index file.

Note	<code>getBlockData</code> uses Guava's com.google.common.io.ByteStreams to skip the bytes.
------	--

`getBlockData` reads the start and end offsets from the index file and then creates a `FileSegmentManagedBuffer` to read the data file for the offsets (using `transportConf` internal property).

Note	The start and end offsets are the offset and the length of the file segment for the block data.
------	---

In the end, `getBlockData` closes the index file.

Checking Consistency of Shuffle Index and Data Files and Returning Block Lengths — `checkIndexAndDataFile` Internal Method

```
checkIndexAndDataFile(index: File, data: File, blocks: Int): Array[Long]
```

`checkIndexAndDataFile` first checks if the size of the input `index` file is exactly the input `blocks` multiplied by `8`.

`checkIndexAndDataFile` returns `null` when the numbers, and hence the shuffle index and data files, don't match.

`checkIndexAndDataFile` reads the shuffle `index` file and converts the offsets into lengths of each block.

`checkIndexAndDataFile` makes sure that the size of the input shuffle `data` file is exactly the sum of the block lengths.

`checkIndexAndDataFile` returns the block lengths if the numbers match, and `null` otherwise.

Note

`checkIndexAndDataFile` is used exclusively when `IndexShuffleBlockResolver` writes shuffle index and data files.

Requesting Shuffle Block Index File (from DiskBlockManager) — `getIndexFile` Internal Method

```
getIndexFile(shuffleId: Int, mapId: Int): File
```

`getIndexFile` requests `BlockManager` for the current `DiskBlockManager`.

Note

`getIndexFile` uses `SparkEnv` to access the current `BlockManager` unless specified when `IndexShuffleBlockResolver` is created.

`getIndexFile` then requests `DiskBlockManager` for the shuffle index file given the input `shuffleId` and `mapId` (as `ShuffleIndexBlockId`)

Note

`getIndexFile` is used when `IndexShuffleBlockResolver` writes shuffle index and data files, creates a `ManagedBuffer` to read a shuffle block data file, and ultimately removes the shuffle index and data files.

Requesting Shuffle Block Data File — `getDataFile` Method

```
getDataFile(shuffleId: Int, mapId: Int): File
```

`getDataFile` requests `BlockManager` for the current `DiskBlockManager`.

Note	<code>getDataFile</code> uses <code>SparkEnv</code> to access the current <code>BlockManager</code> unless specified when <code>IndexShuffleBlockResolver</code> is created.
------	--

`getDataFile` then requests `DiskBlockManager` for the shuffle block data file given the input `shuffleId`, `mapId`, and the special reduce id `0` (as `ShuffleDataBlockId`).

Note	<code>getDataFile</code> is used when: <ol style="list-style-type: none"> 1. <code>IndexShuffleBlockResolver</code> writes an index file, creates a <code>ManagedBuffer</code> for <code>shuffleBlockId</code>, and removes the data and index files that contain the output data from one map 2. <code>ShuffleWriter</code> is requested to write records to shuffle system, i.e. <code>SortShuffleWriter</code>, <code>BypassMergeSortShuffleWriter</code>, and <code>UnsafeShuffleWriter</code>.
------	---

Removing Shuffle Index and Data Files (For Single Map) — `removeDataByMap` Method

```
removeDataByMap(shuffleId: Int, mapId: Int): Unit
```

`removeDataByMap` finds and deletes the shuffle data for the input `shuffleId` and `mapId` first followed by finding and deleting the shuffle data index file.

When `removeDataByMap` fails deleting the files, you should see a WARN message in the logs.

```
WARN Error deleting data [path]
```

or

```
WARN Error deleting index [path]
```

Note	<code>removeDataByMap</code> is used exclusively when <code>SortShuffleManager</code> unregisters a <code>shuffle</code> , i.e. removes a shuffle from a shuffle system.
------	--

Stopping IndexShuffleBlockResolver — `stop` Method

```
stop(): Unit
```

Note

`stop` is a part of [ShuffleBlockResolver contract](#).

`stop` is a noop operation, i.e. does nothing when called.

ShuffleWriter

Caution

[FIXME](#)

ShuffleWriter Contract

```
abstract class ShuffleWriter[K, V] {
  def write(records: Iterator[Product2[K, V]]): Unit
  def stop(success: Boolean): Option[MapStatus]
}
```

Note

`ShuffleWriter` is a `private[spark]` contract.

Table 1. `ShuffleWriter` Contract

Method	Description
<code>write</code>	Writes a sequence of records (for a RDD partition) to a shuffle system when a <code>ShuffleMapTask</code> writes its execution result.
<code>stop</code>	Closes a <code>ShuffleWriter</code> and returns <code>MapStatus</code> if the writing completed successfully. Used when a <code>ShuffleMapTask</code> finishes execution with the input <code>success</code> flag to match the status of the task execution.

BypassMergeSortShuffleWriter

`BypassMergeSortShuffleWriter` is a `ShuffleWriter` that `ShuffleMapTask` uses to write records into one single shuffle block data file when the task runs for a `ShuffleDependency`.

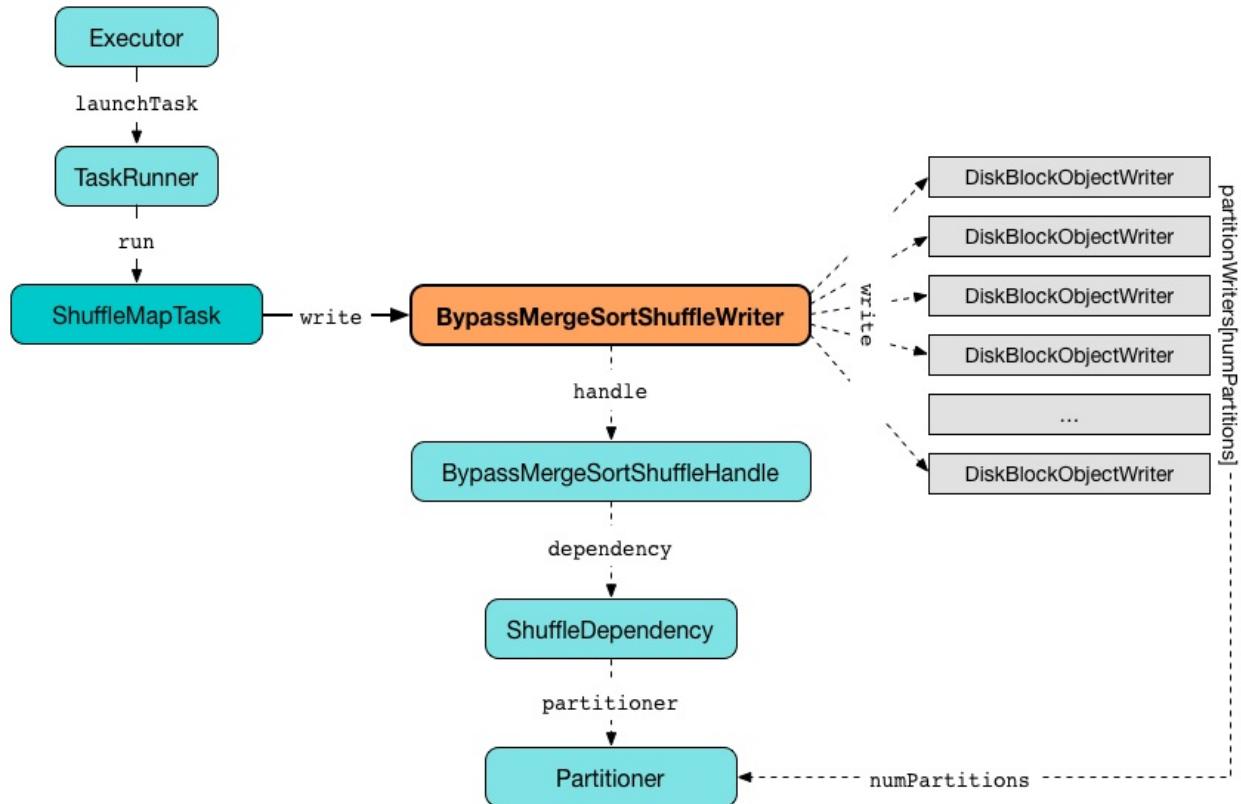


Figure 1. `BypassMergeSortShuffleWriter` writing records (for `ShuffleMapTask`) using `DiskBlockObjectWriters`

`BypassMergeSortShuffleWriter` is created exclusively when `SortShuffleManager` selects a `ShuffleWriter` (for a `BypassMergeSortShuffleHandle`).

Tip

Review the conditions `SortShuffleManager` uses to select `BypassMergeSortShuffleHandle` for a `ShuffleHandle`.

Table 1. BypassMergeSortShuffleWriter's Internal Registries and Counters

Name	Description
numPartitions	FIXME
partitionWriters	FIXME
partitionWriterSegments	FIXME
shuffleBlockResolver	<p><code>IndexShuffleBlockResolver</code>.</p> <p>Initialized when <code>BypassMergeSortShuffleWriter</code> is created.</p> <p>Used when <code>BypassMergeSortShuffleWriter</code> writes records.</p>
mapStatus	<p><code>MapStatus</code> that <code>BypassMergeSortShuffleWriter</code> returns when stopped</p> <p>Initialized every time <code>BypassMergeSortShuffleWriter</code> writes records.</p> <p>Used when <code>BypassMergeSortShuffleWriter</code> stops (with <code>success</code> enabled) as a marker if any records were written and returned if they did.</p>
partitionLengths	<p>Temporary array of partition lengths after records are written to a shuffle system.</p> <p>Initialized every time <code>BypassMergeSortShuffleWriter</code> writes records before passing it in to <code>IndexShuffleBlockResolver</code>). After <code>IndexShuffleBlockResolver</code> finishes, it is used to initialize <code>mapStatus</code> internal property.</p>
transferToEnabled	<p>Internal flag that controls the use of Java New I/O when <code>BypassMergeSortShuffleWriter</code> concatenates per-partition shuffle files into a single shuffle block data file.</p> <p>Specified when <code>BypassMergeSortShuffleWriter</code> is created and controlled by <code>spark.file.transferTo</code> Spark property. Enabled by default.</p>

Tip Enable `ERROR` logging level for `org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter` logger to see what happens in `BypassMergeSortShuffleWriter`.

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

```
log4j.logger.org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter=ERROR
```

Refer to [Logging](#).

Creating BypassMergeSortShuffleWriter Instance

`BypassMergeSortShuffleWriter` takes the following when created:

1. `BlockManager`
2. `IndexShuffleBlockResolver`
3. `BypassMergeSortShuffleHandle`
4. `mapId`
5. `TaskContext`
6. `SparkConf`

`BypassMergeSortShuffleWriter` uses `spark.shuffle.file.buffer` (for `fileBufferSize` as `32k` by default) and `spark.file.transferTo` (for `transferToEnabled` internal flag which is enabled by default) Spark properties.

`BypassMergeSortShuffleWriter` initializes the [internal registries and counters](#).

Writing Records (Into One Single Shuffle Block Data File)

— `write` Method

```
void write(Iterator<Product2<K, V>> records) throws IOException
```

Note `write` is a part of `ShuffleWriter` contract to write a sequence of records to a shuffle system.

Internally, when the input `records` iterator has no more records, `write` creates an empty `partitionLengths` internal array of `numPartitions` size.

`write` then requests the internal `IndexShuffleBlockResolver` to write shuffle index and data files (with `dataTmp` as `null`) and sets the internal `mapStatus` (with the address of `BlockManager` in use and `partitionLengths`).

However, when there are records to write, `write` creates a new `Serializer`.

Note	<code>Serializer</code> was specified when <code>BypassMergeSortShuffleWriter</code> was created and is exactly the <code>Serializer</code> of the <code>ShuffleDependency</code> .
------	---

`write` initializes `partitionWriters` internal array of `DiskBlockObjectWriters` for every partition.

For every partition, `write` requests `DiskBlockManager` for a temporary shuffle block and its file.

Note	<code>write</code> uses <code>BlockManager</code> to access <code>DiskBlockManager</code> . <code>BlockManager</code> was specified when <code>BypassMergeSortShuffleWriter</code> was created.
------	---

`write` requests `BlockManager` for a `DiskBlockObjectWriter` (for the temporary `blockId` and `file`, `SerializerInstance`, `fileBufferSize` and `writeMetrics`).

After `DiskBlockObjectWriters` were created, `write` increments shuffle write time.

`write` initializes `partitionWriterSegments` with `FileSegment` for every partition.

`write` takes records serially, i.e. record by record, and, after computing the partition for a key, requests the corresponding `DiskBlockObjectWriter` to write them.

Note	<code>write</code> uses <code>partitionWriters</code> internal array of <code>DiskBlockObjectWriter</code> indexed by partition number.
------	---

Note	<code>write</code> uses the <code>Partitioner</code> from the <code>ShuffleDependency</code> for which <code>BypassMergeSortShuffleWriter</code> was created.
------	---

Note	<code>write</code> initializes <code>partitionWriters</code> with <code>numPartitions</code> number of <code>DiskBlockObjectWriters</code> .
------	--

After all the `records` have been written, `write` requests every `DiskBlockObjectWriter` to `commitAndGet` and saves the commit results in `partitionWriterSegments`. `write` closes every `DiskBlockObjectWriter`.

`write` requests `IndexShuffleBlockResolver` for the shuffle block data file for `shuffleId` and `mapId`.

Note	<code>IndexShuffleBlockResolver</code> was defined when <code>BypassMergeSortShuffleWriter</code> was created.
------	--

`write` creates a temporary shuffle block data file and writes the per-partition shuffle files to it.

Note

This is the moment when `BypassMergeSortShuffleWriter` concatenates per-partition shuffle file segments into one single map shuffle data file.

In the end, `write` requests `IndexShuffleBlockResolver` to write shuffle index and data files for the `shuffleId` and `mapId` (with `partitionLengths` and the temporary file) and creates a new `mapStatus` (with the location of the `BlockManager` and `partitionLengths`).

Concatenating Per-Partition Files Into Single File (and Tracking Write Time) — `writePartitionedFile` Internal Method

```
long[] writePartitionedFile(File outputFile) throws IOException
```

`writePartitionedFile` creates a file output stream for the input `outputFile` in append mode.

Note

`writePartitionedFile` uses Java's `java.io.FileOutputStream` to create a file output stream.

`writePartitionedFile` starts tracking write time (as `writeStartTime`).

For every `numPartitions` partition, `writePartitionedFile` takes the file from the `FileSegment` (from `partitionWriterSegments`) and creates a file input stream to read raw bytes.

Note

`writePartitionedFile` uses Java's `java.io.FileInputStream` to create a file input stream.

`writePartitionedFile` then copies the raw bytes from each partition segment input stream to `outputFile` (possibly using Java New I/O per `transferToEnabled` flag set when `BypassMergeSortShuffleWriter` was created) and records the length of the shuffle data file (in `lengths` internal array).

Note

`transferToEnabled` is controlled by `spark.file.transferTo` Spark property and is enabled (i.e. `true`) by default.

In the end, `writePartitionedFile` increments shuffle write time, clears `partitionWriters` array and returns the lengths of the shuffle data files per partition.

Note

`writePartitionedFile` uses `ShuffleWriteMetrics` to track shuffle write time that was created when `BypassMergeSortShuffleWriter` was created.

Note

`writePartitionedFile` is used exclusively when `BypassMergeSortShuffleWriter` writes records.

Copying Raw Bytes Between Input Streams (Possibly Using Java New I/O) — `Utils.copyStream` Method

```
copyStream(
    in: InputStream,
    out: OutputStream,
    closeStreams: Boolean = false,
    transferToEnabled: Boolean = false): Long
```

`copyStream` branches off depending on the type of `in` and `out` streams, i.e. whether they are both `FileInputStream` with `transferToEnabled` input flag is enabled.

If they are both `FileInputStream` with `transferToEnabled` enabled, `copyStream` gets their `FileChannels` and transfers bytes from the input file to the output file and counts the number of bytes, possibly zero, that were actually transferred.

Note

`copyStream` uses Java's `java.nio.channels.FileChannel` to manage file channels.

If either `in` and `out` input streams are not `FileInputStream` or `transferToEnabled` flag is disabled (default), `copyStream` reads data from `in` to write to `out` and counts the number of bytes written.

`copyStream` can optionally close `in` and `out` streams (depending on the input `closeStreams` — disabled by default).

Note

`Utils.copyStream` is used when `BypassMergeSortShuffleWriter` writes records into one single shuffle block data file (among other places).

Note

`Utils.copyStream` is here temporarily (until I find a better place).

Tip

Visit the official web site of [JSR 51: New I/O APIs for the Java Platform](#) and read up on [java.nio package](#).

SortShuffleWriter — Fallback ShuffleWriter

`SortShuffleWriter` is a `ShuffleWriter` that is used when `SortShuffleManager` returns a `ShuffleWriter` for `shuffleHandle` (and the more specialized `BypassMergeSortShuffleWriter` and `UnsafeShuffleWriter` could not be used).

Note

`SortShuffleWriter` is parameterized by types for `k` keys, `v` values, and `c` combiner values.

Table 1. SortShuffleWriter's Internal Registries and Counters

Name	Description
<code>mapStatus</code>	MapStatus the <code>SortShuffleWriter</code> has recently persisted (as a shuffle partitioned file in disk store). NOTE: Since <code>write</code> does not return a value, <code>mapStatus</code> attribute is used to be returned when <code>SortShuffleWriter</code> is closed.
<code>stopping</code>	Internal flag to mark that <code>SortShuffleWriter</code> is closed.

Tip

Enable `ERROR` logging level for `org.apache.spark.shuffle.sort.SortShuffleWriter` logger to see what happens in `SortShuffleWriter`.

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

```
log4j.logger.org.apache.spark.shuffle.sort.SortShuffleWriter=ERROR
```

Refer to [Logging](#).

Creating SortShuffleWriter Instance

`SortShuffleWriter` takes the following when created:

1. `IndexShuffleBlockResolver`
2. `BaseShuffleHandle`
3. `mapId` — the mapper task id
4. `TaskContext`

Note

`SortShuffleWriter` is created when `SortShuffleManager` returns a `ShuffleWriter` for the fallback `BaseShuffleHandle`.

Writing Records Into Shuffle Partitioned File In Disk Store

— write Method

```
write(records: Iterator[Product2[K, V]]): Unit
```

Note

`write` is a part of [ShuffleWriter contract](#) to write a sequence of records (for a RDD partition).

Internally, `write` creates a [ExternalSorter](#) with the types `K, V, C` or `K, V, V` depending on `mapSideCombine` flag of the [ShuffleDependency](#) being enabled or not, respectively.

Note

`ShuffleDependency` is defined when [SortShuffleWriter](#) is created (as the dependency of [BaseShuffleHandle](#)).

Note

`write` makes sure that [Aggregator](#) is defined for [ShuffleDependency](#) when `mapSideCombine` flag is enabled.

`write` inserts all the records to [ExternalSorter](#)

`write` requests [IndexShuffleBlockResolver](#) for the shuffle data output file (for the [ShuffleDependency](#) and `mapId`) and creates a temporary file for the shuffle data file in the same directory.

`write` creates a [ShuffleBlockId](#) (for the [ShuffleDependency](#) and `mapId` and the special `IndexShuffleBlockResolver.NOOP_REDUCE_ID` reduce id).

`write` requests [ExternalSorter](#) to write all the records (previously inserted in) into the temporary partitioned file in the disk store.

`write` requests [IndexShuffleBlockResolver](#) to write an index file (for the temporary partitioned file).

`write` creates a [MapStatus](#) (with the location of the shuffle server that serves the executor's shuffle files and the sizes of the shuffle partitioned file's partitions).

Note

The newly-created [MapStatus](#) is available as `mapStatus` internal attribute.

Note

`write` does not handle exceptions so when they occur, they will break the processing.

In the end, `write` deletes the temporary partitioned file. You may see the following ERROR message in the logs if `write` did not manage to do so:

```
ERROR Error while deleting temp file [path]
```

Closing SortShuffleWriter (and Calculating MapStatus)

— stop Method

```
stop(success: Boolean): Option[MapStatus]
```

Note

`stop` is a part of [ShuffleWriter contract](#) to close itself (and return the last written [MapStatus](#)).

`stop` turns [stopping](#) flag on and returns the internal [mapStatus](#) if the input `success` is enabled.

Otherwise, when [stopping](#) flag is already enabled or the input `success` is disabled, `stop` returns no [MapStatus](#) (i.e. `None`).

In the end, `stop` [stops the](#) [ExternalSorter](#) and increments the shuffle write time task metrics.

UnsafeShuffleWriter — ShuffleWriter for SerializedShuffleHandle

`UnsafeShuffleWriter` is a [ShuffleWriter](#) that is used to [write records](#) (i.e. key-value pairs).

`UnsafeShuffleWriter` is chosen when `SortShuffleManager` is requested for a `shufflewriter` for a [SerializedShuffleHandle](#).

`UnsafeShuffleWriter` can use a specialized NIO-based merge procedure that avoids extra serialization/deserialization.

Table 1. UnsafeShuffleWriter's Internal Properties

Name	Initial Value	Description
<code>sorter</code>	(uninitialized)	<p><code>ShuffleExternalSorter</code></p> <p>Initialized when <code>UnsafeShuffleWriter</code> opens (which is when <code>UnsafeShuffleWriter</code> is created) and destroyed when it closes internal resources and writes spill files merged.</p> <p>Used when <code>UnsafeShuffleWriter</code> inserts a record into <code>ShuffleExternalSorter</code>, writes records, <code>forceSorterToSpill</code>, <code>updatePeakMemoryUsed</code>, closes internal resources and writes spill files merged, stops.</p>

Tip	<p>Enable <code>ERROR</code> or <code>DEBUG</code> logging levels for <code>org.apache.spark.shuffle.sort.UnsafeShuffleWriter</code> logger to see what happens in <code>UnsafeShuffleWriter</code>.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.shuffle.sort.UnsafeShuffleWriter=DEBUG</pre> <p>Refer to Logging.</p>
------------	---

mergeSpillsWithTransferTo Method

Caution

[FIXME](#)

forceSorterToSpill Method

Caution

FIXME

mergeSpills Method

Caution

FIXME

updatePeakMemoryUsed Method

Caution

FIXME

Writing Records — write Method

```
void write(Iterator<Product2<K, V>> records) throws IOException
```

Note

write is a part of `shuffleWriter` contract.

Internally, `write` traverses the input sequence of records (for a RDD partition) and `insertRecordIntoSorter` one by one. When all the records have been processed, `write` closes internal resources and writes spill files merged.

In the end, `write` requests `ShuffleExternalSorter` to clean after itself.

Caution

FIXME

Stopping UnsafeShuffleWriter — stop Method

```
Option<MapStatus> stop(boolean success)
```

Caution

FIXME

Note

stop is a part of `shuffleWriter` contract.

Creating UnsafeShuffleWriter Instance

`UnsafeShuffleWriter` takes the following when created:

1. `BlockManager`
2. `IndexShuffleBlockResolver`

3. TaskMemoryManager
4. SerializedShuffleHandle
5. mapId
6. TaskContext
7. SparkConf

`UnsafeShuffleWriter` makes sure that the number of shuffle output partitions (of the `ShuffleDependency` of the input `SerializedShuffleHandle`) is at most `(1 << 24) - 1`, i.e. `16777215`.

Note	The number of shuffle output partitions is first enforced when <code>sortShuffleManager</code> checks if <code>SerializedShuffleHandle</code> can be used for <code>shuffleHandle</code> (that eventually leads to <code>UnsafeShuffleWriter</code>).
------	--

`UnsafeShuffleWriter` uses `spark.file.transferTo` and `spark.shuffle.sort.initialBufferSize` Spark properties to initialize `transferToEnabled` and `initialSortBufferSize` attributes, respectively.

If the number of shuffle output partitions is greater than the maximum, `UnsafeShuffleWriter` throws a `IllegalArgumentException`.

Note	UnsafeShuffleWriter can only be used for shuffles with at most 16777215 reduce partitions
------	---

Note	<code>UnsafeShuffleWriter</code> is created exclusively when <code>SortShuffleManager</code> selects a <code>ShuffleWriter</code> (for a <code>SerializedShuffleHandle</code>).
------	--

Opening UnsafeShuffleWriter (i.e. Creating `ShuffleExternalSorter` and `SerializationStream`) — `open` Internal Method

```
void open() throws IOException
```

`open` makes sure that the internal reference to `ShuffleExternalSorter` (as `sorter`) is not defined and creates one itself.

`open` creates a new byte array output stream (as `serBuffer`) with the buffer capacity of `1M`.

`open` creates a new [SerializationStream](#) for the new byte array output stream using [SerializerInstance](#).

Note	<code>SerializerInstance</code> was defined when UnsafeShuffleWriter was created (and is exactly the one used to create the <code>shuffleDependency</code>).
------	---

Note	<code>open</code> is used exclusively when UnsafeShuffleWriter is created.
------	--

Inserting Record Into ShuffleExternalSorter

— `insertRecordIntoSorter` Method

```
void insertRecordIntoSorter(Product2<K, V> record)
throws IOException
```

`insertRecordIntoSorter` calculates the partition for the key of the input `record`.

Note	<code>Partitioner</code> is defined when UnsafeShuffleWriter is created.
------	--

`insertRecordIntoSorter` then writes the key and the value of the input `record` to [SerializationStream](#) and calculates the size of the serialized buffer.

Note	<code>SerializationStream</code> is created when UnsafeShuffleWriter opens.
------	---

In the end, `insertRecordIntoSorter` inserts the serialized buffer to [ShuffleExternalSorter](#) (as `Platform.BYTE_ARRAY_OFFSET`).

Note	<code>ShuffleExternalSorter</code> is created when UnsafeShuffleWriter opens.
------	---

Note	<code>insertRecordIntoSorter</code> is used exclusively when UnsafeShuffleWriter writes records.
------	--

Closing Internal Resources and Writing Spill Files Merged

— `closeAndWriteOutput` Method

```
void closeAndWriteOutput() throws IOException
```

`closeAndWriteOutput` first updates peak memory used.

`closeAndWriteOutput` removes the internal `ByteArrayOutputStream` and [SerializationStream](#).

`closeAndWriteOutput` requests [ShuffleExternalSorter](#) to close itself and return [SpillInfo](#) metadata.

`closeAndWriteOutput` removes the internal `shuffleExternalSorter`.

`closeAndWriteOutput` requests `IndexShuffleBlockResolver` for the data file for the `shuffleId` and `mapId`.

`closeAndWriteOutput` creates a temporary file to [merge spill files](#), deletes them afterwards, and requests `IndexShuffleBlockResolver` to write index file and commit.

`closeAndWriteOutput` creates a [MapStatus](#) with the [location of the executor's BlockManager](#) and partition lengths in the merged file.

If there is an issue with deleting spill files, you should see the following ERROR message in the logs:

```
ERROR Error while deleting spill file [path]
```

If there is an issue with deleting the temporary file, you should see the following ERROR message in the logs:

```
ERROR Error while deleting temp file [path]
```

Note	<code>closeAndWriteOutput</code> is used exclusively when UnsafeShuffleWriter writes records.
------	---

Settings

Table 2. Spark Properties

Spark Property	Default Value	Description
<code>spark.file.transferTo</code>	true	Controls whether... FIXME
<code>spark.shuffle.sort.initialBufferSize</code>	4096 (bytes)	Default initial sort buffer size

BaseShuffleHandle — Fallback Shuffle Handle

`BaseShuffleHandle` is a `ShuffleHandle` that is created solely to capture the parameters when `SortShuffleManager` is requested for a `ShuffleHandle` (for a `ShuffleDependency`):

1. `shuffleId`
2. `numMaps`
3. `ShuffleDependency`

Note

`BaseShuffleHandle` is the last possible choice when `SortShuffleManager` is requested for a `ShuffleHandle` (after `BypassMergeSortShuffleHandle` and `SerializedShuffleHandle` have already been considered and failed the check).

```

// Start a Spark application, e.g. spark-shell, with the Spark properties to trigger s
election of BaseShuffleHandle:
// 1. spark.shuffle.spill.numElementsForceSpillThreshold=1
// 2. spark.shuffle.sort.bypassMergeThreshold=1

// numSlices > spark.shuffle.sort.bypassMergeThreshold
scala> val rdd = sc.parallelize(0 to 4, numSlices = 2).groupBy(_ % 2)
rdd: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[2] at groupBy at <co
nsole>:24

scala> rdd.dependencies
DEBUG SortShuffleManager: Can't use serialized shuffle for shuffle 0 because an aggreg
ator is defined
res0: Seq[org.apache.spark.Dependency[_]] = List(org.apache.spark.ShuffleDependency@11
60c54b)

scala> rdd.getNumPartitions
res1: Int = 2

scala> import org.apache.spark.ShuffleDependency
import org.apache.spark.ShuffleDependency

scala> val shuffleDep = rdd.dependencies(0).asInstanceOf[ShuffleDependency[Int, Int, I
nt]]
shuffleDep: org.apache.spark.ShuffleDependency[Int, Int, Int] = org.apache.spark.Shuffle
Dependency@1160c54b

// mapSideCombine is disabled
scala> shuffleDep.mapSideCombine
res2: Boolean = false

// aggregator defined
scala> shuffleDep.aggregator
res3: Option[org.apache.spark.Aggregator[Int, Int, Int]] = Some(Aggregator(<function1>,<
function2>,<function2>))

// the number of reduce partitions < spark.shuffle.sort.bypassMergeThreshold
scala> shuffleDep.partitionner.numPartitions
res4: Int = 2

scala> shuffleDep.shuffleHandle
res5: org.apache.spark.shuffle.ShuffleHandle = org.apache.spark.shuffle.BaseShuffleHan
dle@22b0fe7e

```

BypassMergeSortShuffleHandle — Marker Interface for Bypass Merge Sort Shuffle Handles

`BypassMergeSortShuffleHandles` is a `BaseShuffleHandle` with no additional methods or fields and serves only to identify the choice of **bypass merge sort shuffle**.

Like `BaseShuffleHandle`, `BypassMergeSortShuffleHandles` takes `shuffleId`, `numMaps`, and a `ShuffleDependency`.

`BypassMergeSortShuffleHandle` is created when `SortShuffleManager` is requested for a `ShuffleHandle` (for a `ShuffleDependency`).

Note	Review the conditions <code>SortShuffleManager</code> uses to select <code>BypassMergeSortShuffleHandle</code> for a <code>ShuffleHandle</code> .
------	---

```
scala> val rdd = sc.parallelize(0 to 8).groupBy(_ % 3)
rdd: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[2] at groupBy at <console>:24

scala> rdd.dependencies
res0: Seq[org.apache.spark.Dependency[_]] = List(org.apache.spark.ShuffleDependency@65
5875bb)

scala> rdd.getNumPartitions
res1: Int = 8

scala> import org.apache.spark.ShuffleDependency
import org.apache.spark.ShuffleDependency

scala> val shuffleDep = rdd.dependencies(0).asInstanceOf[ShuffleDependency[Int, Int, I
nt]]
shuffleDep: org.apache.spark.ShuffleDependency[Int,Int,Int] = org.apache.spark.Shuffle
Dependency@655875bb

// mapSideCombine is disabled
scala> shuffleDep.mapSideCombine
res2: Boolean = false

// aggregator defined
scala> shuffleDep.aggregator
res3: Option[org.apache.spark.Aggregator[Int,Int,Int]] = Some(Aggregator(<function1>,<
function2>,<function2>))

// spark.shuffle.sort.bypassMergeThreshold == 200
// the number of reduce partitions < spark.shuffle.sort.bypassMergeThreshold
scala> shuffleDep.partitionner.numPartitions
res4: Int = 8

scala> shuffleDep.shuffleHandle
res5: org.apache.spark.shuffle.ShuffleHandle = org.apache.spark.shuffle.sort.BypassMer
geSortShuffleHandle@68893394
```

SerializedShuffleHandle — Marker Interface for Serialized Shuffle Handles

`SerializedShuffleHandle` is a [BaseShuffleHandle](#) with no additional methods or fields and serves only to identify the choice of a **serialized shuffle**.

Like [BaseShuffleHandle](#), `SerializedShuffleHandle` takes `shuffleId`, `numMaps`, and a [ShuffleDependency](#).

`SerializedShuffleHandle` is created when `SortShuffleManager` is requested for a `ShuffleHandle` (for a `ShuffleDependency`) and the [conditions hold](#) (but for [BypassMergeSortShuffleHandle](#) do not which are checked first).

ShuffleReader

Note

[BlockStoreShuffleReader](#) is the one and only `shuffleReader` in Spark.

Caution

[FIXME](#)

BlockStoreShuffleReader

`BlockStoreShuffleReader` is the one and only `ShuffleReader` that fetches and `reads the partitions` (in range [`startPartition` , `endPartition`]) from a shuffle by requesting them from other nodes' block stores.

`BlockStoreShuffleReader` is `created` when the default `SortShuffleManager` is requested for a `ShuffleReader` (for a `ShuffleHandle`).

Creating BlockStoreShuffleReader Instance

`BlockStoreShuffleReader` takes:

1. `BaseShuffleHandle`
2. `startPartition` and `endPartition` partition indices
3. `TaskContext`
4. (optional) `SerializerManager`
5. (optional) `BlockManager`
6. (optional) `MapOutputTracker`

Note

`BlockStoreShuffleReader` uses `SparkEnv` to define the optional `SerializerManager` , `BlockManager` and `MapOutputTracker` .

Reading Combined Key-Value Records For Reduce Task (using `ShuffleBlockFetcherIterator`) — `read` Method

```
read(): Iterator[Product2[K, C]]
```

Note

`read` is a part of `ShuffleReader` contract.

Internally, `read` first creates a `ShuffleBlockFetcherIterator` (passing in the values of `spark.reducer.maxSizeInFlight`, `spark.reducer.maxReqsInFlight` and `spark.shuffle.detectCorrupt` Spark properties).

Note

`read` uses `BlockManager` to access `ShuffleClient` to create `ShuffleBlockFetcherIterator` .

Note

`read` uses `MapOutputTracker` to find the `BlockManagers` with the shuffle blocks and sizes to create `ShuffleBlockFetcherIterator`.

`read` creates a new `SerializerInstance` (using `Serializer` from `ShuffleDependency`).

`read` creates a key/value iterator by `deserializeStream` every shuffle block stream.

`read` updates the `context task metrics` for each record read.

Note

`read` uses `CompletionIterator` (to count the records read) and `InterruptibleIterator` (to support task cancellation).

If the `ShuffleDependency` has an `Aggregator` defined, `read` wraps the current iterator inside an iterator defined by `Aggregator.combineCombinersByKey` (for `mapSideCombine` enabled) or `Aggregator.combineValuesByKey` otherwise.

Note

`run` reports an exception when `ShuffleDependency` has no `Aggregator` defined with `mapSideCombine` flag enabled.

For `keyOrdering` defined in `ShuffleDependency`, `run` does the following:

1. Creates an `ExternalSorter`
2. Inserts all the records into the `ExternalSorter`
3. Updates context `TaskMetrics`
4. Returns a `CompletionIterator` for the `ExternalSorter`

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.reducer.maxSizeInFlight</code>	48m	<p>Maximum size (in bytes) of map outputs to fetch simultaneously from each reduce task.</p> <p>Since each output requires a new buffer to receive it, this represents a fixed memory overhead per reduce task, so keep it small unless you have a large amount of memory.</p> <p>Used when BlockStoreShuffleReader creates a ShuffleBlockFetcherIterator to read records.</p>
<code>spark.reducer.maxReqsInFlight</code>	(unlimited)	<p>The maximum number of remote requests to fetch blocks at any given point.</p> <p>When the number of hosts in the cluster increases, it might lead to very large number of in-bound connections to one or more nodes, causing the workers to fail under load. By allowing it to limit the number of fetch requests, this scenario can be mitigated.</p> <p>Used when BlockStoreShuffleReader creates a ShuffleBlockFetcherIterator to read records.</p>
<code>spark.shuffle.detectCorrupt</code>	true	<p>Controls whether to detect any corruption in fetched blocks.</p> <p>Used when BlockStoreShuffleReader creates a ShuffleBlockFetcherIterator to read records.</p>

ShuffleBlockFetcherIterator

`ShuffleBlockFetcherIterator` is a Scala [Iterator](#) that fetches multiple shuffle blocks (aka *shuffle map outputs*) from local and remote BlockManagers.

`ShuffleBlockFetcherIterator` allows for [iterating over a sequence of blocks](#) as `(BlockId, InputStream)` pairs so a caller can handle shuffle blocks in a pipelined fashion as they are received.

`ShuffleBlockFetcherIterator` [throttles the remote fetches](#) to avoid using too much memory.

Table 1. ShuffleBlockFetcherIterator's Internal Registries and Counters

Name	Description
<code>results</code>	<p>Internal FIFO blocking queue (using Java's <code>java.util.concurrent.LinkedBlockingQueue</code>) to hold <code>FetchResult</code> remote and local fetch results.</p> <p>Used in:</p> <ol style="list-style-type: none"> 1. next to take one <code>FetchResult</code> off the queue, 2. sendRequest to put <code>SuccessFetchResult</code> or <code>FailureFetchResult</code> remote fetch results (as part of <code>BlockFetchingListener</code> callback), 3. fetchLocalBlocks (similarly to sendRequest) to put local fetch results, 4. cleanup to release managed buffers for <code>SuccessFetchResult</code> results.
<code>maxBytesInFlight</code>	<p>The maximum size (in bytes) of all the remote shuffle blocks to fetch.</p> <p>Set when <code>ShuffleBlockFetcherIterator</code> is created.</p>
<code>maxReqsInFlight</code>	<p>The maximum number of remote requests to fetch shuffle blocks.</p> <p>Set when <code>ShuffleBlockFetcherIterator</code> is created.</p>
<code>bytesInFlight</code>	<p>The bytes of fetched remote shuffle blocks in flight</p> <p>Starts at <code>0</code> when <code>ShuffleBlockFetcherIterator</code> is created.</p> <p>Incremented every sendRequest and decremented every next.</p>

	<p><code>ShuffleBlockFetcherIterator</code> makes sure that the invariant of <code>bytesInFlight</code> below <code>maxBytesInFlight</code> holds every remote shuffle block fetch.</p>
<code>reqsInFlight</code>	<p>The number of remote shuffle block fetch requests in flight.</p> <p>Starts at <code>0</code> when <code>ShuffleBlockFetcherIterator</code> is created.</p> <p>Incremented every <code>sendRequest</code> and decremented every <code>next</code>.</p> <p><code>ShuffleBlockFetcherIterator</code> makes sure that the invariant of <code>reqsInFlight</code> below <code>maxReqsInFlight</code> holds every remote shuffle block fetch.</p>
<code>isZombie</code>	<p>Flag whether <code>ShuffleBlockFetcherIterator</code> is still active. It is disabled, i.e. <code>false</code>, when <code>ShuffleBlockFetcherIterator</code> is created.</p> <p>When enabled (when the task using <code>ShuffleBlockFetcherIterator</code> finishes), the block fetch successful callback (registered in <code>sendRequest</code>) will no longer add fetched remote shuffle blocks into <code>results</code> internal queue.</p>
<code>currentResult</code>	<p>The currently-processed <code>SuccessFetchResult</code></p> <p>Set when <code>ShuffleBlockFetcherIterator</code> returns the next <code>(BlockId, InputStream)</code> tuple and released (on cleanup).</p>

Tip	<p>Enable <code>ERROR</code>, <code>WARN</code>, <code>INFO</code>, <code>DEBUG</code> or <code>TRACE</code> logging levels for <code>org.apache.spark.storage.ShuffleBlockFetcherIterator</code> logger to see what happens in <code>ShuffleBlockFetcherIterator</code>.</p>
	<p>Add the following line to <code>conf/log4j.properties</code>:</p> <pre>log4j.logger.org.apache.spark.storage.ShuffleBlockFetcherIterator=TRACE</pre> <p>Refer to Logging.</p>

splitLocalRemoteBlocks Method

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

fetchUpToMaxBytes Method

Caution

FIXME

fetchLocalBlocks Method

Caution

FIXME

Creating ShuffleBlockFetcherIterator Instance

When created, `ShuffleBlockFetcherIterator` takes the following:

1. `TaskContext`
2. `ShuffleClient`
3. `BlockManager`
4. `blocksByAddress` list of blocks to fetch per `BlockManager`.

```
blocksByAddress: Seq[(BlockManagerId, Seq[(BlockId, Long)])]
```

5. `streamWrapper` function to wrap the returned input stream

```
streamWrapper: (BlockId, InputStream) => InputStream
```

6. `maxBytesInFlight`— the maximum size (in bytes) of map outputs to fetch simultaneously from each reduce task (controlled by `spark.reducer.maxSizeInFlight` Spark property)
7. `maxReqsInFlight`— the maximum number of remote requests to fetch blocks at any given point (controlled by `spark.reducer.maxReqsInFlight` Spark property)
8. `detectCorrupt` flag to detect any corruption in fetched blocks (controlled by `spark.shuffle.detectCorrupt` Spark property)

Caution

FIXME

next Method

Caution

FIXME

Initializing ShuffleBlockFetcherIterator — `initializeInternal` Method

```
initialize(): Unit
```

`initialize` registers a task cleanup and fetches shuffle blocks from remote and local [BlockManagers](#).

