CS-E4640 - Big Data Platforms
Lecture 2

Keijo Heljanko

Department of Computer Science
University of Helsinki & Aalto University
keijo.heljanko@helsinki.fi

19.9-2018

Apache Spark

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

Scala Parallel Collections and Apache Spark

Scala Parallel Collections and Apache Spark

- In order to understand Spark, let's first look at Scala parallel collections
- Scala parallel collections are a framework for parallel computing in a single shared memory computer
- Apache Spark is a framework that takes ideas from Scala parallel collections but implements them for a cluster of computers
- Because the cluster has distributed memory instead of shared memory in a single computer, some things are different in the two frameworks

Scala

Scala

- Originally an Acronym Scala - Scalable Language
- A general purpose programming language
- Implemented on top of the Java Virtual Machine (JVM)
- Object oriented
- Functional programming can be done in Scala, prefers the use of immutable data
- Also imperative programs with mutable data can be coded in Scala
- Can use all Java libraries
- The Akka Actor model is integrated into Scala
- Has also other interesting parallel features - **Scala Parallel Collections**

Learning Scala

Learning Scala

- For learning Scala, see the course: CS-A1120 Ohjelmointi 2
- We will not go deep into Scala, examples should be understandable with Java programming background
- Getting started with Scala: `http://www.scala-lang.org/documentation/getting-started.html`
- Main Scala tutorials are at: `http://docs.scala-lang.org/tutorials/`
- Scala Tutorial for Java programmers: `http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html`

Scala Parallel Collections Tutorial

Parallel Collections Tutorial

- New feature of Scala since version 2.9 (version 2.12 is currently the most recent version)
- We are using materials from the Scala Parallel Collections tutorial at:
  `http://docs.scala-lang.org/overviews/parallel-collections/overview.html`
- For design of the internals, see:
  Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, Martin Odersky: A Generic Parallel Collection Framework. Euro-Par (2) 2011: 136-147

Scala Collections

Scala Collections

- Consider the following piece of Scala code using collections:

```scala
val list = (1 to 10000).toList
list.map(_ + 42)
```

- The code adds 42 to each member of the collection, using a single thread to do so
- When run through the Scala interpreter, we get:

```
scala> val list = (1 to 10000).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
scala> list.map(_ + 42)
res0: List[Int] = List(43, 44, 45, 46, 47, 48, 49, 50, 51, 52, ...
```

Scala Parallel Collections

Scala Parallel Collections

- To make this code parallel, we can just use the `par` method on the list to generate a ParVector, a parallel vector datatype

```scala
val list = (1 to 10000).toList
list.par.map(_ + 42)
```

- The code adds 42 to each member of the collection, using several threads running in parallel, we get:

```
scala> val list = (1 to 10000).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
scala> list.par.map(_ + 42)
res0: scala.collection.parallel.immutable.ParSeq[Int] =
ParVector(43, 44, 45, 46, 47, 48, 49, 50, 51, 52, ...
```

Scala Parallel Collections

Scala Parallel Collections

- One can generate parallel collection types from sequential collections
- Operations on parallel collection types can use all the threads available to the Scala runtime to process the collections in parallel
- The load balancing and scheduling of the parallel collections is done by the Scala runtime
- Due to the overhead of creating new threads, better performance is only obtained for operations which are CPU heavy per item in the collection, or for very large collections
- The parallelization uses the functional programming nature Scala collections - The map operations performed should not have side effects (if possible)

Available Parallel Collections

Available Parallel Collections

- ParArray
- ParVector
- mutable.ParHashMap
- mutable.ParHashSet
- immutable.ParHashMap
- immutable.ParHashSet
- ParRange
- ParTrieMap

Example: Using Parallel Map

Parallel Map

- Consider the following piece of Scala code using a parallel map:

```
val lastNames = List("Smith","Jones","Frankenstein","Bach",
"Jackson","Rodin").par
lastNames.map(_.toUpperCase)
```

Example: Using Parallel Map (cnt.)

Parallel Map (cnt.)

- The code converts all elements of the map in parallel to upper case

```
scala> val lastNames = List("Smith","Jones","Frankenstein",
"Bach","Jackson","Rodin").par
lastNames: scala.collection.parallel.immutable.ParSeq[String] =
ParVector(Smith, Jones, Frankenstein, Bach, Jackson, Rodin)

scala> lastNames.map(_.toUpperCase)
res0: scala.collection.parallel.immutable.ParSeq[String] =
ParVector(SMITH, JONES, FRANKENSTEIN, BACH, JACKSON, RODIN)
```

Example: Using Fold

Parallel Fold

- Consider the following piece of Scala code summing up all integers in a list using `fold`, which applies an associative operation to all elements of the collection:

```
val parArray = (1 to 1000000).toArray.par
parArray.fold(0)(_ + _)
```

Example: Using Fold (cnt.)

Parallel Fold

- The output of the operation is well defined as the addition method given to `fold` is an associative operation, and the parameter of fold `0` is the zero element of addition

```
scala> val parArray = (1 to 1000000).toArray.par
parArray: scala.collection.parallel.mutable.ParArray[Int] =
ParArray(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...
scala> parArray.fold(0)(_ + _)
res0: Int = 1784293664
```

- Note: In Scala the operation passed to `fold` does not have to be commutative, only associative!
- Note: Many other frameworks, such as Apache Spark require also operator commutativity, be careful when porting from Scala parallel collections to Spark!

