Mastering Apache Spark

Introduction

Overview of Spark

**Table of Contents**

0

1

Anatomy of Spark Application

SparkConf - Configuration for Spark Applications SparkContext - the door to Spark

RDD - Resilient Distributed Dataset

Operators - Transformations and Actions mapPartitions

Partitions

Caching and Persistence

Shuffling

Checkpointing

Dependencies

Types of RDDs

ParallelCollectionRDD

MapPartitionsRDD

CoGroupedRDD

HadoopRDD

ShuffledRDD

BlockRDD

Spark Tools

Spark Shell

WebUI - UI for Spark Monitoring

Executors Tab

spark-submit

spark-class

Spark Architecture

Driver

Master

Workers

2

2.1

2.2

2.3

2.3.1

2.3.1.1 2.3.2

2.3.3

2.3.4

2.3.5

2.3.6

2.3.7

2.3.7.1 2.3.7.2 2.3.7.3 2.3.7.4 2.3.7.5 2.3.7.6 3

3.1

3.2

3.2.1

3.3

3.4

4

4.1

4.2

4.3

2

Mastering Apache Spark

Executors

Spark Runtime Environment

DAGScheduler

Jobs

Stages

Task Scheduler

Tasks

TaskSets

TaskSetManager

TaskSchedulerImpl - Default TaskScheduler Scheduler Backend

CoarseGrainedSchedulerBackend

Executor Backend

CoarseGrainedExecutorBackend

Shuffle Manager

Block Manager

HTTP File Server

Broadcast Manager

Dynamic Allocation

Data Locality

Cache Manager

Spark, Akka and Netty

OutputCommitCoordinator

RPC Environment (RpcEnv)

Netty-based RpcEnv

ContextCleaner

MapOutputTracker

ExecutorAllocationManager

Deployment Environments

Spark local

Spark on cluster

Spark Standalone

Master

web UI

4.4

5

5.1

5.1.1

5.1.2

5.2

5.2.1

5.2.2

5.2.3

5.2.4

5.3

5.3.1

5.4

5.4.1

5.5

5.6

5.7

5.8

5.9

5.10

5.11

5.12

5.13

5.14

5.14.1 5.15

5.16

5.17

6

6.1

6.2

6.2.1

6.2.1.1 6.2.1.2

3

Mastering Apache Spark

Management Scripts for Standalone Master

Management Scripts for Standalone Workers

Checking Status

6.2.1.3 6.2.1.4 6.2.1.5

Example 2-workers-on-1-node Standalone Cluster (one executor per

worker)

Spark on Mesos

Spark on YARN

Execution Model

Advanced Concepts of Spark

Broadcast variables

Accumulators

Security

Spark Security

Securing Web UI

Data Sources in Spark

Using Input and Output (I/O)

Spark and Parquet

Serialization

Using Apache Cassandra

Using Apache Kafka

Spark Application Frameworks

Spark Streaming

StreamingContext

Stream Operators

Windowed Operators

SaveAs Operators

Stateful Operators

web UI and Streaming Statistics Page Streaming Listeners

Checkpointing

JobScheduler

JobGenerator

DStreamGraph

Discretized Streams (DStreams)

6.2.1.6 6.2.2

6.2.3

7

8

8.1

8.2

9

9.1

9.2

10

10.1

10.1.1 10.1.2 10.2

10.3

11

11.1

11.1.1 11.1.2 11.1.2.1 11.1.2.2 11.1.2.3 11.1.3 11.1.4 11.1.5 11.1.6 11.1.7 11.1.8 11.1.9

4

Mastering Apache Spark

Input DStreams

ReceiverInputDStreams

ConstantInputDStreams

ForEachDStreams

WindowedDStreams

MapWithStateDStreams

StateDStreams

TransformedDStream

Receivers

ReceiverTracker

ReceiverSupervisors

ReceivedBlockHandlers

Ingesting Data from Kafka

KafkaRDD

RecurringTimer

Streaming DataFrames

Backpressure

Dynamic Allocation (Elastic Scaling)

Settings

Spark SQL

SQLContext

Dataset

DataFrame

DataFrameReaders

ContinuousQueryManager

Aggregation (GroupedData)

Windows in DataFrames

Catalyst optimizer

Into the depths

Datasets vs RDDs

Settings

Spark MLlib - Machine Learning in Spark ML Pipelines

11.1.9.1 11.1.9.2 11.1.9.3 11.1.9.4 11.1.9.5 11.1.9.6 11.1.9.7 11.1.9.8

11.1.10 11.1.10.1 11.1.10.2 11.1.10.3

11.1.11 11.1.11.1 11.1.12 11.1.13 11.1.14 11.1.15 11.1.16 11.2

11.2.1

11.2.2

11.2.3

11.2.4

11.2.5

11.2.6

11.2.7

11.2.8

11.2.9

11.2.10 11.2.11 11.3

11.3.1

5

Mastering Apache Spark

Distributed graph computations with GraphX

Monitoring, Tuning and Debugging

Logging

Performance Tuning

Spark Metrics System

Scheduler Listeners

Debugging Spark using sbt

Varia

Building Spark

Spark and Hadoop

Spark and software in-memory file systems

Spark and The Others

Distributed Deep Learning on Spark

Spark Packages

Spark Tips and Tricks

Access private members in Scala in Spark shell

SparkException: Task not serializable

Exercises

One-liners using PairRDDFunctions

Learning Jobs and Partitions Using take Action

Spark Standalone - Using ZooKeeper for High-Availability of Master Spark’s Hello World using Spark shell and Scala

WordCount using Spark shell

Your first complete Spark application (using Scala and sbt)

Spark (notable) use cases

Using Spark SQL to update data in Hive using ORC files

Developing Custom SparkListener to monitor DAGScheduler in Scala Developing RPC Environment

Developing Custom RDD

Further Learning

Courses

Books

Commercial Products using Apache Spark

IBM Analytics for Apache Spark

11.4

12

12.1 12.2 12.3 12.4 12.5 13

13.1 13.2 13.3 13.4 13.5 13.6 14

14.1 14.2 15

15.1 15.2 15.3 15.4 15.5 15.6 15.7 15.8 15.9

15.10 15.11 16

16.1 16.2 17

17.1 6

Mastering Apache Spark

Google Cloud Dataproc

Spark Advanced Workshop

Requirements

Day 1

Day 2

Spark Talks Ideas (STI)

10 Lesser-Known Tidbits about Spark Standalone

Learning Spark internals using groupBy (to cause shuffle) Glossary

17.2 18

18.1 18.2 18.3 19

19.1 19.2

7

Mastering Apache Spark

**Mastering Apache Spark**

Welcome to Mastering Apache Spark!