Internally, `initialize` registers a [TaskCompletionListener](#) (that will [clean up](#) right after the task finishes).

`initialize` [splitLocalRemoteBlocks](#).

`initialize` registers the new remote fetch requests (with [fetchRequests](#) internal registry).

As `ShuffleBlockFetcherIterator` is in initialization phase, `initialize` makes sure that `reqsInFlight` and `bytesInFlight` internal counters are both `0`. Otherwise, `initialize` throws an exception.

`initialize` [fetches shuffle blocks](#) (from remote [BlockManagers](#)).

You should see the following INFO message in the logs:

```
INFO ShuffleBlockFetcherIterator: Started [numFetches] remote fetches in [time] ms
```

`initialize` [fetches local shuffle blocks](#).

You should see the following DEBUG message in the logs:

```
DEBUG ShuffleBlockFetcherIterator: Got local blocks in [time] ms
```

Note	<code>initialize</code> is used when <code>ShuffleBlockFetcherIterator</code> is created.
------	---

Sending Remote Shuffle Block Fetch Request — `sendRequest` Internal Method

```
sendRequest(req: FetchRequest): Unit
```

Internally, when `sendRequest` runs, you should see the following DEBUG message in the logs:

```
DEBUG ShuffleBlockFetcherIterator: Sending request for [blocks.size] blocks ([size] B)  
from [hostPort]
```

`sendRequest` increments `bytesInFlight` and `reqsInFlight` internal counters.

Note

The input `FetchRequest` contains the remote `BlockManagerId` address and the shuffle blocks to fetch (as a sequence of `BlockId` and their sizes).

`sendRequest` requests `shuffleClient` to fetch shuffle blocks (from the host, the port, and the executor as defined in the input `FetchRequest`).

Note

`shuffleClient` was defined when `ShuffleBlockFetcherIterator` was created.

`sendRequest` registers a `BlockFetchingListener` with `shuffleClient` that:

1. For every successfully fetched shuffle block adds it as `SuccessFetchResult` to `results` internal queue.
2. For every shuffle block fetch failure adds it as `FailureFetchResult` to `results` internal queue.

Note

`sendRequest` is used exclusively when `ShuffleBlockFetcherIterator` fetches remote shuffle blocks.

onBlockFetchSuccess Callback

```
onBlockFetchSuccess(blockId: String, buf: ManagedBuffer): Unit
```

Internally, `onBlockFetchSuccess` checks if the iterator is not zombie and does the further processing if it is not.

`onBlockFetchSuccess` marks the input `blockId` as received (i.e. removes it from all the blocks to fetch as requested in `sendRequest`).

`onBlockFetchSuccess` adds the managed `buf` (as `SuccessFetchResult`) to `results` internal queue.

You should see the following DEBUG message in the logs:

```
DEBUG ShuffleBlockFetcherIterator: remainingBlocks: [blocks]
```

Regardless of zombie state of `ShuffleBlockFetcherIterator`, you should see the following TRACE message in the logs:

```
TRACE ShuffleBlockFetcherIterator: Got remote block [blockId] after [time] ms
```

onBlockFetchFailure Callback

```
onBlockFetchFailure(blockId: String, e: Throwable): Unit
```

When `onBlockFetchFailure` is called, you should see the following ERROR message in the logs:

```
ERROR ShuffleBlockFetcherIterator: Failed to get block(s) from [hostPort]
```

`onBlockFetchFailure` adds the block (as `FailureFetchResult`) to `results` internal queue.

Throwing FetchFailedException (for ShuffleBlockId) — `throwFetchFailedException` Internal Method

```
throwFetchFailedException(  
    blockId: BlockId,  
    address: BlockManagerId,  
    e: Throwable): Nothing
```

`throwFetchFailedException` throws a `FetchFailedException` when the input `blockId` is a `ShuffleBlockId`.

Note	<code>throwFetchFailedException</code> creates a <code>FetchFailedException</code> passing on the root cause of a failure, i.e. the input <code>e</code> .
------	--

Otherwise, `throwFetchFailedException` throws a `SparkException`:

```
Failed to get block [blockId], which is not a shuffle block
```

Note	<code>throwFetchFailedException</code> is used when <code>ShuffleBlockFetcherIterator</code> is requested for the next element.
------	---

Releasing Resources — `cleanup` Internal Method

```
cleanup(): Unit
```

Internally, `cleanup` marks `shuffleBlockFetcherIterator` a [zombie](#).

`cleanup` releases the current result buffer.

`cleanup` iterates over `results` internal queue and for every `SuccessFetchResult`, increments remote bytes read and blocks fetched shuffle task metrics, and eventually releases the managed buffer.

Note	<code>cleanup</code> is used when <code>ShuffleBlockFetcherIterator</code> initializes itself.
------	--

Decrementing Reference Count Of and Releasing Result Buffer (for SuccessFetchResult)

— `releaseCurrentResultBuffer` Internal Method

```
releaseCurrentResultBuffer(): Unit
```

`releaseCurrentResultBuffer` decrements the currently-processed `SuccessFetchResult` reference's buffer reference count if there is any.

`releaseCurrentResultBuffer` releases `currentResult`.

Note	<code>releaseCurrentResultBuffer</code> is used when <code>ShuffleBlockFetcherIterator</code> releases resources and <code>BufferReleasingInputStream</code> closes.
------	--

ShuffleExternalSorter — Cache-Efficient Sorter

`ShuffleExternalSorter` is a specialized cache-efficient sorter that sorts arrays of compressed record pointers and partition ids. By using only 8 bytes of space per record in the sorting array, `ShuffleExternalSorter` can fit more of the array into cache.

`ShuffleExternalSorter` is a [MemoryConsumer](#).

Table 1. ShuffleExternalSorter's Internal Properties

Name	Initial Value	Description
<code>inMemSorter</code>	(empty)	<code>ShuffleInMemorySorter</code>

Tip	Enable <code>INFO</code> or <code>ERROR</code> logging levels for <code>org.apache.spark.shuffle.sort.ShuffleExternalSorter</code> logger to see what happens in <code>ShuffleExternalSorter</code> . Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.shuffle.sort.ShuffleExternalSorter=INFO</code>
	Refer to Logging .

getMemoryUsage Method

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

closeAndGetSpills Method

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

insertRecord Method

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

freeMemory Method

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

getPeakMemoryUsedBytes Method

Caution

FIXME

writeSortedFile Method

Caution

FIXME

cleanupResources Method

Caution

FIXME

Creating ShuffleExternalSorter Instance

`ShuffleExternalSorter` takes the following when created:

1. `memoryManager` — [TaskMemoryManager](#)
2. `blockManager` — [BlockManager](#)
3. `taskContext` — [TaskContext](#)
4. `initialSize`
5. `numPartitions`
6. [SparkConf](#)
7. `writeMetrics` — [ShuffleWriteMetrics](#)

`ShuffleExternalSorter` initializes itself as a [MemoryConsumer](#) (with `pagesize` as the minimum of `PackedRecordPointer.MAXIMUM_PAGE_SIZE_BYTES` and `pageSizeBytes`, and Tungsten memory mode).

`ShuffleExternalSorter` uses `spark.shuffle.file.buffer` (for `fileBufferSizeBytes`) and `spark.shuffle.spill.numElementsForceSpillThreshold` (for `numElementsForSpillThreshold`) Spark properties.

`ShuffleExternalSorter` creates a [ShuffleInMemorySorter](#) (with `spark.shuffle.sort.useRadixSort` Spark property enabled by default).

`ShuffleExternalSorter` initializes the internal registries and counters.

Note

`ShuffleExternalSorter` is created when `unsafeShuffleWriter` is open (which is when `UnsafeShuffleWriter` is created).

Freeing Execution Memory by Spilling To Disk — `spill` Method

```
long spill(long size, MemoryConsumer trigger)  
throws IOException
```

Note `spill` is a part of [MemoryConsumer contract](#) to sort and spill the current records due to memory pressure.

`spill` frees execution memory, updates [TaskMetrics](#), and in the end returns the spill size.

Note `spill` returns 0 when `ShuffleExternalSorter` has no [ShuffleInMemorySorter](#) or the `ShuffleInMemorySorter` manages no records.

You should see the following INFO message in the logs:

```
INFO Thread [id] spilling sort data of [memoryUsage] to disk ([size] times so far)
```

`spill` writes sorted file (with `isLastFile` disabled).

`spill` frees memory and records the spill size.

`spill` resets the internal `ShuffleInMemorySorter` (that in turn frees up the underlying in-memory pointer array).

`spill` adds the spill size to [TaskMetrics](#).

`spill` returns the spill size.

ExternalSorter

`ExternalSorter` is a `Spillable` of `WritablePartitionedPairCollection` of `k`-key / `c`-value pairs.

When created `ExternalSorter` expects three different types of data defined, i.e. `k`, `v`, `c`, for keys, values, and combiner (partial) values, respectively.

Note	<code>ExternalSorter</code> is exclusively used when <code>SortShuffleWriter</code> writes records and <code>BlockStoreShuffleReader</code> reads combined key-value pairs (for reduce task when <code>ShuffleDependency</code> has key ordering defined (to sort output)).
------	---

Tip	Enable <code>INFO</code> or <code>WARN</code> logging levels for <code>org.apache.spark.util.collection.ExternalSorter</code> logger to see what happens in <code>ExternalSorter</code> .
-----	---

Tip	Add the following line to <code>conf/log4j.properties</code> :
-----	--

Tip	log4j.logger.org.apache.spark.util.collection.ExternalSorter=INFO
-----	---

Tip	Refer to Logging .
-----	------------------------------------

stop Method

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

writePartitionedFile Method

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

Creating ExternalSorter Instance

`ExternalSorter` takes the following:

1. TaskContext
2. Optional [Aggregator](#)
3. Optional [Partitioner](#)
4. Optional Scala's Ordering
5. Optional [Serializer](#)

Note

`ExternalSorter` uses `SparkEnv` to access the default `Serializer`.

Note

`ExternalSorter` is created when `SortShuffleWriter` writes records and `BlockStoreShuffleReader` reads combined key-value pairs (for reduce task when `ShuffleDependency` has key ordering defined (to sort output)).

spillMemoryIteratorToDisk Internal Method

```
spillMemoryIteratorToDisk(inMemoryIterator: WritablePartitionedIterator): SpilledFile
```

Caution

[FIXME](#)

spill Method

```
spill(collection: WritablePartitionedPairCollection[K, C]): Unit
```

Note

`spill` is a part of Spillable contract.

Caution

[FIXME](#)

maybeSpillCollection Internal Method

```
maybeSpillCollection(usingMap: Boolean): Unit
```

Caution

[FIXME](#)

insertAll Method

```
insertAll(records: Iterator[Product2[K, V]]): Unit
```

Caution

[FIXME](#)

Note

`insertAll` is used when `SortShuffleWriter` writes records and `BlockStoreShuffleReader` reads combined key-value pairs (for reduce task when `ShuffleDependency` has key ordering defined (to sort output)).

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.shuffle.file.buffer</code>	32k	<p>Size of the in-memory buffer for each shuffle file output stream. In bytes unless the unit is specified.</p> <p>These buffers reduce the number of disk seeks and system calls made in creating intermediate shuffle files.</p> <p>Used in <code>ExternalSorter</code>, <code>BypassMergeSortShuffleWriter</code> and <code>ExternalAppendOnlyMap</code> (for <code>fileBufferSize</code>) and in <code>ShuffleExternalSorter</code> (for <code>fileBufferSizeBytes</code>).</p> <p>NOTE: <code>spark.shuffle.file.buffer</code> was previously known as <code>spark.shuffle.file.buffer.kb</code>.</p>
<code>spark.shuffle.spill.batchSize</code>	10000	Size of object batches when reading/writing from serializers.

Serialization

Serialization systems:

- Java serialization
- Kryo
- Avro
- Thrift
- Protobuf

Serializer — Task Serialization and Deserialization

`newInstance` Method

Caution

[FIXME](#)

`deserialize` Method

Caution

[FIXME](#)

`supportsRelocationOfSerializedObjects` Property

`supportsRelocationOfSerializedObjects` should be enabled (i.e. `true`) only when reordering the bytes of serialized objects in serialization stream output is equivalent to having reordered those elements prior to serializing them.

`supportsRelocationOfSerializedObjects` is disabled (i.e. `false`) by default.

Note `KryoSerializer` uses `autoReset` for `supportsRelocationOfSerializedObjects`.

Note `supportsRelocationOfSerializedObjects` is enabled in `UnsafeRowSerializer`.

SerializerInstance

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

serializeStream Method

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

SerializationStream

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

writeKey Method

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

writeValue Method

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

DeserializationStream

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

ExternalClusterManager — Pluggable Cluster Managers

`ExternalClusterManager` is a contract for pluggable cluster managers. It returns a [task scheduler](#) and a [backend scheduler](#) that will be used by [SparkContext](#) to schedule tasks.

Note

The support for pluggable cluster managers was introduced in [SPARK-13904](#) [Add support for pluggable cluster manager](#).

External cluster managers are registered using the `java.util.ServiceLoader` mechanism (with service markers under `META-INF/services` directory). This allows auto-loading implementations of `ExternalClusterManager` interface.

Note

`ExternalClusterManager` is a `private[spark]` trait in `org.apache.spark.scheduler` package.

Note

The two implementations of the [ExternalClusterManager contract](#) in Spark 2.0 are [YarnClusterManager](#) and [MesosClusterManager](#).

ExternalClusterManager Contract

canCreate Method

```
canCreate(masterURL: String): Boolean
```

`canCreate` is a mechanism to match a `ExternalClusterManager` implementation to a given master URL.

Note

`canCreate` is used when `SparkContext` loads the external cluster manager for a master URL.

createTaskScheduler Method

```
createTaskScheduler(sc: SparkContext, masterURL: String): TaskScheduler
```

`createTaskScheduler` creates a [TaskScheduler](#) given a [SparkContext](#) and the input `masterURL`.

createSchedulerBackend Method

```
createSchedulerBackend(sc: SparkContext,  
                      masterURL: String,  
                      scheduler: TaskScheduler): SchedulerBackend
```

`createSchedulerBackend` creates a [SchedulerBackend](#) given a [SparkContext](#), the input `masterURL`, and [TaskScheduler](#).

Initializing Scheduling Components — `initialize` Method

```
initialize(scheduler: TaskScheduler, backend: SchedulerBackend): Unit
```

`initialize` is called after the [task scheduler](#) and the [backend scheduler](#) were created and initialized separately.

Note	There is a cyclic dependency between a task scheduler and a backend scheduler that begs for this additional initialization step.
------	--

There is a cyclic dependency between a task scheduler and a backend scheduler that begs for this additional initialization step.

Note	TaskScheduler and SchedulerBackend (with DAGScheduler) are commonly referred to as scheduling components .
------	--

[TaskScheduler](#) and [SchedulerBackend](#) (with [DAGScheduler](#)) are commonly referred to as **scheduling components**.

BroadcastManager

Broadcast Manager (`BroadcastManager`) is a Spark service to manage [broadcast variables](#) in Spark. It is created for a Spark application when [SparkContext is initialized](#) and is a simple wrapper around [BroadcastFactory](#).

`BroadcastManager` tracks the number of broadcast variables in a Spark application (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 [initialized](#), `BroadcastManager` creates an instance of [TorrentBroadcastFactory](#).

stop Method

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

Creating BroadcastManager Instance

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

Initializing BroadcastManager — initialize Internal Method

<code>initialize(): Unit</code>

`initialize` creates and initializes a [TorrentBroadcastFactory](#).

Note	<code>initialize</code> is executed only once (when <code>BroadcastManager</code> is created) and controlled by the internal <code>initialized</code> flag.
------	---

newBroadcast Method

<code>newBroadcast[T](value_ : T, isLocal: Boolean): Broadcast[T]</code>
--

`newBroadcast` simply requests the current [BroadcastFactory](#) for a new broadcast variable.

Note	The <code>BroadcastFactory</code> is created when <code>BroadcastManager</code> is initialized.
Note	<code>newBroadcast</code> is executed for <code>SparkContext.broadcast</code> method and when <code>MapOutputTracker</code> serializes <code>MapStatuses</code> .

Settings

Table 1. Settings

Name	Default value	Description
<code>spark.broadcast.blockSize</code>	<code>4m</code>	<p>The size of a block (in kB when unit not specified).</p> <p>Used when <code>TorrentBroadcast</code> stores broadcast blocks to <code>BlockManager</code>.</p>
<code>spark.broadcast.compress</code>	<code>true</code>	<p>The flag to enable compression.</p> <p>Refer to CompressionCodec.</p> <p>Used when <code>TorrentBroadcast</code> is created and later when it stores broadcast blocks to <code>BlockManager</code>. Also in <code>SerializerManager</code>.</p>

BroadcastFactory — Pluggable Broadcast Variable Factories

`BroadcastFactory` is the interface for factories of [broadcast variables](#) in Spark.

Note	As of Spark 2.0, it is no longer possible to plug a custom <code>BroadcastFactory</code> in, and TorrentBroadcastFactory is the only implementation.
------	--

`BroadcastFactory` is exclusively used and instantiated inside of [BroadcastManager](#).

Table 1. `BroadcastFactory` Contract

Method	Description
<code>initialize</code>	
<code>newBroadcast</code>	
<code>unbroadcast</code>	
<code>stop</code>	

TorrentBroadcastFactory

`TorrentBroadcastFactory` is a [BroadcastFactory](#) of [TorrentBroadcast](#)s, i.e. BitTorrent-like broadcast variables.

Note

As of Spark 2.0 `TorrentBroadcastFactory` is the only implementation of [BroadcastFactory](#).

`newBroadcast` method creates a `TorrentBroadcast` (passing in the input `value_` and `id` and ignoring the `isLocal` parameter).

Note

`newBroadcast` is executed when `BroadcastManager` is requested to create a new broadcast variable.

`initialize` and `stop` do nothing.

`unbroadcast` removes all the persisted state associated with a `TorrentBroadcast` of a given ID.

TorrentBroadcast — Default Broadcast Implementation

`TorrentBroadcast` is the default and only implementation of the `Broadcast Contract` that describes `broadcast variables`. `TorrentBroadcast` uses a BitTorrent-like protocol for block distribution (that only happens when tasks access broadcast variables on executors).

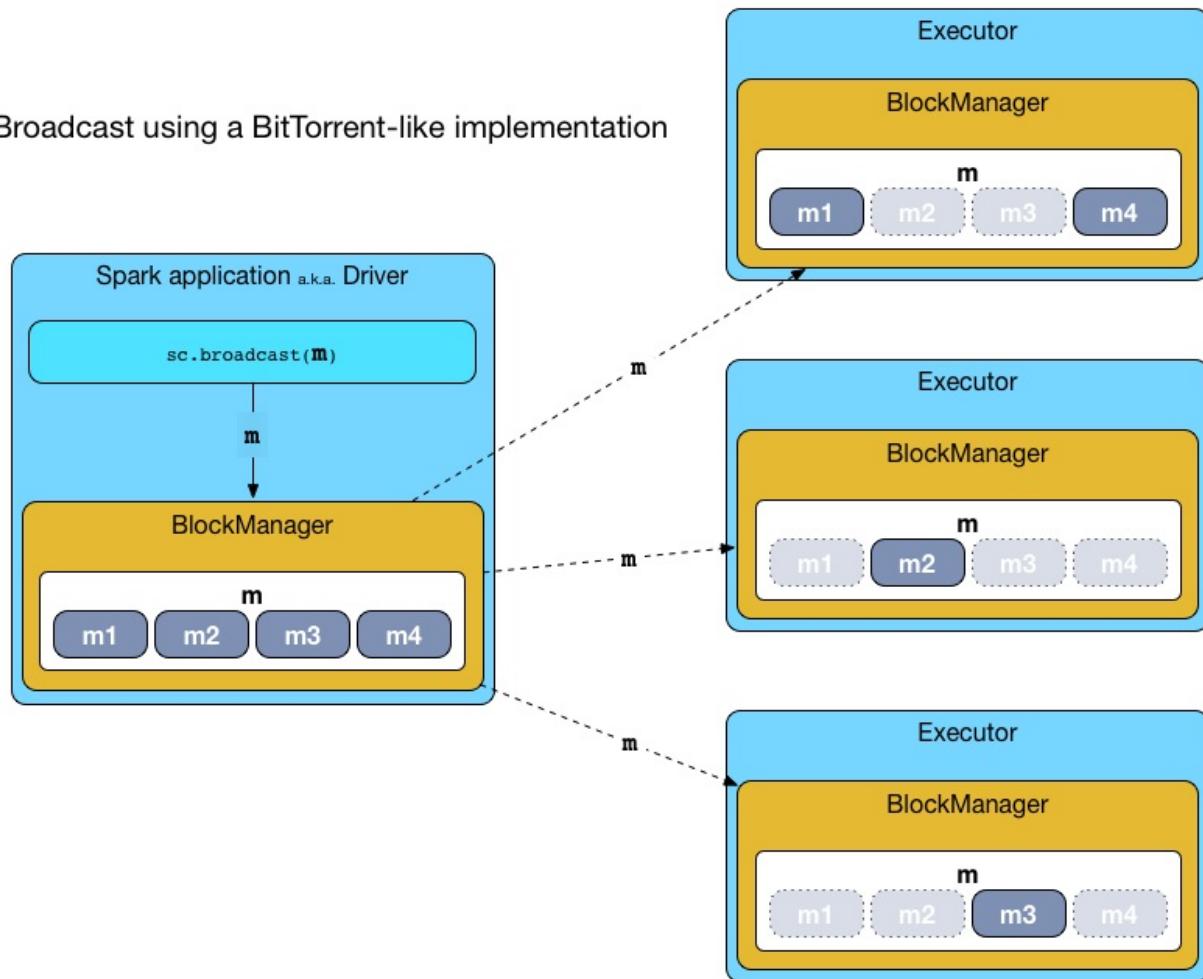


Figure 1. TorrentBroadcast - broadcasting using BitTorrent

When a `broadcast variable is created` (using `SparkContext.broadcast`) on the driver, a new instance of `TorrentBroadcast` is created.

```
// On the driver
val sc: SparkContext = ???
val anyScalaValue = ???
val b = sc.broadcast(anyScalaValue) // <-- TorrentBroadcast is created
```

A broadcast variable is stored on the driver's BlockManager as a single value and separately as broadcast blocks (after it was [divided into broadcast blocks, i.e. blockified](#)). The broadcast block size is the value of [spark.broadcast.blockSize](#) Spark property.

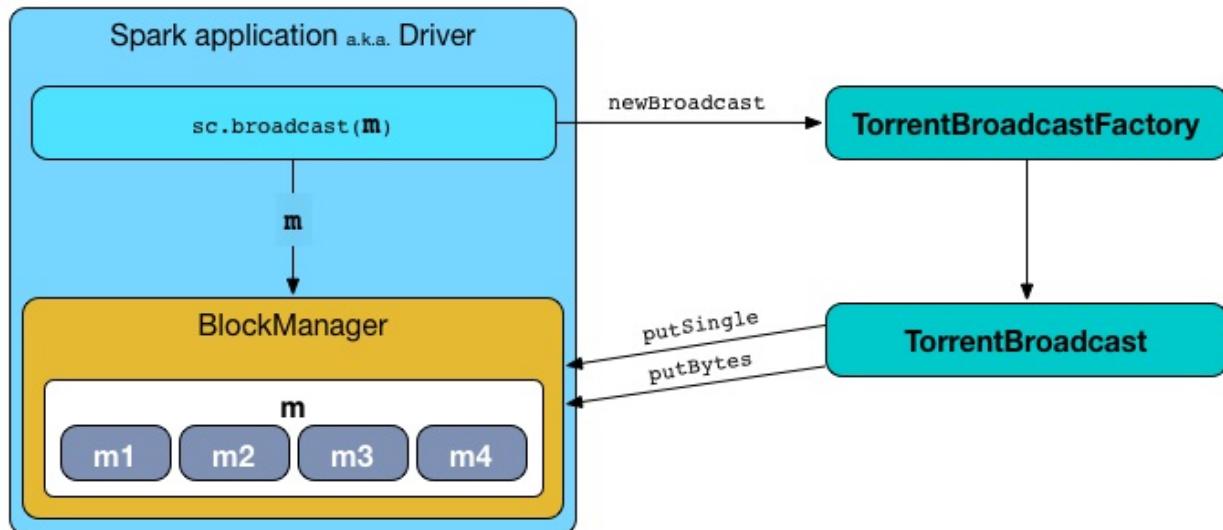


Figure 2. TorrentBroadcast puts broadcast and the chunks to driver's BlockManager

Note

TorrentBroadcast -based broadcast variables are created using [TorrentBroadcastFactory](#).

Note

TorrentBroadcast belongs to `org.apache.spark.broadcast` package.

Tip

Enable `INFO` or `DEBUG` logging levels for `org.apache.spark.broadcast.TorrentBroadcast` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.broadcast.TorrentBroadcast=DEBUG
```

Refer to [Logging](#).

unBlockifyObject Method

Caution

[FIXME](#)

readBlocks Method

Caution

[FIXME](#)

releaseLock Method

Caution

FIXME

Creating TorrentBroadcast Instance

```
TorrentBroadcast[T](obj: T, id: Long)
extends Broadcast[T](id)
```

When created, `TorrentBroadcast` reads broadcast blocks (to the internal `_value`).

Note	The internal <code>_value</code> is transient so it is not serialized and sent over the wire to executors. It is later recreated lazily on executors when requested.
------	--

`TorrentBroadcast` then sets the internal optional `CompressionCodec` and the size of broadcast block (as controlled by `spark.broadcast.blockSize` Spark property in `SparkConf` per driver and executors).

Note	Compression is controlled by <code>spark.broadcast.compress</code> Spark property and is enabled by default.
------	--

The internal `broadcastId` is `BroadcastBlockId` for the input `id`.

The internal `numBlocks` is set to the number of the pieces the broadcast was divided into.

Note	A broadcast's blocks are first stored in the local <code>BlockManager</code> on the driver.
------	---

Getting Value of Broadcast Variable — `getValue` Method

```
def getValue(): T
```

`getValue` returns the value of a broadcast variable.

Note	<code>getValue</code> is a part of the <code>Broadcast</code> Variable Contract and is the only way to access the value of a broadcast variable.
------	--

Internally, `getValue` reads the internal `_value` that, once accessed, reads broadcast blocks from the local or remote BlockManagers.

Note	The internal <code>_value</code> is <i>transient</i> and <i>lazy</i> , i.e. it is not preserved when serialized and (re)created only when requested, respectively. That "trick" allows for serializing broadcast values on the driver before they are transferred to executors over the wire.
------	---

readBroadcastBlock Internal Method

`readBroadcastBlock(): T`

Internally, `readBroadcastBlock` sets the `SparkConf`

Note	The current <code>SparkConf</code> is available using <code>SparkEnv.get.conf</code> .
------	--

`readBroadcastBlock` requests the local `BlockManager` for values of the broadcast.

Note	The current <code>BlockManager</code> is available using <code>SparkEnv.get.blockManager</code> .
------	---

If the broadcast was available locally, `readBroadcastBlock` releases a lock for the broadcast and returns the value.

If however the broadcast was not found locally, you should see the following INFO message in the logs:

`INFO Started reading broadcast variable [id]`

`readBroadcastBlock` reads blocks (as chunks) of the broadcast.

You should see the following INFO message in the logs:

`INFO Reading broadcast variable [id] took [usedTimeMs]`

`readBroadcastBlock` unblockifies the collection of `ByteBuffer` blocks

Note	<code>readBroadcastBlock</code> uses the current <code>Serializer</code> and the internal <code>CompressionCodec</code> to bring all the blocks together as one single broadcast variable.
------	--

`readBroadcastBlock` stores the broadcast variable with `MEMORY_AND_DISK` storage level to the local `BlockManager`. When storing the broadcast variable was unsuccessful, a `SparkException` is thrown.

`Failed to store [broadcastId] in BlockManager`

The broadcast variable is returned.

Note	<code>readBroadcastBlock</code> is exclusively used to recreate a broadcast variable on executors.
------	--

setConf Internal Method

```
setConf(conf: SparkConf): Unit
```

`setConf` uses the input `conf` [SparkConf](#) to set compression codec and the block size.

Internally, `setConf` reads [spark.broadcast.compress](#) Spark property and if enabled (which it is by default) sets a [CompressionCodec](#) (as an internal `compressionCodec` property).

`setConf` also reads [spark.broadcast.blockSize](#) Spark property and sets the block size (as the internal `blockSize` property).

Note	<code>setConf</code> is executed when TorrentBroadcast is created or re-created when deserialized on executors.
------	---

Storing Broadcast and Its Blocks in Local BlockManager — writeBlocks Internal Method

```
writeBlocks(value: T): Int
```

`writeBlocks` is an internal method to store the broadcast's `value` and blocks in the driver's [BlockManager](#). It returns the number of the broadcast blocks the broadcast was divided into.

Note	<code>writeBlocks</code> is exclusively used when a TorrentBroadcast is created that happens on the driver only. It sets the internal <code>numBlocks</code> property that is serialized as a number before the broadcast is sent to executors (after they have called <code>value</code> method).
------	--

Internally, `writeBlocks` stores the block for `value` broadcast to the local [BlockManager](#) (using a new [BroadcastBlockId](#), `value`, [MEMORY_AND_DISK](#) storage level and without telling the driver).

If storing the broadcast block fails, you should see the following [SparkException](#) in the logs:

```
Failed to store [broadcastId] in BlockManager
```

`writeBlocks` divides `value` into blocks (of [spark.broadcast.blockSize](#) size) using the [Serializer](#) and an optional [CompressionCodec](#) (enabled by [spark.broadcast.compress](#)). Every block gets its own `BroadcastBlockId` (with `piece` and an index) that is wrapped inside a `ChunkedByteBuffer`. Blocks are stored in the local [BlockManager](#) (using the `piece` block id, [MEMORY_AND_DISK_SER](#) storage level and informing the driver).

Note	The entire broadcast value is stored in the local <code>BlockManager</code> with <code>MEMORY_AND_DISK</code> storage level, and the pieces with <code>MEMORY_AND_DISK_SER</code> storage level.
------	--

If storing any of the broadcast pieces fails, you should see the following `SparkException` in the logs:

```
Failed to store [pieceId] of [broadcastId] in local BlockManager
```

Chunking Broadcast Into Blocks — `blockifyObject` Method

```
blockifyObject[T](
  obj: T,
  blockSize: Int,
  serializer: Serializer,
  compressionCodec: Option[CompressionCodec]): Array[ByteBuffer]
```

`blockifyObject` divides (aka *blockifies*) the input `obj` broadcast variable into blocks (of `ByteBuffer`). `blockifyObject` uses the input `serializer` `Serializer` to write `obj` in a serialized format to a `chunkedByteBufferOutputStream` (of `blocksize` size) with the optional `CompressionCodec`.

Note	<code>blockifyObject</code> is executed when <code>TorrentBroadcast</code> stores a broadcast and its blocks to a local <code>BlockManager</code> .
------	---

`doUnpersist` Method

```
doUnpersist(blocking: Boolean): Unit
```

`doUnpersist` removes all the persisted state associated with a broadcast variable on executors.

Note	<code>doUnpersist</code> is a part of the <code>Broadcast</code> Variable Contract and is executed from <code>unpersist</code> method.
------	--

`doDestroy` Method

```
doDestroy(blocking: Boolean): Unit
```

`doDestroy` removes all the persisted state associated with a broadcast variable on all the nodes in a Spark application, i.e. the driver and executors.

Note

`doDestroy` is executed when `Broadcast` removes the persisted data and metadata related to a broadcast variable.

unpersist Internal Method

```
unpersist(  
    id: Long,  
    removeFromDriver: Boolean,  
    blocking: Boolean): Unit
```

`unpersist` removes all broadcast blocks from executors and possibly the driver (only when `removeFromDriver` flag is enabled).

Note

`unpersist` belongs to `TorrentBroadcast` private object and is executed when `TorrentBroadcast` unpersists a broadcast variable and removes a broadcast variable completely.

When executed, you should see the following DEBUG message in the logs:

```
DEBUG TorrentBroadcast: Unpersisting TorrentBroadcast [id]
```

`unpersist` requests `BlockManagerMaster` to remove the `id` broadcast.

Note

`unpersist` uses `SparkEnv` to get the `BlockManagerMaster` (through `blockManager` property).

CompressionCodec

With `spark.broadcast.compress` enabled (which is the default), `TorrentBroadcast` uses compression for broadcast blocks.

Caution	FIXME What's compressed?
---------	--

Table 1. Built-in Compression Codecs

Codec Alias	Fully-Qualified Class Name	Notes
lz4	<code>org.apache.spark.io.LZ4CompressionCodec</code>	The default implementation
lzf	<code>org.apache.spark.io.LZFCompressionCodec</code>	
snappy	<code>org.apache.spark.io.SnappyCompressionCodec</code>	The fallback when the default codec is not available.

An implementation of `CompressionCodec` trait has to offer a constructor that accepts a single argument being `SparkConf`. Read [Creating `compressionCodec` — `createCodec` Factory Method](#) in this document.

You can control the default compression codec in a Spark application using `spark.io.compression.codec` Spark property.

Creating `CompressionCodec` — `createCodec` Factory Method

```
createCodec(conf: SparkConf): CompressionCodec (1)
createCodec(conf: SparkConf, codecName: String): CompressionCodec (2)
```

`createCodec` uses the internal `shortCompressionCodecNames` lookup table to find the input `codecName` (regardless of the case).

`createCodec` finds the constructor of the compression codec's implementation (that accepts a single argument being `SparkConf`).

If a compression codec could not be found, `createCodec` throws a `IllegalArgumentException` exception:

Codec [<codecName>] is not available. Consider setting spark.io.compression.codec=snap
py

getCodecName Method

getCodecName(conf: SparkConf): String

getCodecName reads `spark.io.compression.codec` Spark property from the input `conf` `SparkConf` or assumes `lz4`.

Note getCodecName is used when `SparkContext` sets up event logging (for History Server) or when creating a `CompressionCodec`.

Settings

Table 2. Settings

Name	Default value	Description
<code>spark.io.compression.codec</code>	<code>lz4</code>	The compression codec to use. Used when <code>getCodecName</code> is called to find the current compression codec.

ContextCleaner — Spark Application Garbage Collector

`ContextCleaner` is a Spark service that is responsible for [application-wide cleanup](#) of [shuffles](#), [RDDs](#), [broadcasts](#), [accumulators](#) and [checkpointed RDDs](#) that is aimed at reducing the memory requirements of long-running data-heavy Spark applications.

`ContextCleaner` runs on the driver. It is created and immediately started when `SparkContext` starts (and `spark.cleaner.referenceTracking` Spark property is enabled, which it is by default). It is stopped when `SparkContext` is stopped.

Table 1. ContextCleaner's Internal Registries and Counters

Name	Description
<code>referenceBuffer</code>	Used when ???
<code>referenceQueue</code>	Used when ???
<code>listeners</code>	Used when ???

It uses a daemon **Spark Context Cleaner** thread that cleans RDD, shuffle, and broadcast states (using `keepCleaning` method).

[ShuffleDependencies register themselves for cleanup.](#)

Tip	Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.ContextCleaner</code> logger to see what happens in <code>ContextCleaner</code> . Add the following line to <code>conf/log4j.properties</code> : <pre>log4j.logger.org.apache.spark.ContextCleaner=DEBUG</pre>
	Refer to Logging .

doCleanupRDD Method

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

keepCleaning Internal Method

```
keepCleaning(): Unit
```

`keepCleaning` runs indefinitely until `ContextCleaner` is stopped. It...[FIXME](#)

You should see the following DEBUG message in the logs:

```
DEBUG Got cleaning task [task]
```

Note	<code>keepCleaning</code> is exclusively used in Spark Context Cleaner Cleaning Thread that is started once when <code>ContextCleaner</code> is started.
------	--

Spark Context Cleaner Cleaning Thread — `cleaningThread` Attribute

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

The name of the daemon thread is **Spark Context Cleaner**.

```
$ jstack -l [sparkPID] | grep "Spark Context Cleaner"
"Spark Context Cleaner" #80 daemon prio=5 os_prio=31 tid=0x00007fc304677800 nid=0xa103
in Object.wait() [0x0000000120371000]
```

Note	<code>cleaningThread</code> is started as a daemon thread when <code>ContextCleaner</code> starts.
------	--

registerRDDCheckpointDataForCleanup Method

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

registerBroadcastForCleanup Method

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

registerRDDForCleanup Method

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

registerAccumulatorForCleanup Method

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

stop Method

Caution

[FIXME](#)

Creating ContextCleaner Instance

ContextCleaner takes a [SparkContext](#).

ContextCleaner initializes the internal registries and counters.

Starting ContextCleaner — start Method

`start(): Unit`

start starts [cleaning thread](#) and an action to request the JVM garbage collector (using `System.gc()`) every [spark.cleaner.periodicGC.interval](#) interval.

Note

The action to request the JVM GC is scheduled on `periodicGCSchedule` executor service.

periodicGCSchedule Single-Thread Executor Service

`periodicGCSchedule` is an internal single-thread [executor service](#) with the name `context-cleaner-periodic-gc` to request the JVM garbage collector.

Note

Requests for JVM GC are scheduled every [spark.cleaner.periodicGC.interval](#) interval.

The periodic runs are started when ContextCleaner starts and stopped when ContextCleaner stops.

Registering ShuffleDependency for Cleanup — registerShuffleForCleanup Method

`registerShuffleForCleanup(shuffleDependency: ShuffleDependency[_, _, _]): Unit`

registerShuffleForCleanup registers a [ShuffleDependency](#) for cleanup.

Internally, registerShuffleForCleanup simply executes [registerForCleanup](#) for the input `ShuffleDependency`.

Note	<code>registerShuffleForCleanup</code> is exclusively used when <code>ShuffleDependency</code> is created.
------	--

Registering Object Reference For Cleanup — `registerForCleanup` Internal Method

```
registerForCleanup(objectForCleanup: AnyRef, task: CleanupTask): Unit
```

Internally, `registerForCleanup` adds the input `objectForCleanup` to `referenceBuffer` internal queue.

Note	Despite the widest-possible <code>AnyRef</code> type of the input <code>objectForCleanup</code> , the type is really <code>CleanupTaskWeakReference</code> which is a custom Java's <code>java.lang.ref.WeakReference</code> .
------	--

Removing Shuffle Blocks From `MapOutputTrackerMaster` and `BlockManagerMaster` — `doCleanupShuffle` Method

```
doCleanupShuffle(shuffleId: Int, blocking: Boolean): Unit
```

`doCleanupShuffle` performs a shuffle cleanup which is to remove the shuffle from the current `MapOutputTrackerMaster` and `BlockManagerMaster`. `doCleanupShuffle` also notifies `CleanerListeners`.

Internally, when executed, you should see the following DEBUG message in the logs:

```
DEBUG Cleaning shuffle [id]
```

`doCleanupShuffle` unregisters the input `shuffleId` from `MapOutputTrackerMaster`.

Note	<code>doCleanupShuffle</code> uses <code>SparkEnv</code> to access the current <code>MapOutputTracker</code> .
------	--

`doCleanupShuffle` removes the shuffle blocks of the input `shuffleId` from `BlockManagerMaster`.

Note	<code>doCleanupShuffle</code> uses <code>SparkEnv</code> to access the current <code>BlockManagerMaster</code> .
------	--

`doCleanupShuffle` informs all registered `CleanerListener` listeners (from `listeners` internal queue) that the input `shuffleId` was cleaned.

In the end, you should see the following DEBUG message in the logs:

```
DEBUG Cleaned shuffle [id]
```

In case of any exception, you should see the following ERROR message in the logs and the exception itself.

