Apache Spark

Distributed programming framework for Big Data processing
Based on functional programming
Implements distributed Scala collections like interfaces for Scala, Java, Python, and R
Implemented on top of the Akka actor framework
Original paper:
Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012: 15-28
Resilient Distributed Datasets

Resilient Distributed Datasets (RDDs) are Scala collection-like entities that are distributed over several computers.

- The framework stores the RDDs in partitions, where a separate thread can process each partition.

- To implement fault tolerance, the RDD partitions record lineage: A recipe to recompute the RDD partition based on the parent RDDs and the operation used to generate the partition.

- If a server goes down, the lineage information can be used to recompute its partitions on another server.
Spark Tutorials

- Quick start:
  http://spark.apache.org/docs/latest/quick-start.html

- Spark Programming Guide:
  http://spark.apache.org/docs/latest/programming-guide.html

- Dataframes and Spark SQL:
After Spark has been installed, the command `spark-shell` can be used to create an interactive Scala shell to run spark code in.

Shell initializes a Spark context in variable `sc`.

For the shell to work, a Spark master has to be running. A local Spark master can be started with the command `start-master.sh` and stopped with `stop-master.sh`.

To create an RDD from a local file, count the number of lines, and show the first line use:

```scala
1 val textFile = sc.textFile("kalevala.txt")
2 textFile.count()
3 textFile.first()
```
Spark Quick Start (cnt.)

Spark Quick Start

- The log is:
  
  Spark context available as sc.
  SQL context available as sqlContext.
  Welcome to

  ___
  / __/ __ __ __
  / _ \ _ _ _/
  /_/\_/\_/\_/\_/\_/\_/\_/

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

  scala> val textFile = sc.textFile("kalevala.txt")

  scala> textFile.count()
  res0: Long = 14366

  scala> textFile.first()
  res1: String = Kalevala
To find out how many lines contain the name “Louhi”:

```scala
val textFile = sc.textFile("kalevala.txt")
textFile.filter(line => line.contains("Louhi")).count()  // How many lines contain "Louhi"?
```

The log is:

```scala
scala> val textFile = sc.textFile("kalevala.txt")

scala> textFile.filter(line => line.contains("Louhi")).count()
// How many lines contain "Louhi"?
res0: Long = 29
```
Creating RDDs

RDDs can be created from:

- From other RDDs using RDD transformations
- Scala collections
- Local files
- Usually with Big Data: Files stored in distributed storage systems: Hadoop Distributed Filesystem (HDFS), Amazon S3, HBase, ... 

When an RDD is created, it is initially split to a number of partitions, which should be large enough (e.g., at least 2-10 x the number of cores) to allow for efficient load balancing between cores.

Each partition should still be large enough to take more than 100 ms to process, so not to waste too much time in starting and finishing the task processing a partition.
Example Standalone Spark App

```scala
/* SimpleApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your system
    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
  }
}
```
The following RDD transformations are available:

- `map(func)`
- `filter(func)`
- `flatMap(func)`
- `mapPartitions(func)`
- `mapPartitionsWithIndex(func)`
- `sample(withReplacement, fraction, seed)`
- `union(otherDataset)`
- `intersection(otherDataset)`
- `distinct([numTasks])`
RDD Transformations (cnt.)

The following RDD transformations are available:

- `groupByKey([numTasks])`
- `reduceByKey(func, [numTasks])`
- `aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])`
- `sortByKey([ascending], [numTasks])`
- `join(otherDataset, [numTasks])`
- `cogroup(otherDataset, [numTasks])`
- `cartesian(otherDataset)`
- `pipe(command, [envVars])`
- `coalesce(numPartitions)`
 RDD Transformations (cnt.)

The following RDD transformations are available:

- `repartition(numPartitions)`
- `repartitionAndSortWithinPartitions(partitioner)`
- ...
RDD Transformations (cnt.)

RDD Transformations:

- RDD transformations build a DAG of dependencies between RDD partitions but do not yet start processing.
- The actual data processing is done lazily.
- The RDD Actions are needed to start the computation.
RDD Actions

Available RDD Actions:

- `reduce(func)`
- `collect()`
- `count()`
- `first()`
- `take(n)`
- `takeSample(withReplacement, num, [seed])`
- `takeOrdered(n, [ordering])`
- `saveAsTextFile(path)`
- `saveAsSequenceFile(path)`
Available RDD Actions:

- `saveAsObjectFile(path)`
- `countByKey()`
- `foreach(func)`
- `...`
RDD Operations and Actions

RDD Operations and Actions:

- Many well known functional programming costructs such as `map` (or `flatMap`) and `reduce` (`reduceByKey`) are available as operations.
- One can implement relational database like functionality easily on top of RDD operations, if needed.
- This is how Spark SQL, a Big Data analytics framework, was originally implemented.
- Many operations take a user function to be called back as argument.
- Freely mixing user code with RDD operations limits the optimization capabilities of Spark RDDs - You get the data processing operations you write.
Broadcast Variables