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

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

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

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

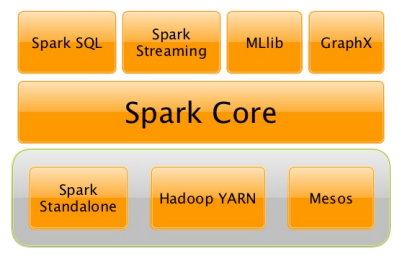
Expect text and code snippets from Spark’s mailing lists, the official documentation of Apache Spark, StackOverflow, blog posts, books from O’Reilly, press releases, YouTube/Vimeo videos, Quora, the source code of Apache Spark, etc. Attribution follows.

Introduction 8

Mastering Apache Spark

**Overview of Spark**

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

Figure 1. The Spark Platform

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

**Why Spark**

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

**Diverse Workloads**

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

Overview of Spark 9

Mastering Apache Spark

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

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

Spark also supports **near real-time streaming workloads** via Spark Streaming application framework.

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

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

There is also support for interactive workloads using Spark shell.

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

**Leverages the Best in distributed batch data processing**

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

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

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

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

**RDD - Distributed Parallel Scala Collections**

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

It is also exposed in Java, Python and R (as well as SQL, i.e. SparkSQL, in a sense). Overview of Spark 10

Mastering Apache Spark

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

**Rich Standard Library**

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

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

**Unified development and deployment environment for all**

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

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

**Interactive exploration**

It is also called *ad hoc queries*.

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

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

Overview of Spark 11

Mastering Apache Spark

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

**Single environment**

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

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

Or use them all in a single application.

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

**Rich set of supported data sources**

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

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

**Tools unavailable then, at your fingertips now**

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

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

Machine learning that is so tool- and feature-rich in Python, e.g. SciKit library, can now be used by Scala developers (as Pipeline API in Spark MLlib or calling pipe() ).

DataFrames from R are available in Scala, Java, Python, R APIs.

Single node computations in machine learning algorithms are migrated to their distributed versions in Spark MLlib.

Overview of Spark 12

Mastering Apache Spark

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

Mind the proverb if all you have is a hammer, everything looks like a nail, too. **Low-level Optimizations**

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

Mind the proverb less is more.

**Excels at low-latency iterative workloads**

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

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

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

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

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

**ETL done easier**

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

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

Overview of Spark 13

Mastering Apache Spark

**Unified API (for different computation models)**

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

**Different kinds of data processing using unified API**

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

**Little to no disk use for better performance**

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

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

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

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

**Fault Tolerance included**

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

**Small Codebase Invites Contributors**

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

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

Overview of Spark 14

Mastering Apache Spark

**Overview**

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

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

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

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

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

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

Using Spark Application Frameworks, Spark simplifies access to machine learning and predictive analytics at scale.

Spark is mainly written in Scala, but supports other languages, i.e. Java, Python, and R.

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

Access any data type across any data source.

Huge demand for storage and data processing.

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

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

Spark can access data from many data sources.

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

Overview of Spark 15

Mastering Apache Spark

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

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

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

In Going from Hadoop to Spark: A Case Study, Sujee Maniyam 20150223: Spark is like emacs - once you join emacs, you can’t leave emacs.

Overview of Spark 16

Mastering Apache Spark

**Anatomy of Spark Application**

Every Spark application starts at instantiating a Spark context. Without a Spark context no computation can ever be started using Spark services.

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

For it to work, you have to create a Spark configuration using SparkConf or use a custom SparkContext constructor.

package pl.japila.spark

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

object SparkMeApp {

def main(args: Array[String]) {

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

val conf = new SparkConf() (2)

.setAppName("SparkMe Application")

.setMaster(masterURL)

val sc = new SparkContext(conf) (3)

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

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

val c = lines.count() (5)

println(s"There are $c lines in $fileName")

}

}

1. Master URL to connect the application to

2. Create Spark configuration

3. Create Spark context

4. Create lines RDD

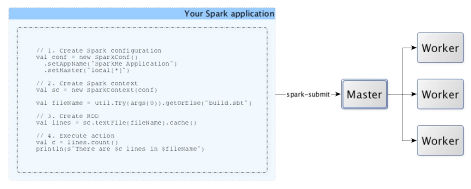
5. Execute count action

| Tip | Spark shell creates a Spark context and SQL context for you at startup. |
| --- | --- |

Anatomy of Spark Application 17

Mastering Apache Spark

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.

Anatomy of Spark Application 18

Mastering Apache Spark

**SparkConf - Configuration for Spark Applications**

| Tip | Refer to Spark Configuration for extensive coverage of how to configure Spark and user programs. |
| --- | --- |

| Caution | TODO  Describe SparkConf object for the application configuration. the default configs  system properties |
| --- | --- |

There are three ways to configure Spark and user programs:

Spark Properties - use Web UI to learn the current properties.

…

**Spark Properties**

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

| Tip | 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:

SparkConf - Configuration for Spark Applications 19

Mastering Apache Spark

scala> sc.getConf.getOption("spark.local.dir")

res0: Option[String] = None

scala> sc.getConf.getOption("spark.app.name")

res1: Option[String] = Some(Spark shell)

scala> sc.getConf.get("spark.master")

res2: String = local[\*]

**Setting up Properties**

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

conf/spark-defaults.conf - the default

--conf - the command line option used by spark-shell and spark-submit SparkConf

**Default Configuration**

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

import org.apache.spark.SparkConf

val conf = new SparkConf

It merely loads any spark.\* system properties.

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

SparkConf - Configuration for Spark Applications 20

Mastering Apache Spark

scala> conf.getAll

res0: Array[(String, String)] = Array((spark.app.name,Spark shell), (spark.jars,""), (spark.master,l

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

SparkConf - Configuration for Spark Applications 21

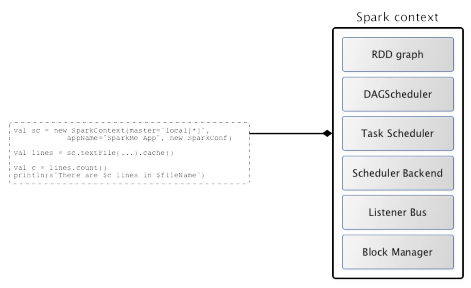
Mastering Apache Spark

**SparkContext - the door to Spark**

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

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

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

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

SparkContext offers the following functions:

Default Level of Parallelism

Specifying mandatory master URL

Specifying mandatory application name

Creating RDDs

Creating accumulators

Creating broadcast variables

SparkContext - the door to Spark 22

Mastering Apache Spark

Accessing services, e.g. Task Scheduler, Listener Bus, Block Manager, Scheduler Backends, Shuffle Manager.