```
ERROR Error cleaning shuffle [id]
```

Note

`doCleanupShuffle` is executed when `ContextCleaner` cleans a shuffle reference and (interestingly) while fitting a `ALSModel` (in Spark MLlib).

Settings

Table 2. Spark Properties

Spark Property	Default Value	Descr
<code>spark.cleaner.periodicGC.interval</code>	<code>30min</code>	Controls how often to trigger collection.
<code>spark.cleaner.referenceTracking</code>	<code>true</code>	Controls whether a <code>ContextCleaner</code> is created when a <code>SparkContext</code> is created.
<code>spark.cleaner.referenceTracking.blocking</code>	<code>true</code>	Controls whether the cleanup tasks block on cleanup tasks (which is controlled by <code>spark.cleaner.referenceTracking.blocking</code> Spark property). It is <code>true</code> as a workaround for <code>sparkContext.broadcast</code> . Removing broadcast in <code>ContextCleaner</code> causes Akka timeout.
<code>spark.cleaner.referenceTracking.blocking.shuffle</code>	<code>false</code>	Controls whether the cleanup tasks block on shuffle cleanup. It is <code>false</code> as a workaround for <code>sparkContext.broadcast</code> . Akka timeouts from ContextCleaner when cleaning shuffles.
<code>spark.cleaner.referenceTracking.cleanCheckpoints</code>	<code>false</code>	Controls whether to clean checkpoints when a reference is out of scope.

CleanerListener

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

shuffleCleaned Callback Method

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

Dynamic Allocation (of Executors)

Dynamic Allocation (of Executors) (aka *Elastic Scaling*) is a Spark feature that allows for adding or removing [Spark executors](#) dynamically to match the workload.

Unlike the "traditional" static allocation where a Spark application reserves CPU and memory resources upfront (irrespective of how much it may eventually use), in dynamic allocation you get as much as needed and no more. It scales the number of executors up and down based on workload, i.e. idle executors are removed, and when there are pending tasks waiting for executors to be launched on, dynamic allocation requests them.

Dynamic allocation is enabled using `spark.dynamicAllocation.enabled` setting. When enabled, it is assumed that the [External Shuffle Service](#) is also used (it is not by default as controlled by `spark.shuffle.service.enabled` property).

[ExecutorAllocationManager](#) is responsible for dynamic allocation of executors. With [dynamic allocation enabled](#), it is [started when `SparkContext` is initialized](#).

Dynamic allocation reports the current state using [ExecutorAllocationManager metric source](#).

Dynamic Allocation comes with the policy of scaling executors up and down as follows:

1. **Scale Up Policy** requests new executors when there are pending tasks and increases the number of executors exponentially since executors start slow and Spark application may need slightly more.
2. **Scale Down Policy** removes executors that have been idle for `spark.dynamicAllocation.executorIdleTimeout` seconds.

Dynamic allocation is available for all the currently-supported [cluster managers](#), i.e. Spark Standalone, Hadoop YARN and Apache Mesos.

Tip

Read about [Dynamic Allocation on Hadoop YARN](#).

Tip

Review the excellent slide deck [Dynamic Allocation in Spark](#) from Databricks.

Is Dynamic Allocation Enabled?

— `Utils.isDynamicAllocationEnabled` Method

```
isDynamicAllocationEnabled(conf: SparkConf): Boolean
```

`isDynamicAllocationEnabled` returns `true` if all the following conditions hold:

1. `spark.dynamicAllocation.enabled` is enabled (i.e. `true`)
2. Spark on cluster is used (i.e. `spark.master` is non-`local`)
3. `spark.dynamicAllocation.testing` is enabled (i.e. `true`)

Otherwise, `isDynamicAllocationEnabled` 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).
Note	<code>isDynamicAllocationEnabled</code> is used when Spark calculates the initial number of executors for coarse-grained scheduler backends for YARN , Spark Standalone , and Mesos . It is also used for Spark Streaming .
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>log4j.logger.org.apache.spark.util.Utils=WARN</pre> <p>Refer to Logging.</p>

Programmable Dynamic Allocation

`SparkContext` offers a [developer API](#) to scale executors up or down.

Getting Initial Number of Executors for Dynamic Allocation — `Utils.getDynamicAllocationInitialExecutors` Method

```
getDynamicAllocationInitialExecutors(conf: SparkConf): Int
```

`getDynamicAllocationInitialExecutors` first makes sure that `spark.dynamicAllocation.initialExecutors` is equal or greater than `spark.dynamicAllocation.minExecutors`.

Note	<code>spark.dynamicAllocation.initialExecutors</code> falls back to <code>spark.dynamicAllocation.minExecutors</code> if not set. Why to print the <code>WARN</code> message to the logs?
------	---

If not, you should see the following `WARN` message in the logs:

```
spark.dynamicAllocation.initialExecutors less than  
spark.dynamicAllocation.minExecutors is invalid, ignoring its  
setting, please update your configs.
```

`getDynamicAllocationInitialExecutors` makes sure that `spark.executor.instances` is greater than `spark.dynamicAllocation.minExecutors`.

Note

Both `spark.executor.instances` and `spark.dynamicAllocation.minExecutors` fall back to `0` when no defined explicitly.

If not, you should see the following WARN message in the logs:

```
spark.executor.instances less than  
spark.dynamicAllocation.minExecutors is invalid, ignoring its  
setting, please update your configs.
```

`getDynamicAllocationInitialExecutors` sets the initial number of executors to be the maximum of:

- `spark.dynamicAllocation.minExecutors`
- `spark.dynamicAllocation.initialExecutors`
- `spark.executor.instances`
- `0`

You should see the following INFO message in the logs:

```
Using initial executors = [initialExecutors], max of  
spark.dynamicAllocation.initialExecutors,  
spark.dynamicAllocation.minExecutors and  
spark.executor.instances
```

Note

`getDynamicAllocationInitialExecutors` is used when `ExecutorAllocationManager` sets the initial number of executors and in YARN to set initial target number of executors.

Settings

Table 1. Spark Properties

Spark Property	Default Value
<code>spark.dynamicAllocation.enabled</code>	false
<code>spark.dynamicAllocation.initialExecutors</code>	<code>spark.dynamicAllocation.minExecutors</code>
<code>spark.dynamicAllocation.minExecutors</code>	0
<code>spark.dynamicAllocation.maxExecutors</code>	<code>Integer.MAX_VALUE</code>
<code>spark.dynamicAllocation.schedulerBacklogTimeout</code>	1s
<code>spark.dynamicAllocation.sustainedSchedulerBacklogTimeout</code>	<code>spark.dynamicAllocation.schedulerBacklogTimeout</code>
<code>spark.dynamicAllocation.executorIdleTimeout</code>	60s
<code>spark.dynamicAllocation.cachedExecutorIdleTimeout</code>	<code>Integer.MAX_VALUE</code>
<code>spark.dynamicAllocation.testing</code>	

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).

`ExecutorAllocationManager` is created when `SparkContext` is created and dynamic allocation of executors is enabled.

Note	SparkContext expects that SchedulerBackend follows the <code>ExecutorAllocationClient contract</code> when dynamic allocation of executors is enabled.
------	--

Table 1. `FIXME`'s Internal Properties

Name	Initial Value	Description
<code>executorAllocationManagerSource</code>	<code>ExecutorAllocationManagerSource</code>	<code>FIXME</code>

Table 2. ExecutorAllocationManager's Internal Registries and Counters

Name	Description
executorsPendingToRemove	Internal cache with... FIXME Used when... FIXME
removeTimes	Internal cache with... FIXME Used when... FIXME
executorIds	Internal cache with... FIXME Used when... FIXME
initialNumExecutors	FIXME
numExecutorsTarget	FIXME
numExecutorsToAdd	FIXME
initializing	Flag whether... FIXME Starts enabled (i.e. <code>true</code>).

Tip	Enable <code>INFO</code> logging level for <code>org.apache.spark.ExecutorAllocationManager</code> logger to see what happens inside. Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.ExecutorAllocationManager=INFO</code>
	Refer to Logging .

addExecutors Method

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

removeExecutor Method

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

maxNumExecutorsNeeded Method

Caution

FIXME

Starting ExecutorAllocationManager — start Method

```
start(): Unit
```

`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

`100` milliseconds for the period between successive `scheduling` 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 Method

```
updateAndSyncNumExecutorsTarget(now: Long): Int
```

`updateAndSyncNumExecutorsTarget` ...[FIXME](#)

If `ExecutorAllocationManager` is `initializing` it returns `0`.

Resetting ExecutorAllocationManager — 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.

Stopping ExecutorAllocationManager — stop Method

```
stop(): Unit
```

`stop` shuts down `spark-dynamic-executor-allocation allocation executor`.

Note

`stop` waits 10 seconds for the termination to be complete.

Creating ExecutorAllocationManager Instance

`ExecutorAllocationManager` takes the following when created:

- `ExecutorAllocationClient`
- `LiveListenerBus`
- `SparkConf`

`ExecutorAllocationManager` initializes the `internal registries and counters`.

Validating Configuration of Dynamic Allocation — validateSettings Internal Method

```
validateSettings(): Unit
```

`validateSettings` makes sure that the `settings for dynamic allocation` are correct.

`validateSettings` validates the following and throws a `SparkException` if not set correctly.

1. `spark.dynamicAllocation.minExecutors` must be positive
2. `spark.dynamicAllocation.maxExecutors` must be `> 0` or greater
3. `spark.dynamicAllocation.minExecutors` must be less than or equal to `spark.dynamicAllocation.maxExecutors`
4. `spark.dynamicAllocation.executorIdleTimeout` must be greater than `0`
5. `spark.shuffle.service.enabled` must be enabled.
6. The number of tasks per core, i.e. `spark.executor.cores` divided by `spark.task.cpus`, is not zero.

Note	<code>validateSettings</code> is used when <code>ExecutorAllocationManager</code> is created.
------	---

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.

ExecutorAllocationClient Contract

```
trait ExecutorAllocationClient {
  def getExecutorIds(): Seq[String]
  def requestTotalExecutors(numExecutors: Int, localityAwareTasks: Int, hostToLocalTasksCount: Map[String, Int]): Boolean
  def requestExecutors(numAdditionalExecutors: Int): Boolean
  def killExecutor(executorId: String): Boolean
  def killExecutors(executorIds: Seq[String]): Seq[String]
  def killExecutorsOnHost(host: String): Boolean
}
```

Note

`ExecutorAllocationClient` is a `private[spark]` contract.

Table 1. ExecutorAllocationClient Contract

Method	Description
<code>getExecutorIds</code>	<p>Finds identifiers of the executors in use.</p> <p>Used when <code>SparkContext</code> calculates the executors in use and also when <code>Spark Streaming</code> manages executors.</p>
<code>requestTotalExecutors</code>	<p>Updates the cluster manager with the exact number of executors desired. It returns whether the request has been acknowledged by the cluster manager (<code>true</code>) or not (<code>false</code>).</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>SparkContext</code> requests executors (for coarse-grained scheduler backends only). • <code>ExecutorAllocationManager</code> starts, does <code>updateAndSyncNumExecutorsTarget</code>, and <code>addExecutors</code>. • <code>Streaming</code> <code>ExecutorAllocationManager</code> requests executors. • <code>YarnSchedulerBackend</code> stops.

<code>requestExecutors</code>	<p>Requests additional executors from a cluster manager and returns whether the request has been acknowledged by the cluster manager (<code>true</code>) or not (<code>false</code>).</p> <p>Used when SparkContext requests additional executors (for coarse-grained scheduler backends only).</p>
<code>killExecutor</code>	<p>Requests 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 (<code>true</code>) or not (<code>false</code>).</p> <p>The default implementation simply calls killExecutors (with a single-element collection of executors to kill).</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>ExecutorAllocationManager</code> removes an executor. • <code>SparkContext</code> is requested to kill executors. • <code>Streaming ExecutorAllocationManager</code> is requested to kill executors.
<code>killExecutors</code>	<p>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 (<code>true</code>) or not (<code>false</code>).</p> <p><i>Interestingly, it is only used for killExecutor.</i></p>
<code>killExecutorsOnHost</code>	Used exclusively when <code>BlacklistTracker</code> kills blacklisted executors.

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 ExecutorAllocationManager 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 Metrics 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.

HTTP File Server

It is started on a [driver](#).

Caution	FIXME Review HttpFileServer
---------	---

Settings

- `spark.filesServer.port` (default: `0`) - the port of a file server
- `spark.filesServer.uri` (Spark internal) - the URI of a file server

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	FIXME 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	FIXME
---------	-----------------------

Uncaching Query (`uncacheQuery` method)

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

OutputCommitCoordinator

`OutputCommitCoordinator` service is authority that coordinates [result commits](#) by means of **commit locks** (using the internal [authorizedCommittersByStage](#) registry).

Result commits are the outputs of running tasks (and a running task is described by a task attempt for a partition in a stage).

Tip

A partition (of a stage) is **unlocked** when it is marked as `-1` in [authorizedCommittersByStage](#) internal registry.

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.

Authorized committers are task attempts (per partition and stage) that can...[FIXME](#)

Table 1. `outputCommitCoordinator` Internal Registries and Counters

Name	Description
<code>authorizedCommittersByStage</code>	Tracks commit locks for task attempts for a partition in a stage. Used in taskCompleted to authorize task completions to... FIXME

Tip

Enable `INFO` or `DEBUG` logging level for `org.apache.spark.scheduler.OutputCommitCoordinator` logger to see what happens in `OutputCommitCoordinator`.

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

```
log4j.logger.org.apache.spark.scheduler.OutputCommitCoordinator=DEBUG
```

Refer to [Logging](#).

stop Method

Caution

FIXME

stageStart Method

Caution

FIXME

taskCompleted Method

```
taskCompleted(  
    stage: StageId,  
    partition: PartitionId,  
    attemptNumber: TaskAttemptNumber,  
    reason: TaskEndReason): Unit
```

`taskCompleted` marks the `partition` (in the `stage`) completed (and hence a result committed), but only when the `attemptNumber` is amongst [authorized committers](#) per stage (for the `partition`).

Internally, `taskCompleted` first finds [authorized committers](#) for the `stage`.

For task completions with no stage registered in [authorizedCommittersByStage internal registry](#), you should see the following DEBUG message in the logs and `taskCompleted` simply exits.

```
DEBUG OutputCommitCoordinator: Ignoring task completion for completed stage
```

For the `reason` being `Success` `taskCompleted` does nothing and exits.

For the `reason` being `TaskCommitDenied`, you should see the following INFO message in the logs and `taskCompleted` exits.

```
INFO OutputCommitCoordinator: Task was denied committing, stage: [stage], partition: [partition], attempt: [attemptNumber]
```

Note

For no `stage` registered or `reason` being `Success` or `TaskCommitDenied`, `taskCompleted` does nothing (important).

For task completion reasons other than `Success` or `TaskCommitDenied` and `attemptNumber` amongst [authorized committers](#), `taskCompleted` marks `partition` unlocked.

Note

A task attempt can never be -1.

When the lock for `partition` is cleared, You should see the following DEBUG message in the logs:

```
DEBUG OutputCommitCoordinator: Authorized committer (attemptNumber=[attemptNumber], stage=[stage], partition=[partition]) failed; clearing lock
```

Note

`taskCompleted` is executed only when [DAGScheduler](#) informs that a task has completed.

RpcEnv — RPC Environment

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**.

You can create a RPC Environment using [RpcEnv.create](#) factory methods.

The only implementation of RPC Environment is [Netty-based implementation](#).

A [RpcEndpoint](#) defines 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](#).

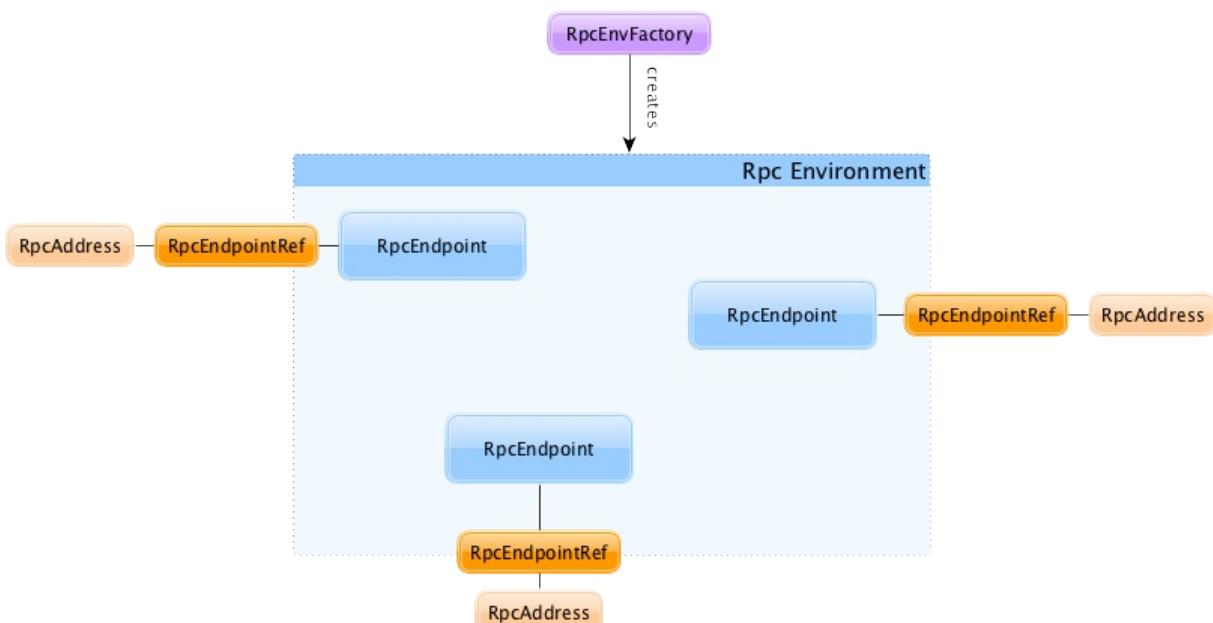


Figure 1. RpcEnvironment with RpcEndpoints and RpcEndpointRefs

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.

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.

Creating RpcEndpointRef For URI

— `asyncSetupEndpointRefByURI` Method

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

Creating RpcEndpointRef For URI

— `setupEndpointRefByURI` Method

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

shutdown Method

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

Registering RPC Endpoint — `setupEndpoint` Method

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

awaitTermination Method

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

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 .

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]`.

Stopping RpcEndpointRef — stop Method

```
stop(endpoint: RpcEndpointRef): Unit
```

Caution

[FIXME](#)

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.

RpcEnvConfig

`RpcEnvConfig` is a placeholder for an instance of `SparkConf`, the name of the RPC Environment, host and port, a security manager, and `clientMode`.

Creating RpcEnv — `create` Factory Methods

```
create(
    name: String,
    host: String,
    port: Int,
    conf: SparkConf,
    securityManager: SecurityManager,
    clientMode: Boolean = false): RpcEnv (1)

create(
    name: String,
    bindAddress: String,
    advertiseAddress: String,
    port: Int,
    conf: SparkConf,
    securityManager: SecurityManager,
    clientMode: Boolean): RpcEnv
```

1. The 6-argument `create` (with `clientMode` disabled) simply passes the input arguments on to the second `create` making `bindAddress` and `advertiseAddress` the same.

`create` creates a [RpcEnvConfig](#) (with the input arguments) and creates a [NettyRpcEnv](#).

	<p>Copied (almost verbatim) from SPARK-10997 Netty-based RPC env should support a "client-only" mode and the commit:</p> <p>"Client mode" means the RPC env will not listen for incoming connections.</p> <p>This allows certain processes in the Spark stack (such as Executors or the YARN client-mode AM) to act as pure clients when using the netty-based RPC backend, reducing the number of sockets Spark apps need to use and also the number of open ports.</p>
Note	<p>The AM connects to the driver in "client mode", and that connection is used for all driver—AM communication, and so the AM is properly notified when the connection goes down.</p> <p>In "general", non-YARN case, <code>clientMode</code> flag is therefore enabled for executors and disabled for the driver.</p> <p>In Spark on YARN in client deploy mode, <code>clientMode</code> flag is however enabled explicitly when Spark on YARN's ApplicationMaster creates the <code>sparkYarnAM</code> RPC Environment.</p>

	<p><code>create</code> is used when:</p> <ol style="list-style-type: none"> 1. SparkEnv creates a RpcEnv (for the driver and executors). 2. Spark on YARN's ApplicationMaster creates the <code>sparkYarnAM</code> RPC Environment (with <code>clientMode</code> enabled). 3. CoarseGrainedExecutorBackend creates the temporary <code>driverPropsFetcher</code> RPC Environment (to fetch the current Spark properties from the driver). 4. org.apache.spark.deploy.Client standalone application creates the <code>driverClient</code> RPC Environment. 5. Spark Standalone's master creates the <code>sparkMaster</code> RPC Environment. 6. Spark Standalone's worker creates the <code>sparkWorker</code> RPC Environment. 7. Spark Standalone's <code>DriverWrapper</code> creates the <code>Driver</code> RPC Environment.
Note	

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
spark.rpc.lookupTimeout	120s	Timeout to use for RPC remote endpoint lookup. Refer to Endpoint Lookup Timeout
spark.rpc.numRetries	3	Number of attempts to send a message to and receive a response from a remote endpoint.
spark.rpc.retry.wait	3s	Time to wait between retries.
spark.rpc.askTimeout	120s	Timeout for RPC ask calls. Refer to Ask Operation Timeout .
spark.network.timeout	120s	Network timeout to use for RPC remote endpoint lookup. Fallback for spark.rpc.askTimeout .

RpcEndpoint

`RpcEndpoint` defines a RPC endpoint that processes **messages** using **callbacks**, i.e. **functions** to execute when a message arrives.

`RpcEndpoint` lives in [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 ThreadSafeRpcEndpoint .
-----	---

`onError` method is called for any exception thrown.

onStart Method

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

stop Method

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

RpcEndpointRef — Reference to RPC Endpoint

`RpcEndpointRef` is a reference to a `RpcEndpoint` in a `RpcEnv`.

`RpcEndpointRef` is a 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](#).

send Method

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

askWithRetry Method

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

RpcEnvFactory

`RpcEnvFactory` is the [contract](#) to create a [RPC Environment](#).

Note	As of this commit in Spark 2, the one and only <code>RpcEnvFactory</code> is Netty-based <code>NettyRpcEnvFactory</code> .
------	--

RpcEnvFactory Contract

```
trait RpcEnvFactory {  
    def create(config: RpcEnvConfig): RpcEnv  
}
```

Note	<code>RpcEnvFactory</code> is a <code>private[spark]</code> contract.
------	---

Table 1. `RpcEnvFactory` Contract

Method	Description
<code>create</code>	Used when <code>RpcEnv</code> creates a RPC Environment .

Netty-based RpcEnv

Tip

Read up [RpcEnv — RPC Environment](#) on 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 `:`.

Creating NettyRpcEnv — `create` Method

Caution

[FIXME](#)

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 <code>NioEventLoopGroup</code> .
---------	---

Caution	FIXME Where are <code>SO_BACKLOG</code> , <code>SO_RCVBUF</code> , <code>SO_SNDBUF</code> 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=0x7103 waiting on condition [0x000000011f78b000]
```

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](#).

Creating TransportConf from SparkConf

— `fromSparkConf` Method

```
fromSparkConf(_conf: SparkConf, module: String, numUsableCores: Int = 0): TransportConf
```

Note

`fromSparkConf` belongs to `SparkTransportConf` object.

`fromSparkConf` creates a `TransportConf` for `module` from the given `SparkConf`.

Internally, `fromSparkConf` calculates the default number of threads for both the Netty client and server thread pools.

`fromSparkConf` uses `spark.[module].io.serverThreads` and `spark.[module].io.clientThreads` if specified for the number of threads to use. If not defined, `fromSparkConf` sets them to the default number of threads calculated earlier.

Calculating Default Number of Threads (8 Maximum)

— `defaultNumThreads` Internal Method

```
defaultNumThreads(numUsableCores: Int): Int
```

Note

`defaultNumThreads` belongs to `SparkTransportConf` object.

`defaultNumThreads` calculates the default number of threads for both the Netty client and server thread pools that is 8 maximum or `numUsableCores` is smaller. If `numUsableCores` is not specified, `defaultNumThreads` uses the number of processors available to the Java virtual machine.

Note

8 is the maximum number of threads for Netty and is not configurable.

Note

`defaultNumThreads` uses Java's Runtime for the number of processors in JVM.

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. $2 \times \# \text{cores}$)—the number of threads used in the server thread pool.
- `io.clientThreads` (default: `0` i.e. $2 \times \# \text{cores}$)—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`).

Spark Streaming — Streaming RDDs

Spark Streaming is the incremental **micro-batching stream processing framework** for Spark.

Tip

You can find more information about Spark Streaming in my separate book in the [notebook repository at GitBook](#).

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 is composed of the driver and executors that can run locally (on a single JVM) or using cluster resources (like CPU, RAM and disk that are managed by a cluster manager).

Note

You can specify where to run the driver using the [deploy mode](#) (using `--deploy-mode` option of `spark-submit` or `spark.submit.deployMode` Spark property).

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` executors (threads), `cores` CPUs and `memory` locally (aka [Spark local-cluster](#))
- `spark://host:port,host1:port1,...` for connecting to [Spark Standalone cluster\(s\)](#)
- `mesos://` for [Spark on Mesos cluster](#)
- `yarn` for [Spark on YARN](#)

You can specify the master URL of a Spark application as follows:

1. `spark-submit`'s `--master` command-line option,
2. `spark.master` Spark property,
3. When creating a `SparkContext` (using `setMaster` method),
4. When creating a `SparkSession` (using `master` method of the builder interface).

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**, **LocalSchedulerBackend**, 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.

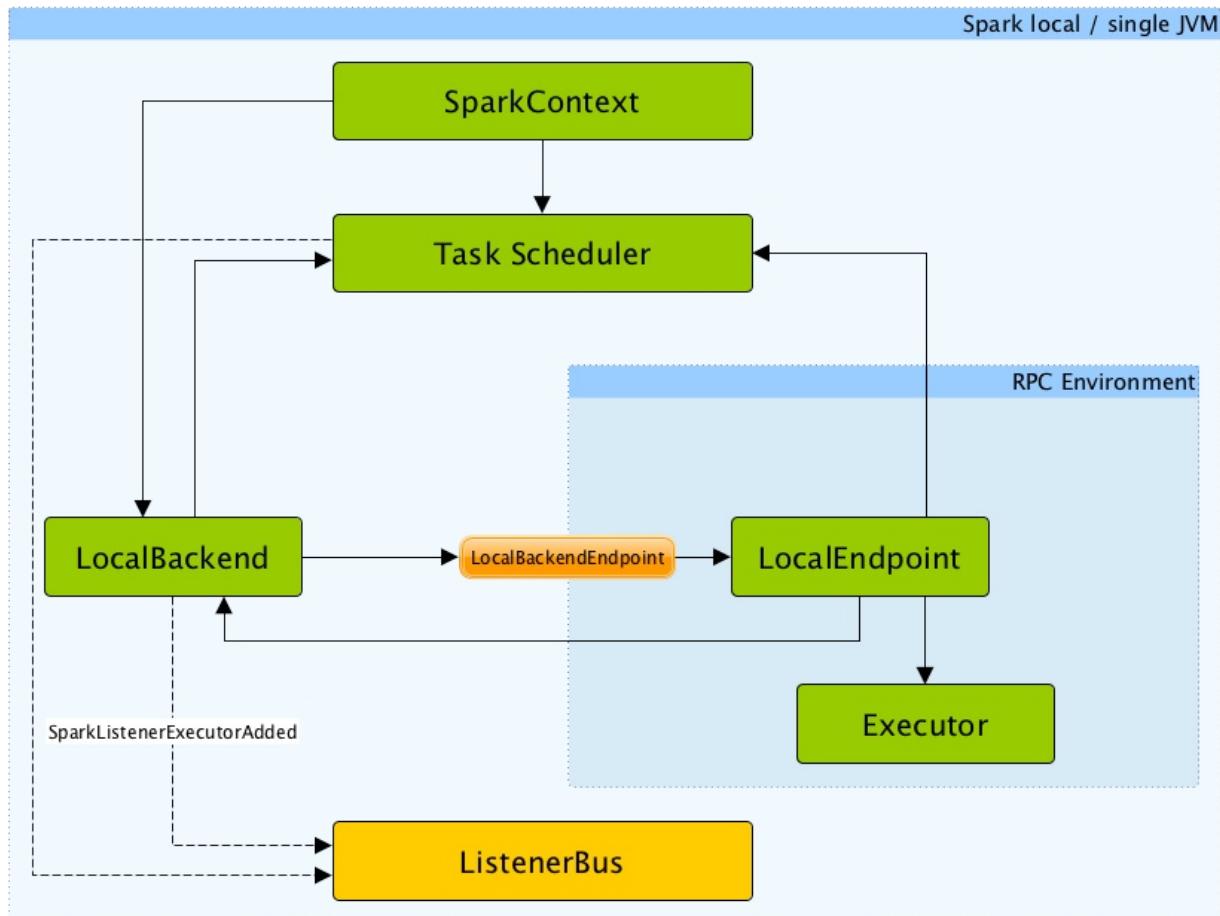


Figure 1. Architecture of Spark local

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 [LocalSchedulerBackend](#) 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, maxFailures]` (called **local-with-retries**) with `N` being `*` or the number of threads to use (as explained above) and `maxFailures` being the value of [spark.task.maxFailures](#).

Task Submission a.k.a. `reviveOffers`

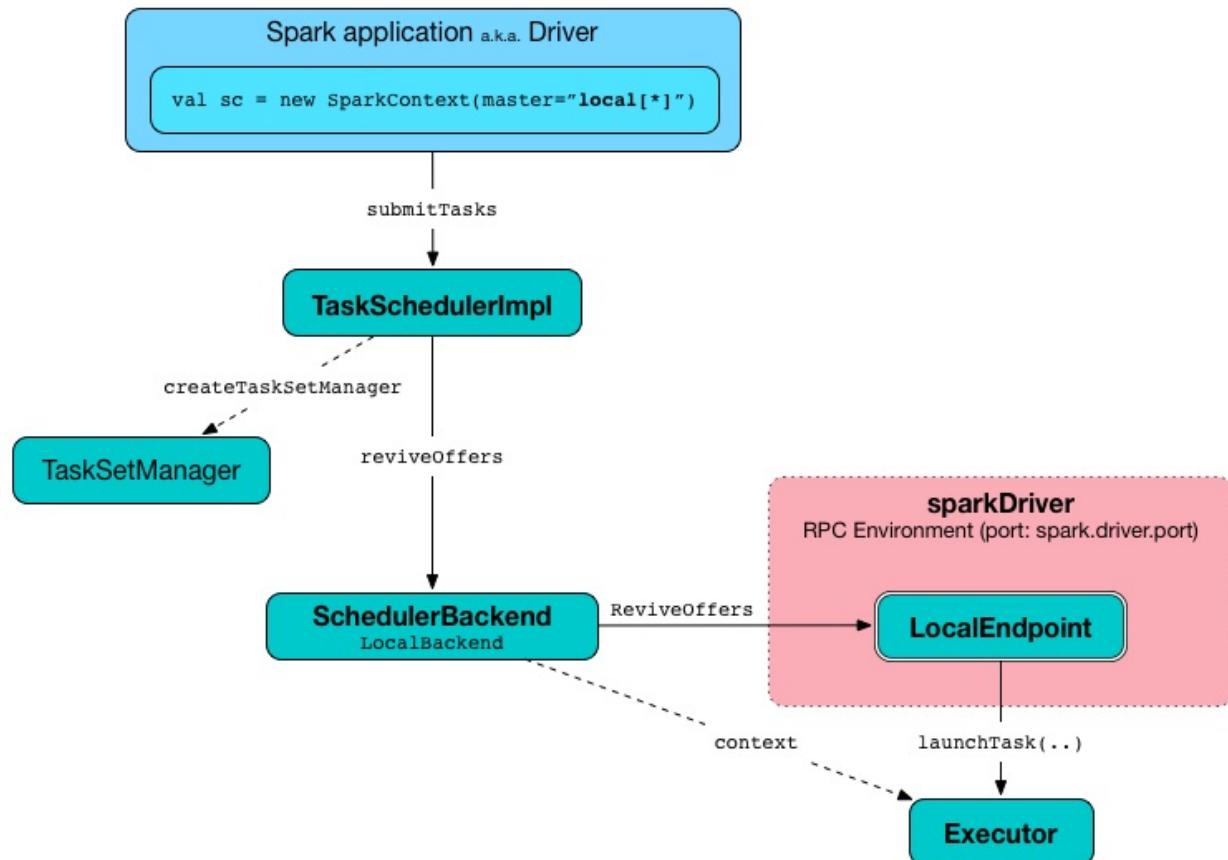


Figure 2. TaskSchedulerImpl.submitTasks in local mode

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.

LocalSchedulerBackend

`LocalSchedulerBackend` is a [scheduler backend](#) and a [ExecutorBackend](#) for [Spark local run mode](#).

`LocalSchedulerBackend` 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](#).

Note

[WorkerOffer](#) represents a resource offer with CPU cores available on an executor.

When an executor sends task status updates (using `ExecutorBackend.statusUpdate`), they are passed along as [StatusUpdate](#) to [LocalEndpoint](#).

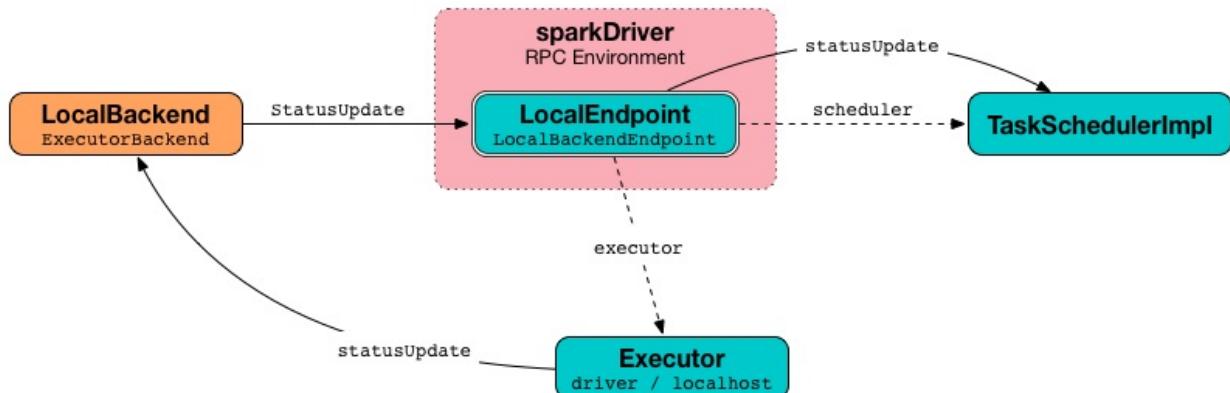


Figure 1. Task status updates flow in local mode

When `LocalSchedulerBackend` starts up, it registers a new [RpcEndpoint](#) called **LocalSchedulerBackendEndpoint** 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` property.