Broadcast variables are a way to send some read-only data to all Spark workers in a coordinated fashion:

```scala
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```
Accumulators

Accumulators allow a way to compute statistics done during operations:

```scala
scala> val accum = sc.accumulator(0, "My Accumulator")
accum: spark.Accumulator[Int] = 0

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s

scala> accum.value
res2: Int = 10
```

- Note: If a job is rescheduled during operation execution, the accumulators are increased several times

- Thus the accumulators can also count processing that is “wasted” due to fault tolerance / speculation
Implementing RDD Operations and Actions

As the RDD operations are run in several computers, there is no shared memory to use to share state between operations.

The functions run in all of the distributed nodes need to copy all of the data they need to all of the Spark workers using so called closures.

To minimize the amount of data that needs to be copied to all nodes, the functions should refer to as little data as possible to keep the closures small.

Note that variables of the closure modified in a Worker node are lost and thus cannot be used to communicate back to the driver program.

The only communication is through operations and actions on RDDs.
Buggy Code misusing Closures

When Spark is run in truly distributed mode, the following code will not work, as changes to `counter` do not propagate back to the driver:

```
// Buggy code!!!

var counter = 0
var rdd = sc.parallelize(data)

// Wrong: Don’t do this!!
rdd.foreach(x => counter += x)
println("Counter value: " + counter)
```
RDD Persistence levels

RDDs can be configured with different persistence levels for caching RDDs:

- MEMORY_ONLY
- MEMORY_AND_DISK
- MEMORY_ONLY_SER
- MEMORY_AND_DISK_SER
- DISK_ONLY
- MEMORY_ONLY_2, MEMORY_AND_DISK_ONLY_2
- OFF_HEAP (experimental)
Lineage can contain both narrow and wide dependencies:

Lineage (cnt.)

Lineage can contain both narrow and wide dependencies:

**Dependency Types**

"Narrow" (pipeline-able)

- map, filter
- union

"Wide" (shuffle)

- join with inputs co-partitioned
- groupByKey on non-partitioned data
- join with inputs not co-partitioned
Narrow and Wide Dependencies

In narrow dependencies, the data partition from an input RDD can be scheduled on the same physical machine as the output RDD. This is called “pipelining”.

Thus computing RDD operations with only narrow dependencies can be scheduled to be done without network traffic.

Wide dependencies require all RDD partitions at the previous level of the lineage to be inputs to computing an RDD partition.

This requires sending most of the input RDD data over the network.

This operation is called the “shuffle” in Spark (and MapReduce) terminology.

Shuffles are unavoidable for applications needing e.g., global sorting of the output data.
DAG Scheduling

DAG scheduler is used to schedule RDD operations:

Job Scheduling Process

- RDD Objects:
  - rdd1.join(rdd2)
  - rdd1.groupBy(...)
  - rdd1.filter(...)
  - rdd1.count()
- Build operator DAG
- Scheduler (DAGScheduler):
  - Split graph into stages of tasks
  - Submit each stage as ready
- Executors:
  - Threads
  - Block manager
  - Execute tasks
  - Store and serve blocks
Pipelining into Stages
Scheduler pipelines work into stages separated by wide dependencies:

Scheduler Optimizations

- Pipelines operations within a stage
- Picks join algorithms based on partitioning (minimize shuffles)
- Reuses previously cached data

= previously computed partition
Spark Motivation

The MapReduce framework is one of the most robust Big Data processing frameworks.

MapReduce has several problems that Spark is able to address:

- Reading and storing data from main memory instead of hard disks - Main memory is much faster than the hard drives.
- Running iterative algorithms much faster - Especially important for many machine learning algorithms.
Spark Benchmarking

The original papers claim Spark to be upto 100x faster when data fits into RAM vs MapReduce, or upto 10x faster when data is on disks.

Large speedups can be observed in the RAM vs HD case.

However, independent benchmarks show when both use HDs, Spark is upto 2.5-5x faster for CPU bound workloads, however MapReduce can still sort faster:


http://www.slideshare.net/ssuser6bb12d/a-comparative-performance-evaluation-of-apache-flink
Spark Extensions

- MLlib - Distributed Machine learning Library
- Spark SQL - Distributed Analytics SQL Database - Can be tightly integrated with Apache Hive Data Warehouse, uses HQL (Hive SQL variant)
- Spark Streaming - A Streaming Data Processing Framework
- GraphX - A Graph processing system
Data Frames

- The Spark project has introduced a new data storage and processing framework - Data Frames - to eventually replace RDDs for many applications.

- Problem with RDDs: As RDDs are operated by arbitrary user functions in Scala/Java/Python, optimizing them is very hard.

- DataFrames allow only a limited number of DataFrame native operations, and allows the Spark SQL optimizer to rewrite user DataFrame code to equivalent but faster codes.

- Instead of executing what the user wrote (as with RDDs), execute an optimized sequence of operations.

- Allows DataFrame operations to be also implemented in C/C++ (LLVM) level.