Running jobs

Setting up custom Scheduler Backend, Task Scheduler and DAGScheduler Closure Cleaning

Submitting Jobs Asynchronously

Unpersisting RDDs, i.e. marking RDDs as non-persistent

Read the scaladoc of org.apache.spark.SparkContext.

**Master URL**

| Caution | FIXME |
| --- | --- |

Connecting to a cluster

**Application Name**

| Caution | FIXME |
| --- | --- |

Specifying mandatory application name

**Default Level of Parallelism**

**Default level of parallelism** is the number of partitions when not specified explicitly by a user.

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

SparkContext queries TaskScheduler for the default level of parallelism (refer to TaskScheduler Contract).

**SparkContext.makeRDD**

| Caution | FIXME |
| --- | --- |

SparkContext - the door to Spark 23

Mastering Apache Spark

**Submitting Jobs Asynchronously**

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

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

| Caution | FIXME What are resultFunc ? |
| --- | --- |

It is used in:

AsyncRDDActions methods

Spark Streaming for ReceiverTrackerEndpoint.startReceiver

**Spark Configuration**

| Caution | FIXME |
| --- | --- |

**Creating SparkContext**

You create a SparkContext instance using a SparkConf object.

scala> import org.apache.spark.SparkConf

import org.apache.spark.SparkConf

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

scala> import org.apache.spark.SparkContext

import org.apache.spark.SparkContext

scala> val sc = new SparkContext(conf) (1)

sc: org.apache.spark.SparkContext = org.apache.spark.SparkContext@50ee2523

1. You can also use the other constructor of SparkContext , i.e. new

SparkContext(master="local[\*]", appName="SparkMe App", new SparkConf) , with master and application name specified explicitly

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

INFO SparkContext: Running Spark version 1.6.0-SNAPSHOT

SparkContext - the door to Spark 24

Mastering Apache Spark

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

**spark.driver.allowMultipleContexts**

Quoting the scaladoc of org.apache.spark.SparkContext:

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

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

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

**SparkContext and RDDs**

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

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

SparkContext - the door to Spark 25

Mastering Apache Spark

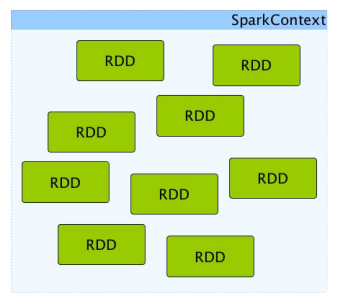


Figure 2. A Spark context creates a living space for RDDs.

**SparkContext in Spark shell**

In Spark shell, an instance of SparkContext is automatically created for you under the name sc .

Read Spark shell.

**Creating RDD**

SparkContext allows you to create many different RDDs from input sources like: Scala’s collections, i.e. sc.parallelize(0 to 100)

local or remote filesystems, i.e. sc.textFile("README.md")

Any Hadoop InputSource using sc.newAPIHadoopFile

Read Creating RDDs in RDD - Resilient Distributed Dataset.

**Unpersisting RDDs (Marking RDDs as non-persistent)**

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

SparkContext - the door to Spark 26

Mastering Apache Spark

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

**Setting Checkpoint Directory (setCheckpointDir method)** setCheckpointDir(directory: String)

setCheckpointDir method is used to set up the checkpoint directory…FIXME

| Caution | FIXME |
| --- | --- |

**Creating accumulators**

| Caution | FIXME |
| --- | --- |

Read Use accumulators.

**Creating Broadcast Variables**

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

SparkContext comes with broadcast method to broadcast a value among Spark executors. def broadcast[T: ClassTag](value: T): Broadcast[T]

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

scala> sc.broadcast("hello")

INFO MemoryStore: Ensuring 1048576 bytes of free space for block broadcast\_0(free: 535953408, max: 5INFO MemoryStore: Ensuring 80 bytes of free space for block broadcast\_0(free: 535953408, max: 535953INFO MemoryStore: Block broadcast\_0 stored as values in memory (estimated size 80.0 B, free 80.0 B)

INFO MemoryStore: Ensuring 34 bytes of free space for block broadcast\_0\_piece0(free: 535953328, max: INFO MemoryStore: Block broadcast\_0\_piece0 stored as bytes in memory (estimated size 34.0 B, free 11INFO BlockManagerInfo: Added broadcast\_0\_piece0 in memory on localhost:61505 (size: 34.0 B, free: 51INFO SparkContext: Created broadcast 0 from broadcast at <console>:25

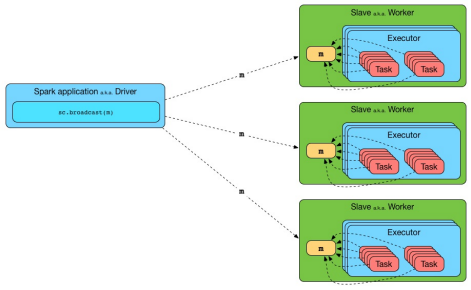
res0: org.apache.spark.broadcast.Broadcast[String] = Broadcast(0)



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

SparkContext - the door to Spark 27

Mastering Apache Spark

Figure 3. Broadcasting a value to executors

When a broadcast value is created the following INFO message appears in the logs: INFO SparkContext: Created broadcast [id] from broadcast at <console>:25

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

scala> val rdd = sc.parallelize(0 to 10)

rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:24

scala> sc.broadcast(rdd)

WARN SparkContext: Can not directly broadcast RDDs; instead, call collect() and broadcast the result 

**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.

SparkContext - the door to Spark 28

Mastering Apache Spark

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 

| Caution | FIXME Why is HttpFileServer used for addJar? |
| --- | --- |

**SparkContext as the global configuration for services** SparkContext keeps track of:

shuffle ids using nextShuffleId internal field for registering shuffle dependencies to Shuffle Service.

**Running Jobs**

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

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

scala> import org.apache.spark.TaskContext

import org.apache.spark.TaskContext

scala> sc.runJob(lines, (t: TaskContext, i: Iterator[String]) => 1) (1)

res0: Array[Int] = Array(1, 1) (2)

1. Run a job using runJob on lines RDD with a function that returns 1 for every partition (of lines RDD).

2. What can you say about the number of partitions of the lines RDD? Is your result res0 different than mine? Why?

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

SparkContext.runJob prints out the following INFO message:

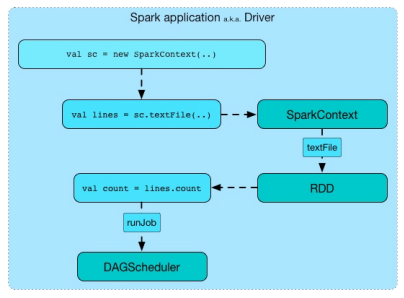
INFO Starting job: ...