LocalEndpoint

`LocalEndpoint` is the communication channel between [Task Scheduler](#) and [LocalSchedulerBackend](#). It is a (thread-safe) [RpcEndpoint](#) 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
```

reviveOffers Method

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

Creating LocalEndpoint Instance

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

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.

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">• Spark execution in cluster - Diagram of the communication between driver, cluster manager, workers with executors and tasks. See Cluster Mode Overview.◦ Show Spark's driver with the main code in Scala in the box◦ Nodes with executors with tasks• Hosts drivers• Manages a cluster |
|---------|--|

The workers are in charge of communicating the cluster manager the availability of their resources.

Communication with a driver is through a RPC interface (at the moment Akka), except [Mesos in fine-grained mode](#).

Executors remain alive after jobs are finished for future ones. This allows for better data utilization as intermediate data is cached in memory.

Spark reuses resources in a cluster for:

- efficient data sharing
- fine-grained partitioning
- low-latency scheduling

Reusing also means the the resources can be hold onto for a long time.

Spark reuses long-running executors for speed (contrary to Hadoop MapReduce using short-lived containers for each task).

Spark Application Submission to Cluster

When you submit a Spark application to the cluster this is what happens (see the answers to [the answer to What are workers, executors, cores in Spark Standalone cluster?](#) on StackOverflow):

- The Spark driver is launched to invoke the `main` method of the Spark application.
- The driver asks the cluster manager for resources to run the application, i.e. to launch executors that run tasks.
- The cluster manager launches executors.
- The driver runs the Spark application and sends tasks to the executors.
- Executors run the tasks and save the results.
- Right after `SparkContext.stop()` is executed from the driver or the `main` method has exited all the executors are terminated and the cluster resources are released by the cluster manager.

Note	"There's not a good reason to run more than one worker per machine." by Sean Owen in What is the relationship between workers, worker instances, and executors?
------	--

Caution

One executor per node may not always be ideal, esp. when your nodes have lots of RAM. On the other hand, using fewer executors has benefits like more efficient broadcasts.

Two modes of launching executors

Warning

[Review core/src/main/scala/org/apache/spark/deploy/master/Master.scala](#)

Others

Spark application can be split into the part written in Scala, Java, and Python with the cluster itself in which the application is going to run.

Spark application runs on a cluster with the help of **cluster manager**.

A Spark application consists of a single driver process and a set of executor processes scattered across nodes on the cluster.

Both the driver and the executors usually run as long as the application. The concept of **dynamic resource allocation** has changed it.

Caution

[FIXME Figure](#)

A node is a machine, and there's not a good reason to run more than one worker per machine. So two worker nodes typically means two machines, each a Spark worker.

Workers hold many executors for many applications. One application has executors on many workers.

Spark 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 **2.0.0**, `yarn master URL` is the only proper master URL and you can use `--deploy-mode` to choose between `client` (default) or `cluster` modes.

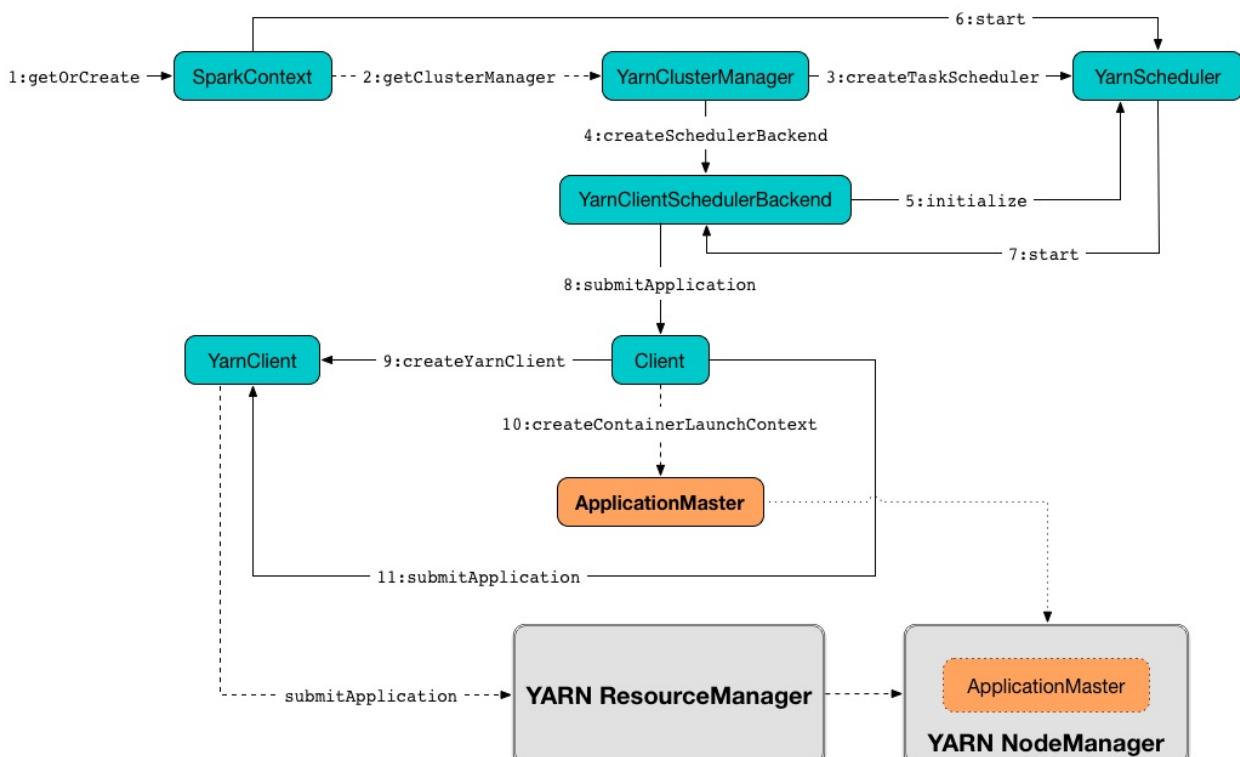


Figure 1. Submitting Spark Application to YARN Cluster (aka Creating SparkContext with `yarn Master URL` and client Deploy Mode)

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 `Spark driver` 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	FIXME
---------	-----------------------

serviceStop

```
void serviceStop()
```

`serviceStop` is a part of YARN's `AuxiliaryService` contract and is called when...[FIXME](#)

Caution	FIXME The contract
---------	------------------------------------

Caution	FIXME What are <code>shuffleServer</code> and <code>blockHandler</code> ? What's their lifecycle?
---------	---

When an exception occurs, you should see the following ERROR message in the logs:

```
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	FIXME 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	FIXME 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	FIXME 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 <code>ExternalShuffleBlockHandler#applicationRemoved</code> 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 <code>yarn classpath</code> 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` standalone application.

`ExecutorRunnable` is created when `YarnAllocator` launches Spark executors in allocated YARN containers (and for debugging purposes when `ApplicationMaster` requests cluster resources for executors).

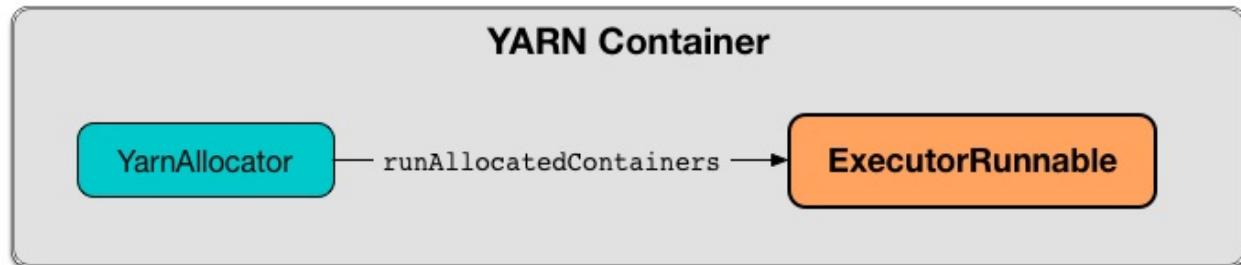


Figure 1. ExecutorRunnable and YarnAllocator in YARN Resource Container
If external shuffle service is used, it is set in the `ContainerLaunchContext` context as a service under the name of `spark_shuffle`.

Table 1. ExecutorRunnable's Internal Properties

Name	Description
<code>rpc</code>	<code>YarnRPC</code> for...FIXME
<code>nmClient</code>	<code>NMClient</code> for...FIXME

Note	Despite the name <code>ExecutorRunnable</code> is not a <code>java.lang.Runnable</code> anymore after SPARK-12447.
------	--

Tip	<p>Enable <code>INFO</code> or <code>DEBUG</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=DEBUG</pre> <p>Refer to Logging.</p>
-----	--

Creating ExecutorRunnable Instance

`ExecutorRunnable` takes the following when created:

1. YARN Container to run a Spark executor in
2. `YarnConfiguration`
3. `sparkConf` — `SparkConf`
4. `masterAddress`
5. `executorId`
6. `hostname` of the YARN container
7. `executorMemory`
8. `executorCores`
9. `appId`
10. `SecurityManager`
11. `localResources` — `Map[String, LocalResource]`

`ExecutorRunnable` initializes the internal registries and counters.

Note	<code>executorMemory</code> and <code>executorCores</code> input arguments are from <code>YarnAllocator</code> but really are <code>spark.executor.memory</code> and <code>spark.executor.cores</code> properties.
------	--

Note	Most of the input parameters are exactly as <code>YarnAllocator</code> was created with.
------	--

Building Command to Run CoarseGrainedExecutorBackend in YARN Container — `prepareCommand` Internal Method

```
prepareCommand(
  masterAddress: String,
  slaveId: String,
  hostname: String,
  executorMemory: Int,
  executorCores: Int,
  appId: String): List[String]
```

`prepareCommand` prepares 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.

`prepareCommand` builds the command that will be executed in a YARN container.

Note	JVM options are defined using <code>-Dkey=value</code> format.
------	--

`prepareCommand` builds `-Xmx` JVM option using `executorMemory` (in MB).

Note	<code>prepareCommand</code> uses <code>executorMemory</code> that is given when <code>ExecutorRunnable</code> is created.
------	---

`prepareCommand` adds the optional `spark.executor.extraJavaOptions` property to the JVM options (if defined).

`prepareCommand` adds the optional `SPARK_JAVA_OPTS` environment variable to the JVM options (if defined).

`prepareCommand` adds the optional `spark.executor.extraLibraryPath` to the library path (changing the path to be YARN NodeManager-aware).

`prepareCommand` adds `-Djava.io.tmpdir=<LOG_DIR>./tmp` to the JVM options.

`prepareCommand` adds all the Spark properties for executors to the JVM options.

Note	<code>prepareCommand</code> uses <code>SparkConf</code> that is given when <code>ExecutorRunnable</code> is created.
------	--

`prepareCommand` adds `-Dspark.yarn.app.container.log.dir=<LOG_DIR>` to the JVM options.

`prepareCommand` adds `-XX:MaxPermSize=256m` unless already defined or IBM JVM or Java 8 are used.

`prepareCommand` reads the list of URLs representing the user classpath and adds `--user-class-path` and `file:[path]` for every entry.

`prepareCommand` adds `-xx:onOutOfMemoryError` to the JVM options unless already defined.

In the end, `prepareCommand` combines the parts together to build the entire command with the following (in order):

1. Extra library path
2. `JAVA_HOME/bin/java`
3. `-server`
4. JVM options
5. `org.apache.spark.executor.CoarseGrainedExecutorBackend`
6. `--driver-url` followed by `masterAddress`
7. `--executor-id` followed by `executorId`

8. --hostname followed by `hostname`
9. --cores followed by `executorCores`
10. --app-id followed by `appId`
11. --user-class-path with the arguments
12. 1><LOG_DIR>/stdout
13. 2><LOG_DIR>/stderr

Note

`prepareCommand` uses the arguments for `--driver-url`, `--executor-id`, `--hostname`, `--cores` and `--app-id` as given when `ExecutorRunnable` is created.

Note

You can see the result of `prepareCommand` as `command` in the INFO message in the logs when `ApplicationMaster` registers itself with YARN ResourceManager (to print it out once and avoid flooding the logs when starting Spark executors).

Note

`prepareCommand` is used when `ExecutorRunnable` starts `CoarseGrainedExecutorBackend` in a YARN resource container and (only for debugging purposes) when `ExecutorRunnable` builds launch context diagnostic information (to print it out as an INFO message to the logs).

Collecting Environment Variables for CoarseGrainedExecutorBackend Containers — `prepareEnvironment` Internal Method

```
prepareEnvironment(): HashMap[String, String]
```

`prepareEnvironment` collects environment-related entries.

`prepareEnvironment` populates class path (passing in `YarnConfiguration`, `SparkConf`, and `spark.executor.extraClassPath` property)

Caution

`FIXME` How does `populateClasspath` use the input `env` ?

`prepareEnvironment` collects the executor environment variables set on the current `SparkConf`, i.e. the Spark properties with the prefix `spark.executorEnv.`, and `YarnSparkHadoopUtil.addPathToEnvironment(env, key, value)`.

Note

`SPARK_YARN_USER_ENV` is deprecated.

`prepareEnvironment` reads YARN's `yarn.http.policy` property (with `YarnConfiguration.YARN_HTTP_POLICY_DEFAULT`) to choose a secure HTTPS scheme for container logs when `HTTPS_ONLY`.

With the input `container` defined and `SPARK_USER` environment variable available, `prepareEnvironment` registers `SPARK_LOG_URL_STDERR` and `SPARK_LOG_URL_STDOUT` environment entries with `stderr?start=-4096` and `stdout?start=-4096` added to `[httpScheme][address]/node/containerlogs/[containerId]/[user]`, respectively.

In the end, `prepareEnvironment` collects all the System environment variables with `SPARK` prefix.

Note

`prepareEnvironment` is used when `ExecutorRunnable` starts `CoarseGrainedExecutorBackend` in a container and (for debugging purposes) builds launch context diagnostic information (to print it out as an INFO message to the logs).

Starting ExecutorRunnable (with CoarseGrainedExecutorBackend) — `run` Method

```
run(): Unit
```

When called, you should see the following DEBUG message in the logs:

```
DEBUG ExecutorRunnable: Starting Executor Container
```

`run` creates a YARN `NMClient` (to communicate with YARN NodeManager service), inits it with `YarnConfiguration` and starts it.

Note

`run` uses `YarnConfiguration` that was given when `ExecutorRunnable` was created.

In the end, `run` starts `CoarseGrainedExecutorBackend` in the YARN container.

Note

`run` is used exclusively when `YarnAllocator` schedules `ExecutorRunnables` in allocated YARN resource containers.

Starting YARN Resource Container — `startContainer` Method

```
startContainer(): java.util.Map[String, ByteBuffer]
```

`startContainer` uses YARN NodeManager's `NMClient` API to start a `CoarseGrainedExecutorBackend` in a YARN container.

`startContainer` follows the design pattern to request YARN NodeManager to start a container:

```
val ctx = Records.newRecord(classOf[ContainerLaunchContext]).asInstanceOf[ContainerLaunchContext]
ctx.setLocalResources(...)
ctx.setEnvironment(...)
ctx.setTokens(...)
ctx.setCommands(...)
ctx.setApplicationACLs(...)
ctx.setServiceData(...)
nmClient.startContainer(container, ctx)
```

Tip

`startContainer` creates a YARN [ContainerLaunchContext](#).

Note

YARN [ContainerLaunchContext](#) represents all of the information for the YARN NodeManager to launch a resource container.

`startContainer` then sets [local resources](#) and [environment](#) to the `ContainerLaunchContext`.

Note

`startContainer` uses [local resources](#) given when `ExecutorRunnable` was created.

`startContainer` sets security tokens to the `ContainerLaunchContext` (using Hadoop's `UserGroupInformation` and the current user's credentials).

`startContainer` sets the [command](#) (to launch `CoarseGrainedExecutorBackend`) to the `ContainerLaunchContext`.

`startContainer` sets the [application ACLs](#) to the `ContainerLaunchContext`.

If `spark.shuffle.service.enabled` property is enabled, `startContainer` registers the `ContainerLaunchContext` with the YARN shuffle service started on the YARN NodeManager under `spark_shuffle` service name.

In the end, `startContainer` requests the YARN NodeManager to start the YARN container with the `ContainerLaunchContext` context.

Note

`startContainer` uses `nmClient` internal reference to send the request with the YARN resource container given when `ExecutorRunnable` was created.

If any exception happens, `startContainer` reports `SparkException`.

```
Exception while starting container [containerId] on host [hostname]
```

Note

`startContainer` is used exclusively when `ExecutorRunnable` is started.

Building Launch Context Diagnostic Information (with Command, Environment and Resources) — `launchContextDebugInfo` Method

```
launchContextDebugInfo(): String
```

`launchContextDebugInfo` prepares the command to launch `CoarseGrainedExecutorBackend` (as `commands value`) and collects environment variables for `coarseGrainedExecutorBackend containers` (as `env value`).

`launchContextDebugInfo` returns the launch context debug info.

```
=====
YARN executor launch context:
env:
  [key] -> [value]
  ...

command:
  [commands]

resources:
  [key] -> [value]
=====
```

Note `resources` entry is the input `localResources` given when `ExecutorRunnable was created`.

Note `launchContextDebugInfo` is used when `ApplicationMaster` registers itself with `YARN ResourceManager`.

Client

`client` is a handle to a YARN cluster to submit [ApplicationMaster](#) (that represents a Spark application submitted to a YARN cluster).

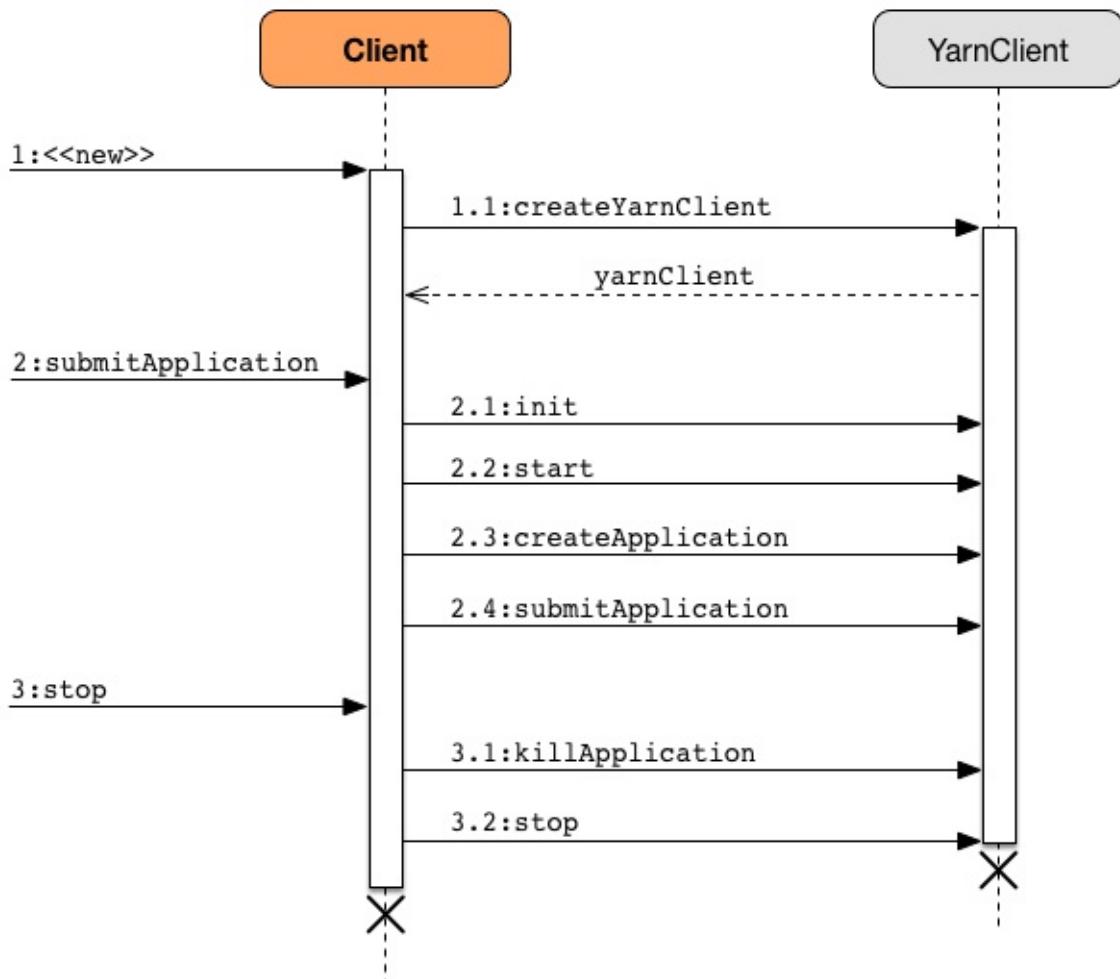


Figure 1. Client and Hadoop's `YarnClient` Interactions

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).

Note

`client` was initially used as a [standalone application](#) to [submit Spark applications](#) to a YARN cluster, but is currently considered obsolete.

Table 1. Client's Internal Properties

Name	Initial Value	Description
executorMemoryOverhead	spark.yarn.executor.memoryOverhead and falls back to 10% of the spark.executor.memory or 384 whatever is larger.	FIXME NOTE: 10% and 384 are constants and cannot be changed.
Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.deploy.yarn.Client</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.Client=DEBUG</pre> <p>Refer to Logging.</p>	

isUserClassPathFirst Method

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

getUserClasspath Method

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

clientArguments

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

Setting Up Environment to Launch ApplicationMaster Container— setupLaunchEnv Method

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

launcherBackend Property

`launcherBackend ...`FIXME

loginFromKeytab Method

Caution

FIXME

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

```
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 Client.run .
------	--

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	FIXME How to configure <code>YarnClient</code> ? What is YARN's <code>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	FIXME Why is this important?
---------	--

`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](#)).

Note

`submitApplication` is used when Client runs or `YarnClientSchedulerBackend` is started.

Creating YARN ApplicationSubmissionContext — `createApplicationSubmissionContext` Method

```
createApplicationSubmissionContext(  
    newApp: YarnClientApplication,  
    containerContext: ContainerLaunchContext): ApplicationSubmissionContext
```

`createApplicationSubmissionContext` creates YARN's [ApplicationSubmissionContext](#).

Note

YARN's `ApplicationSubmissionContext` represents all of the information needed by the [YARN ResourceManager](#) to launch the [ApplicationMaster](#) 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 Resource Capabilities for ApplicationMaster — Memory and Virtual CPU Cores section below
Rolled Log Aggregation for the Spark application	See Rolled Log Aggregation Configuration for Spark Application 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 Resource 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	spark.yarn.am.nodeLabelExpression configuration setting
ResourceRequest of AM container	spark.yarn.am.nodeLabelExpression 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 LogAggregationContext represents all of the information needed by the YARN NodeManager 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	spark.yarn.rolledLog.includePattern configuration setting
Exclude Pattern	spark.yarn.rolledLog.excludePattern configuration setting

Verifying Maximum Memory Capability of YARN Cluster — `verifyClusterResources` Internal 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 YARN ContainerLaunchContext to Launch ApplicationMaster — `createContainerLaunchContext` Internal Method

```
createContainerLaunchContext(newAppResponse: GetNewApplicationResponse): ContainerLaunchContext
```

Note

The input `GetNewApplicationResponse` is Hadoop YARN's `GetNewApplicationResponse`.

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).

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	FIXME prefixEnv ? How is path calculated? ApplicationConstants.LOG_DIR_EXPANSION_VAR ?
---------	---

```
[JAVA_HOME]/bin/java -server [javaOpts] [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	FIXME <code>setApplicationACLs</code> ? Set up security tokens?
---------	---

Note	<code>createContainerLaunchContext</code> is used when <code>Client</code> submits a Spark application to a YARN cluster.
------	---

prepareLocalResources Method

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

```
prepareLocalResources(  
    destDir: Path,  
    pySparkArchives: Seq[String]): HashMap[String, LocalResource]
```

`prepareLocalResources` is...[FIXME](#)

Caution	FIXME Describe <code>credentialManager</code>
---------	---

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 <code>credentials</code> 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 `org.apache.hadoop.fs.FileSystem` to know a list of HDFS-compatible file systems, e.g. [Amazon S3](#) or [Windows Azure](#).

If a keytab is used to log in, ...[FIXME](#)

Caution

[FIXME](#) if (`loginFromKeytab`)

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

[FIXME](#) Describe `case Some(jars)`

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

[FIXME](#) `YarnCommandBuilderUtils.findJarsDir`

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 `spark-submit` using command-line options `--jars`, `--files`, and `--archives`.

Caution

FIXME Describe `distribute`

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` Internal 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` 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 prepareLocalResources .
------	---

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 <code>args</code> is <code>null</code> when preparing environment for <code>ExecutorRunnable</code> and the constructor's <code>args</code> for <code>Client</code> .
------	---

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	<code>extraClassPath</code> corresponds to <code>spark.driver.extraClassPath</code> for the driver and <code>spark.executor.extraClassPath</code> 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` Internal 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, `isClusterMode` flag effectively "splits" `Client` on two parts per deploy mode—one responsible for `client` and the other for `cluster` deploy mode.

Caution

[FIXME](#) Replace the internal fields used below with their true meanings.

Table 2. Internal Attributes of `Client` per Deploy Mode (`isClusterMode`)

Internal attribute	cluster deploy mode	client 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.am.libraryPath</code>
--class command-line argument for ApplicationMaster	<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.

SPARK_YARN_MODE flag

`SPARK_YARN_MODE` flag controls... [FIXME](#)

Note	Any environment variable with the <code>SPARK_</code> prefix is propagated to all (remote) processes.
Caution	FIXME 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 YarnSparkHadoopUtil .

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

FIXME Why is this needed? `git blame` it.

`SPARK_YARN_MODE` flag is checked when [YarnSparkHadoopUtil](#) or [SparkHadoopUtil](#) are accessed.

It is cleared later when [Client](#) is requested to stop.

Internal Hadoop's YarnClient — `yarnClient` Property

```
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.

Launching Client Standalone Application — `main` Method

`main` method is invoked while a Spark application is being deployed to a YARN cluster.

Note

It is executed by [spark-submit](#) with `--master yarn` command-line argument.

Note

When you start the `main` method when starting the `Client` standalone application `org.apache.spark.deploy.yarn.Client`, 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.

Stopping Client (with LauncherBackend and YarnClient) — `stop` Method

```
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).

Running Client — run Method

`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 <code>LauncherBackend</code> lost the connection since it was connected in <code>submitApplication</code> ?
---------	--

Caution	FIXME What is <code>fireAndForget</code> ?
---------	--

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 <code>YarnApplicationState</code> and <code>FinalApplicationStatus</code> statuses?
---------	--

monitorApplication Method

```
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	<code>cleanupStagingDir</code> won't be called when <code>returnOnRunning</code> is enabled and an application turns RUNNING. <i>I guess it is likely a left-over since the Client is deprecated now.</i>
------	---

The current state is recorded for future checks (in the loop).

cleanupStagingDir Method

`cleanupStagingDir` clears the staging directory of an application.

Note	It is used in <code>submitApplication</code> when there is an exception and <code>monitorApplication</code> 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 Method

```
reportLauncherState(state: SparkAppHandle.State): Unit
```

`reportLauncherState` merely passes the call on to `LauncherBackend.setState`.

Caution	What does <code>setState</code> do?
---------	-------------------------------------

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).

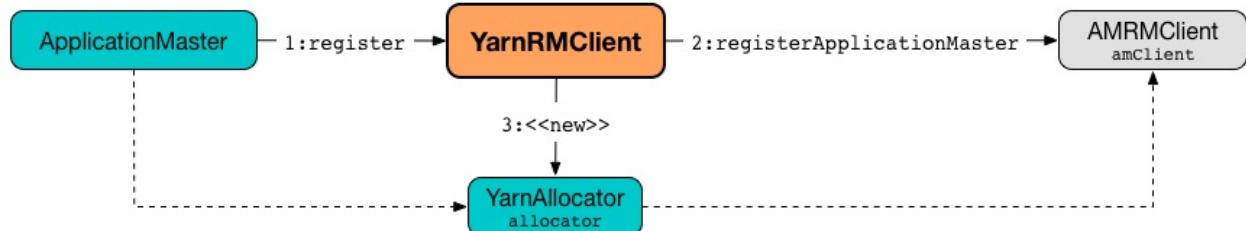


Figure 1. Registering ApplicationMaster with YARN ResourceManager

`YarnRMClient` is just a wrapper for [AMRMClient\[ContainerRequest\]](#) that is started when [registering](#) `ApplicationMaster` (and never stopped explicitly!).

`YarnRMClient` tracks the [application attempt identifiers](#) and the [maximum number of attempts to register](#) `ApplicationMaster`.

Table 1. YarnRMClient's Internal Registries and Counters

Name	Description
<code>amClient</code>	<p><code>AMRMClient</code> using <code>ContainerRequest</code> for YARN ResourceManager.</p> <p>Created (initialized and started) when <code>YarnRMClient</code> registers <code>ApplicationMaster</code>.</p> <p>Used when <code>YarnRMClient</code> creates a <code>YarnAllocator</code> (after registering <code>ApplicationMaster</code>) and to unregister <code>ApplicationMaster</code>.</p>
<code>uiHistoryAddress</code>	
<code>registered</code>	Flag to say whether <code>YarnRMClient</code> is connected to YARN ResourceManager (i.e. <code>true</code>) or not. Disabled by default. Used when unregistering <code>ApplicationMaster</code> .

Tip	<p>Enable <code>INFO</code> logging level for <code>org.apache.spark.deploy.yarn.YarnRMClient</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.YarnRMClient=INFO</pre> <p>Refer to Logging.</p>

Registering ApplicationMaster with YARN ResourceManager (and Creating YarnAllocator)

— register Method

```
register(
  driverUrl: String,
  driverRef: RpcEndpointRef,
  conf: YarnConfiguration,
  sparkConf: SparkConf,
  uiAddress: String,
  uiHistoryAddress: String,
  securityMgr: SecurityManager,
  localResources: Map[String, LocalResource]): YarnAllocator
```

`register` creates a [AMRMClient](#), initializes it (using the input [YarnConfiguration](#)) and starts immediately.

Note	<code>AMRMClient</code> is used in YARN to register an application's <code>ApplicationMaster</code> with the YARN ResourceManager.
------	--

	<code>register</code> connects to YARN ResourceManager using the following design pattern:
--	--

	<pre>val amClient: AMRMClient[ContainerRequest] = AMRMClient.createAMRMClient() amClient.init(conf) amClient.start()</pre>
--	--

`register` saves the input `uiHistoryAddress` as [uiHistoryAddress](#).

You should see the following INFO message in the logs (in stderr in YARN):

```
INFO YarnRMClient: Registering the ApplicationMaster
```

`register` then uses [AMRMClient](#) to register the Spark application's `ApplicationMaster` (using the local hostname, the port `0` and the input `uiAddress`).

	The input <code>uiAddress</code> is the web UI of the Spark application and is specified using the SparkContext (when the application runs in <code>cluster</code> deploy mode) or using spark.driver.appUIAddress property.
--	--

`registered` flag is enabled.

In the end, `register` creates a new `YarnAllocator` (using the input parameters of `register` and the internal [AMRMClient](#)).

Note

`register` is used exclusively when `ApplicationMaster` registers itself with the YARN ResourceManager.

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 Method

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

ApplicationMaster (aka ExecutorLauncher)

`ApplicationMaster` is the [YARN ApplicationMaster](#) for a Spark application submitted to a YARN cluster (which is commonly called [Spark on YARN](#)).

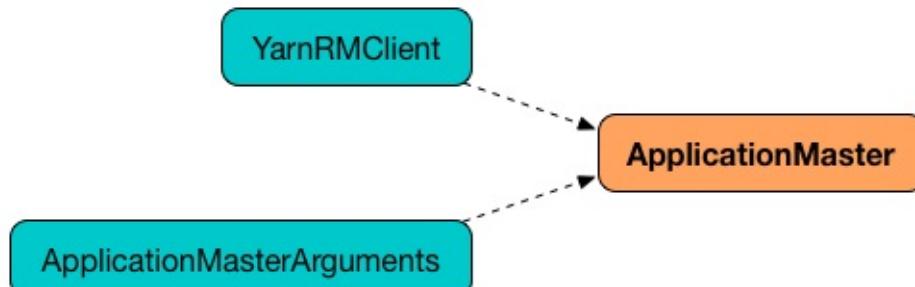


Figure 1. ApplicationMaster's Dependencies

`ApplicationMaster` is a [standalone application](#) that [YARN NodeManager](#) runs in a YARN container to manage a Spark application running in a YARN cluster.

	<p>From the official documentation of Apache Hadoop YARN (with some minor changes of mine):</p> <p>Note The per-application ApplicationMaster is actually a framework-specific library and is tasked with negotiating cluster resources from the YARN ResourceManager and working with the YARN NodeManager(s) to execute and monitor the tasks.</p>
--	---

`ApplicationMaster` (and `ExecutorLauncher`) is launched as a result of `client` creating a `ContainerLaunchContext` to launch a Spark application on YARN.

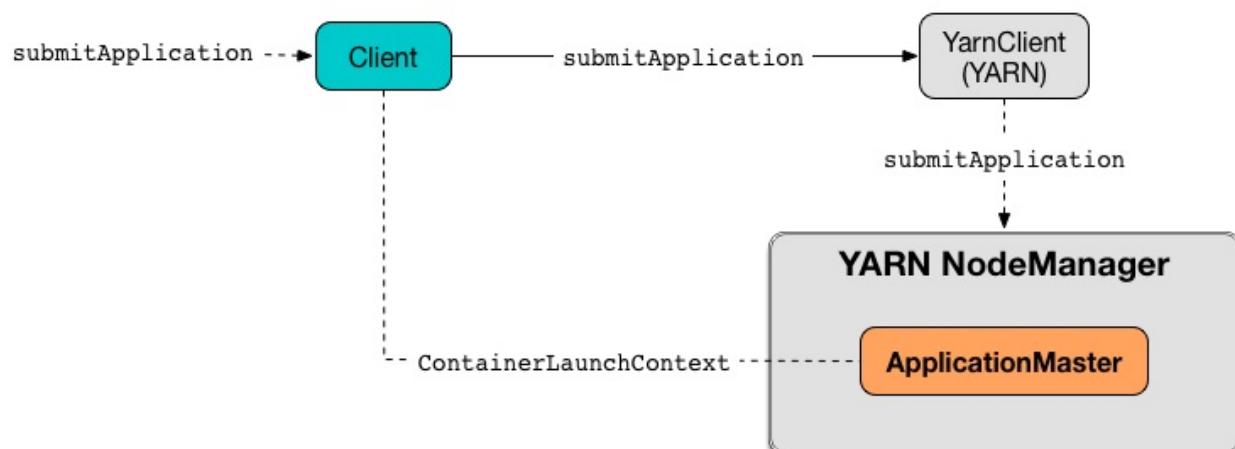


Figure 2. Launching ApplicationMaster

	<p><code>ContainerLaunchContext</code> represents all of the information needed by the YARN NodeManager to launch a container.</p>
--	--

`ExecutorLauncher` is a custom `ApplicationMaster` for `client deploy mode` only for distinguishing client and cluster deploy modes when using `ps` or `jps`.

Note

```
$ jps -lm
```

```
71253 org.apache.spark.deploy.yarn.ExecutorLauncher --arg
192.168.99.1:50188 --properties-file /tmp/hadoop-jacek/nm-1
dir/usercache/jacek/appcache/.../_spark_conf__/_spark_con
```

When `created` `ApplicationMaster` takes a `YarnRMClient` (to handle communication with `YARN ResourceManager` for YARN containers for `ApplicationMaster` and executors).

`ApplicationMaster` uses `YarnAllocator` to manage YARN containers with executors.

Table 1. ApplicationMaster's Internal Properties

Name	Initial Value	Description
amEndpoint	(uninitialized)	<p><code>RpcEndpointRef</code> to the <code>YarnAM</code> RPC endpoint initialized when <code>ApplicationMaster runAMEndpoint</code>.</p> <p>CAUTION: <code>FIXME</code> When, in a Spark application's lifecycle, does <code>runAMEndpoint</code> really happen?</p> <p>Used exclusively when <code>ApplicationMaster registers the web UI security filters</code> (in <code>client deploy mode</code> when the driver runs outside <code>ApplicationMaster</code>).</p>
client	<code>YarnRMClient</code>	<p>Used to <code>register the ApplicationMaster</code> and request containers for executors from YARN and later <code>unregister ApplicationMaster</code> from YARN <code>ResourceManager</code>.</p> <p>Used to get an application attempt <code>id</code> and the <code>allowed number of attempts to register ApplicationMaster</code>.</p> <p>Used to get filter parameters to secure <code>ApplicationMaster's UI</code>.</p>
sparkConf	<code>New SparkConf</code>	<code>FIXME</code>

finished	false	Flag to...FIXME
yarnConf	Hadoop's YarnConfiguration	Flag to...FIXME Created using SparkHadoopUtil.newConfiguration
exitCode	0	FIXME
userClassThread	(uninitialized)	FIXME
sparkContextPromise	SparkContext Scala's Promise	<p>Used only in <code>cluster</code> deploy mode (when the driver and <code>ApplicationMaster</code> run together in a YARN container) as a communication bus between <code>ApplicationMaster</code> and the separate <code>Driver</code> thread that runs a Spark application.</p> <p>Used to inform <code>ApplicationMaster</code> when a Spark application's <code>SparkContext</code> has been initialized successfully or failed.</p> <p>Non-<code>null</code> value allows <code>ApplicationMaster</code> to access the driver's <code>RpcEnv</code> (available as <code>rpcEnv</code>).</p> <p>NOTE: A successful initialization of a Spark application's <code>SparkContext</code> is when YARN-specific <code>TaskScheduler</code>, i.e. <code>YarnClusterScheduler</code>, gets informed that the Spark application has started. <i>What a clever solution!</i></p>
rpcEnv	(uninitialized)	<p><code>RpcEnv</code> which is:</p> <ul style="list-style-type: none"> • <code>sparkYarnAM</code> RPC environment from a Spark application submitted to YARN in <code>client</code> deploy mode. • <code>sparkDriver</code> RPC environment from the Spark application submitted to YARN in <code>cluster</code> deploy mode.
isClusterMode	true (when --class was specified)	Flag...FIXME

maxNumExecutorFailures	FIXME
------------------------	-------

maxNumExecutorFailures Property

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

Computed 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.

Creating ApplicationMaster Instance

`ApplicationMaster` takes the following when created:

- [ApplicationMasterArguments](#)
- [YarnRMClient](#)

`ApplicationMaster` initializes the [internal registries and counters](#).

Caution	FIXME Review the initialization again
---------	---------------------------------------

reporterThread Method

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

Launching Progress Reporter Thread — launchReporterThread Method

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

Setting Internal SparkContext Reference — sparkContextInitialized Method

`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` Method

```
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`).

Registering web UI Security Filters — `addAmIpFilter` Method

```
addAmIpFilter(): Unit
```

`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](#)).

finish Method

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

allocator Internal Reference to YarnAllocator

`allocator` is the internal reference to `YarnAllocator` that `ApplicationMaster` uses to request new or release outstanding containers for executors.

`allocator` is created when `ApplicationMaster` is registered (using the internal `YarnRMClient` reference).

Launching ApplicationMaster Standalone Application

— main Method

`ApplicationMaster` is started as a standalone application inside a YARN container on a node.

Note	<code>ApplicationMaster</code> standalone application is launched as a result of sending a <code>ContainerLaunchContext</code> request to launch <code>ApplicationMaster</code> for a Spark application to YARN ResourceManager.
------	--

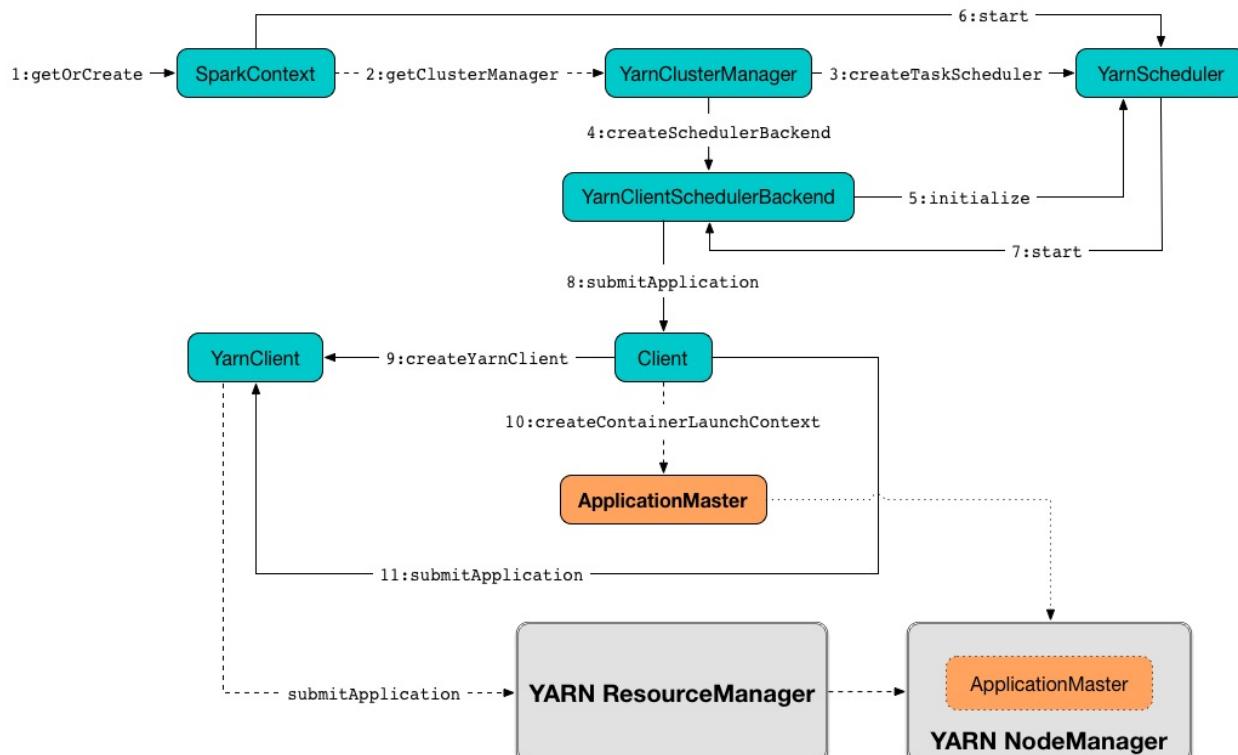


Figure 3. Submitting ApplicationMaster to YARN NodeManager

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.