And it follows up on spark.logLineage and then hands over the execution to

DAGScheduler.runJob().

SparkContext - the door to Spark 29

Mastering Apache Spark

Figure 4. Executing action

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

| Caution | Spark can only run jobs when a Spark context is available and active, i.e. started. See Stopping Spark context.  Since SparkContext runs inside a Spark driver, i.e. a Spark application, it must be alive to run jobs. |
| --- | --- |

**Stopping SparkContext**

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

Calling stop many times leads to the following INFO message in the logs: INFO SparkContext: SparkContext already stopped.

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

java.lang.IllegalStateException: SparkContext has been shutdown .

SparkContext - the door to Spark 30

Mastering Apache Spark

scala> sc.stop

scala> sc.parallelize(0 to 5)

java.lang.IllegalStateException: Cannot call methods on a stopped SparkContext.

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

Posts a application end event SparkListenerApplicationEnd to Listener Bus Stops web UI

Requests MetricSystem to report metrics from all registered sinks (using MetricsSystem.report() )

metadataCleaner.cancel()

Stops ContextCleaner

Stops ExecutorAllocationManager

Stops DAGScheduler

Stops Listener Bus

Stops EventLoggingListener

Stops HeartbeatReceiver

Stops ConsoleProgressBar

Stops SparkEnv

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

**HeartbeatReceiver**

| Caution | FIXME |
| --- | --- |

HeartbeatReceiver is a SparkListener.

**Custom SchedulerBackend, TaskScheduler and DAGScheduler**

SparkContext - the door to Spark 31

Mastering Apache Spark

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

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

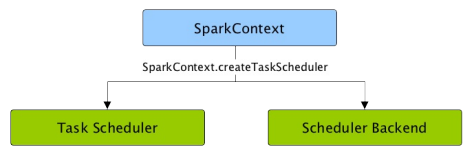
It is also applicable to SchedulerBackend and TaskScheduler using schedulerBackend\_=(sb: SchedulerBackend) and taskScheduler\_=(ts: TaskScheduler) methods, respectively.

| Caution | FIXME Make it an advanced exercise. |
| --- | --- |

**Creating SchedulerBackend and TaskScheduler**

SparkContext.createTaskScheduler is executed as part of SparkContext’s initialization to create Task Scheduler and Scheduler Backend.

It uses the master URL to select right implementations.

Figure 5. SparkContext creates Task Scheduler and Scheduler Backend

**Events**

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

Refer to the section SparkContext’s initialization.

**Persisted RDDs**

FIXME When is the internal field persistentRdds used?

**Setting Default Log Level Programatically**

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

SparkContext - the door to Spark 32

Mastering Apache Spark

| Tip | sc.setLogLevel("INFO") becomes org.apache.log4j.Level.toLevel(logLevel) and org.apache.log4j.Logger.getRootLogger().setLevel(l) internally.  See org/apache/spark/SparkContext.scala. |
| --- | --- |

**SparkStatusTracker**

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

SparkStatusTracker is only used by ConsoleProgressBar.

**ConsoleProgressBar**

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

[Stage 0:====> (316 + 4) / 1000][Stage 1:> (0 + 0) / 1000][Stage 2:> The progress includes the stage’s id, the number of completed, active, and total tasks. It is useful when you ssh to workers and want to see the progress of active stages.

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

import org.apache.log4j.\_

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

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

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

See the progress bar in Spark shell with the following:

SparkContext - the door to Spark 33

Mastering Apache Spark

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

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

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

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

1. Make sure spark.ui.showConsoleProgress is true . It is by default.

2. Disable ( OFF ) the root logger (that includes Spark’s logger)

3. Make sure org.apache.spark.SparkContext logger is at least WARN .

4. Run a job with 4 tasks with 500ms initial sleep and 200ms sleep chunks to see the progress bar.

Watch the short video that show ConsoleProgressBar in action.

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

> ./bin/spark-shell --conf spark.scheduler.mode=FAIR

scala> val a = sc.makeRDD(1 to 1000, 10000).map(x => (x, x)).reduceByKey(\_ + \_) scala> val b = sc.makeRDD(1 to 1000, 10000).map(x => (x, x)).reduceByKey(\_ + \_) scala> a.union(b).count()

**Closure Cleaning (clean method)**

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

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

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

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

SparkContext - the door to Spark 34

Mastering Apache Spark

| Tip | Enable DEBUG logging level for org.apache.spark.util.ClosureCleaner logger to see what happens inside the class.  Add the following line to conf/log4j.properties :  log4j.logger.org.apache.spark.util.ClosureCleaner=DEBUG |
| --- | --- |

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

+++ Cleaning closure [func] ([func.getClass.getName]) +++

+ declared fields: [declaredFields.size]

[field]

...

+++ closure [func] ([func.getClass.getName]) is now cleaned +++

Serialization is verified using a new instance of Serializer (as

SparkEnv.get.closureSerializer ). Refer to Serialization.

| Caution | FIXME an example, please. |
| --- | --- |

**Creating a SparkContext**

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

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

// 1. Create Spark configuration

val conf = new SparkConf()

.setAppName("SparkMe Application")

.setMaster("local[\*]")

// 2. Create Spark context

val sc = new SparkContext(conf)

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

It all starts with checking whether SparkContexts can be shared or not using spark.driver.allowMultipleContexts .

The very first information printed out is the version of Spark as an INFO message: INFO SparkContext: Running Spark version 1.6.0-SNAPSHOT

SparkContext - the door to Spark 35

Mastering Apache Spark

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

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

scala> sc.sparkUser

res0: String = jacek

| Caution | FIXME Where is sparkUser useful? |
| --- | --- |

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

When spark.logConf is true (default: false ) SparkConf.toDebugString is called.

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

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

spark.executor.id is set as driver .

| Tip | Use sc.getConf.get("spark.executor.id") to know where the code is executed - driver or executors. |
| --- | --- |

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

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

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

| Caution | FIXME: What’s externalBlockStoreFolderName ? |
| --- | --- |

For yarn-client master URL, the system property SPARK\_YARN\_MODE is set to true . An instance of JobProgressListener is created and registered to Listener Bus.

SparkContext - the door to Spark 36

Mastering Apache Spark

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

MetadataCleaner is created.

| Caution | FIXME What’s MetadataCleaner? |
| --- | --- |

Optional ConsoleProgressBar with SparkStatusTracker are created.

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

| Caution | FIXME Step through SparkUI.createLiveUI . Where’s \_ui used? |
| --- | --- |

A Hadoop configuration is created. See Hadoop Configuration.

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

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

\_executorMemory is later available as sc.executorMemory and used for LOCAL\_CLUSTER\_REGEX, Spark Standalone’s SparkDeploySchedulerBackend, to set executorEnvs("SPARK\_EXECUTOR\_MEMORY") , MesosSchedulerBackend,

CoarseMesosSchedulerBackend.

The value of SPARK\_PREPEND\_CLASSES environment variable is included in executorEnvs .

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

The Mesos scheduler backend’s configuration is included in executorEnvs , i.e. SPARK\_EXECUTOR\_MEMORY, \_conf.getExecutorEnv , and SPARK\_USER .

**HeartbeatReceiver** RPC Endpoint is created using HeartbeatReceiver (as \_heartbeatReceiver ). Refer to HeartbeatReceiver.

SparkContext.createTaskScheduler is executed (using the master URL). SparkContext - the door to Spark 37

Mastering Apache Spark

| Caution | FIXME Why is \_heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet) important? |
| --- | --- |

Task Scheduler is started.

The internal fields, \_applicationId and \_applicationAttemptId , are set. Application and attempt ids are specific to the implementation of Task Scheduler.

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

scala> sc.getConf.get("spark.app.id")

res1: String = local-1447834845413

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

Metric System is started (after the application id is set using spark.app.id ).

| Caution | FIXME Why does Metric System need the application id? |
| --- | --- |

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

\_eventLogger is created and started if isEventLogEnabled . It uses EventLoggingListener that gets registered to Listener Bus.

| Caution | FIXME Why is \_eventLogger required to be the internal field of SparkContext? Where is this used? |
| --- | --- |

If dynamic allocation is enabled, \_executorAllocationManager is set to ExecutorAllocationManager and started.

\_cleaner is set to ContextCleaner if spark.cleaner.referenceTracking is true (default: true ).

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

setupAndStartListenerBus registers user-defined listeners and starts Listener Bus that starts event delivery to the listeners.

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

SparkContext - the door to Spark 38

Mastering Apache Spark

postApplicationStart is called to post SparkListenerApplicationStart event over Listener Bus.

TaskScheduler.postStartHook() is called (see TaskScheduler Contract) Two new metrics sources are registered (via \_env.metricsSystem ):

BlockManagerSource

ExecutorAllocationManagerSource (only when \_executorAllocationManager is set) ShutdownHookManager.addShutdownHook() is called to do SparkContext’s cleanup.

| Caution | FIXME What exactly does ShutdownHookManager.addShutdownHook() do? |
| --- | --- |

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

| Caution | FIXME What does NonFatal represent in Scala? |
| --- | --- |

nextShuffleId and nextRddId start with 0 .

| Caution | FIXME Where are nextShuffleId and nextRddId used? |
| --- | --- |

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

**Hadoop Configuration**

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

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

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

If AWS\_ACCESS\_KEY\_ID and AWS\_SECRET\_ACCESS\_KEY are both available, the following settings are set for the Hadoop configuration:

fs.s3.awsAccessKeyId , fs.s3n.awsAccessKeyId , fs.s3a.access.key are set to the value of AWS\_ACCESS\_KEY\_ID

fs.s3.awsSecretAccessKey , fs.s3n.awsSecretAccessKey , and fs.s3a.secret.key are set to the value of AWS\_SECRET\_ACCESS\_KEY

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

SparkContext - the door to Spark 39

Mastering Apache Spark

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

**Environment Variables**

SPARK\_EXECUTOR\_MEMORY sets the amount of memory to allocate to each executor. See Executor Memory.

SparkContext - the door to Spark 40

Mastering Apache Spark

**RDD - Resilient Distributed Dataset**

**Introduction**

The origins of RDD

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

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

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

Learning about RDD by its name:

**Resilient**, i.e. fault-tolerant and so able to recompute missing or damaged partitions on node failures with the help of RDD lineage graph.

**Distributed** across clusters.

**Dataset** is a collection of partitioned data.

Figure 1. RDDs

From the scaladoc of org.apache.spark.rdd.RDD:

RDD - Resilient Distributed Dataset 41

Mastering Apache Spark

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

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

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

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

**Immutable**, i.e. it does not change once created.

**Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.

**Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).

**Parallel**, i.e. process data in parallel.

RDDs are distributed by design and to achieve even **data distribution** as well as leverage data locality (in distributed systems like HDFS or Cassandra in which data is partitioned by default), they are **partitioned** to a fixed number of partitions - logical chunks (parts) of data.

The logical division is for processing only and internally it is not divided whatsoever. Each partition comprises of **records**.

Figure 2. RDDs

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

RDD - Resilient Distributed Dataset 42

Mastering Apache Spark

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

RDDs support two kinds of operations:

transformations - lazy operations that return another RDD.

actions - operations that trigger computation and return values.

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

**iterative algorithms** in machine learning and graph computations.

**interactive data mining tools** as ad-hoc queries on the same dataset.

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

Each RDD is characterized by five main properties:

An array of partitions that a dataset is divided to

A function to do a computation for a partition

List of parent RDDs

An optional partitioner that defines how keys are hashed, and the pairs partitioned (for key-value RDDs)

Optional preferred locations, i.e. hosts for a partition where the data will have been loaded.

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

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

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

RDD - Resilient Distributed Dataset 43

Mastering Apache Spark

scala> val ns = sc.parallelize(0 to 10)

ns: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:24

scala> ns.id

res0: Int = 2

scala> ns.name

res1: String = null

scala> ns.name = "Friendly name"

ns.name: String = Friendly name

scala> ns.name

res2: String = Friendly name

scala> ns.toDebugString

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

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

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

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

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

**Types of RDDs**

There are some of the most interesting types of RDDs:

ParallelCollectionRDD

CoGroupedRDD

HadoopRDD is an RDD that provides core functionality for reading data stored in HDFS using the older MapReduce API. The most notable use case is the return RDD of SparkContext.textFile .

**MapPartitionsRDD** - a result of calling operations like map , flatMap , filter , mapPartitions, etc.

**CoalescedRDD** - a result of calling operations like repartition and coalesce RDD - Resilient Distributed Dataset 44

Mastering Apache Spark

ShuffledRDD - a result of shuffling, e.g. after repartition and coalesce **PipedRDD** - an RDD created by piping elements to a forked external process.