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 Logging.</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 new `YarnRMClient`) and then [runs it](#).

Running ApplicationMaster — `run` Method

```
run(): Int
```

`run` reads the [application attempt id](#).

(only in `cluster` deploy mode) `run` sets `cluster` deploy mode-specific settings and sets the application attempt id (from YARN).

`run` sets a `CallerContext` for `APPMASTER`.

Caution

[**FIXME** Why is `CallerContext` required? It's only executed when `hadoop.caller.context.enabled` is enabled and `org.apache.hadoop.ipc.CallerContext` class is on CLASSPATH.](#)

You should see the following INFO message in the logs:

```
INFO ApplicationAttemptId: [appAttemptId]
```

`run` creates a Hadoop [FileSystem](#) (using the internal [YarnConfiguration](#)).

`run` registers the [cleanup shutdown hook](#).

`run` creates a [SecurityManager](#).

(only when `spark.yarn.credentials.file` is defined) `run` creates a `ConfigurableCredentialManager` to get a `AMCredentialRenewer` and schedules login from keytab.

Caution

[**FIXME** Security stuff begs for more details.](#)

In the end, `run` registers `ApplicationMaster` (with YARN ResourceManager) for the Spark application — either calling `runDriver` (in `cluster` deploy mode) or `runExecutorLauncher` (for `client` deploy mode).

`run` exits with `0` exit code.

In case of an exception, you should see the following ERROR message in the logs and `run` finishes with `FAILED` final application status.

```
ERROR Uncaught exception: [exception]
```

Note

`run` is used exclusively when `ApplicationMaster` is launched as a standalone application (inside a YARN container on a YARN cluster).

Creating sparkYarnAM RPC Environment and Registering ApplicationMaster with YARN ResourceManager (Client Deploy Mode) — `runExecutorLauncher` Internal Method

```
runExecutorLauncher(securityMgr: SecurityManager): Unit
```

`runExecutorLauncher` creates `sparkYarnAM` RPC environment (on `spark.yarn.am.port` port, the internal `SparkConf` and `clientMode` enabled).

Tip

Read the note in [Creating RpcEnv](#) to learn the meaning of `clientMode` input argument.

`clientMode` is enabled for so-called a client-mode `ApplicationMaster` which is when a Spark application is submitted to YARN in [client deploy mode](#).

`runExecutorLauncher` then waits until the driver accepts connections and creates `RpcEndpointRef` to communicate.

`runExecutorLauncher` registers web UI security filters.

Caution

FIXME Why is this needed? `addAmIpFilter`

In the end, `runExecutorLauncher` registers `ApplicationMaster` with YARN ResourceManager and requests resources and then pauses until `reporterThread` finishes.

Note

`runExecutorLauncher` is used exclusively when `ApplicationMaster` is started in [client deploy mode](#).

Running Spark Application's Driver and Registering ApplicationMaster with YARN ResourceManager (Cluster Deploy Mode) — `runDriver` Internal Method

```
runDriver(securityMgr: SecurityManager): Unit
```

`runDriver` starts a Spark application on a [separate thread](#), registers `YarnAM` endpoint in the application's `RpcEnv` followed by registering `ApplicationMaster` with YARN ResourceManager. In the end, `runDriver` waits for the Spark application to finish.

Internally, `runDriver` registers web UI security filters and [starts a Spark application](#) (on a [separate Thread](#)).

You should see the following INFO message in the logs:

```
INFO Waiting for spark context initialization...
```

`runDriver` waits `spark.yarn.am.waitTime` time till the Spark application's `SparkContext` is available and accesses the [current `RpcEnv`](#) (and saves it as the internal `rpcEnv`).

Note	<code>runDriver</code> uses <code>SparkEnv</code> to access the current <code>RpcEnv</code> that the Spark application's <code>SparkContext</code> manages.
------	---

`runDriver` creates `RpcEndpointRef` to the driver's `YarnScheduler` endpoint and registers `YarnAM` endpoint (using `spark.driver.host` and `spark.driver.port` properties for the driver's host and port and `isClusterMode` enabled).

`runDriver` registers `ApplicationMaster` with YARN ResourceManager and requests cluster resources (using the Spark application's `RpcEnv`, the driver's RPC endpoint reference, `webUrl` if web UI is enabled and the input `securityMgr`).

`runDriver` pauses until the Spark application finishes.

Note	<code>runDriver</code> uses Java's <code>Thread.join</code> on the internal <code>Thread</code> reference to the Spark application running on it.
------	---

If the Spark application has not started in `spark.yarn.am.waitTime` time, `runDriver` reports a `IllegalStateException` :

```
SparkContext is null but app is still running!
```

If `TimeoutException` is reported while waiting for the Spark application to start, you should see the following ERROR message in the logs and `runDriver` finishes with `FAILED` final application status and the error code `13`.

```
ERROR SparkContext did not initialize after waiting for [spark.yarn.am.waitTime] ms. Please check earlier log output for errors. Failing the application.
```

Note	<code>runDriver</code> is used exclusively when <code>ApplicationMaster</code> is started in <code>cluster deploy mode</code> .
------	---

Starting Spark Application (in Separate Driver Thread)

— `startUserApplication` Method

```
startUserApplication(): Thread
```

`startUserApplication` starts a Spark application as a separate `Driver` thread.

Internally, when `startUserApplication` is executed, you should see the following INFO message in the logs:

```
INFO Starting the user application in a separate Thread
```

`startUserApplication` takes the [user-specified jars](#) and maps them to use the `file`: protocol.

`startUserApplication` then creates a class loader to load the main class of the Spark application given the [precedence of the Spark system jars and the user-specified jars](#).

`startUserApplication` works on custom configurations for Python and R applications (which I don't bother including here).

`startUserApplication` loads the main class (using the custom class loader created above with the user-specified jars) and creates a reference to the `main` method.

Note	The main class is specified as <code>userClass</code> in ApplicationMasterArguments when <code>ApplicationMaster</code> was created.
------	--

`startUserApplication` starts a Java [Thread](#) (with the name `Driver`) that invokes the `main` method (with the application arguments from `userArgs` from [ApplicationMasterArguments](#)). The `Driver` thread uses the internal [sparkContextPromise](#) to notify `ApplicationMaster` about the execution status of the `main` method (success or failure).

When the main method (of the Spark application) finishes successfully, the `Driver` thread will [finish](#) with `SUCCEEDED` final application status and code status `0` and you should see the following DEBUG message in the logs:

```
DEBUG Done running users class
```

Any exceptions in the `Driver` thread are reported with corresponding ERROR message in the logs, `FAILED` final application status, appropriate code status.

```
// SparkUserAppException
ERROR User application exited with status [exitCode]

// non-SparkUserAppException
ERROR User class threw exception: [cause]
```

Note A Spark application's exit codes are passed directly to `finish ApplicationMaster` and recorded as `exitCode` for future reference.

Note `startUserApplication` is used exclusively when `ApplicationMaster` runs a **Spark application's driver** and registers itself with `YARN ResourceManager` for cluster deploy mode.

Registering ApplicationMaster with YARN ResourceManager and Requesting YARN Cluster Resources — `registerAM` Internal Method

```
registerAM(
  _sparkConf: SparkConf,
  _rpcEnv: RpcEnv,
  driverRef: RpcEndpointRef,
  uiAddress: String,
  securityMgr: SecurityManager): Unit
```

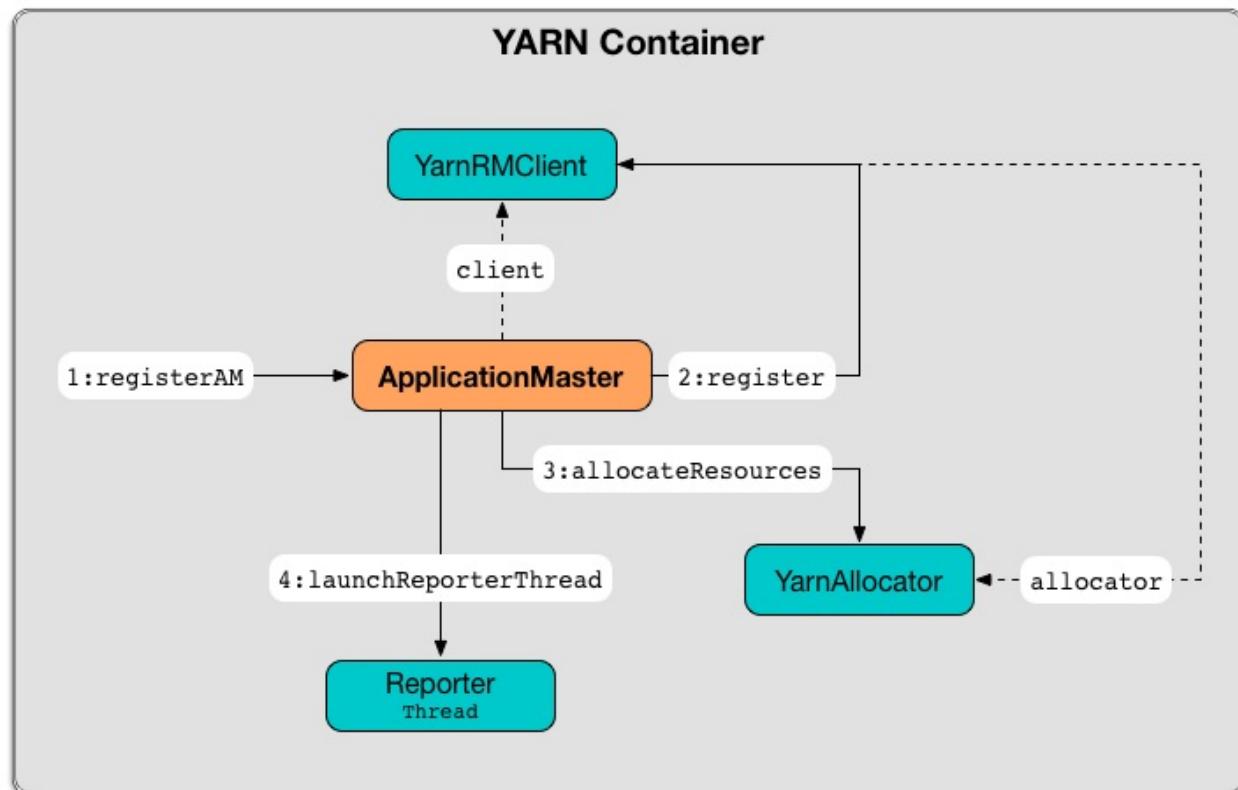


Figure 4. Registering ApplicationMaster with YARN ResourceManager

Internally, `registerAM` first takes the application and attempt ids, and creates the URL of [Spark History Server](#) for the Spark application, i.e. `[address]/history/[appId]/[attemptId]`, by substituting Hadoop variables (using the internal [YarnConfiguration](#)) in the optional `spark.yarn.historyServer.address` setting.

`registerAM` then creates a [RpcEndpointAddress](#) for the driver's [CoarseGrainedScheduler](#) [RPC endpoint](#) available at `spark.driver.host` and `spark.driver.port`.

`registerAM` prints YARN launch context diagnostic information (with command, environment and resources) for executors (with `spark.executor.memory`, `spark.executor.cores` and dummy `<executorId>` and `<hostname>`)

`registerAM` requests [YarnRMClient](#) to register [ApplicationMaster](#) (with YARN ResourceManager) and the internal [YarnAllocator](#) to allocate required cluster resources (given placement hints about where to allocate resource containers for executors to be as close to the data as possible).

Note	<code>registerAM</code> uses YarnRMClient that was given when ApplicationManager was created.
------	---

In the end, `registerAM` launches reporter thread.

Note	<code>registerAM</code> is used when ApplicationMaster runs a Spark application in cluster deploy mode and client deploy mode .
------	---

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.
```

localResources Property

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 <code>NodeManager</code> .
------	--

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]
```

Cluster Mode Settings

When in `cluster` [deploy mode](#), `ApplicationMaster` sets the following system properties (in `run`):

- `spark.ui.port` to `0`
- `spark.master` as `yarn`
- `spark.submit.deployMode` as `cluster`
- `spark.yarn.app.id` as YARN-specific application id

Caution	<code>FIXME</code> Why are the system properties required? Who's expecting them?
---------	--

`isClusterMode` Internal Flag

Caution	<code>FIXME</code> Since <code>org.apache.spark.deploy.yarn.ExecutorLauncher</code> is used for client deploy mode , the <code>isClusterMode</code> flag could be set there (not depending on <code>--class</code> 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` (to launch the `ApplicationMaster` for a Spark application).

`isClusterMode` is used to set additional system properties in `run` and `runDriver` (the flag is enabled) or `runExecutorLauncher` (when disabled).

Besides, `isClusterMode` controls the default final status of a Spark application being `FinalApplicationStatus.FAILED` (when the flag is enabled) or `FinalApplicationStatus.UNDEFINED`.

`isClusterMode` 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 `cleanup shutdown hook` (that was registered in `ApplicationMaster` when it `started running`) 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`.

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

```
getAttemptId(): ApplicationAttemptId
```

`getAttemptId` returns YARN's `ApplicationAttemptId` (of the Spark application to which the container was assigned).

Internally, it queries YARN by means of [YarnRMClient](#).

Waiting Until Driver is Network-Accessible and Creating `RpcEndpointRef` to Communicate — `waitForSparkDriver` Internal Method

```
waitForSparkDriver(): RpcEndpointRef
```

`waitForSparkDriver` waits until the driver is network-accessible, i.e. accepts connections on a given host and port, and returns a `RpcEndpointRef` to the driver.

When executed, you should see the following INFO message in the logs:

```
INFO yarn.ApplicationMaster: Waiting for Spark driver to be reachable.
```

`waitForSparkDriver` takes the driver's host and port (using [ApplicationMasterArguments](#) passed in when [ApplicationMaster was created](#)).

Caution	FIXME <code>waitForSparkDriver</code> expects the driver's host and port as the 0-th element in <code>ApplicationMasterArguments.userArgs</code> . Why?
---------	--

`waitForSparkDriver` tries to connect to the driver's host and port until the driver accepts the connection but no longer than `spark.yarn.am.waitTime` setting or `finished` internal flag is enabled.

You should see the following INFO message in the logs:

```
INFO yarn.ApplicationMaster: Driver now available: [driverHost]:[driverPort]
```

While `waitForSparkDriver` tries to connect (while the socket is down), you can see the following ERROR message and `waitForSparkDriver` pauses for 100 ms and tries to connect again (until the `waitTime` elapses).

```
ERROR Failed to connect to driver at [driverHost]:[driverPort], retrying ...
```

Once `waitForSparkDriver` could connect to the driver, `waitForSparkDriver` sets `spark.driver.host` and `spark.driver.port` properties to `driverHost` and `driverPort`, respectively (using the internal [SparkConf](#)).

In the end, `waitForSparkDriver` [runAMEndpoint](#).

If `waitForSparkDriver` did not manage to connect (before `waitTime` elapses or `finished` internal flag was enabled), `waitForSparkDriver` reports a `SparkException`:

```
Failed to connect to driver!
```

Note

`waitForSparkDriver` is used exclusively when client-mode ApplicationMaster creates the `sparkYarnAM` RPC environment and registers itself with YARN ResourceManager.

Creating RpcEndpointRef to Driver's YarnScheduler Endpoint and Registering YarnAM Endpoint

— `runAMEndpoint` Internal Method

```
runAMEndpoint(host: String, port: String, isClusterMode: Boolean): RpcEndpointRef
```

`runAMEndpoint` sets up a `RpcEndpointRef` to the driver's `YarnScheduler` endpoint and registers `YarnAM` endpoint.

Note

`sparkDriver` RPC environment when the driver lives in YARN cluster (in cluster deploy mode)

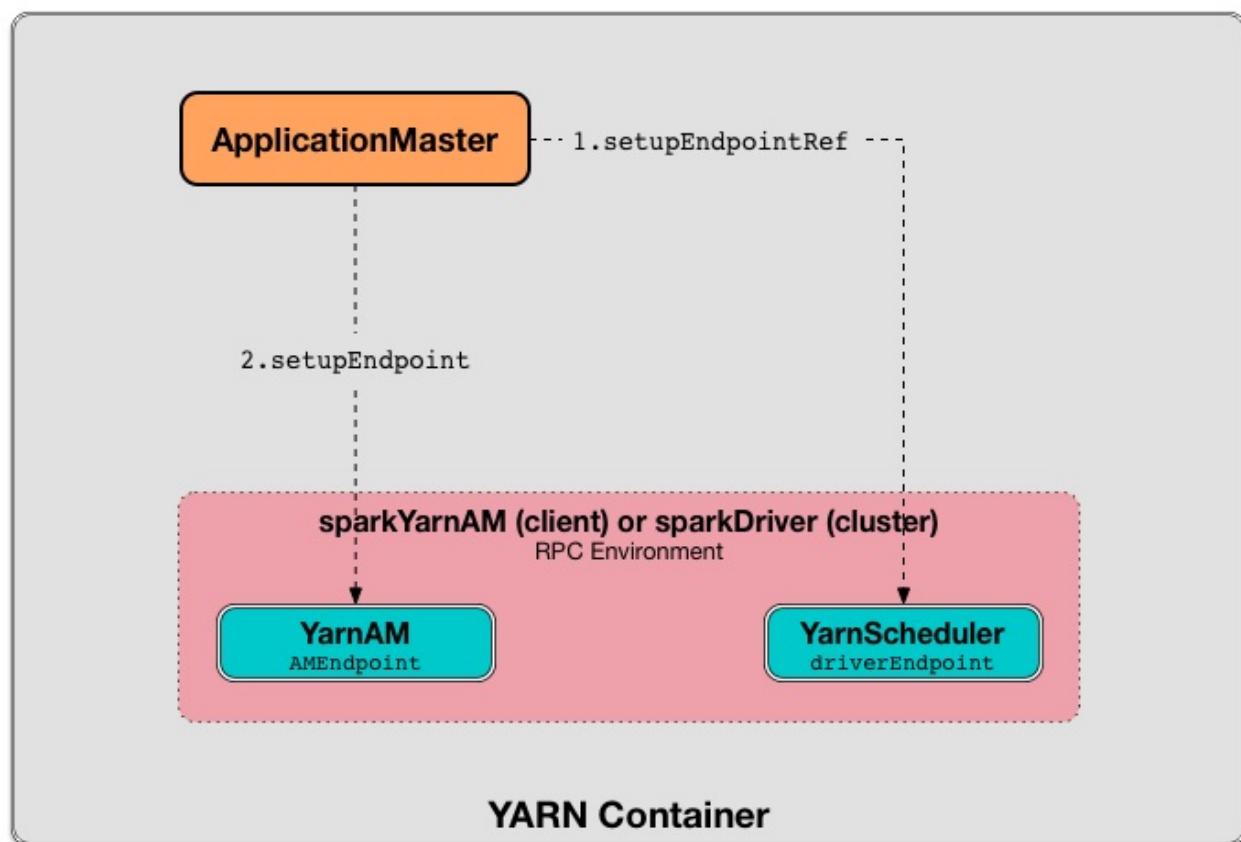


Figure 5. Registering YarnAM Endpoint

Internally, `runAMEndpoint` gets a `RpcEndpointRef` to the driver's `YarnScheduler` endpoint (available on the `host` and `port`).

Note

`YarnScheduler` RPC endpoint is registered when the `Spark coarse-grained scheduler backends for YARN` are created.

`runAMEndpoint` then registers the RPC endpoint as **YarnAM** (and **AMEndpoint** implementation with `ApplicationMaster`'s `RpcEnv`, `YarnScheduler` endpoint reference, and `isClusterMode` flag).

Note

`runAMEndpoint` is used when `ApplicationMaster` waits for the driver (in client deploy mode) and runs the driver (in cluster deploy mode).

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
```

Initializing YarnClusterManager — initialize Method

`initialize` simply [initializes the input `TaskSchedulerImpl`](#).

TaskSchedulers for YARN

There are 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.

Tip Enable `INFO` or `DEBUG` logging levels for `org.apache.spark.scheduler.cluster.YarnScheduler` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.scheduler.cluster.YarnScheduler=DEBUG
```

Refer to [Logging](#).

Tracking Racks per Hosts and Ports (getRackForHost method)

`getRackForHost` attempts to compute the rack for a host.

Note `getRackForHost` overrides the [parent TaskSchedulerImpl's getRackForHost](#)

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 Callback

`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 — Foundation for Coarse-Grained Scheduler Backends for YARN

`YarnSchedulerBackend` is a `CoarseGrainedSchedulerBackend` that acts as the foundation for the concrete deploy mode-specific Spark scheduler backends for YARN, i.e.

`YarnClientSchedulerBackend` and `YarnClusterSchedulerBackend` for `client` deploy mode and `cluster` deploy mode, respectively.

`YarnSchedulerBackend` registers itself as `YarnScheduler` RPC endpoint in the RPC Environment.

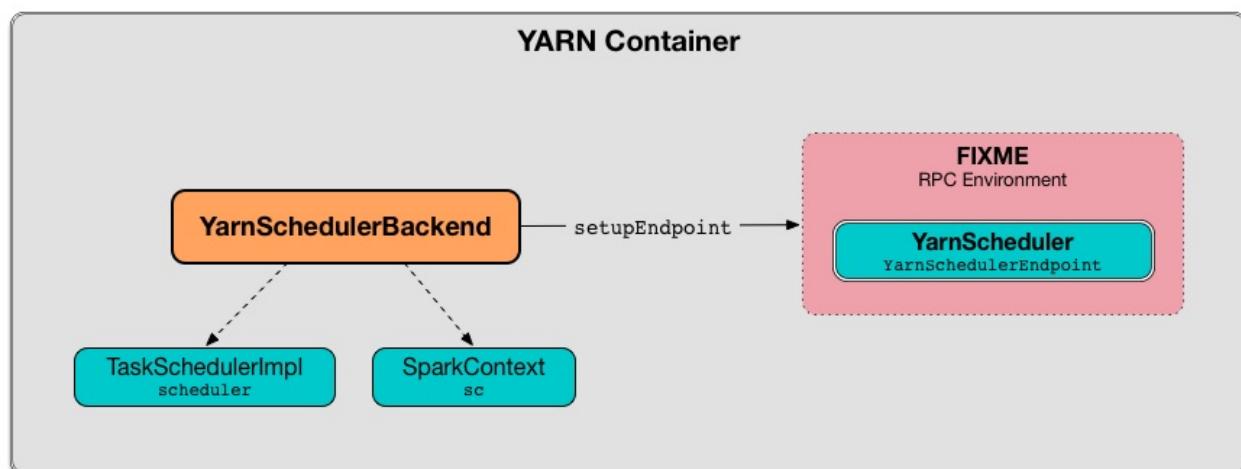


Figure 1. YarnSchedulerBackend in YARN Container

`YarnSchedulerBackend` is ready to accept task launch requests right after the sufficient executors are registered (that varies on dynamic allocation being enabled or not).

Note	With no extra configuration, <code>YarnSchedulerBackend</code> is ready for task launch requests when 80% of all the requested executors are available.
Note	<code>YarnSchedulerBackend</code> is an <code>private[spark]</code> abstract class and is never created directly (but only indirectly through the concrete implementations <code>YarnClientSchedulerBackend</code> and <code>YarnClusterSchedulerBackend</code>).

Table 1. YarnSchedulerBackend's Internal Properties

Name	Initial Value	
<code>minRegisteredRatio</code>	<p>Ratio for minimum number of registered executors to claim <code>YarnSchedulerBackend</code> is ready for task launch requests.</p> <ul style="list-style-type: none"> • <code>0.8</code> (when <code>spark.scheduler.minRegisteredResourcesRatio</code> property is undefined) 	Minir exec that s avail task

	<ul style="list-style-type: none"> • minRegisteredRatio from the parent CoarseGrainedSchedulerBackend 	
yarnSchedulerEndpoint	YarnSchedulerEndpoint object	
yarnSchedulerEndpointRef	RPC endpoint reference to yarnScheduler RPC endpoint	Crea Yarn creat
totalExpectedExecutors	0	Total exec that s avail task
askTimeout	FIXME	Upda Spar mode
appId	FIXME	FIXM
attemptId	(undefined)	YARI a Sp Only mode
shouldResetOnAmRegister		Set v Yarn starts using Appl Use which Sche
		Cont Yarn anoth RPC allow after Appli new can c deplo

Disal
Yarn:
creat

Resetting YarnSchedulerBackend — `reset` Method

Note

`reset` is a part of [CoarseGrainedSchedulerBackend Contract](#).

`reset` [resets](#) the parent `CoarseGrainedSchedulerBackend` scheduler backend and `ExecutorAllocationManager` (accessible by `sparkContext.executorAllocationManager`).

doRequestTotalExecutors Method

```
def doRequestTotalExecutors(requestedTotal: Int): Boolean
```

Note

`doRequestTotalExecutors` is a part of the [CoarseGrainedSchedulerBackend Contract](#).

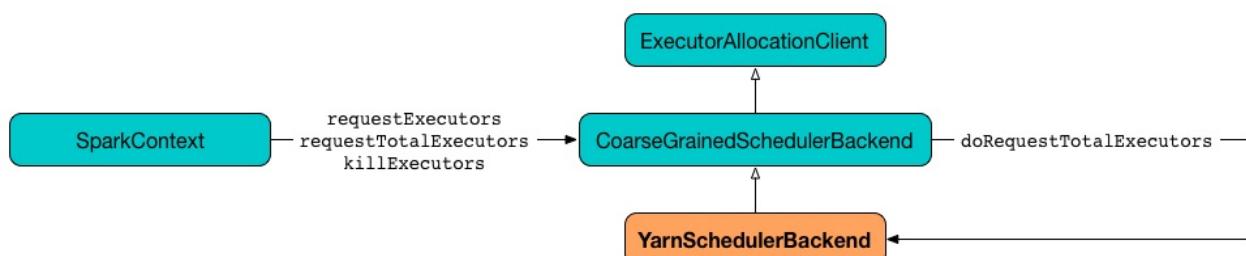


Figure 2. Requesting Total Executors in YarnSchedulerBackend (`doRequestTotalExecutors` method)

`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?

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`.

Requesting YARN for Spark Application's Current Attempt Id — `applicationAttemptId` Method

```
applicationAttemptId(): Option[String]
```

Note

`applicationAttemptId` is a part of [SchedulerBackend Contract](#).

`applicationAttemptId` requests the internal YARN's `ApplicationAttemptId` for the Spark application's current attempt id.

Creating YarnSchedulerBackend Instance

Note	This section is only to take notes about the required components to instantiate the base services.
------	--

`YarnSchedulerBackend` takes the following when created:

1. `TaskSchedulerImpl`
2. `SparkContext`

`YarnSchedulerBackend` initializes the [internal properties](#).

Checking if Enough Executors Are Available — `sufficientResourcesRegistered` Method

```
sufficientResourcesRegistered(): Boolean
```

Note	<code>sufficientResourcesRegistered</code> is a part of the CoarseGrainedSchedulerBackend contract that makes sure that sufficient resources are available.
------	---

`sufficientResourcesRegistered` is positive, i.e. `true`, when `totalRegisteredExecutors` is exactly or above `minRegisteredRatio` of `totalExpectedExecutors`.

YarnClientSchedulerBackend — SchedulerBackend for YARN in Client Deploy Mode

`YarnClientSchedulerBackend` is the [YarnSchedulerBackend](#) used when a Spark application is submitted to a YARN cluster in `client` deploy mode.

Note

`client` deploy mode is the default deploy mode of Spark applications submitted to a YARN cluster.

`YarnClientSchedulerBackend` submits a Spark application when [started](#) and [waits for the Spark application](#) until it finishes (successfully or not).

Table 1. `YarnClientSchedulerBackend`'s Internal Properties

Name	Initial Value	Description
<code>client</code>	(undefined)	<p>Client to submit and monitor a Spark application (when <code>YarnClientSchedulerBackend</code> is started).</p> <p>Created when <code>YarnClientSchedulerBackend</code> is started and stopped when <code>YarnClientSchedulerBackend</code> stops.</p>
<code>monitorThread</code>	(undefined)	MonitorThread

Tip

Enable `DEBUG` logging level for `org.apache.spark.scheduler.cluster.YarnClientSchedulerBackend` logger to see what happens inside `YarnClientSchedulerBackend`.

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

```
log4j.logger.org.apache.spark.scheduler.cluster.YarnClientSchedulerBackend=DEBUG
```

Refer to [Logging](#).

Enable `DEBUG` logging level for `org.apache.hadoop` logger to see what happens inside Hadoop YARN.

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

```
log4j.logger.org.apache.hadoop=DEBUG
```

Tip

Refer to [Logging](#).

Use with caution though as there will be a flood of messages in the logs every second.

Starting YarnClientSchedulerBackend — `start` Method

```
start(): Unit
```

Note

`start` is a part of [SchedulerBackend contract](#) executed when `TaskSchedulerImpl` starts.

`start` creates [Client](#) (to communicate with YARN ResourceManager) and [submits a Spark application](#) to a YARN cluster.

After the application is launched, `start` starts a [MonitorThread](#) state monitor thread. In the meantime it also calls the supertype's `start`.

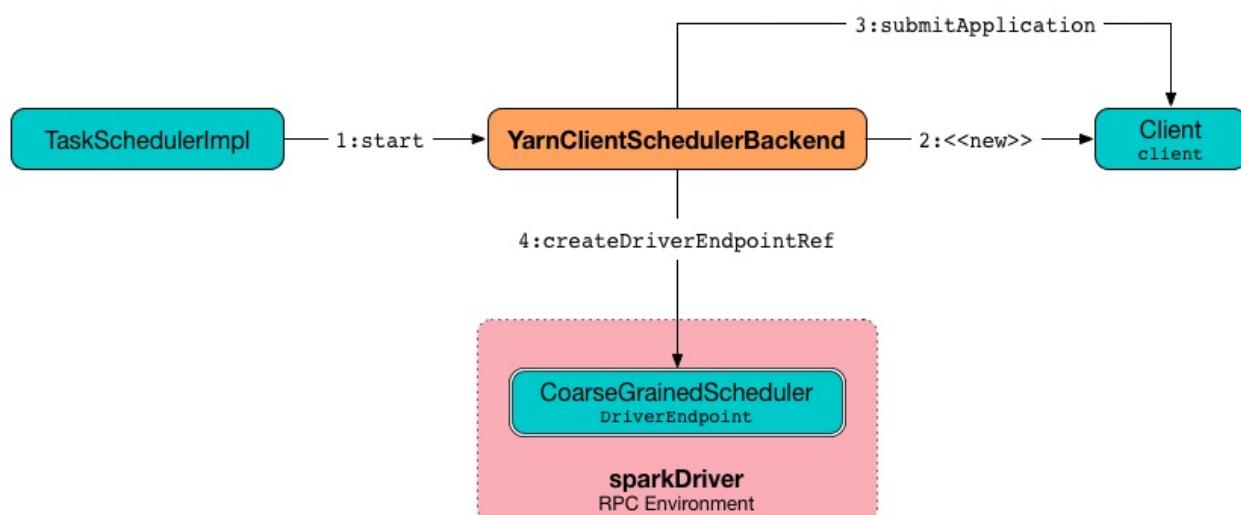


Figure 1. Starting YarnClientSchedulerBackend

Internally, `start` takes `spark.driver.host` and `spark.driver.port` properties for the driver's host and port, respectively.

If [web UI is enabled](#), `start` sets `spark.driver.appUIAddress` as `webUrl`.

You should see the following DEBUG message in the logs:

```
DEBUG YarnClientSchedulerBackend: ClientArguments called with: --arg [hostport]
```

Note	<code>hostport</code> is <code>spark.driver.host</code> and <code>spark.driver.port</code> properties separated by <code>:</code> , e.g. <code>192.168.99.1:64905</code> .
------	---

`start` creates a [ClientArguments](#) (passing in a two-element array with `--arg` and `hostport`).

`start` sets the [total expected number of executors](#) to the initial number of executors.

Caution	FIXME Why is this part of subtypes since they both set it to the same value?
---------	--

`start` creates a [Client](#) (with the [ClientArguments](#) and `SparkConf`).

`start` submits the Spark application to YARN (through [Client](#)) and saves `ApplicationId` (with undefined `ApplicationAttemptId`).

`start` starts [YarnSchedulerBackend](#) (that in turn starts the top-level [CoarseGrainedSchedulerBackend](#)).

Caution	FIXME Would be very nice to know why <code>start</code> does so in a NOTE.
---------	--

`start` waits until the Spark application is running.

(only when `spark.yarn.credentials.file` is defined) `start` starts [ConfigurableCredentialManager](#).

Caution	FIXME Why? Include a NOTE to make things easier.
---------	--

`start` creates and starts [monitorThread](#) (to monitor the Spark application and stop the current `SparkContext` when it stops).

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 Until Spark Application Runs — `waitForApplication` Internal Method

```
waitForApplication(): Unit
```

`waitForApplication` 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.
```

Note	<code>waitForApplication</code> is used when <code>YarnClientSchedulerBackend</code> is started.
------	--

asyncMonitorApplication

```
asyncMonitorApplication(): MonitorThread
```

`asyncMonitorApplication` internal method creates a separate daemon `MonitorThread` thread called "Yarn application state monitor".

Note	<code>asyncMonitorApplication</code> does not start the daemon thread.
------	--

MonitorThread

`MonitorThread` internal class is to monitor a Spark application submitted to a YARN cluster in client deploy mode.

When started, `MonitorThread` requests `Client`> to [monitor a Spark application](#) (with `logApplicationReport` disabled).

Note

`Client.monitorApplication` is a blocking operation and hence it is wrapped in `MonitorThread` to be executed on 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 SchedulerBackend Contract .
------	--

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.

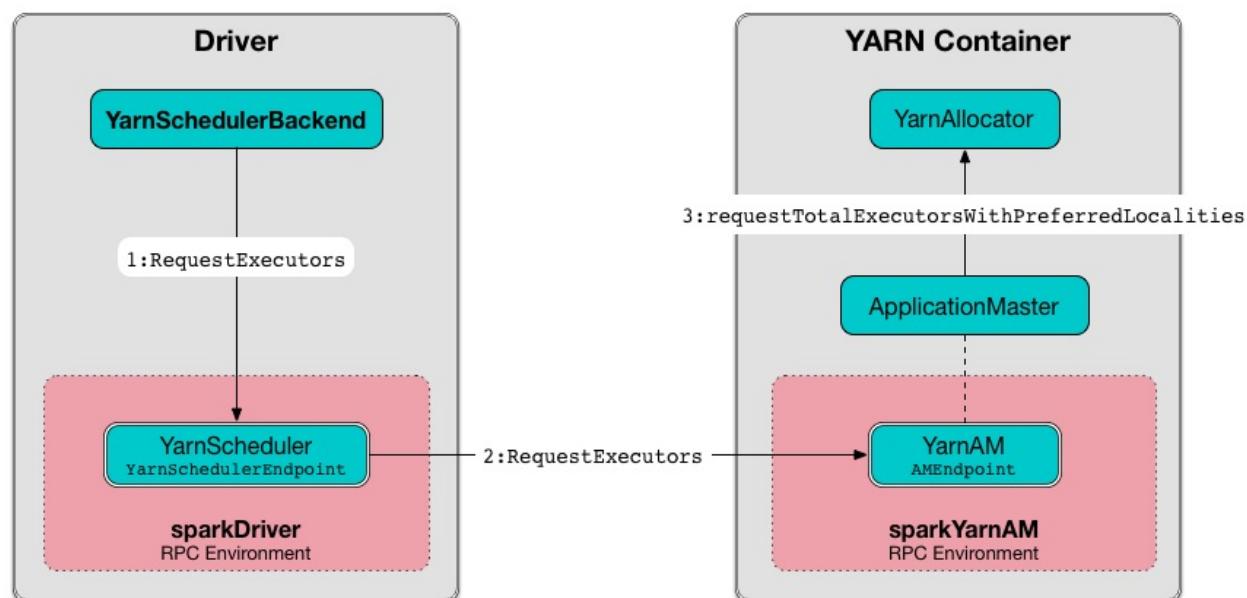


Figure 1. RequestExecutors Message Flow (client deploy mode)

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... FIXME
------	--

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... FIXME
------	---

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 — YARN Resource Container Allocator

`YarnAllocator` requests resources from a YARN cluster (in a form of containers from YARN ResourceManager) and manages the container allocations by allocating them to Spark executors and releasing them when no longer needed by a Spark application.

`YarnAllocator` manages resources using `AMRMClient` (that `YarnRMClient` passes in when creating a `YarnAllocator`).

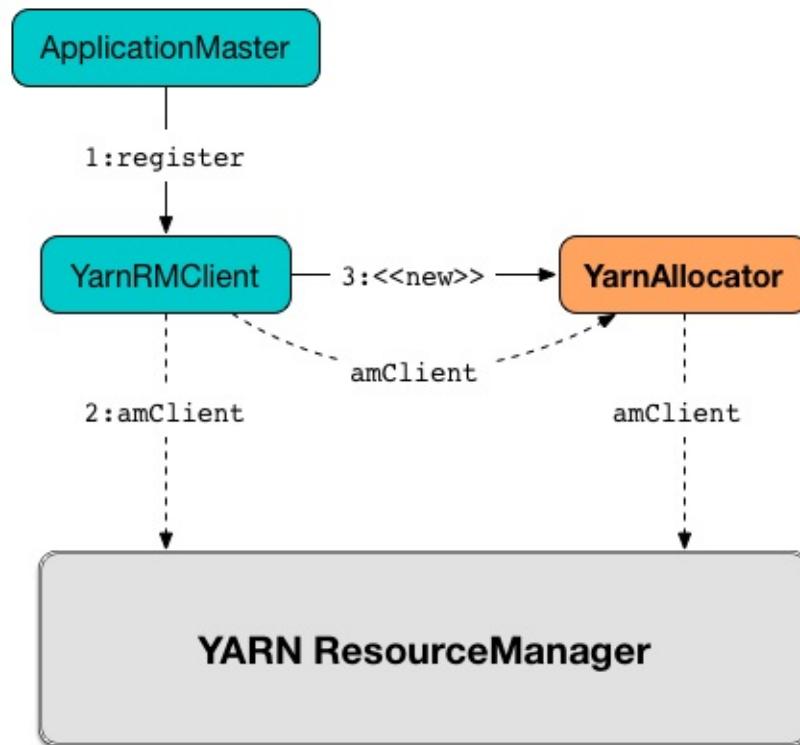


Figure 1. Creating YarnAllocator

`YarnAllocator` is a part of the internal state of `ApplicationMaster` (via the internal `allocator` reference).

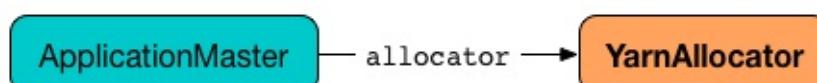


Figure 2. ApplicationMaster uses YarnAllocator (via allocator attribute)

`YarnAllocator` later launches Spark executors in allocated YARN resource containers.

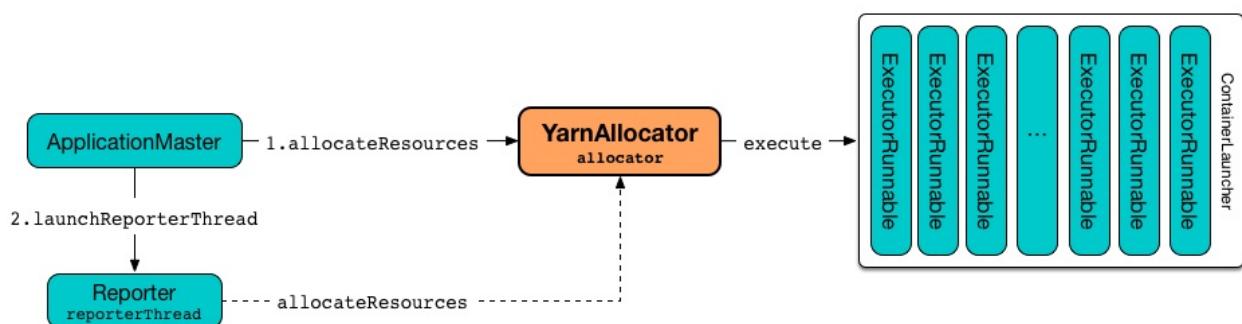


Figure 3. YarnAllocator Runs ExecutorRunnables in Allocated YARN Containers

Table 1. YarnAllocator's Internal Registries and Counters

Name	Description
resource	<p>The YARN Resource that sets capacity requirement (i.e. memory and virtual cores) of a single executor.</p> <p>NOTE: <code>Resource</code> models a set of computer resources in the cluster. Currently both memory and virtual CPU cores (vcores).</p> <p>Created when <code>YarnAllocator</code> is created and is the sum <code>executorMemory</code> and <code>memoryOverhead</code> for the amount memory and <code>executorCores</code> for the number of virtual cores.</p>
executorIdCounter	<p>Used to set executor id when launching Spark executor allocated YARN resource containers.</p> <p>Set to the last allocated executor id (received through a RPC system when <code>YarnAllocator</code> is created).</p>
targetNumExecutors	<p>Current desired total number of executors (as YARN resource containers).</p> <p>Set to the initial number of executors when <code>YarnAllocator</code> is created.</p> <p><code>targetNumExecutors</code> is eventually reached after <code>YarnAllocator</code> updates YARN container allocation requests.</p> <p>May later be changed when <code>YarnAllocator</code> is requested for total number of executors given locality preferences.</p> <p>Used when requesting missing resource containers and launching Spark executors in the allocated resource containers.</p>
numExecutorsRunning	<p>Current number of...FIXME</p> <p>Used to update YARN container allocation requests and get the current number of executors running.</p>

	Incremented when launching Spark executors in allocat YARN resource containers and decremented when releasing a resource container for a Spark executor.
currentNodeBlacklist	List of...FIXME
releasedContainers	<p>Unneeded containers that are of no use anymore by the globally unique identifier ContainerId (for a Container in the cluster).</p> <p>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.</p>
allocatedHostToContainersMap	Lookup table
allocatedContainerToHostMap	Lookup Table
pendingLossReasonRequests	
releasedExecutorLossReasons	
executorIdToContainer	
numUnexpectedContainerRelease	
containerIdToExecutorId	
hostToLocalTaskCounts	Lookup table
failedExecutorsTimeStamps	
executorMemory	
memoryOverhead	
executorCores	
launchContainers	
labelExpression	
nodeLabelConstructor	
containerPlacementStrategy	
launcherPool	ContainerLauncher Thread Pool

	<p>Number of locality-aware tasks to be used as container placement hint when <code>YarnAllocator</code> is requested for executors given locality preferences.</p> <p><code>numLocalityAwareTasks</code></p> <p>Set to <code>0</code> when <code>YarnAllocator</code> is created.</p> <p>Used as an input to <code>containerPlacementStrategy.localityOfRequestedContainer</code> when <code>YarnAllocator</code> updates YARN container allocation requests.</p>
Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.deploy.yarn.YarnAllocator</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.YarnAllocator=DEBUG</pre> <p>Refer to Logging.</p>

Creating YarnAllocator Instance

`YarnAllocator` takes the following when created:

1. `driverUrl`
2. `driverRef` — `RpcEndpointRef` to the driver's [FIXME](#)
3. [YarnConfiguration](#)
4. `sparkConf` — [SparkConf](#)
5. `amClient` [AMRMClient](#) for `ContainerRequest`
6. `ApplicationAttemptId`
7. `SecurityManager`
8. `localResources` — `Map[String, LocalResource]`

All the input parameters for `YarnAllocator` (but `appAttemptId` and `amClient`) are passed directly from the input parameters of `YarnRMClient`.

`YarnAllocator` sets the `org.apache.hadoop.yarn.util.RackResolver` logger to `WARN` (unless set to some log level already).

`YarnAllocator` initializes the internal registries and counters.

It sets the following internal counters:

- `numExecutorsRunning` to 0
- `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?
---------	--

getNumExecutorsRunning Method

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

updateInternalState Method

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

killExecutor Method

Caution

[FIXME](#)

Specifying Current Total Number of Executors with Locality Preferences

— `requestTotalExecutorsWithPreferredLocalities` Method

```
requestTotalExecutorsWithPreferredLocalities(
    requestedTotal: Int,
    localityAwareTasks: Int,
    hostToLocalTaskCount: Map[String, Int],
    nodeBlacklist: Set[String]): Boolean
```

`requestTotalExecutorsWithPreferredLocalities` returns whether the [current desired total number of executors](#) is different than the input `requestedTotal`.

Note

`requestTotalExecutorsWithPreferredLocalities` should instead have been called `shouldRequestTotalExecutorsWithPreferredLocalities` since it answers the question whether to request new 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](#) you should see the following INFO message in the logs:

```
INFO YarnAllocator: Driver requested a total number of [requestedTotal] executor(s).
```

`requestTotalExecutorsWithPreferredLocalities` saves the input `requestedTotal` to be the [current desired total number of executors](#).

`requestTotalExecutorsWithPreferredLocalities` updates blacklist information to YARN ResourceManager for this application in order to avoid allocating new Containers on the problematic nodes.

Caution

[FIXME](#) Describe the blacklisting

Note

`requestTotalExecutorsWithPreferredLocalities` is executed in response to [RequestExecutors](#) message to [ApplicationMaster](#).

Adding or Removing Container Requests to Launch Executors — `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.

Note

`updateResourceRequests` is used when `YarnAllocator` requests new resource containers.

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 longer needed)
```

It computes locality of requested containers (based on the internal `numLocalityAwareTasks`, `hostToLocalTaskCounts` and `allocatedHostToContainersMap` lookup table).

Caution	<code>FIXME Review containerPlacementStrategy.localityOfRequestedContainers + the code that follows.</code>
---------	---

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.
```

Handling Allocated Containers for Executors — `handleAllocatedContainers` Internal Method

```
handleAllocatedContainers(allocatedContainers: Seq[Container]): Unit
```

`handleAllocatedContainers` handles allocated YARN containers, i.e. runs Spark executors on matched containers or releases unneeded containers.

Note	A YARN Container represents an allocated resource in the cluster. The allocated Container is always on a single node and has a unique <code>containerId</code> . It has a specific amount of Resource allocated.
------	--

Internally, `handleAllocatedContainers` matches requests to host, rack, and any host (a container allocation).

If `handleAllocatedContainers` did not manage to allocate some containers, you should see the following DEBUG message in the logs:

```
DEBUG Releasing [size] unneeded containers that were allocated to us
```

`handleAllocatedContainers` releases the unneeded containers (if there are any).

`handleAllocatedContainers` runs the allocated and matched containers.

You should see the following INFO message in the logs:

```
INFO Received [allocatedContainersSize] containers from YARN, launching executors on [containersToUseSize] of them.
```

Note

`handleAllocatedContainers` is used exclusively when `YarnAllocator` allocates YARN resource containers for Spark executors.

Running ExecutorRunnables (with CoarseGrainedExecutorBackends) in Allocated YARN Resource Containers — `runAllocatedContainers` Internal Method

```
runAllocatedContainers(containersToUse: ArrayBuffer[Container]): Unit
```

`runAllocatedContainers` traverses the YARN Container collection (as the input `containersToUse`) and schedules execution of `ExecutorRunnables` per YARN container on `ContainerLauncher` thread pool.

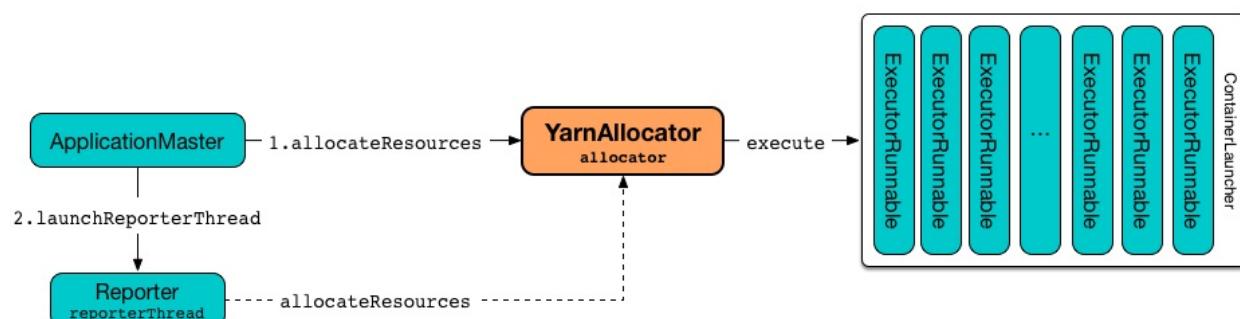


Figure 4. YarnAllocator Runs ExecutorRunnables in Allocated YARN Containers

Note

A `Container` in YARN represents allocated resources (memory and cores) in the cluster.

Internally, `runAllocatedContainers` increments `executorIdCounter` internal counter.

Note

`runAllocatedContainers` asserts that the amount of memory of a container not less than the [requested memory for executors](#). And only memory!

You should see the following INFO message in the logs:

```
INFO YarnAllocator: Launching container [containerId] for on host [executorHostname]
```

`runAllocatedContainers` checks if the [number of executors running](#) is less than the [number of required executors](#).

If there are executors still missing (and `runAllocatedContainers` is not in [testing mode](#)), `runAllocatedContainers` schedules execution of a [ExecutorRunnable](#) on [ContainerLauncher thread pool](#) and [updates internal state](#). When executing a [ExecutorRunnable](#) `runAllocatedContainers` first creates a [ExecutorRunnable](#) and starts it.

When `runAllocatedContainers` catches a non-fatal exception and you should see the following ERROR message in the logs and immediately [releases the container](#) (using the internal [AMRMClient](#)).

```
ERROR Failed to launch executor [executorId] on container [containerId]
```

If `yarnAllocator` has reached [target number of executors](#), you should see the following INFO message in the logs:

```
INFO Skip launching executorRunnable as running Executors count: [numExecutorsRunning] reached target Executors count: [targetNumExecutors].
```

Note

`runAllocatedContainers` is used exclusively when `YarnAllocator` handles [allocated YARN containers](#).

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.

processCompletedContainers Method

```
processCompletedContainers(completedContainers: Seq[ContainerStatus]): Unit
```

`processCompletedContainers` accepts a collection of YARN's [ContainerStatus](#)'es.

`ContainerStatus` represents the current status of a YARN Container and provides details such as:

- Note
- Id
 - State
 - Exit status of a completed container.
 - Diagnostic message for a failed container.

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	FIXME 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]. Diag  
nostics: [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.

Requesting and Allocating YARN Resource Containers to Spark Executors (and Cancelling Outstanding Containers) — `allocateResources` Method

```
allocateResources(): Unit
```

`allocateResources` claims new resource containers from `YARN ResourceManager` and cancels any outstanding resource container requests.

Note

In YARN, you first have to submit requests for YARN resource containers to `YARN ResourceManager` (using `AMRMClient.addContainerRequest`) before claiming them by calling `AMRMClient.allocate`.

Internally, `allocateResources` submits requests for new containers and cancels 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
application_1469955900130_0001	jacek	Spark shell	SPARK	default	Sun Jul 31 11:05:33 +0200 2016	N/A	RUNNING	UNDEFINED	<div style="width: 10%;">10%</div>	ApplicationMaster	0

Figure 5. 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 resource containers.

`allocateResources` gets the list of completed containers' statuses from YARN ResourceManager.

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].
```

Note

`allocateResources` is used when ApplicationMaster is registered to the YARN ResourceManager and launches progress Reporter thread.

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	FIXME
---------	-----------------------

Further reading or watching

- (video training) [Introduction to Hadoop Security](#)
- [Hadoop Security](#)
- [Kerberos: The Definitive Guide](#)

ConfigurableCredentialManager

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

Creating ConfigurableCredentialManager Instance

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

credentialRenewer Method

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

Obtaining Security Tokens from Credential Providers — **obtainCredentials** Method

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

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](#)

`YarnSparkHadoopUtil` can only be created when [SPARK_YARN_MODE](#) flag is enabled.

Note	<code>YarnSparkHadoopUtil</code> belongs to <code>org.apache.spark.deploy.yarn</code> package.
------	--

Enable `DEBUG` logging level for
`org.apache.spark.deploy.yarn.YarnSparkHadoopUtil` logger to see what happens inside.

Tip

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

```
log4j.logger.org.apache.spark.deploy.yarn.YarnSparkHadoopUtil=DEBUG
```

Refer to [Logging](#).

startCredentialUpdater Method

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

Getting YarnSparkHadoopUtil Instance — get Method

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

addPathToEnvironment Method

```
addPathToEnvironment(env: HashMap[String, String], key: String, value: String): Unit
```

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

startExecutorDelegationTokenRenewer

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

stopExecutorDelegationTokenRenewer

Caution

FIXME

getApplicationAclsForYarn Method

Caution

FIXME

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.

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).

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`.

Calculating Initial Number of Executors — getInitialTargetExecutorNumber Method

```
getInitialTargetExecutorNumber(conf: SparkConf, numExecutors: Int = 2): 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 <code>DEFAULT_NUMBER_EXECUTORS</code>) is 2.
------	--

With [dynamic allocation enabled](#), `getInitialTargetExecutorNumber` is `spark.dynamicAllocation.initialExecutors` or `spark.dynamicAllocation.minExecutors` to fall back to 0 if the others are undefined.

With [dynamic allocation disabled](#), `getInitialTargetExecutorNumber` is the value of `spark.executor.instances` property or `SPARK_EXECUTOR_INSTANCES` environment variable, or the default value (of the input parameter `numExecutors`) 2.

Note	<code>getInitialTargetExecutorNumber</code> is used to calculate <code>totalExpectedExecutors</code> to start Spark on YARN in client or cluster modes.
------	---

Settings

The following settings (aka system properties) are specific to Spark on YARN.

Spark Property	Default Value	Description
<code>spark.yarn.am.port</code>	0	Port that ApplicationMaster uses to create the sparkYarnAM RPC environment.
<code>spark.yarn.am.waitTime</code>	100s	In milliseconds unless the unit is specified.
<code>spark.yarn.app.id</code>		
<code>spark.yarn.executor.memoryOverhead</code>	10% of spark.executor.memory but not less than 384	(in MiBs) is an optional setting for the executor memory overhead (in addition to spark.executor.memory when requesting YARN resource containers from a YARN cluster). Used when Client calculates memory overhead for executors

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	FIXME
---------	-----------------------

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.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	<code>spark.yarn.jar</code> 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.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	FIXME
---------	-----------------------

```
→ spark git:(master) ✘ ./bin/spark-submit --help
...
Usage: spark-submit --kill [submission ID] --master [spark://...]
Usage: spark-submit --status [submission ID] --master [spark://...]
...
```

Refer to [Command-line Options](#) in `spark-submit`.

Round-robin Scheduling Across Nodes

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	FIXME
---------	-----------------------

```
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](#).

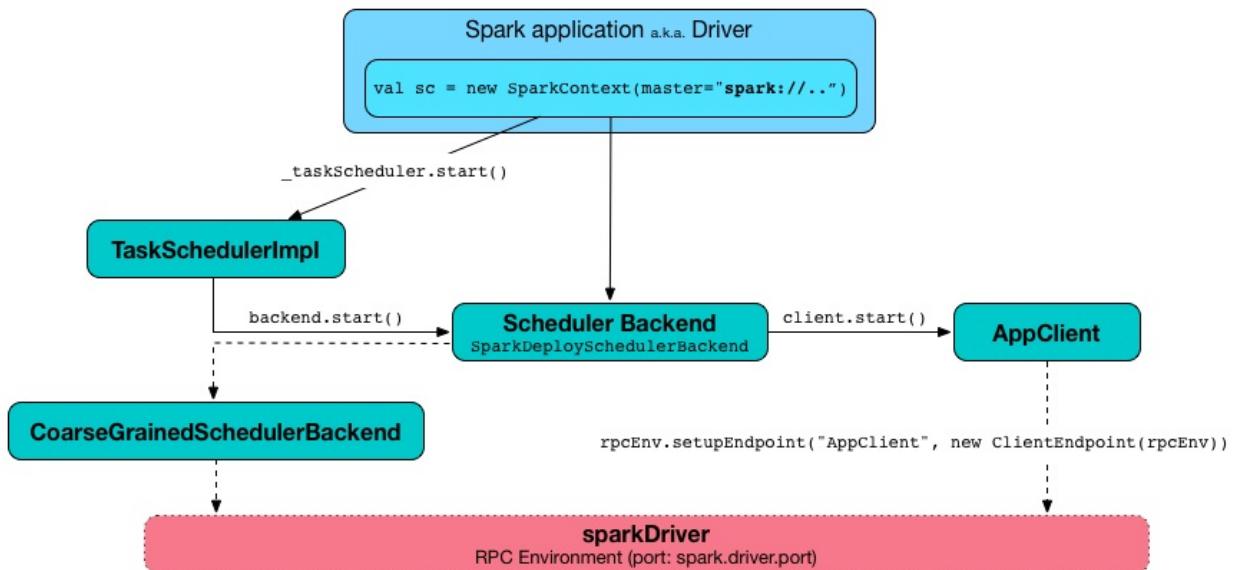


Figure 1. `SparkDeploySchedulerBackend.start()` (while `SparkContext` starts)

Caution	FIXME <code>AppClientListener</code> & <code>LauncherBackend</code> & <code>ApplicationDescription</code>
---------	---

It uses [AppClient](#) to talk to executors.

AppClient

`AppClient` is an interface to allow Spark applications to talk to a Standalone cluster (using a RPC Environment). It takes an RPC Environment, a collection of master URLs, a `ApplicationDescription`, and a `AppClientListener`.

It is solely used by [SparkDeploySchedulerBackend](#).

`AppClient` registers **AppClient** RPC endpoint (using `ClientEndpoint` class) to a given RPC Environment.

`AppClient` uses a daemon cached thread pool (`askAndReplyThreadPool`) with threads' name in the format of `appclient-receive-and-reply-threadpool-ID`, where `ID` is a unique integer for asynchronous asks and replies. It is used for requesting executors (via `RequestExecutors` message) and kill executors (via `KillExecutors`).

`sendToMaster` sends one-way `ExecutorStateChanged` and `UnregisterApplication` messages to master.

Initialization - `AppClient.start()` method

When AppClient starts, `AppClient.start()` method is called that merely registers [AppClient RPC Endpoint](#).

Others

- killExecutors
- start
- stop

AppClient RPC Endpoint

[AppClient](#) RPC endpoint is started as part of [AppClient's initialization](#) (that is in turn part of [SparkDeploySchedulerBackend's initialization](#), i.e. the scheduler backend for [Spark Standalone](#)).

It is a [ThreadSafeRpcEndpoint](#) that knows about the RPC endpoint of the primary active standalone Master (there can be a couple of them, but only one can be active and hence primary).

When it starts, it sends [RegisterApplication](#) message to register an application and itself.

RegisterApplication RPC message

An AppClient registers the Spark application to a single master (regardless of [the number of the standalone masters given in the master URL](#)).

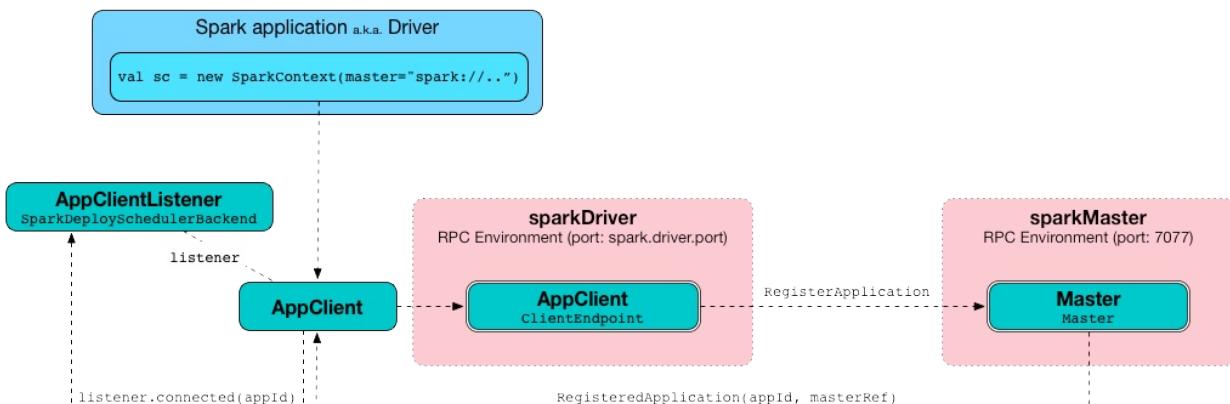


Figure 2. AppClient registers application to standalone Master

It uses a dedicated thread pool **appclient-register-master-threadpool** to asynchronously send `RegisterApplication` messages, one per standalone master.

```
INFO AppClient$ClientEndpoint: Connecting to master spark://localhost:7077...
```

An AppClient tries connecting to a standalone master 3 times every 20 seconds per master before giving up. They are not configurable parameters.

The appclient-register-master-threadpool thread pool is used until the registration is finished, i.e. AppClient is connected to the primary standalone Master or the registration fails. It is then `shutdown`.

RegisteredApplication RPC message

`RegisteredApplication` is a one-way message from the primary master to confirm successful application registration. It comes with the application id and the master's RPC endpoint reference.

The `AppClientListener` gets notified about the event via `listener.connected(appId)` with `appId` being an application id.

ApplicationRemoved RPC message

`ApplicationRemoved` is received from the primary master to inform about having removed the application. AppClient RPC endpoint is stopped afterwards.

It can come from the standalone Master after a kill request from Web UI, application has finished properly or the executor where the application was still running on has been killed, failed, lost or exited.

ExecutorAdded RPC message

`ExecutorAdded` is received from the primary master to inform about...[FIXME](#)

Caution	FIXME the message
---------	-----------------------------------

```
INFO Executor added: %s on %s (%s) with %d cores
```

ExecutorUpdated RPC message

`ExecutorUpdated` is received from the primary master to inform about...[FIXME](#)

Caution	FIXME the message
---------	-----------------------------------

```
INFO Executor updated: %s is now %s%
```

MasterChanged RPC message

`MasterChanged` is received from the primary master to inform about...[FIXME](#)

Caution	FIXME the message
---------	-----------------------------------

INFO Master has changed, new master is at

StopAppClient RPC message

`StopAppClient` is a reply-response message from the `SparkDeploySchedulerBackend` to stop the `AppClient` after the `SparkContext` has been stopped (and so should the running application on the standalone cluster).

It stops the `AppClient` RPC endpoint.

RequestExecutors RPC message

`RequestExecutors` is a reply-response message from the `SparkDeploySchedulerBackend` that is passed on to the master to request executors for the application.

KillExecutors RPC message

`KillExecutors` is a reply-response message from the `SparkDeploySchedulerBackend` that is passed on to the master to kill executors assigned to the application.

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
```

Creating Master Instance

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

startRpcEnvAndEndpoint Method

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

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.

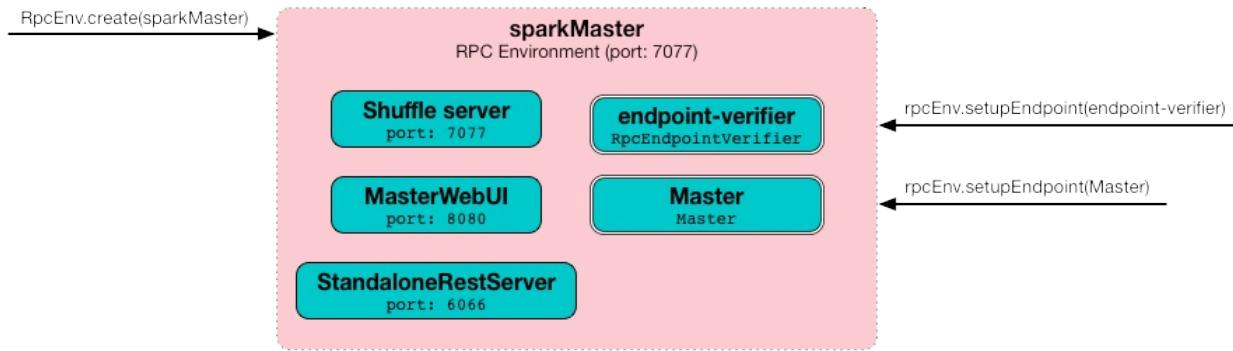


Figure 1. sparkMaster - the RPC Environment for Spark Standalone's master
Master endpoint is a [ThreadSafeRpcEndpoint](#) and [LeaderElectable](#) (see [Leader Election](#)).

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**

	FIXME
Caution	<ul style="list-style-type: none"> • Review <code>org.apache.spark.metrics.MetricsConfig</code> • How to access the metrics for master? See <code>Master#onStart</code> • Review <code>masterMetricsSystem</code> and <code>applicationMetricsSystem</code>

REST Server

The standalone Master starts the REST Server service for alternative application submission that is supposed to work across Spark versions. It is enabled by default (see [spark.master.rest.enabled](#)) and used by [spark-submit](#) for the [standalone cluster mode](#), i.e. `-deploy-mode is cluster`.

`RestSubmissionClient` is the client.

The server includes a JSON representation of `SubmitRestProtocolResponse` in the HTTP body.

The following INFOs show up when the Master Endpoint starts up (`Master#onStart` is called) with REST Server enabled:

```
INFO Utils: Successfully started service on port 6066.
INFO StandaloneRestServer: Started REST server for submitting applications on port 6066
```

Recovery Mode

A standalone Master can run with **recovery mode** enabled and be able to recover state among the available swarm of masters. By default, there is no recovery, i.e. no persistence and no election.

Note

Only a master can schedule tasks so having one always on is important for cases where you want to launch new tasks. Running tasks are unaffected by the state of the master.

Master uses `spark.deploy.recoveryMode` to set up the recovery mode (see [spark.deploy.recoveryMode](#)).

The Recovery Mode enables [election of the leader master](#) among the masters.

Tip

Check out the exercise [Spark Standalone - Using ZooKeeper for High-Availability of Master](#).

Leader Election

Master endpoint is `LeaderElectable`, i.e. [FIXME](#)

Caution

[FIXME](#)

RPC Messages

Master communicates with drivers, executors and configures itself using **RPC messages**.

The following message types are accepted by master (see `Master#receive` or `Master#receiveAndReply` methods):

- `ElectedLeader` for [Leader Election](#)
- `CompleteRecovery`
- `RevokedLeadership`
- [RegisterApplication](#)
- `ExecutorStateChanged`
- `DriverStateChanged`
- `Heartbeat`
- `MasterChangeAcknowledged`
- `WorkerSchedulerStateResponse`
- `UnregisterApplication`
- `CheckForWorkerTimeOut`
- `RegisterWorker`
- `RequestSubmitDriver`
- `RequestKillDriver`
- `RequestDriverStatus`
- `RequestMasterState`
- `BoundPortsRequest`
- `RequestExecutors`
- `KillExecutors`

RegisterApplication event

A **RegisterApplication** event is sent by [AppClient](#) to the standalone Master. The event holds information about the application being deployed (`ApplicationDescription`) and the driver's endpoint reference.

`ApplicationDescription` describes an application by its name, maximum number of cores, executor's memory, command, appUiUrl, and user with optional eventLogDir and eventLogCodec for Event Logs, and the number of cores per executor.

Caution

FIXME Finish

A standalone Master receives `RegisterApplication` with a `ApplicationDescription` and the driver's `RpcEndpointRef`.

```
INFO Registering app " + description.name
```

Application ids in Spark Standalone are in the format of `app-[yyyyMMddHHmmss]-[4-digit nextAppNumber]`.

Master keeps track of the number of already-scheduled applications (`nextAppNumber`).

`ApplicationDescription` (AppClient) → `ApplicationInfo` (Master) - application structure enrichment

```
ApplicationSource metrics + applicationMetricsSystem
```

```
INFO Registered app " + description.name + " with ID " + app.id
```

Caution

FIXME `persistenceEngine.addApplication(app)`

`schedule()` schedules the currently available resources among waiting apps.

FIXME When is `schedule()` method called?

It's only executed when the Master is in `RecoveryState.ALIVE` state.

Worker in `workerState.ALIVE` state can accept applications.

A driver has a state, i.e. `driver.state` and when it's in `DriverState.RUNNING` state the driver has been assigned to a worker for execution.

LaunchDriver RPC message

Warning

It seems a dead message. Disregard it for now.

A **LaunchDriver** message is sent by an active standalone Master to a worker to launch a driver.

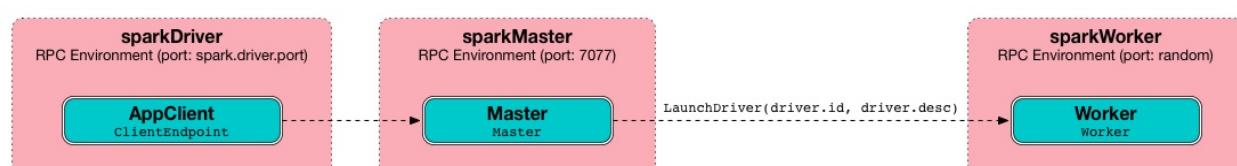


Figure 2. Master finds a place for a driver (posts LaunchDriver)

You should see the following INFO in the logs right before the message is sent out to a worker:

```
INFO Launching driver [driver.id] on worker [worker.id]
```

The message holds information about the id and name of the driver.

A driver can be running on a single worker while a worker can have many drivers running.

When a worker receives a `LaunchDriver` message, it prints out the following INFO:

```
INFO Asked to launch driver [driver.id]
```

It then creates a `DriverRunner` and starts it. It starts a separate JVM process.

Workers' free memory and cores are considered when assigning some to waiting drivers (applications).

Caution	FIXME Go over <code>waitingDrivers</code> ...
---------	---

DriverRunner

Warning	It seems a dead piece of code. Disregard it for now.
---------	--

A `DriverRunner` manages the execution of one driver.

It is a `java.lang.Process`

When started, it spawns a thread `DriverRunner` for `[driver.id]` that:

1. Creates the working directory for this driver.
2. Downloads the user jar [FIXME](#) `downloadUserJar`
3. Substitutes variables like `WORKER_URL` or `USER_JAR` that are set when...[FIXME](#)

Internals of org.apache.spark.deploy.master.Master

Tip	You can debug a Standalone master using the following command:
-----	--

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005 -cp /Use
```

	The above command suspends (<code>suspend=y</code>) the process until a JPDA debugging cli
--	--

When `Master` starts, it first creates the [default SparkConf configuration](#) whose values it then overrides using [environment variables](#) and [command-line options](#).

A fully-configured master instance requires `host`, `port` (default: `7077`), `webUiPort` (default: `8080`) settings defined.

Tip

When in troubles, consult [Spark Tips and Tricks](#) document.

It starts [RPC Environment](#) with necessary endpoints and lives until the RPC environment terminates.

Worker Management

Master uses `master-forward-message-thread` to schedule a thread every `spark.worker.timeout` to check workers' availability and remove timed-out workers.

It is that Master sends `CheckForWorkerTimeOut` message to itself to trigger verification.

When a worker hasn't responded for `spark.worker.timeout`, it is assumed dead and the following WARN message appears in the logs:

```
WARN Removing [worker.id] because we got no heartbeat in [spark.worker.timeout] seconds
```

System Environment Variables

Master uses the following system environment variables (directly or indirectly):

- `SPARK_LOCAL_HOSTNAME` - the custom host name
- `SPARK_LOCAL_IP` - the custom IP to use when `SPARK_LOCAL_HOSTNAME` is not set
- `SPARK_MASTER_HOST` (not `SPARK_MASTER_IP` as used in `start-master.sh` script above!) - the master custom host
- `SPARK_MASTER_PORT` (default: `7077`) - the master custom port
- `SPARK_MASTER_IP` (default: `hostname` command's output)
- `SPARK_MASTER_WEBUI_PORT` (default: `8080`) - the port of the master's WebUI. Overridden by `spark.master.ui.port` if set in the properties file.
- `SPARK_PUBLIC_DNS` (default: `hostname`) - the custom master hostname for WebUI's http URL and master's address.

- `SPARK_CONF_DIR` (default: `$SPARK_HOME/conf`) - the directory of the default properties file [spark-defaults.conf](#) from which all properties that start with `spark.` prefix are loaded.

Settings

Caution	FIXME <ul style="list-style-type: none"> • Where are `RETAINED_`'s properties used?
----------------	--

Master uses the following properties:

- `spark.cores.max` (default: `0`) - total expected number of cores. 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](#).

Creating Worker Instance

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

startRpcEnvAndEndpoint Method

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

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.

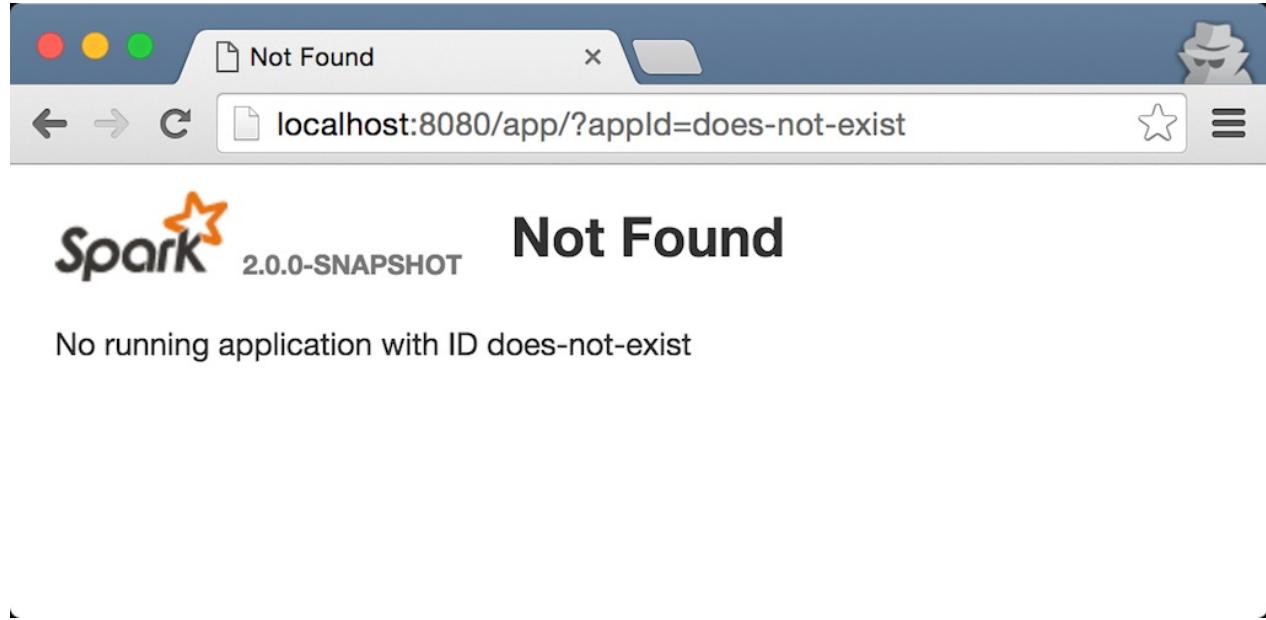


Figure 2. Application Not Found Page

Submission Gateways

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

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 Print Launch Command of Spark Scripts .
------	--

It has support for starting Tachyon using `--with-tachyon` command line option. It assumes `tachyon/bin/tachyon` command be available in Spark's home directory.

The script uses the following helper scripts:

- `sbin/spark-config.sh`
- `bin/load-spark-env.sh`
- `conf/spark-env.sh` contains environment variables of a Spark executable.

Ultimately, the script calls `sbin/spark-daemon.sh start` to kick off `org.apache.spark.deploy.master.Master` with parameter `1` and `--ip`, `--port`, and `--webui-port` [command-line options](#).

Command-line Options

You can use the following command-line options:

- `--host` or `-h` the hostname to listen on; overrides [SPARK_MASTER_HOST](#).
- `--ip` or `-i` (deprecated) the IP to listen on

- `--port` or `-p` - command-line version of `SPARK_MASTER_PORT` that overrides it.
- `--webui-port` - command-line version of `SPARK_MASTER_WEBUI_PORT` that overrides it.
- `--properties-file` (default: `$SPARK_HOME/conf/spark-defaults.conf`) - the path to a custom Spark properties file. Refer to [spark-defaults.conf](#).
- `--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. Refer to [spark-defaults.conf](#).
- `--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">• Operating Spark master• Starting Spark workers on node using sbin/start-slave.sh
Important	You can use the Spark Standalone cluster in the following ways: <ul style="list-style-type: none">• Use <code>spark-shell</code> with <code>--master MASTER_URL</code>• Use <code>SparkConf.setMaster(MASTER_URL)</code> in your Spark application 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.

The screenshot shows a web browser window titled "Spark Master at spark://japila.local:7077". The URL in the address bar is "localhost:8080". The page displays the following information:

- Spark Logo** 1.6.0-SNAPSHOT
- Spark Master at spark://japila.local:7077**
- URL:** spark://japila.local:7077
- REST URL:** spark://japila.local:6066 (cluster mode)
- Alive Workers:** 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

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 1. Master's web UI with no workers and applications

- 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`

- Check out master's web UI at <http://localhost:8080> to know the current setup - one worker.

The screenshot shows a web browser window titled "Spark Master at spark://japila.local:7077" with the URL "localhost:8080". The UI displays cluster statistics and lists of workers, running applications, and completed applications.

Cluster 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
worker-20150920142418-192.168.99.1-55334	192.168.99.1:55334	ALIVE	8 (0 Used)	15.0 GB (0.0 B Used)

Running Applications:

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.

The screenshot shows the Spark Master web UI at <http://localhost:8080>. The title bar says "Spark Master at spark://japila.local:7077". The UI displays the following statistics:

- 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 page displays system statistics and lists workers and applications.