**PairRDD** (implicit conversion as org.apache.spark.rdd.PairRDDFunctions ) that is an RDD of key-value pairs that is a result of groupByKey and join operations.

**DoubleRDD** (implicit conversion as org.apache.spark.rdd.DoubleRDDFunctions ) that is an RDD of Double type.

**SequenceFileRDD** (implicit conversion as

org.apache.spark.rdd.SequenceFileRDDFunctions ) that is an RDD that can be saved as a SequenceFile .

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

**Transformations**

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

Go in-depth in the section Transformations in Operations - Transformations and Actions. **Actions**

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

Go in-depth in the section Actions in Operations - Transformations and Actions. **Creating RDDs**

**SparkContext.parallelize**

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

scala> val rdd = sc.parallelize(1 to 1000)

rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:25 You may also want to randomize the sample data:

RDD - Resilient Distributed Dataset 45

Mastering Apache Spark

scala> val data = Seq.fill(10)(util.Random.nextInt)

data: Seq[Int] = List(-964985204, 1662791, -1820544313, -383666422, -111039198, 310967683, 1114081267

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 map it over to have an RDD of sequences of words):

scala> val words = sc.textFile("README.md").flatMap(\_.split("\\s+")).cache()

words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[27] at flatMap at <console>:24 

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

**Creating RDDs from Input**

Refer to Using Input and Output (I/O) to learn about the IO API to create RDDs. **Transformations**

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

RDD - Resilient Distributed Dataset 46

Mastering Apache Spark

Refer to Transformations section to learn more.

**RDDs in Web UI**

It is quite informative to look at RDDs in the Web UI that is at http://localhost:4040 for Spark shell.

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

val ints = sc.parallelize(1 to 100) (1)

ints.setName("Hundred ints") (2)

ints.cache (3)

ints.count (4)

1. Creates an RDD with hundreds of numbers (with as many partitions as possible) 2. Sets the name of the RDD

3. Caches the RDD (so it shows up in Storage in UI)

4. Executes action (and materializes the RDD)

With the above executed, you should see the following in the Web UI: Figure 3. RDD with custom name

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

RDD - Resilient Distributed Dataset 47

Mastering Apache Spark

Figure 4. RDD Storage Info

Execute the following Spark job and you will see how the number of partitions decreases. ints.repartition(2).count

Figure 5. Number of tasks after repartition

**Computing Partition of RDD (using compute method)**

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

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

RDD - Resilient Distributed Dataset 48

Mastering Apache Spark

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

compute method runs on a driver.

**Preferred Locations**

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

def getPreferredLocations(split: Partition): Seq[String] specifies placement preferences for a partition in an RDD.

**RDD Lineage Graph**

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



Figure 6. RDD lineage

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

val r00 = sc.parallelize(0 to 9)

val r01 = sc.parallelize(0 to 90 by 10)

val r10 = r00 cartesian r01

val r11 = r00.map(n => (n, n))

val r12 = r00 zip r01

val r13 = r01.keyBy(\_ / 20)

val r20 = Seq(r11, r12, r13).foldLeft(r10)(\_ union \_)

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

You can learn about a RDD lineage graph using RDD.toDebugString method. RDD - Resilient Distributed Dataset 49

Mastering Apache Spark

**toDebugString**

You can learn about a RDD lineage graph using RDD.toDebugString method.

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

scala> wordsCount.toDebugString

res2: String =

(2) ShuffledRDD[24] at reduceByKey at <console>:24 []

+-(2) MapPartitionsRDD[23] at map at <console>:24 []

| MapPartitionsRDD[22] at flatMap at <console>:24 []

| MapPartitionsRDD[21] at textFile at <console>:24 []

| README.md HadoopRDD[20] at textFile at <console>:24 []

**spark.logLineage**

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

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

scala> sc.textFile("README.md", 4).count

...

15/10/17 14:46:42 INFO SparkContext: Starting job: count at <console>:25

15/10/17 14:46:42 INFO SparkContext: RDD's recursive dependencies:

(4) MapPartitionsRDD[1] at textFile at <console>:25 []

| README.md HadoopRDD[0] at textFile at <console>:25 []

**Execution Plan**

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

RDD - Resilient Distributed Dataset 50

Mastering Apache Spark

**Operators - Transformations and Actions** RDDs have two fundamental types of operations: Transformations and Actions. **Transformations**

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

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

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

Operators - Transformations and Actions 51

Mastering Apache Spark

Figure 1. From SparkContext by transformations to the result

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

scala> val file = sc.textFile("README.md")

file: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[54] at textFile at <console>:24

scala> val allWords = file.flatMap(\_.split("\\s+"))

allWords: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[55] at flatMap at <console>:

scala> val words = allWords.filter(!\_.isEmpty)

words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[56] at filter at <console>:28

scala> val pairs = words.map((\_,1))

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

scala> val reducedByKey = pairs.reduceByKey(\_ + \_)

reducedByKey: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[59] at reduceByKey at <console>:

scala> val top10words = reducedByKey.takeOrdered(10)(Ordering[Int].reverse.on(\_.\_2)) INFO SparkContext: Starting job: takeOrdered at <console>:34

...

INFO DAGScheduler: Job 18 finished: takeOrdered at <console>:34, took 0.074386 s top10words: Array[(String, Int)] = Array((the,21), (to,14), (Spark,13), (for,11), (and,10

There are two kinds of transformations:

narrow transformations

wide transformations

**Narrow Transformations**

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

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

Spark groups narrow transformations as a stage.

**Wide Transformations**

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

Operators - Transformations and Actions 52

Mastering Apache Spark

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

**Actions**

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

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

They trigger execution of RDD transformations to return values. Simply put, an action evaluates the RDD lineage graph.

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

Actions in org.apache.spark.rdd.RDD:

aggregate

collect

count

countApprox\*

countByValue\*

first

fold

foreach

foreachPartition

max

min

reduce

saveAs\* actions, e.g. saveAsTextFile , saveAsHadoopFile

take

Operators - Transformations and Actions 53

Mastering Apache Spark

takeOrdered

takeSample

toLocalIterator

top

treeAggregate

treeReduce

Actions run jobs using SparkContext.runJob or directly DAGScheduler.runJob.

scala> words.count (1)

res0: Long = 502

1. words is an RDD of String .

| Tip | You should cache an RDD you work with when you want to execute two or more actions on it for better performance. Refer to RDD Caching / Persistence. |
| --- | --- |

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

| Note | Spark uses ClosureCleaner to clean closures. |
| --- | --- |

**AsyncRDDActions**

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

The following asynchronous methods are available:

countAsync

collectAsync

takeAsync

foreachAsync

foreachPartitionAsync

**FutureActions**

Operators - Transformations and Actions 54

Mastering Apache Spark

| Caution | FIXME |
| --- | --- |

**Gotchas - things to watch for**

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

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

Operators - Transformations and Actions 55

Mastering Apache Spark

**mapPartitions Operator**

| Caution | FIXME |
| --- | --- |

Operators - Transformations and Actions 56

Mastering Apache Spark

**Partitions**

**Introduction**

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

A **partition** (aka *split*) is…FIXME

| Caution | FIXME  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 | Read the following documentations to learn what experts say on the topic: How Many Partitions Does An RDD Have?  Tuning Spark (the official documentation of Spark) |
| --- | --- |

Partitions 57

Mastering Apache 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.