System Statistics:

- URL: `spark://japila.local:7077`
- REST URL: `spark://japila.local:6066 (cluster mode)`
- Alive Workers: 1
- Cores in use: 2 Total, 0 Used
- Memory in use: 4.0 GB Total, 0.0 B Used
- Applications: 0 Running, 0 Completed
- Drivers: 0 Running, 0 Completed
- Status: ALIVE

Workers:

Worker Id	Address	State	Cores	Memory
<code>worker-20150920142418-192.168.99.1-55334</code>	192.168.99.1:55334	DEAD	8 (0 Used)	15.0 GB (0.0 B Used)
<code>worker-20150920144742-192.168.99.1-55538</code>	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)

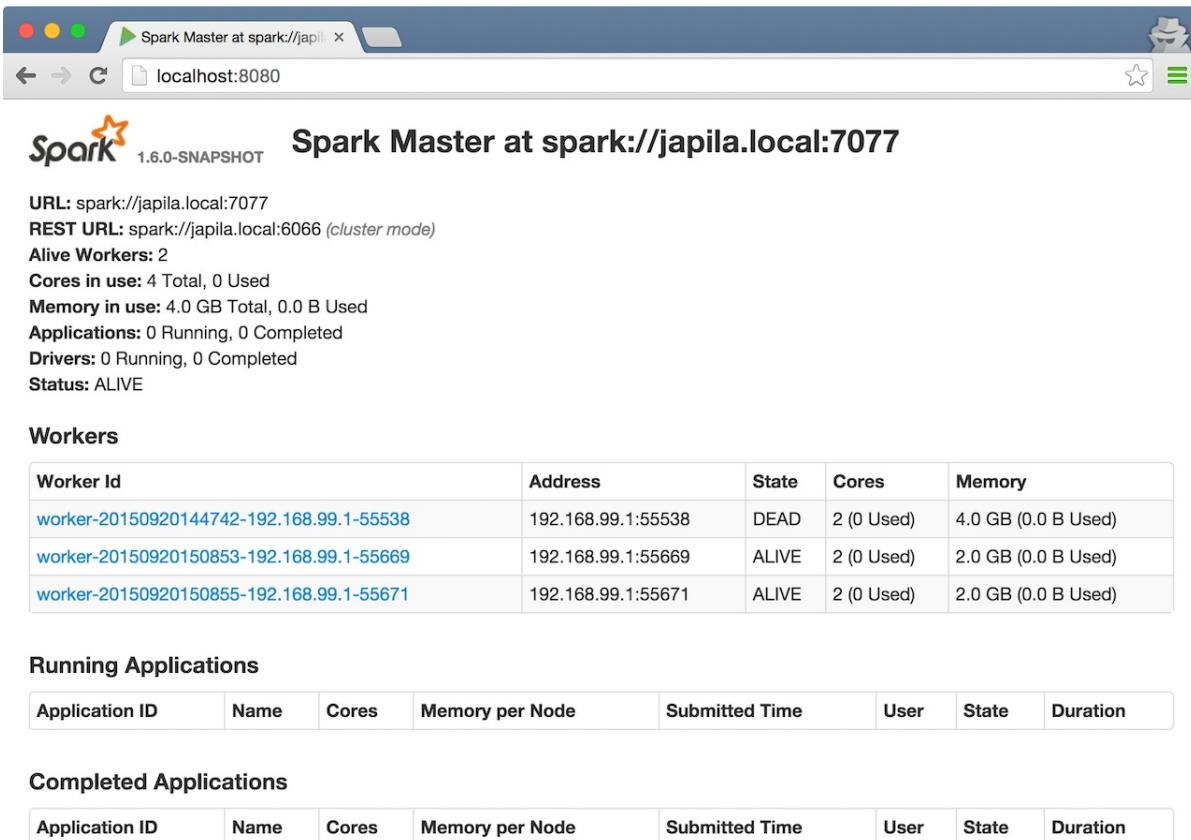
10. Start the workers.

```
./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
```

- 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 page 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

Note	<p>Use <code>jps</code> on master to see the instances given they all run on the same machine, e.g. <code>localhost</code>).</p> <pre>\$ jps 6580 Worker 4872 Master 6874 Jps 6539 Worker</pre>
------	--

- Stop all instances - the driver and the workers.

```
./sbin/stop-all.sh
```

StandaloneSchedulerBackend

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

Starting StandaloneSchedulerBackend — `start` Method

start(): Unit	
---------------	--

Spark on Mesos

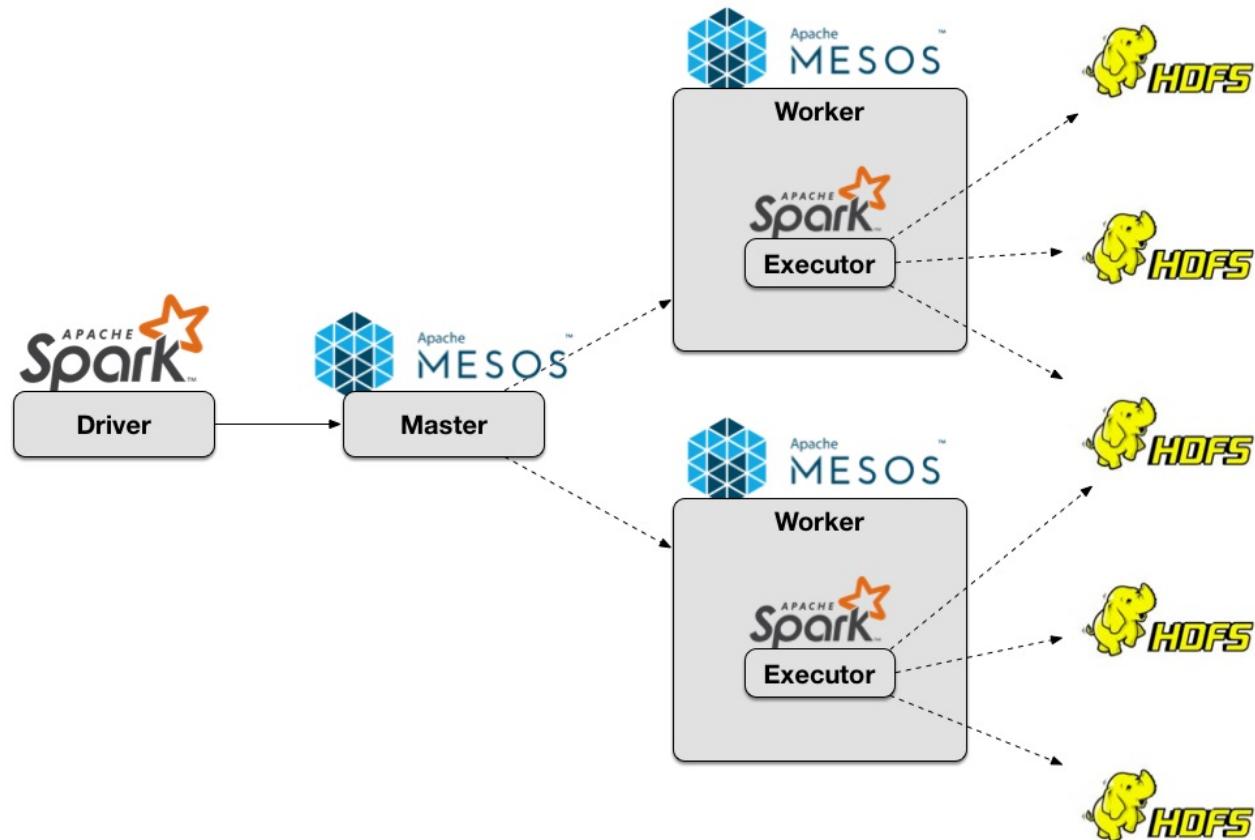


Figure 1. Spark on Mesos Architecture

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. On the left, there's a sidebar with various metrics and status indicators:

- Cluster:** (Unnamed)
- Server:** 127.0.0.1:5050
- Version:** 0.28.0
- Built:** 2 weeks ago by brew
- Started:** 2 minutes ago
- Elected:** 2 minutes ago
- LOG**
- Slaves**
 - Activated: 0
 - Deactivated: 0
- Tasks**
 - Staging: 0
 - Starting: 0
 - Running: 0
 - Killing: 0
 - Finished: 0
 - Killed: 0
 - Failed: 0
 - Lost: 0
- Resources**

	CPU	Mem	Disk
Total	0	0 B	0 B
Used	0	0 B	0 B
Offered	0	0 B	0 B
Idle	0	0 B	0 B

On the right, there are two tables:

- Active Tasks**: An empty table with columns ID, Name, State, Started ▾, and Host.
- Completed Tasks**: An empty table with columns ID, Name, State, Started ▾, Stopped, and Host.

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 interface with the Slaves tab selected. At the top, there's a navigation bar with tabs for Mesos, Frameworks, Slaves, and Offers. Below the navigation bar, it says "Master / Slaves".

On the left, there's a table header for "Slaves" with columns: ID ▾, Host, CPUs, Mem, Disk, Registered, and Re-Registered.

There is one row in the table:

ID ▾	Host	CPUs	Mem	Disk	Registered	Re-Registered
...aaea-5dffbfd44e5d-S0	192.168.1.4	8	15.0 GB	459.8 GB	a minute ago	

Figure 3. Mesos Management Console (Slaves tab) with one slave running

Important

You have to export `MESOS_NATIVE_JAVA_LIBRARY` environment variable before connecting to the Mesos cluster.

```
$ export MESOS_NATIVE_JAVA_LIBRARY=/usr/local/lib/libmesos.dylib
```

Note

The preferred approach to launch Spark on Mesos and to give the location of Spar binaries is through `spark.executor.uri` setting.

```
--conf spark.executor.uri=/Users/jacek/Downloads/spark-1.5.2-bin-hadoop2.6.tgz
```

For us, on a bleeding edge of Spark development, it is very convenient to use `spark.mesos.executor.home` setting, instead.

```
-c spark.mesos.executor.home=`pwd`
```

```
$ ./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	CPUs	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

Tip

Consult slave logs under `/tmp/mesos/slaves` when facing troubles.

Important

Ensure that the versions of Spark of `spark-shell` and as pointed out by `spark.executor.uri` are the same or compatible.

```
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`.

Settings

- `spark.cores.max` (default: `Int.MaxValue`) - maximum number of cores to acquire
- `spark.mesos.extra.cores` (default: `0`) - extra cores per slave (`extraCoresPerSlave`)
FIXME
- `spark.mesos.constraints` (default: (empty)) - offer constraints **FIXME**
`slaveOfferConstraints`
- `spark.mesos.rejectOfferDurationForUnmetConstraints` (default: `120s`) - reject offers with mismatched constraints in seconds
- `spark.mesos.executor.home` (default: `SPARK_HOME`) - the home directory of Spark for executors. It is only required when no `spark.executor.uri` is set.

MesosExternalShuffleClient

FIXME

(Fine)MesosSchedulerBackend

When `spark.mesos.coarse` is `false`, Spark on Mesos uses `MesosSchedulerBackend`

reviveOffers

It calls `mesosDriver.reviveOffers()`.

Caution**FIXME**

Settings

- `spark.mesos.coarse` (default: `true`) controls whether the scheduler backend for Mesos works in coarse- (`CoarseMesosSchedulerBackend`) or fine-grained mode (`MesosSchedulerBackend`).

	FIXME Review
Caution	<ul style="list-style-type: none"> • MesosClusterScheduler.scala • MesosExternalShuffleService

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

[FIXME](#)

executorLimitOption Property

`executorLimitOption` is an internal attribute to...[FIXME](#)

resourceOffers Method

Caution

[FIXME](#)

Note

`resourceOffers` is a part of Mesos' [Scheduler](#) callback interface to be implemented by frameworks' schedulers.

handleMatchedOffers Internal Method

Caution

[FIXME](#)

Note

`handleMatchedOffers` is used exclusively when MesosCoarseGrainedSchedulerBackend [resourceOffers](#).

buildMesosTasks Internal Method

Caution

[FIXME](#)

Note

`buildMesosTasks` is used exclusively when MesosCoarseGrainedSchedulerBackend launches Spark executors on accepted offers.

createCommand Method

Caution

[FIXME](#)

Note

`createCommand` is used exclusively when MesosCoarseGrainedSchedulerBackend builds Mesos tasks for given offers.

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.

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	FIXME Likely move to a separate page with references here.
---------	--

Securing Web UI

Tip	Read the official document Web UI .
-----	---

To secure Web UI you implement a security filter and use `spark.ui.filters` setting to refer to the class.

Examples of filters implementing basic authentication:

- [Servlet filter for HTTP basic auth](#)
- [neolitec/BasicAuthenticationFilter.java](#)

Data Sources in Spark

Spark can access data from many data sources, including [Hadoop Distributed File System \(HDFS\)](#), [Cassandra](#), [HBase](#), [S3](#) and many more.

Spark offers different APIs to read data based upon the content and the storage.

There are two groups of data based upon the content:

- binary
- text

You can also group data by the storage:

- [files](#)
- databases, e.g. [Cassandra](#)

Using Input and Output (I/O)

Caution

FIXME What are the differences between `textFile` and the rest methods in `SparkContext` like `newAPIHadoopRDD`, `newAPIHadoopFile`, `hadoopFile`, `hadoopRDD` ?

From [SPARK AND MERGED CSV FILES](#):

Spark is like Hadoop - uses Hadoop, in fact - for performing actions like outputting data to HDFS. You'll know what I mean the first time you try to save "all-the-data.csv" and are surprised to find a directory named all-the-data.csv/ containing a 0 byte _SUCCESS file and then several part-0000n files for each partition that took part in the job.

The read operation is lazy - it is [a transformation](#).

Methods:

- [SparkContext.textFile\(path: String, minPartitions: Int = defaultMinPartitions\): RDD\[String\]](#) reads a text data from a file from a remote HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI (e.g. sources in HBase or S3) at `path`, and automatically distributes the data across a Spark cluster as an RDD of Strings.
 - Uses Hadoop's [org.apache.hadoop.mapred.InputFormat](#) interface and file-based [org.apache.hadoop.mapred.FileInputFormat](#) class to read.
 - Uses the global Hadoop's `configuration` with all `spark.hadoop.xxx=yyy` properties mapped to `xxx=yyy` in the configuration.
 - `io.file.buffer.size` is the value of `spark.buffer.size` (default: `65536`).
 - Returns [HadoopRDD](#)
 - When using `textFile` to read an HDFS folder with multiple files inside, the number of partitions are equal to the number of HDFS blocks.
- What does `sc.binaryFiles` ?

URLs supported:

- `s3://...` or `s3n://...`
- `hdfs://...`
- `file://...;`

The general rule seems to be to use HDFS to read files multiple times with S3 as a storage for a one-time access.

Creating RDDs from Input

[FIXME](#)

```
sc.newAPIHadoopFile("filepath1, filepath2", classOf[NewTextInputFormat], classOf[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).

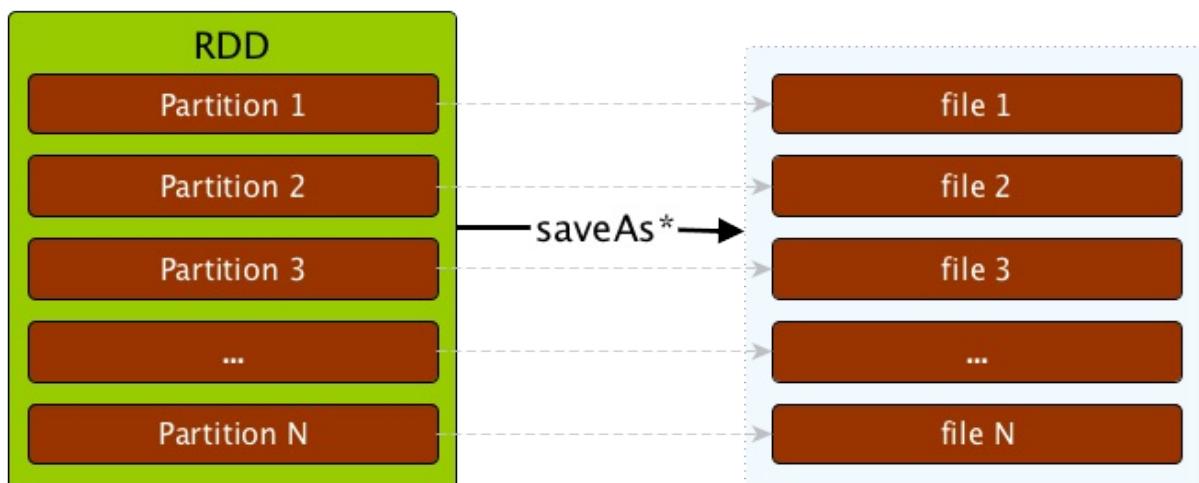


Figure 1. `saveAs` on RDD

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](#).

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>FIXME:</p> <ol style="list-style-type: none">1. Kafka Direct API in Spark Streaming2. Getting information on the current topic being consumed by each executor
---------	--

Couchbase Spark Connector

The Couchbase Spark Connector provides an open source integration between Apache Spark and Couchbase Server.

Tip	Read the official documentation in Spark Connector 1.2 .
Caution	FIXME Describe the features listed in the document and how Spark features contributed
Caution	FIXME 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 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.

Unified Memory Management

Unified Memory Management was introduced in [SPARK-10000: Consolidate storage and execution memory management](#).

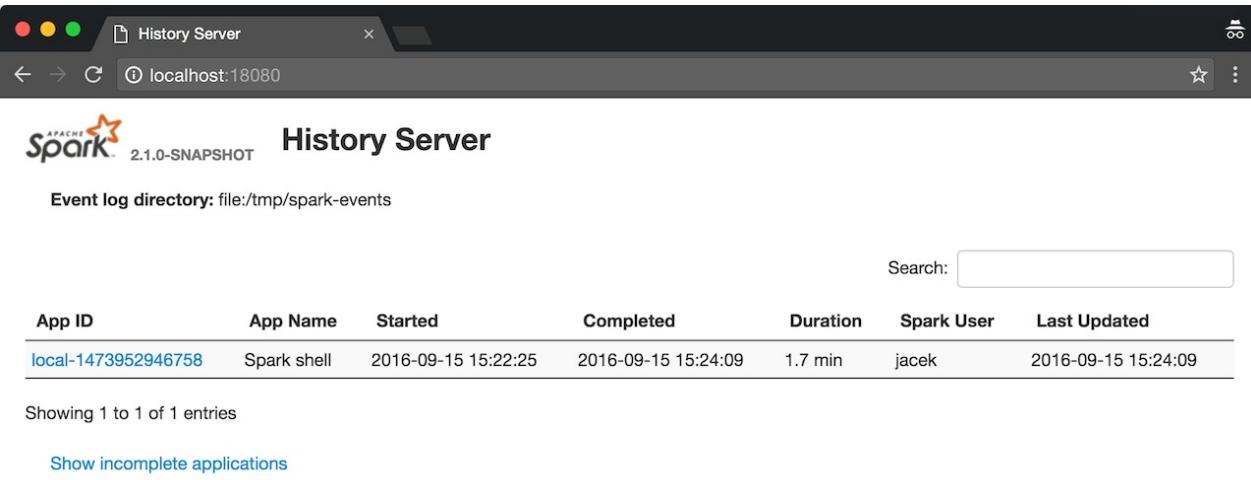
It uses the custom memory manager [UnifiedMemoryManager](#).

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](#)

Spark History Server

Spark History Server is the web UI for [completed](#) and running (aka *incomplete*) Spark applications. It is an extension of Spark's [web UI](#).



The screenshot shows a browser window titled "History Server" with the URL "localhost:18080". The page displays the Apache Spark logo and the text "History Server". Below this, it says "Event log directory: file:/tmp/spark-events". A search bar is present with the placeholder "Search:". A table lists one application entry:

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
local-1473952946758	Spark shell	2016-09-15 15:22:25	2016-09-15 15:24:09	1.7 min	jacek	2016-09-15 15:24:09

Below the table, it says "Showing 1 to 1 of 1 entries" and "Show incomplete applications".

Figure 1. History Server's web UI

Tip Enable collecting events in your Spark applications using [spark.eventLog.enabled](#) Spark property.

You can start History Server by executing [start-history-server.sh](#) shell script and stop it using [stop-history-server.sh](#).

`start-history-server.sh` accepts `--properties-file [propertiesFile]` command-line option that specifies the properties file with the custom [Spark properties](#).

```
$ ./sbin/start-history-server.sh --properties-file history.properties
```

If not specified explicitly, Spark History Server uses the default configuration file, i.e. [spark-defaults.conf](#).

Tip Enable `INFO` logging level for `org.apache.spark.deploy.history` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.deploy.history=INFO
```

Refer to [Logging](#).

Starting History Server— `start-history-server.sh` script

You can start a `HistoryServer` instance by executing `$SPARK_HOME/sbin/start-history-server.sh` script (where `SPARK_HOME` is the directory of your Spark installation).

```
$ ./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
```

Internally, `start-history-server.sh` script starts [org.apache.spark.deploy.history.HistoryServer](#) standalone application for execution (using `spark-daemon.sh` shell script).

```
$ ./bin/spark-class org.apache.spark.deploy.history.HistoryServer
```

Tip	Using the more explicit approach with <code>spark-class</code> to start Spark History Server could be easier to trace execution by seeing the logs printed out to the standard output and hence terminal directly.
-----	--

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 <code>initSecurity</code>
---------	--

It creates a `SecurityManager`.

It creates a [ApplicationHistoryProvider](#) (by reading `spark.history.provider`).

It [creates a `HistoryServer`](#) and requests it to bind to `spark.history.ui.port` port.

Tip	The host's IP can be specified using <code>SPARK_LOCAL_IP</code> environment variable (defaults to <code>0.0.0.0</code>).
-----	--

You should see the following INFO message in the logs:

```
INFO HistoryServer: Bound HistoryServer to [host], and started at [webUrl]
```

It registers a shutdown hook to call `stop` on the `HistoryServer` instance.

Tip

Use [stop-history-server.sh](#) shell script to stop a running History Server.

Stopping History Server— `stop-history-server.sh` script

You can stop a running instance of `HistoryServer` using `$SPARK_HOME/sbin/stop-history-server.sh` shell script.

```
$ ./sbin/stop-history-server.sh
stopping org.apache.spark.deploy.history.HistoryServer
```

Settings

Table 1. Spark Properties

Setting	Default Value
spark.history.ui.port	18080
spark.history.fs.logDirectory	file:/tmp/spark-events
spark.history.retainedApplications	50
spark.history.ui.maxApplications	(unbounded)
spark.history.kerberos.enabled	false
spark.history.kerberos.principal	(empty)
spark.history.kerberos.keytab	(empty)
spark.history.provider	org.apache.spark.deploy.history.FsHistoryProvider

HistoryServer

HistoryServer extends WebUI abstract class.

Tip

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 :

```
log4j.logger.org.apache.spark.deploy.history.HistoryServer=INFO
```

Refer to [Logging](#).

Starting HistoryServer Standalone Application — main Method

Caution

[FIXME](#)

Creating HistoryServer Instance

Caution

[FIXME](#)

Initializing HistoryServer — initialize Method

Caution

[FIXME](#)

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	FIXME 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	FIXME
---------	-----------------------

Creating SQLHistoryListener Instance

`SQLHistoryListener` is created using a (`private[sql]`) `SQLHistoryListenerFactory` class (which is `SparkHistoryListenerFactory`).

The `SQLHistoryListenerFactory` class is registered when `SparkUI` creates a web UI for `History Server` 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.

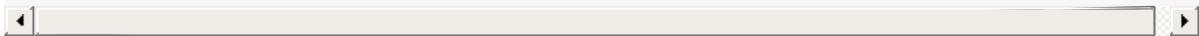
```
stop(): Unit
```

- `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
```



HistoryServerArguments

`HistoryServerArguments` is the command-line parser for the [History Server](#).

When `HistoryServerArguments` is executed with a single command-line parameter it is assumed to be the event logs directory.

```
$ ./sbin/start-history-server.sh /tmp/spark-events
```

This is however deprecated since Spark 1.1.0 and you should see the following WARN message in the logs:

```
WARN HistoryServerArguments: Setting log directory through the command line is deprecated as of Spark 1.1.0. Please set this through spark.history.fs.logDirectory instead.
```

The same WARN message shows up for `--dir` and `-d` command-line options.

`--properties-file [propertiesFile]` command-line option specifies the file with the custom [Spark properties](#).

Note	When not specified explicitly, History Server uses the default configuration file, i.e. spark-defaults.conf .
------	---

Tip	Enable <code>WARN</code> logging level for <code>org.apache.spark.deploy.history.HistoryServerArguments</code> logger to see what happens inside.
-----	---

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

```
log4j.logger.org.apache.spark.deploy.history.HistoryServerArguments=WARN
```

Refer to [Logging](#).

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

MetricsSystem

Spark uses [Metrics 3.1.0](#) Java library to give you insight into the [Spark subsystems](#) (aka *instances*), e.g. [DAGScheduler](#), [BlockManager](#), [Executor](#), [ExecutorAllocationManager](#), [ExternalShuffleService](#), etc.

Note	Metrics are only available for cluster modes, i.e. <code>local</code> mode turns metrics off.
------	---

Table 1. Subsystems and Their MetricsSystems (in alphabetical order)

Subsystem Name	When created
driver	SparkEnv is created for the driver.
executor	SparkEnv is created for an executor.
shuffleService	ExternalShuffleService is created .
applications	Spark Standalone's Master is created .
master	Spark Standalone's Master is created .
worker	Spark Standalone's worker is created .
mesos_cluster	Spark on Mesos' MesosClusterScheduler is created .

Subsystems access their `MetricsSystem` using [SparkEnv](#).

```
val metricsSystem = SparkEnv.get.metricsSystem
```

Caution	FIXME Mention TaskContextImpl and Task.run
---------	--

[org.apache.spark.metrics.source.Source](#) is the top-level class for the metric registries in Spark. Sources expose their internal status.

Metrics System is available at <http://localhost:4040/metrics/json/> (for the default setup of a Spark application).

```
$ http http://localhost:4040/metrics/json/
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Content-Length: 2200
Content-Type: text/json;charset=utf-8
Date: Sat, 25 Feb 2017 14:14:16 GMT
```

```

Server: Jetty(9.2.z-SNAPSHOT)
X-Frame-Options: SAMEORIGIN

{
  "counters": {
    "app-20170225151406-0000.driver.HiveExternalCatalog.fileCacheHits": {
      "count": 0
    },
    "app-20170225151406-0000.driver.HiveExternalCatalog.filesDiscovered": {
      "count": 0
    },
    "app-20170225151406-0000.driver.HiveExternalCatalog.hiveClientCalls": {
      "count": 2
    },
    "app-20170225151406-0000.driver.HiveExternalCatalog.parallelListingJobCount": {
      "count": 0
    },
    "app-20170225151406-0000.driver.HiveExternalCatalog.partitionsFetched": {
      "count": 0
    }
  },
  "gauges": {
    ...
  },
  "timers": {
    "app-20170225151406-0000.driver.DAGScheduler.messageProcessingTime": {
      "count": 0,
      "duration_units": "milliseconds",
      "m15_rate": 0.0,
      "m1_rate": 0.0,
      "m5_rate": 0.0,
      "max": 0.0,
      "mean": 0.0,
      "mean_rate": 0.0,
      "min": 0.0,
      "p50": 0.0,
      "p75": 0.0,
      "p95": 0.0,
      "p98": 0.0,
      "p99": 0.0,
      "p999": 0.0,
      "rate_units": "calls/second",
      "stddev": 0.0
    }
  },
  "version": "3.0.0"
}

```

Note

You can access a Spark subsystem's MetricsSystem using its corresponding "leading" port, e.g. 4040 for the driver, 8080 for Spark Standalone's master and applications.

Note	You have to use the trailing slash (/) to have the output.
------	--

Enable `org.apache.spark.metrics.sink.JmxSink` in `conf/metrics.properties` and use jconsole to access Spark metrics through JMX.

```
*.sink.jmx.class=org.apache.spark.metrics.sink.JmxSink
```

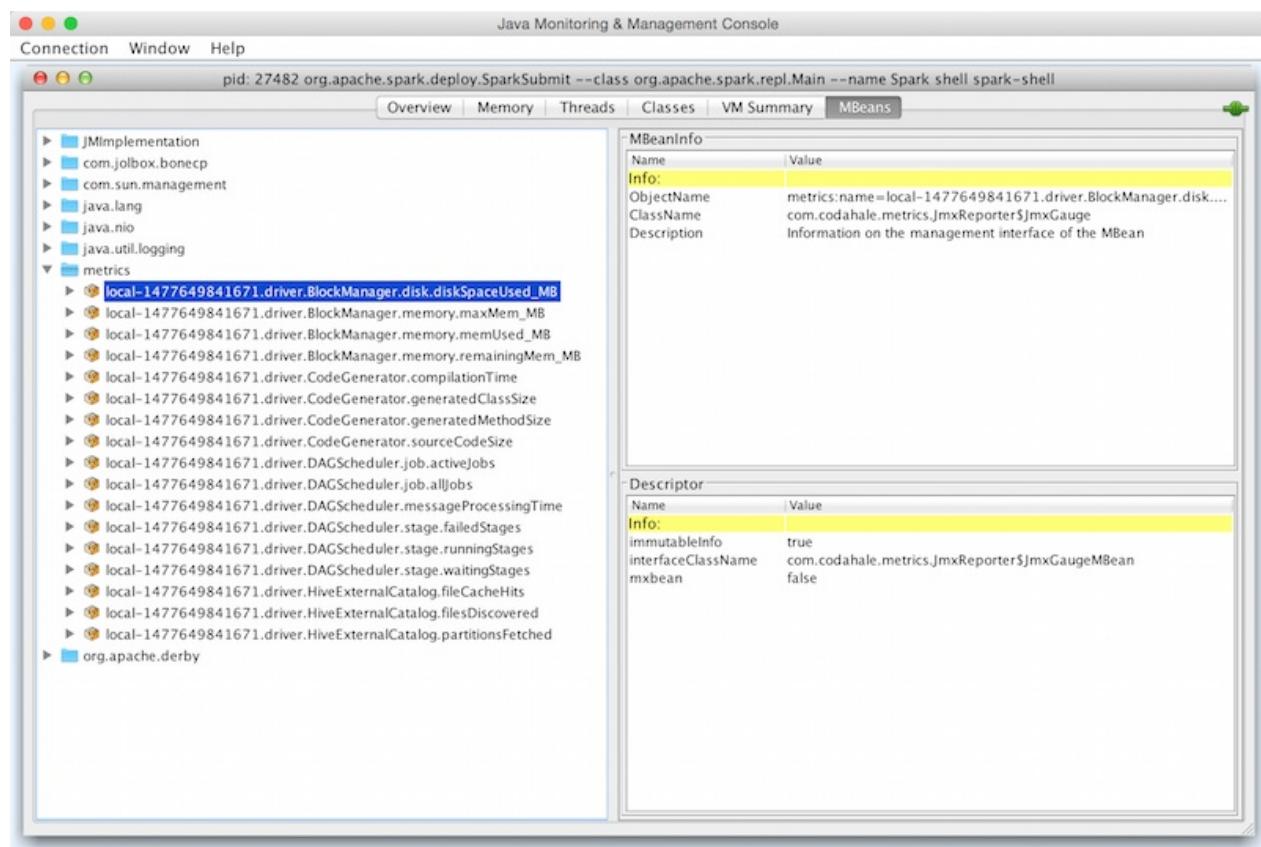


Figure 1. jconsole and JmxSink in spark-shell

Table 2. MetricsSystem's Internal Properties

Name	Initial Value	Description
metricsConfig	MetricsConfig	Initialized when <code>MetricsSystem</code> is created. Used when <code>MetricsSystem</code> registers sinks and sources.
running	Flag whether <code>MetricsSystem</code> has already been started or not	FIXME
metricsServlet	(uninitialized)	FIXME

Table 3. MetricsSystem's Internal Registries and Counters

Name	Description
registry	com.codahale.metrics.MetricRegistry
FIXME	sinks
Metrics sinks in a Spark application. Used when MetricsSystem registers a new metrics sink and starts them eventually.	sources

Tip	<p>Enable <code>WARN</code> or <code>ERROR</code> logging levels for <code>org.apache.spark.metrics.MetricsSystem</code> logger to see what happens in <code>MetricsSystem</code>.</p> <p>Add the following line to <code>conf/log4j.properties</code>:</p> <pre>log4j.logger.org.apache.spark.metrics.MetricsSystem=WARN</pre> <p>Refer to Logging.</p>
-----	--

"Static" Metrics Sources for Spark SQL — StaticSources

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

registerSinks Internal Method

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

stop Method

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

removeSource Method

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

report Method

Caution

FIXME

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"
}
```

Creating MetricsSystem Instance For Subsystem — `createMetricsSystem` Factory Method

```
createMetricsSystem(
  instance: String,
  conf: SparkConf,
  securityMgr: SecurityManager): MetricsSystem
```

`createMetricsSystem` creates a `MetricsSystem`.

Note	<code>createMetricsSystem</code> is used when subsystems create their <code>MetricsSystems</code> .
------	---

Creating MetricsSystem Instance

MetricsSystem takes the following when created:

- Subsystem name
- SparkConf
- SecurityManager

MetricsSystem initializes the internal registries and counters.

When created, MetricsSystem requests MetricsConfig to initialize.

Note	createMetricsSystem is used to create MetricsSystems instead.
------	---

Registering Metrics Source — registerSource Method

```
registerSource(source: Source): Unit
```

registerSource adds source to sources internal registry.

registerSource creates an identifier for the metrics source and registers it with MetricRegistry.

Note	registerSource uses Metrics' MetricRegistry.register to register a metrics source under a given name.
------	---

When registerSource tries to register a name more than once, you should see the following INFO message in the logs:

```
INFO Metrics already registered
```

	<p><code>registerSource</code> is used when:</p> <ul style="list-style-type: none"> • <code>SparkContext</code> registers metrics sources for: <ul style="list-style-type: none"> ◦ <code>DAGScheduler</code> ◦ <code>BlockManager</code> ◦ <code>ExecutorAllocationManager</code> (when dynamic allocation is enabled) • <code>MetricsSystem</code> is started (and registers the "static" metrics sources — <code>CodegenMetrics</code> and <code>HiveCatalogMetrics</code>) and does <code>registerSources</code>. • <code>Executor</code> is created (and registers a <code>ExecutorSource</code>) • <code>ExternalShuffleService</code> is started (and registers <code>ExternalShuffleServiceSource</code>) • Spark Structured Streaming's <code>StreamExecution</code> runs batches as data arrives (when metrics are enabled). • Spark Streaming's <code>StreamingContext</code> is started (and registers <code>StreamingSource</code>) • Spark Standalone's <code>Master</code> and <code>Worker</code> start (and register their <code>MasterSource</code> and <code>WorkerSource</code>, respectively) • Spark Standalone's <code>Master</code> registers a Spark application (and registers a <code>ApplicationSource</code>) • Spark on Mesos' <code>MesosClusterScheduler</code> is started (and registers a <code>MesosClusterSchedulerSource</code>)
Note	

Building Metrics Source Identifier — `buildRegistryName` Method

```
buildRegistryName(source: Source): String
```

Note	<code>buildRegistryName</code> is used to build the metrics source identifiers for a Spark application's driver and executors, but also for other Spark framework's components (e.g. Spark Standalone's master and workers).
Note	<code>buildRegistryName</code> uses <code>spark.metrics.namespace</code> and <code>spark.executor.id</code> Spark properties to differentiate between a Spark application's driver and executors, and the other Spark framework's components.

(only when `instance` is `driver` or `executor`) `buildRegistryName` builds metrics source name that is made up of `spark.metrics.namespace`, `spark.executor.id` and the name of the `source`.

Note

`buildRegistryName` uses Metrics' [MetricRegistry](#) to build metrics source identifiers.

Caution

[FIXME](#) Finish for the other components.

Note

`buildRegistryName` is used when `MetricsSystem` [registers](#) or [removes](#) a metrics source.

Starting MetricsSystem — `start` Method

```
start(): Unit
```

`start` turns [running](#) flag on.

Note

`start` can only be called once and reports an [IllegalArgumentException](#) otherwise.

`start` registers the "static" metrics sources for Spark SQL, i.e. `CodegenMetrics` and `HiveCatalogMetrics`.

`start` then [registerSources](#) followed by [registerSinks](#).

In the end, `start` [starts registered metrics sinks](#) (from `sinks` registry).

Note

`start` is used when:

- `SparkContext` [is created](#) (on the driver)
- `SparkEnv` [is created](#) (on executors)
- `ExternalShuffleService` [is started](#)
- Spark Standalone's `Master` and `worker` start
- Spark on Mesos' `MesosClusterScheduler` is started

Registering Metrics Sources for Current Subsystem — `registerSources` Internal Method

```
registerSources(): Unit
```

`registerSources` finds `metricsConfig` configuration for the current subsystem (aka `instance`).

Note

`instance` is defined when `MetricsSystem` [is created](#).

`registerSources` finds the configuration of all the [metrics sources](#) for the subsystem (as described with `source.` prefix).

For every metrics source, `registerSources` finds `class` property, creates an instance, and in the end [registers it](#).

When `registerSources` fails, you should see the following ERROR message in the logs followed by the exception.

```
ERROR Source class [classPath] cannot be instantiated
```

Note

`registerSources` is used exclusively when `MetricsSystem` is started.

Settings

Table 4. Spark Properties

Spark Property	Default Value	Description
<code>spark.metrics.namespace</code>	Spark application's ID (aka <code>spark.app.id</code>)	<p>Root namespace for metrics reporting.</p> <p>Given a Spark application's ID changes with every invocation of a Spark application, a custom <code>spark.metrics.namespace</code> can be specified for metrics reporting.</p> <p>Used when <code>MetricsSystem</code> is requested for a metrics source identifier.</p>

MetricsConfig — Metrics System Configuration

`MetricsConfig` is the configuration of the `MetricsSystem` (i.e. metrics `sources` and `sinks`).

`MetricsConfig` uses `metrics.properties` as the default metrics configuration file that can however be changed using `spark.metrics.conf` property. The file is first loaded from the path directly before using Spark's CLASSPATH.

`MetricsConfig` lets you also configure the metrics configuration using `spark.metrics.conf`. - prefixed Spark properties.

`MetricsConfig` makes sure that the default metrics properties are always defined.

Table 1. MetricsConfig's Default Metrics Properties

Name	Description
<code>*.sink.servlet.class</code>	<code>org.apache.spark.metrics.sink.MetricsServlet</code>
<code>*.sink.servlet.path</code>	<code>/metrics/json</code>
<code>master.sink.servlet.path</code>	<code>/metrics/master/json</code>
<code>applications.sink.servlet.path</code>	<code>/metrics/applications/json</code>

Note	The order of precedence of metrics configuration settings is as follows:
	1. Default metrics properties
	2. <code>spark.metrics.conf</code> or <code>metrics.properties</code> configuration file
	3. <code>spark.metrics.conf</code> . -prefixed Spark properties

`MetricsConfig` is created when `MetricsSystem` is created.

Table 2. MetricsConfig's Internal Registries and Counters

Name	Description
<code>properties</code>	<code>java.util.Properties</code> with metrics properties Used to initialize per-subsystem's <code>perInstanceSubProperties</code> .
<code>perInstanceSubProperties</code>	Lookup table of metrics properties per subsystem

Creating MetricsConfig Instance

`MetricsConfig` takes the following when created:

- `SparkConf`

`MetricsConfig` initializes the internal registries and counters.

Initializing MetricsConfig — `initialize` Method

```
initialize(): Unit
```

`initialize` sets the default properties and loads the properties from the configuration file (that is defined using `spark.metrics.conf` Spark property).

`initialize` takes all Spark properties that start with `spark.metrics.conf`. prefix from `SparkConf` and adds them to `properties` (without the prefix).

In the end, `initialize` splits configuration per Spark subsystem with the default configuration (denoted as `*`) assigned to the configured subsystems afterwards.

Note	<code>initialize</code> accepts <code>*</code> (star) for the default configuration or any combination of lower- and upper-case letters for Spark subsystem names.
------	--

Note	<code>initialize</code> is used exclusively when <code>MetricsSystem</code> is created.
------	---

setDefaultProperties Internal Method

```
setDefaultProperties(prop: Properties): Unit
```

`setDefaultProperties` sets the default properties (in the input `prop`).