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

Partitions 58

Mastering Apache Spark

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 59

Mastering Apache Spark

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

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

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

It will only work as described for uncompressed files.

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

Some operations, e.g. map , flatMap , filter , don’t preserve partitioning. map , flatMap , filter operations apply a function to every partition.

**Repartitioning**

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

With the following computation you can see that repartition(5) causes 5 tasks to be started using NODE\_LOCAL data locality.

Partitions 60

Mastering Apache Spark

scala> lines.repartition(5).count

...

15/10/07 08:10:00 INFO DAGScheduler: Submitting 5 missing tasks from ResultStage 7 (MapPartitionsRDD[15/10/07 08:10:00 INFO TaskSchedulerImpl: Adding task set 7.0 with 5 tasks

15/10/07 08:10:00 INFO TaskSetManager: Starting task 0.0 in stage 7.0 (TID 17, localhost, partition 15/10/07 08:10:00 INFO TaskSetManager: Starting task 1.0 in stage 7.0 (TID 18, localhost, partition 15/10/07 08:10:00 INFO TaskSetManager: Starting task 2.0 in stage 7.0 (TID 19, localhost, partition 215/10/07 08:10:00 INFO TaskSetManager: Starting task 3.0 in stage 7.0 (TID 20, localhost, partition 15/10/07 08:10:00 INFO TaskSetManager: Starting task 4.0 in stage 7.0 (TID 21, localhost, partition ...



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 (MapPartitions15/10/07 08:14:09 INFO TaskSchedulerImpl: Adding task set 8.0 with 2 tasks

15/10/07 08:14:09 INFO TaskSetManager: Starting task 0.0 in stage 8.0 (TID 22, localhost, partition 15/10/07 08:14:09 INFO TaskSetManager: Starting task 1.0 in stage 8.0 (TID 23, localhost, partition ...



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**

Partitions 61

Mastering Apache Spark

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

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

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

scala> val rdd = sc.parallelize(0 to 10, 8)

rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> rdd.partitions.size

res0: Int = 8

scala> rdd.coalesce(numPartitions=8, shuffle=false) (1)

res1: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[1] at coalesce at <console>:27

scala> res1.toDebugString

res2: String =

(8) CoalescedRDD[1] at coalesce at <console>:27 []

| ParallelCollectionRDD[0] at parallelize at <console>:24 []

scala> rdd.coalesce(numPartitions=8, shuffle=true)

res3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[5] at coalesce at <console>:27

scala> res3.toDebugString

res4: String =

(8) MapPartitionsRDD[5] at coalesce at <console>:27 []

| CoalescedRDD[4] at coalesce at <console>:27 []

| ShuffledRDD[3] at coalesce at <console>:27 []

+-(8) MapPartitionsRDD[2] at coalesce at <console>:27 []

| ParallelCollectionRDD[0] at parallelize at <console>:24 []



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

**PairRDDFunctions - groupByKey, reduceByKey, partitionBy** You may want to look at the number of partitions from another angle.

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

Partitions 62

Mastering Apache Spark

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

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

rdd.keyBy(\_.kind)

.partitionBy(new HashPartitioner(PARTITIONS))

.foreachPartition(...)

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

You could do as follows:

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

or mapValues or plenty of other solutions. *FIXME, man*.

**Partitioner**

| Caution | FIXME |
| --- | --- |

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

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

| Caution | FIXME |
| --- | --- |

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

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

Partitions 63

Mastering Apache Spark

**RDD Caching and Persistence**

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

The cache() operation is a synonym of persist() that uses the default storage level MEMORY\_ONLY .

RDDs can be unpersisted.

**Storage Levels**

StorageLevel describes how an RDD is persisted (and addresses the following concerns): Does RDD use disk?

How much of RDD is in memory?

Does RDD use off-heap memory?

Should an RDD be serialized (while persisting)?

How many replicas (default: 1 ) to use (can only be less than 40 )?

There are the following StorageLevel (number \_2 in the name denotes 2 replicas): NONE (default)

DISK\_ONLY

DISK\_ONLY\_2

MEMORY\_ONLY (default for cache() operation)

MEMORY\_ONLY\_2

MEMORY\_ONLY\_SER

MEMORY\_ONLY\_SER\_2

MEMORY\_AND\_DISK

MEMORY\_AND\_DISK\_2

MEMORY\_AND\_DISK\_SER

MEMORY\_AND\_DISK\_SER\_2

Caching and Persistence 64

Mastering Apache Spark

OFF\_HEAP

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

scala> val lines = sc.textFile("README.md")

lines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at textFile at <console>:24

scala> lines.getStorageLevel

res2: org.apache.spark.storage.StorageLevel = StorageLevel(false, false, false, false, 1) 

**Unpersisting RDDs (Clearing Blocks)**

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

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

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

Caching and Persistence 65

Mastering Apache Spark

**RDD shuffling**

| Tip | Read the official documentation about the topic Shuffle operations. It is *still* better than this page. |
| --- | --- |

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

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

By default, shuffling doesn’t change the number of partitions, but their content. Avoid groupByKey and use reduceByKey or combineByKey instead.

groupByKey shuffles all the data, which is slow.

reduceByKey shuffles only the results of sub-aggregations in each partition of the data.

**Example - join**

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

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

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

Shuffling 66

Mastering Apache Spark

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,

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)

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) |
| --- | --- |

Shuffling 67

Mastering Apache Spark

join operation is one of the **cogroup operations** that uses defaultPartitioner , i.e. walks through the RDD lineage graph (sorted by the number of partitions decreasing) and picks the partitioner with positive number of output partitions. Otherwise, it checks

spark.default.parallelism setting and if defined picks HashPartitioner with the default parallelism of the scheduler backend.

join is almost CoGroupedRDD.mapValues .

| Caution | FIXME the default parallelism of scheduler backend |
| --- | --- |

Shuffling 68

Mastering Apache Spark

**Checkpointing**

**Introduction**

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

There are two types of checkpointing:

**reliable** - in Spark (core), RDD checkpointing that saves the actual intermediate RDD data to a reliable distributed file system, e.g. HDFS.