Note	<code>setDefaultProperties</code> is used exclusively when <code>MetricsConfig</code> is initialized.
------	---

Loading Custom Metrics Configuration File or metrics.properties — `loadPropertiesFromFile` Method

```
loadPropertiesFromFile(path: Option[String]): Unit
```

`loadPropertiesFromFile` tries to open the input `path` file (if defined) or the default metrics configuration file `metrics.properties` (on CLASSPATH).

If either file is available, `loadPropertiesFromFile` loads the properties (to `properties` registry).

In case of exceptions, you should see the following ERROR message in the logs followed by the exception.

```
ERROR Error loading configuration file [file]
```

Note

`loadPropertiesFromFile` is used exclusively when `MetricsConfig` is initialized.

Grouping Properties Per Subsystem — `subProperties` Method

```
subProperties(prop: Properties, regex: Regex): mutable.HashMap[String, Properties]
```

`subProperties` takes `prop` properties and destructures keys given `regex`. `subProperties` takes the matching prefix (of a key per `regex`) and uses it as a new key with the value(s) being the matching suffix(es).

```
driver.hello.world => (driver, (hello.world))
```

Note

`subProperties` is used when `MetricsConfig` is initialized (to apply the default metrics configuration) and when `MetricsSystem` registers metrics sources and sinks.

Settings

Table 3. Spark Properties

Spark Property	Default Value	Description
<code>spark.metrics.conf</code>	<code>metrics.properties</code>	The metrics configuration file.

Metrics Source

`Source` is an interface for metrics sources in Spark.

Any `Source` has the following attributes:

1. `sourceName` — the name of a source
2. `metricRegistry` — [com.codahale.metrics.MetricRegistry](#)

Metrics Sink

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

start Method

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

Spark Listeners — Intercepting Events from Spark Scheduler

`SparkListener` is a mechanism in Spark to intercept events from the Spark scheduler that are emitted over the course of execution of a Spark application.

`SparkListener` extends `SparkListenerInterface` with all the *callback methods* being no-op/do-nothing.

Spark [relies on](#) `SparkListeners` [internally](#) to manage communication between internal components in the distributed environment for a Spark application, e.g. [web UI](#), [event persistence](#) (for History Server), [dynamic allocation of executors](#), [keeping track of executors](#) ([using](#) `HeartbeatReceiver`) and others.

You can develop your own custom `sparkListener` and register it using `SparkContext.addSparkListener` method or `spark.extraListeners` Spark property.

With `SparkListener` you can focus on Spark events of your liking and process a subset of all scheduling events.

Tip	Developing a custom <code>SparkListener</code> 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 .
-----	---

Tip	Enable <code>INFO</code> logging level for <code>org.apache.spark.SparkContext</code> logger to see when <code>Spark listeners</code> are registered. <pre style="background-color: #f0f0f0; padding: 5px;">INFO SparkContext: Registered listener org.apache.spark.scheduler.StatsReportList...</pre> See SparkContext — Entry Point to Spark (Core) .
-----	---

SparkListenerInterface — Internal Contract for Spark Listeners

`SparkListenerInterface` is an `private[spark]` contract for Spark listeners to intercept events from the Spark scheduler.

Note	<code>SparkListener</code> and <code>SparkFirehoseListener</code> Spark listeners are direct implementations of <code>SparkListenerInterface</code> contract to help developing more sophisticated Spark listeners.
------	---

Table 1. `SparkListenerInterface` Methods (in alphabetical order)

Method	Event	Reason
onApplicationEnd	SparkListenerApplicationEnd	SparkContext does postApplicationEnd
onApplicationStart	SparkListenerApplicationStart	SparkContext does postApplicationStart
onBlockManagerAdded	SparkListenerBlockManagerAdded	BlockManagerMaster registered a BlockManager
onBlockManagerRemoved	SparkListenerBlockManagerRemoved	BlockManagerMaster removed a BlockManager is when... (FIXME)
onBlockUpdated	SparkListenerBlockUpdated	BlockManagerMaster receives a UpdateBlock message (which is what BlockManager reports status update to driver)
onEnvironmentUpdate	SparkListenerEnvironmentUpdate	SparkContext does postEnvironmentUpdate
onExecutorMetricsUpdate	SparkListenerExecutorMetricsUpdate	
onExecutorAdded	SparkListenerExecutorAdded	DriverEndpoint RPC CoarseGrainedScheduler receives RegisterExecutor message, MesosFineGrainedScheduler does resourceOffers LocalSchedulerBacker starts.
onExecutorBlacklisted	SparkListenerExecutorBlacklisted	(FIXME)
onExecutorRemoved	SparkListenerExecutorRemoved	DriverEndpoint RPC CoarseGrainedScheduler does removeExecutor, MesosFineGrainedScheduler does removeExecutor
onExecutorUnblacklisted	SparkListenerExecutorUnblacklisted	(FIXME)
onJobEnd	SparkListenerJobEnd	DAGScheduler does cleanUpAfterSchedule handleTaskCompletion failJobAndIndependent markMapStageJobAs

onJobStart	SparkListenerJobStart	DAGScheduler handles JobSubmitted and MapStageSubmitted
onNodeBlacklisted	SparkListenerNodeBlacklisted	FIXME
onNodeUnblacklisted	SparkListenerNodeUnblacklisted	FIXME
onStageCompleted	SparkListenerStageCompleted	DAGScheduler marks finished.
onStageSubmitted	SparkListenerStageSubmitted	DAGScheduler submits missing tasks of a stage in a Spark job).
onTaskEnd	SparkListenerTaskEnd	DAGScheduler handles task completion
onTaskGettingResult	SparkListenerTaskGettingResult	DAGScheduler handles GettingResultEvent
onTaskStart	SparkListenerTaskStart	DAGScheduler is informed that a task is about to start.
onUnpersistRDD	SparkListenerUnpersistRDD	SparkContext unpersists RDD i.e. removes RDD blocks from BlockManagerMaster triggered explicitly or
onOtherEvent	SparkListenerEvent	

Built-In Spark Listeners

Table 2. Built-In Spark Listeners

Spark Listener	Description
EventLoggingListener	Logs JSON-encoded events to a file that can later be read by History Server
StatsReportListener	
SparkFirehoseListener	Allows users to receive all SparkListenerEvent events by overriding the single <code>onEvent</code> method only.
ExecutorAllocationListener	
HeartbeatReceiver	
StreamingJobProgressListener	
ExecutorsListener	Prepares information for Executors tab in web UI
StorageStatusListener, RDDOperationGraphListener, EnvironmentListener, BlockStatusListener and StorageListener	For web UI
SpillListener	
ApplicationEventListener	
StreamingQueryListenerBus	
SQLListener / SQLHistoryListener	Support for History Server
StreamingListenerBus	
JobProgressListener	

LiveListenerBus

`LiveListenerBus` is used to announce application-wide events in a Spark application. It asynchronously passes [listener events](#) to registered Spark listeners.

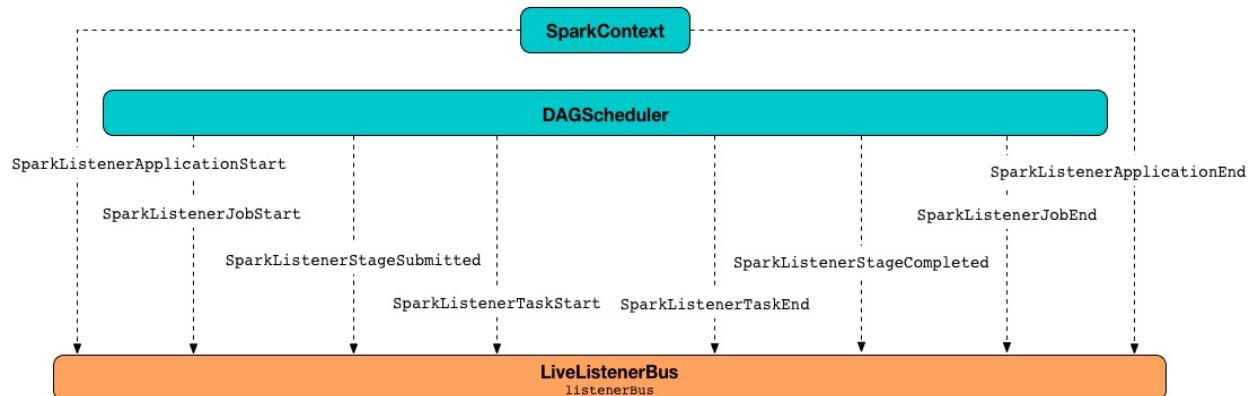


Figure 1. LiveListenerBus, SparkListenerEvents, and Senders

`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 and started when <code>SparkContext</code> is initialized.
------	---

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 <code>stopped</code> 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 <code>logDroppedEvent</code> atomic variable to track the state.
------	---

Stopping `LiveListenerBus` — `stop` method

```
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

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.extraListeners</code>	(empty)	The comma-separated list of fully-qualified class names of <code>Spark listeners</code> that should be registered (when <code>SparkContext</code> is initialized)

ReplayListenerBus

`ReplayListenerBus` is a custom `SparkListenerBus` that can replay JSON-encoded `SparkListenerEvent` events.

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` to convert JSON-encoded events to `SparkListenerEvent` objects.

Note	<code>replay</code> uses jackson 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.
------	---

SparkListenerBus — Internal Contract for Spark Event Buses

`SparkListenerBus` is a `private[spark]` [ListenerBus](#) for [SparkListenerInterface](#) listeners that process [SparkListenerEvent](#) events.

`SparkListenerBus` comes with a custom `doPostEvent` method that simply relays `SparkListenerEvent` events to appropriate `sparkListenerInterface` methods.

Note	There are two custom <code>SparkListenerBus</code> listeners: LiveListenerBus and ReplayListenerBus .
------	---

There are two custom `SparkListenerBus` listeners: [LiveListenerBus](#) and [ReplayListenerBus](#).

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

ListenerBus Event Bus Contract

```
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 `doPostEvent` is provided by more specialized `ListenerBus` 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 `ListenerBus` listeners: [SparkListenerBus](#), [StreamingQueryListenerBus](#), and [StreamingListenerBus](#).

Tip Enable `ERROR` logging level for `org.apache.spark.util.ListenerBus` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.util.ListenerBus=ERROR
```

Refer to [Logging](#).

EventLoggingListener — Spark Listener for Persisting Events

`EventLoggingListener` is a [SparkListener](#) that persists JSON-encoded events to a file.

When [event logging is enabled](#), `EventLoggingListener` writes events to a log file under `spark.eventLog.dir` directory. All [Spark events](#) are logged (except [SparkListenerBlockUpdated](#) and [SparkListenerExecutorMetricsUpdate](#)).

Tip

Use [Spark History Server](#) to view the event logs in a browser.

Events can optionally be [compressed](#).

In-flight log files are with `.inprogress` extension.

`EventLoggingListener` is a `private[spark]` class in `org.apache.spark.scheduler` package.

Enable `INFO` logging level for `org.apache.spark.scheduler.EventLoggingListener` logger to see what happens inside `EventLoggingListener`.

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

Tip

```
log4j.logger.org.apache.spark.scheduler.EventLoggingListener=INFO
```

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 `Configuration` it calls `SparkHadoopUtil.get.newConfiguration(sparkConf)`.

Starting EventLoggingListener — `start` method

```
start(): Unit
```

`start` checks whether `logBaseDir` is really a directory, and if it is not, it throws a `IllegalArgumentException` with the following message:

```
Log directory [logBaseDir] does not exist.
```

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]
```

Note	<code>start</code> is executed while <code>SparkContext</code> is created.
------	--

Logging Event as JSON — `logEvent` method

```
logEvent(event: SparkListenerEvent, flushLogger: Boolean = false)
```

`logEvent` logs `event` as JSON.

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

Stopping EventLoggingListener — `stop` method

```
stop(): Unit
```

`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])
```

Note	<code>stop</code> is executed while <code>SparkContext</code> is stopped.
------	---

Compressing Logged Events

If `event compression is enabled`, events are compressed using `CompressionCodec`.

Tip	Refer to <code>CompressionCodec</code> to learn about the available compression codecs.
-----	---

Settings

Table 1. Spark Properties

Spark Property	Default Value	Description
<code>spark.eventLog.enabled</code>	<code>false</code>	Enables (<code>true</code>) or disables (<code>false</code>) persisting Spark events.
<code>spark.eventLog.dir</code>	<code>/tmp/spark-events</code>	Directory where events are logged, e.g. <code>hdfs://namenode:8021/directory</code> . The directory must exist before Spark starts up .
<code>spark.eventLog.buffer.kb</code>	<code>100</code>	Size of the buffer to use when writing to output streams.
<code>spark.eventLog.overwrite</code>	<code>false</code>	Enables (<code>true</code>) or disables (<code>false</code>) deleting (or at least overwriting) an existing <code>.inprogress</code> log file.
<code>spark.eventLog.compress</code>	<code>false</code>	Enables (<code>true</code>) or disables (<code>false</code>) event compression .
<code>spark.eventLog.testing</code>	<code>false</code>	Internal flag for testing purposes that enables adding JSON events to the internal <code>loggedEvents</code> array.

StatsReportListener — Logging Summary Statistics

`org.apache.spark.scheduler.StatsReportListener` (see the listener's scaladoc) is a [SparkListener](#) that logs summary statistics when each stage completes.

`StatsReportListener` listens to [SparkListenerTaskEnd](#) and [SparkListenerStageCompleted](#) events and prints them out at `INFO` logging level.

Enable `INFO` logging level for `org.apache.spark.scheduler.StatsReportListener` logger to see Spark events.

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

Tip

```
log4j.logger.org.apache.spark.scheduler.StatsReportListener=INFO
```

Refer to [Logging](#).

Intercepting Stage Completed Events — `onStageCompleted` Callback

Caution

[FIXME](#)

Example

```
$ ./bin/spark-shell -c spark.extraListeners=org.apache.spark.scheduler.StatsReportListener
...
INFO SparkContext: Registered listener org.apache.spark.scheduler.StatsReportListener
...
scala> spark.read.text("README.md").count
...
INFO StatsReportListener: Finished stage: Stage(0, 0); Name: 'count at <console>:24';
Status: succeeded; numTasks: 1; Took: 212 msec
INFO StatsReportListener: task runtime:(count: 1, mean: 198.000000, stdev: 0.000000, max: 198.000000, min: 198.000000)
INFO StatsReportListener:          0%      5%     10%     25%     50%     75%     90%
                           95%     100%
INFO StatsReportListener:          198.0 ms      198.0 ms      198.0 ms      198.0 ms      198.0 ms
ms      198.0 ms      198.0 ms      198.0 ms      198.0 ms      198.0 ms
INFO StatsReportListener: shuffle bytes written:(count: 1, mean: 59.000000, stdev: 0.0
00000, max: 59.000000, min: 59.000000)
```

INFO StatsReportListener:	0%	5%	10%	25%	50%	75%	90%
95% 100%							
INFO StatsReportListener:	59.0 B	59.0 B	59.0 B	59.0 B	59.0 B	59.0 B	59.0 B
59.0 B 59.0 B							
INFO StatsReportListener: fetch wait time:(count: 1, mean: 0.000000, stdev: 0.000000, max: 0.000000, min: 0.000000)							
INFO StatsReportListener:	0%	5%	10%	25%	50%	75%	90%
95% 100%							
INFO StatsReportListener:	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
0.0 ms 0.0 ms							
INFO StatsReportListener: remote bytes read:(count: 1, mean: 0.000000, stdev: 0.000000, max: 0.000000, min: 0.000000)							
INFO StatsReportListener:	0%	5%	10%	25%	50%	75%	90%
95% 100%							
INFO StatsReportListener:	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B
0.0 B 0.0 B							
INFO StatsReportListener: task result size:(count: 1, mean: 1885.000000, stdev: 0.000000, max: 1885.000000, min: 1885.000000)							
INFO StatsReportListener:	0%	5%	10%	25%	50%	75%	90%
95% 100%							
INFO StatsReportListener:	1885.0 B		1885.0 B		1885.0 B		1885.0 B
1885.0 B 1885.0 B 1885.0 B 1885.0 B 1885.0 B							
INFO StatsReportListener: executor (non-fetch) time pct: (count: 1, mean: 73.737374, stdev: 0.000000, max: 73.737374, min: 73.737374)							
INFO StatsReportListener:	0%	5%	10%	25%	50%	75%	90%
95% 100%							
INFO StatsReportListener:	74 %	74 %	74 %	74 %	74 %	74 %	74 %
74 % 74 %							
INFO StatsReportListener: fetch wait time pct: (count: 1, mean: 0.000000, stdev: 0.000000, max: 0.000000, min: 0.000000)							
INFO StatsReportListener:	0%	5%	10%	25%	50%	75%	90%
95% 100%							
INFO StatsReportListener:	0 %	0 %	0 %	0 %	0 %	0 %	0 %
0 % 0 %							
INFO StatsReportListener: other time pct: (count: 1, mean: 26.262626, stdev: 0.000000, max: 26.262626, min: 26.262626)							
INFO StatsReportListener:	0%	5%	10%	25%	50%	75%	90%
95% 100%							
INFO StatsReportListener:	26 %	26 %	26 %	26 %	26 %	26 %	26 %
26 % 26 %							
INFO StatsReportListener: Finished stage: Stage(1, 0); Name: 'count at <console>:24'; Status: succeeded; numTasks: 1; Took: 34 msec							
INFO StatsReportListener: task runtime:(count: 1, mean: 33.000000, stdev: 0.000000, max: 33.000000, min: 33.000000)							
INFO StatsReportListener:	0%	5%	10%	25%	50%	75%	90%
95% 100%							
INFO StatsReportListener:	33.0 ms	33.0 ms	33.0 ms	33.0 ms	33.0 ms	33.0 ms	33.0 ms
33.0 ms 33.0 ms							
INFO StatsReportListener: shuffle bytes written:(count: 1, mean: 0.000000, stdev: 0.000000, max: 0.000000, min: 0.000000)							
INFO StatsReportListener:	0%	5%	10%	25%	50%	75%	90%
95% 100%							
INFO StatsReportListener:	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

```

    0.0 B   0.0 B
INFO StatsReportListener: fetch wait time:(count: 1, mean: 0.000000, stdev: 0.000000,
max: 0.000000, min: 0.000000)
INFO StatsReportListener:          0%     5%    10%    25%    50%    75%    90%
                                95%   100%
INFO StatsReportListener:          0.0 ms  0.0 ms  0.0 ms  0.0 ms  0.0 ms  0.0 ms  0.0 ms
                                0.0 ms  0.0 ms
INFO StatsReportListener: remote bytes read:(count: 1, mean: 0.000000, stdev: 0.000000
, max: 0.000000, min: 0.000000)
INFO StatsReportListener:          0%     5%    10%    25%    50%    75%    90%
                                95%   100%
INFO StatsReportListener:          0.0 B   0.0 B   0.0 B   0.0 B   0.0 B   0.0 B   0.0 B
                                0.0 B   0.0 B
INFO StatsReportListener: task result size:(count: 1, mean: 1960.000000, stdev: 0.0000
00, max: 1960.000000, min: 1960.000000)
INFO StatsReportListener:          0%     5%    10%    25%    50%    75%    90%
                                95%   100%
INFO StatsReportListener:          1960.0 B   1960.0 B   1960.0 B   1960.0 B   1960.0 B
                                1960.0 B   1960.0 B
INFO StatsReportListener: executor (non-fetch) time pct: (count: 1, mean: 75.757576, s
tdev: 0.000000, max: 75.757576, min: 75.757576)
INFO StatsReportListener:          0%     5%    10%    25%    50%    75%    90%
                                95%   100%
INFO StatsReportListener:          76 %   76 %   76 %   76 %   76 %   76 %   76 %
                                76 %   76 %
INFO StatsReportListener: fetch wait time pct: (count: 1, mean: 0.000000, stdev: 0.000
000, max: 0.000000, min: 0.000000)
INFO StatsReportListener:          0%     5%    10%    25%    50%    75%    90%
                                95%   100%
INFO StatsReportListener:          0 %   0 %   0 %   0 %   0 %   0 %   0 %
                                0 %   0 %
INFO StatsReportListener: other time pct: (count: 1, mean: 24.242424, stdev: 0.000000,
max: 24.242424, min: 24.242424)
INFO StatsReportListener:          0%     5%    10%    25%    50%    75%    90%
                                95%   100%
INFO StatsReportListener:          24 %   24 %   24 %   24 %   24 %   24 %   24 %
                                24 %   24 %
res0: Long = 99

```

JsonProtocol

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

taskInfoFromJson Method

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

taskMetricsFromJson Method

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

taskMetricsToJson Method

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

sparkEventFromJson Method

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

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 Apache Spark from Sources

You can download pre-packaged versions of Apache Spark from [the project's web site](#). The packages are built for a different Hadoop versions for Scala 2.11.

Note

Since [\[SPARK-6363\]](#)[\[BUILD\]](#) Make Scala 2.11 the default Scala version the default version of Scala in Apache Spark is 2.11.

The build process for Scala 2.11 takes less than 15 minutes (on a decent machine like my shiny MacBook Pro with 8 cores and 16 GB RAM) 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 -Phadoop-2.7,yarn,mesos,hive,hive-thriftserver -DskipTests clean assembly
```

Using Java 8 to build Spark using sbt takes ca 10 minutes.

```
→ spark git:(master) ✘ ./build/sbt -Phadoop-2.7,yarn,mesos,hive,hive-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

hive-thriftserver Maven profile for Spark Thrift Server

Caution

[FIXME](#)

Tip

Read [Thrift JDBC/ODBC Server—Spark Thrift Server \(STS\)](#).

Using Apache Maven as the build tool

The build command with Apache Maven is as follows:

```
$ ./build/mvn -Phadoop-2.7,yarn,mesos,hive,hive-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_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)

→ spark git:(master) ✘ ./build/mvn -Phadoop-2.7,yarn,mesos,hive,hive-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 `./bin/spark-shell --version`.

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 -Phadoop-2.7,yarn,mesos,hive,hive
-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

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 partition files into so called **splits** and distributes them across multiple nodes in a cluster to achieve fail-over and resiliency.

MapReduce happens in three phases: **Map**, **Shuffle**, and **Reduce**.

Further reading

- [Introducing Kudu: The New Hadoop Storage Engine for Fast Analytics on Fast Data](#)

SparkHadoopUtil

Tip

Enable `DEBUG` logging level for `org.apache.spark.deploy.SparkHadoopUtil` logger to see what happens inside.

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

```
log4j.logger.org.apache.spark.deploy.SparkHadoopUtil=DEBUG
```

Refer to [Logging](#).

Creating SparkHadoopUtil Instance — `get` Method

Caution

[FIXME](#)

`substituteHadoopVariables` Method

Caution

[FIXME](#)

`transferCredentials` Method

Caution

[FIXME](#)

`newConfiguration` Method

Caution

[FIXME](#)

`conf` Method

Caution

[FIXME](#)

`stopCredentialUpdater` Method

Caution

[FIXME](#)

Running Executable Block As Spark User — `runAsSparkUser` Method

```
runAsSparkUser(func: () => Unit)
```

`runAsSparkUser` runs `func` function with Hadoop's `UserGroupInformation` of the current user as a thread local variable (and distributed to child threads). It is later used for authenticating HDFS and YARN calls.

Internally, `runAsSparkUser` reads the current username (as `SPARK_USER` environment variable or the short user name from Hadoop's `UserGroupInformation`).

Caution

[FIXME](#) How to use `SPARK_USER` 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.

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.

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](#)
 - [Apache Toree](#)

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 Spark Notebook 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

    __
   / \
  _\ \_ \_ \_ \_ \_ \
 /__\ .__/\_,/_/ /__\ \
                  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@2700
f556)
  - 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 Applications on Windows

Running Spark applications on Windows in general is no different than running it on other operating systems like Linux or macOS.

Note	A Spark application could be spark-shell or your own custom Spark application.
------	--

What makes the huge difference between the operating systems is Hadoop that is used internally for file system access in Spark.

You may run into few minor issues when you are on Windows due to the way Hadoop works with Windows' POSIX-incompatible NTFS filesystem.

Note	You do not have to install Apache Hadoop to work with Spark or run Spark applications.
------	--

Tip	Read the Apache Hadoop project's Problems running Hadoop on Windows .
-----	---

Among the issues is the infamous `java.io.IOException` when running Spark Shell (below a stacktrace from Spark 2.0.2 on Windows 10 so the line numbers may be different in your case).

```
16/12/26 21:34:11 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:379)
    at org.apache.hadoop.util.Shell.getWinUtilsPath(Shell.java:394)
    at org.apache.hadoop.util.Shell.<clinit>(Shell.java:387)
    at org.apache.hadoop.hive.conf.HiveConf$ConfVars.findHadoopBinary(HiveConf.java:2327)
)
    at org.apache.hadoop.hive.conf.HiveConf$ConfVars.<clinit>(HiveConf.java:365)
    at org.apache.hadoop.hive.conf.HiveConf.<clinit>(HiveConf.java:105)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:348)
    at org.apache.spark.util.Utils$.classForName(Utils.scala:228)
    at org.apache.spark.sql.SparkSession$.hiveClassesArePresent(SparkSession.scala:963)
    at org.apache.spark.repl.Main$.createSparkSession(Main.scala:91)
```

Note	You need to have Administrator rights on your laptop. All the following commands must be executed in a command-line window (<code>cmd</code>) ran as Administrator, i.e. using Run as administrator option while executing <code>cmd</code> . Read the official document in Microsoft TechNet— Start a Command Prompt as an Administrator .
------	---

Download `winutils.exe` binary from <https://github.com/steveloughran/winutils> repository.

Note You should select the version of Hadoop the Spark distribution was compiled with, e.g. use `hadoop-2.7.1` for Spark 2 ([here is the direct link to `winutils.exe` binary](#)).

Save `winutils.exe` binary to a directory of your choice, e.g. `c:\hadoop\bin`.

Set `HADOOP_HOME` to reflect the directory with `winutils.exe` (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 so any Windows program would use them.

Create `C:\tmp\hive` directory.

Note `C:\tmp\hive` directory is the default value of [hive.exec.scratchdir configuration property](#) in Hive 0.14.0 and later and Spark uses a custom build of Hive 1.2.1.

You can change `hive.exec.scratchdir` configuration property to another directory as described in [Changing `hive.exec.scratchdir` Configuration Property](#) in this document.

Execute the following command in `cmd` that you started using the option **Run as administrator**.

```
winutils.exe chmod -R 777 C:\tmp\hive
```

Check the permissions (that is one of the commands that are executed under the covers):

```
winutils.exe ls -F C:\tmp\hive
```

Open `spark-shell` and observe the output (perhaps with few WARN messages that you can simply disregard).

As a verification step, execute the following line to display the content of a `DataFrame`:

```
scala> spark.range(1).withColumn("status", lit("All seems fine. Congratulations!")).show(false)
+---+-----+
|id |status
+---+-----+
|0  |All seems fine. Congratulations!
+---+-----+
```

Note

Disregard WARN messages when you start `spark-shell`. They are harmless.

```
16/12/26 22:05:41 WARN General: Plugin (Bundle) "org.datanucleus" is already registered, and you are trying to register an identical plugin located at URL "file:/C:/spark-2.0.2-bin-hadoop2.7/plugins/org.datanucleus-api-jdo-3.2.10.jar."
16/12/26 22:05:41 WARN General: Plugin (Bundle) "org.datanucleus.api.jdo" is already registered, and you are trying to register an identical plugin located at URL 'hadoop2.7/bin/../jars/datanucleus-api-jdo-3.2.6.jar.'
16/12/26 22:05:41 WARN General: Plugin (Bundle) "org.datanucleus.store.rdbms" is already registered, and you are trying to register an identical plugin located at URL 'hadoop2.7/jars/datanucleus-rdbms-3.2.9.jar.'
```

If you see the above output, you're done. You should now be able to run Spark applications on your Windows. Congrats!

Changing `hive.exec.scratchdir` Configuration Property

Create a `hive-site.xml` file with the following content:

```
<configuration>
<property>
  <name>hive.exec.scratchdir</name>
  <value>/tmp/mydir</value>
  <description>Scratch space for Hive jobs</description>
</property>
</configuration>
```

Start a Spark application, e.g. `spark-shell`, with `HADOOP_CONF_DIR` environment variable set to the directory with `hive-site.xml`.

```
HADOOP_CONF_DIR=conf ./bin/spark-shell
```


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	FIXME Snapshot from web UI - note the number of tasks
---------	---

However, when you execute `r1.take(2)` two jobs get run as the implementation assumes one job with one partition, and if the elements didn't total to the number of elements requested in `take`, quadruple the partitions to work on in the following jobs.

Caution	FIXME Snapshot from web UI - note the number of tasks
---------	---

Can you guess how many jobs are run for `r1.take(15)`? How many tasks per job?

Caution	FIXME Snapshot from web UI - note the number of tasks
---------	---

Answer: 3.

Spark Standalone - Using ZooKeeper for High-Availability of Master

Tip

Read [Recovery Mode](#) to know the theory.

You're going to start two standalone Masters.

You'll need 4 terminals (adjust addresses as needed):

Start ZooKeeper.

Create a configuration file `ha.conf` with the content as follows:

```
spark.deploy.recoveryMode=ZOOKEEPER
spark.deploy.zookeeper.url=<zookeeper_host>:2181
spark.deploy.zookeeper.dir=/spark
```

Start the first standalone Master.

```
./sbin/start-master.sh -h localhost -p 7077 --webui-port 8080 --properties-file ha.conf
```

Note

It is not possible to start another instance of standalone Master on the same machine using `./sbin/start-master.sh`. The reason is that the script assumes one instance per machine only. We're going to change the script to make it possible.

```
$ cp ./sbin/start-master{,-2}.sh

$ grep "CLASS 1" ./sbin/start-master-2.sh
"${SPARK_HOME}/sbin"/spark-daemon.sh start $CLASS 1 \

$ sed -i -e 's/CLASS 1/CLASS 2/' sbin/start-master-2.sh

$ grep "CLASS 1" ./sbin/start-master-2.sh

$ grep "CLASS 2" ./sbin/start-master-2.sh
"${SPARK_HOME}/sbin"/spark-daemon.sh start $CLASS 2 \

$ ./sbin/start-master-2.sh -h localhost -p 17077 --webui-port 18080 --properties-file
ha.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">• FIXME Offer a complete working solution in Scala• FIXME Load ORC files into dataframe<ul style="list-style-type: none">◦ <code>val df = hiveContext.read.format("orc").load(to/path)</code>
---------	--

Solution was to use Hive in ORC format with partitions:

- A table in Hive stored as an ORC file (using partitioning)
- Using `SQLContext.sql` to insert data into the table
- Using `SQLContext.sql` to periodically run `ALTER TABLE...CONCATENATE` to merge your many small files into larger files optimized for your HDFS block size
 - Since the `CONCATENATE` command operates on files in place it is transparent to any downstream processing
- Hive solution is just to concatenate the files
 - it does not alter or change records.
 - it's possible to update data in Hive using ORC format
 - With transactional tables in Hive together with insert, update, delete, it does the "concatenate" for you automatically in regularly intervals. Currently this works only with tables in `orc.format` (stored as `orc`)
 - Alternatively, use Hbase with Phoenix as the SQL layer on top
 - Hive was originally not designed for updates, because it was purely warehouse focused, the most recent one can do updates, deletes etc in a transactional way.

Criteria:

- [Spark Streaming](#) jobs are receiving a lot of small events (avg 10kb)
- Events are stored to HDFS, e.g. for Pig jobs
- There are a lot of small files in HDFS (several millions)

Exercise: Developing Custom SparkListener to monitor DAGScheduler in Scala

The example shows how to develop a custom Spark Listener. You should read [Spark Listeners — Intercepting Events from Spark Scheduler](#) first to understand the motivation for the example.

Requirements

1. [IntelliJ IDEA](#) (or eventually `sbt` alone if you're adventurous).
2. Access to Internet to download Apache Spark's dependencies.

Setting up Scala project using IntelliJ IDEA

Create a new project `custom-spark-listener`.

Add the following line to `build.sbt` (the main configuration file for the sbt project) that adds the dependency on Apache Spark.

```
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.0.1"
```

`build.sbt` should look as follows:

```
name := "custom-spark-listener"
organization := "pl.jaceklaskowski.spark"
version := "1.0"

scalaVersion := "2.11.8"

libraryDependencies += "org.apache.spark" %% "spark-core" % "2.0.1"
```

Custom Listener - `pl.jaceklaskowski.spark.CustomSparkListener`

Create a Scala class — `CustomSparkListener` — for your custom `SparkListener`. It should be under `src/main/scala` directory (create one if it does not exist).

The aim of the class is to intercept scheduler events about jobs being started and tasks completed.

```

package pl.jaceklaskowski.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 the custom Spark listener. Execute `sbt package` command in the `custom-spark-listener` project's main directory.

```

$ sbt package
[info] Loading global plugins from /Users/jacek/.sbt/0.13/plugins
[info] Loading project definition from /Users/jacek/dev/workshops/spark-workshop/solut
ions/custom-spark-listener/project
[info] Updating {file:/Users/jacek/dev/workshops/spark-workshop/solutions/custom-spark
-listener/project/}custom-spark-listener-build...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Set current project to custom-spark-listener (in build file:/Users/jacek/dev/wo
rkshops/spark-workshop/solutions/custom-spark-listener/)
[info] Updating {file:/Users/jacek/dev/workshops/spark-workshop/solutions/custom-spark
-listener/}custom-spark-listener...
[info] Resolving jline#jline;2.12.1 ...
[info] Done updating.
[info] Compiling 1 Scala source to /Users/jacek/dev/workshops/spark-workshop/solutions
/custom-spark-listener/target/scala-2.11/classes...
[info] Packaging /Users/jacek/dev/workshops/spark-workshop/solutions/custom-spark-list
ener/target/scala-2.11/custom-spark-listener_2.11-1.0.jar ...
[info] Done packaging.
[success] Total time: 8 s, completed Oct 27, 2016 11:23:50 AM

```

You should find the result jar file with the custom scheduler listener ready under `target/scala-2.11` directory, e.g. `target/scala-2.11/custom-spark-listener_2.11-1.0.jar`.

Activating Custom Listener in Spark shell

Start [spark-shell](#) with additional configurations for the extra custom listener and the jar that includes the class.

```
$ spark-shell --conf spark.logConf=true --conf spark.extraListeners=pl.jaceklaskowski.spark.CustomSparkListener --driver-class-path target/scala-2.11/custom-spark-listener_2.11-1.0.jar
```

Create a [Dataset](#) and execute an action like `show` to start a job as follows:

```
scala> spark.read.text("README.md").count
[CustomSparkListener] Job started with 2 stages: SparkListenerJobStart(1,1473946006715
,WrappedArray(org.apache.spark.scheduler.StageInfo@71515592, org.apache.spark.schedule
r.StageInfo@6852819d),{spark.rdd.scope.noOverride=true, spark.rdd.scope={"id":"14","na
me":"collect"}, spark.sql.execution.id=2})
[CustomSparkListener] Stage 1 completed with 1 tasks.
[CustomSparkListener] Stage 2 completed with 1 tasks.
res0: Long = 7
```

The lines with `[CustomSparkListener]` came from your custom Spark listener.
Congratulations! The exercise's over.

BONUS Activating Custom Listener in Spark Application

Tip	Read Registering SparkListener — <code>addSparkListener</code> method.
-----	--

Questions

1. What are the pros and cons of using the command line version vs inside a Spark application?

Developing RPC Environment

Caution	<p>FIXME</p> <ul style="list-style-type: none"> • Create the exercise • It could be easier to have an exercise to register a custom RpcEndpoint (it can receive network events known to all endpoints, e.g. <code>RemoteProcessConnected = "a new node connected"</code> or <code>RemoteProcessDisconnected = a node disconnected</code>). That could be the only way to know about the current runtime configuration of <code>RpcEnv</code>. Use <code>SparkEnv.rpcEnv</code> and <code>rpcEnv.setupEndpoint(name, endpointCreator)</code> to register a RPC Endpoint.
---------	--

Start simple using the following command:

```
$ ./bin/spark-shell --conf spark.rpc=doesnotexist
...
15/10/21 12:06:11 INFO SparkContext: Running Spark version 1.6.0-SNAPSHOT
...
15/10/21 12:06:11 ERROR SparkContext: Error initializing SparkContext.
java.lang.ClassNotFoundException: doesnotexist
    at scala.reflect.internal.util.AbstractFileClassLoader.findClass(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	FIXME
---------	-----------------------

Working with Datasets using JDBC (and PostgreSQL)

Start `spark-shell` with the proper JDBC driver.

Note	Download PostgreSQL JDBC Driver JDBC 4.1 from the Maven repository .
------	--

Tip	Execute the command to have the jar downloaded into <code>~/.ivy2/jars</code> directory by <code>spark-shell</code> :
-----	---

```
./bin/spark-shell --packages org.postgresql:postgresql:9.4.1211
```

Tip	The entire path to the driver file is then like
-----	---

```
/Users/jacek/.ivy2/jars/org.postgresql_postgresql-9.4.1211.jar .
```

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/.ivy2/jars/org.postgresql_postgresql-9.4.1211.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.

Enable `all` logs in PostgreSQL to see query statements.

Tip `log_statement = 'all'`

Add `log_statement = 'all'` to `/usr/local/var/postgres/postgresql.conf` on Mac OS X with PostgreSQL installed using `brew`.

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 (9.5.4)  
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 9.5.4 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 (5.2) Sandbox with Hadoop 2.7 uses **Spark 1.6.1** and is available as a VMware or VirtualBox VM.

The documentation is available at <http://maprdocs.mapr.com/home/>

Spark Advanced Workshop

Taking the notes and leading [Scala/Spark meetups in Warsaw, Poland](#) gave me opportunity to create the initial version of the **Spark Advanced workshop**. It is a highly-interactive in-depth 2-day workshop about Spark with many practical exercises.

Contact me at jacek@japila.pl to discuss having one at your convenient location and/or straight in the office. We could also host the workshop remotely.

It's is a hands-on workshop with lots of exercises and do-fail-fix-rinse-repeat cycles.

1. Requirements
2. Day 1
3. Day 2

Spark Advanced Workshop - Requirements

1. Linux or Mac OS (please no Windows - if you insist, use a virtual machine with Linux using [VirtualBox](#)).
2. The latest release of [Java™ Platform, Standard Edition Development Kit](#).
3. The latest release of Apache Spark **pre-built for Hadoop 2.6 and later** from [Download Spark](#).
4. Basic experience in developing simple applications using [Scala programming language](#) and [sbt](#).

Spark Advanced Workshop - Day 1

Agenda

1. RDD - Resilient Distributed Dataset - 45 mins
2. Setting up Spark Standalone cluster - 45 mins
3. Using Spark shell with Spark Standalone - 45 mins
4. WebUI - UI for Spark Monitoring - 45 mins
5. Developing Spark applications using Scala and sbt and deploying to the Spark Standalone cluster - 2 x 45 mins

Spark Advanced Workshop - Day 2

Agenda

1. [Using Listeners to monitor Spark's Scheduler](#) - 45 mins
2. [TaskScheduler and Speculative execution of tasks](#) - 45 mins
3. [Developing Custom RPC Environment \(RpcEnv\)](#) - 45 mins
4. [Spark Metrics System](#) - 45 mins
5. [Don't fear the logs - Learn Spark by Logs](#) - 45 mins

Spark Talks Ideas (STI)

This is the collection of talks I'm going to present at conferences, meetups, webinars, etc.

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.