**local** - in Spark Streaming or GraphX - RDD checkpointing that truncates RDD lineage graph.

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

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

**Reliable Checkpointing**

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

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

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

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

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

Checkpointing 69

Mastering Apache Spark

**ReliableRDDCheckpointData**

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

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

**ReliableCheckpointRDD**

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

**Local Checkpointing**

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

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

Local checkpointing trades fault-tolerance for performance.

The checkpoint directory set through SparkContext.setCheckpointDir is not used. **LocalRDDCheckpointData**

FIXME

**LocalCheckpointRDD**

FIXME

Checkpointing 70

Mastering Apache Spark

**Dependencies**

**Dependency** (represented by Dependency class) is a connection between RDDs after applying a transformation.

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

scala> val r1 = sc.parallelize(0 to 9)

r1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[20] at parallelize at <console>:18

scala> val r2 = sc.parallelize(0 to 9)

r2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[21] at parallelize at <console>:18

scala> val r3 = sc.parallelize(0 to 9)

r3: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[22] at parallelize at <console>:18

scala> val r4 = sc.union(r1, r2, r3)

r4: org.apache.spark.rdd.RDD[Int] = UnionRDD[23] at union at <console>:24

scala> r4.dependencies

res0: Seq[org.apache.spark.Dependency[\_]] = ArrayBuffer(org.apache.spark.RangeDependency@6f2ab3f6, or

scala> r4.toDebugString

res1: String =

(24) UnionRDD[23] at union at <console>:24 []

| ParallelCollectionRDD[20] at parallelize at <console>:18 []

| ParallelCollectionRDD[21] at parallelize at <console>:18 []

| ParallelCollectionRDD[22] at parallelize at <console>:18 []

scala> r4.collect

...

res2: Array[Int] = Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 

**Kinds of Dependencies**

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

scala> val r = sc.parallelize(0 to 9).groupBy(identity)

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

scala> r.dependencies.map(\_.rdd).foreach(println)

MapPartitionsRDD[11] at groupBy at <console>:18

Dependencies 71

Mastering Apache Spark

There are the following more specialized Dependency extensions:

NarrowDependency

OneToOneDependency

PruneDependency

RangeDependency

ShuffleDependency

**ShuffleDependency**

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

scala> val r = sc.parallelize(0 to 9).groupBy(identity)

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

scala> r.dependencies

res0: Seq[org.apache.spark.Dependency[\_]] = List(org.apache.spark.ShuffleDependency@493b0b09) 

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

A ShuffleDependency has a **shuffleId** (FIXME from SparkContext.newShuffleId ).

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

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

The places where ShuffleDependency is used:

CoGroupedRDD and SubtractedRDD when partitioner differs among RDDs

ShuffledRDD and ShuffledRowRDD that are RDDs from a shuffle

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

repartition

cogroup

Dependencies 72

Mastering Apache Spark

intersection

subtractByKey

subtract

sortByKey

sortBy

repartitionAndSortWithinPartitions

combineByKeyWithClassTag

combineByKey

aggregateByKey

foldByKey

reduceByKey

countApproxDistinctByKey

groupByKey

partitionBy

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

**NarrowDependency**

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

NarrowDependency extends the base with the additional method:

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

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

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

Dependencies 73

Mastering Apache Spark

scala> val r1 = sc.parallelize(0 to 9)

r1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[13] at parallelize at <console>:18

scala> val r3 = r1.map((\_, 1))

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

scala> r3.dependencies

res32: Seq[org.apache.spark.Dependency[\_]] = List(org.apache.spark.OneToOneDependency@7353a0fb)

scala> r3.toDebugString

res33: String =

(8) MapPartitionsRDD[19] at map at <console>:20 []

| ParallelCollectionRDD[13] at parallelize at <console>:18 []

**PruneDependency**

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

**RangeDependency**

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

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

scala> val r1 = sc.parallelize(0 to 9)

r1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[13] at parallelize at <console>:18

scala> val r2 = sc.parallelize(10 to 19)

r2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[14] at parallelize at <console>:18

scala> val unioned = sc.union(r1, r2)

unioned: org.apache.spark.rdd.RDD[Int] = UnionRDD[16] at union at <console>:22

scala> unioned.dependencies

res19: Seq[org.apache.spark.Dependency[\_]] = ArrayBuffer(org.apache.spark.RangeDependency@28408ad7,

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 []

Dependencies 74

Mastering Apache Spark

**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 .

Dependencies 75

Mastering Apache Spark

**MapPartitionsRDD**

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

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

MapPartitionsRDD is the result of the following transformations:

RDD.map

RDD.flatMap

RDD.filter

RDD.glom

RDD.mapPartitions

RDD.mapPartitionsWithIndex

PairRDDFunctions.mapValues

PairRDDFunctions.flatMapValues

Dependencies 76

Mastering Apache Spark

**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.

Dependencies 77

Mastering Apache Spark

**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.

Dependencies 78

Mastering Apache Spark

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

| Caution | FIXME  What are InputMetrics ?  What is JobConf ?  What are the InputSplits: FileSplit and CombineFileSplit ? \* What are InputFormat and Configurable subtypes?  What’s InputFormat’s RecordReader? It creates a key and a value. What are they?  What does TaskContext do?  What’s Hadoop Split? input splits for Hadoop reads? See  InputFormat.getSplits |
| --- | --- |

**getPartitions**

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

For SparkContext.textFile the input format class is

org.apache.hadoop.mapred.TextInputFormat.

The javadoc of org.apache.hadoop.mapred.FileInputFormat says:

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

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

Dependencies 79

Mastering Apache Spark

**ShuffledRDD**

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

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

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

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

scala> val r = sc.parallelize(0 to 9, 3).groupBy(\_ / 3)

r: org.apache.spark.rdd.RDD[(Int, Iterable[Int])] = ShuffledRDD[2] at groupBy at <console>:18

scala> r.toDebugString

res0: String =

(3) ShuffledRDD[2] at groupBy at <console>:18 []

+-(3) MapPartitionsRDD[1] at groupBy at <console>:18 []

| ParallelCollectionRDD[0] at parallelize at <console>:18 []



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

Figure 1. Two stages in a job due to shuffling

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

repartitionAndSortWithinPartitions

sortByKey (be very careful due to [SPARK-1021] sortByKey() launches a cluster job when it shouldn’t)

partitionBy (only when the input partitioner is different from the current one in an RDD)

It uses Partitioner.

Dependencies 80