Example: Using Reduce

Parallel Reduce

- The `reduce` operation is like `fold` except that because you do not give the zero element, it can not be applied to empty collections (it will throw an exception in that case). As fold, it also requires the applied operation to be associative:

```scala
val parArray = (1 to 1000000).toArray.par
parArray.reduce(_ + _)
```

```
scala> val parArray = (1 to 1000000).toArray.par
parArray: scala.collection.parallel.mutable.ParArray[Int] =
ParArray(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...
scala> parArray.reduce(_ + _)
res0: Int = 1784293664
```

Example: Using Filter

Parallel Filter

- Consider the following piece of Scala code filtering last names starting with letters larger than 'J':

```scala
val lastNames = List("Smith","Jones","Frankenstein","Bach",
"Jackson","Rodin").par
lastNames.filter(_.head >= 'J')
```

Example: Using Filter (cnt.)

Parallel Filter

- Notice that in Scala the filtered collection still preserves order of the original collection:

```
scala> val lastNames = List("Smith","Jones","Frankenstein",
"Bach","Jackson","Rodin").par
lastNames: scala.collection.parallel.immutable.ParSeq[String] =
ParVector(Smith, Jones, Frankenstein, Bach, Jackson, Rodin)

scala> lastNames.filter(_.head >= 'J')
res0: scala.collection.parallel.immutable.ParSeq[String] =
ParVector(Smith, Jones, Jackson, Rodin)
```

Creating Parallel Collections

Creating Parallel Collections

- By using the `new` keyword after importing the right package

```scala
import scala.collection.parallel.immutable.ParVector
val pv = new ParVector[Int]
```

```
scala> import scala.collection.parallel.immutable.ParVector
import scala.collection.parallel.immutable.ParVector

scala> val pv = new ParVector[Int]
pv: scala.collection.parallel.immutable.ParVector[Int] =
ParVector()
```

Creating Parallel Collections (cnt.)

Creating Parallel Collections

- By constructing a parallel collection from an existing sequential collection using the `par` method of the sequential collection:

```scala
val pv = Vector(1,2,3,4,5,6,7,8,9).par
```

```
scala> val pv = Vector(1,2,3,4,5,6,7,8,9).par
pv: scala.collection.parallel.immutable.ParVector[Int] =
ParVector(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Semantics of Parallel Collections

Semantics

- Code with side-effects will result in non-deterministic behaviour. Proper locking needs to be taken if operations on parallel collections manipulate shared state
- Using operations that are not associative will result in non-deterministic behaviour as evaluation order is based on scheduling of concurrently executing threads

Buggy! Summation using Side-effects

Buggy code due to side effects!

- The following code uses the variable sum in a racy manner, the outcome of the code depends on the interleaving:

```
val list = (1 to 1000).toList.par

var sum = 0;
list.foreach(sum += _); sum

var sum = 0;
list.foreach(sum += _); sum

var sum = 0;
list.foreach(sum += _); sum
```

Buggy! Summation using Side-effects (cnt.)

Different results on different runs!

```
scala> val list = (1 to 1000).toList.par
list: scala.collection.parallel.immutable.ParSeq[Int] =
ParVector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...
scala> var sum=0
sum: Int = 0

scala> list.foreach(sum += _); sum
res0: Int = 481682

scala> var sum=0
sum: Int = 0

scala> list.foreach(sum += _); sum
res1: Int = 486426

scala> var sum=0
sum: Int = 0

scala> list.foreach(sum += _); sum
res2: Int = 500500
```

Buggy! Code due to Non-Associativity

Non-Associative Operations are Non-Deterministic

- The subtraction operator is not associative (e.g, $(1-2)-3 \neq 1-(2-3)$). Thus the order of scheduling of operations affects the outcome of the reduce:

```
val list = (1 to 1000).toList.par
list.reduce(_-_)
list.reduce(_-_)
list.reduce(_-_)
```

Buggy! Code due to Non-Associativity (cnt.)

Different results on different runs!

```
scala> val list = (1 to 1000).toList.par
list: scala.collection.parallel.immutable.ParSeq[Int] =
ParVector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
scala> list.reduce(_-_)
res0: Int = 169316

scala> list.reduce(_-_)
res1: Int = 497564

scala> list.reduce(_-_)
res2: Int = -331818
```

Correct Associative but Non-Commutative Operators

In Scala Parallel Collections Commutativity is not needed!

- The following code is correct in Scala parallel collections, as String concatenation is associative (even if it is not commutative!)

```
val strings = List("abc","def","ghi","jk","lmnop","qrs","tuv","wx","yz")
val alphabet = strings.reduce(_++_)
```

Correct Associative but Non-Commutative Operators

In Scala Parallel Collections Commutativity is not needed!

- The outcome is the same regardless of thread scheduling:

```
scala> val strings = List("abc","def","ghi","jk",
"lmnop","qrs","tuv","wx","yz").par
strings: scala.collection.parallel.immutable.ParSeq[String] =
ParVector(abc, def, ghi, jk, lmnop, qrs, tuv, wx, yz)

scala> val alphabet = strings.reduce(_++_)
alphabet: String = abcdefghijklmnopqrstuvwxyz
```

- Note: Other frameworks, such as Apache Spark, require the operator applied by reduce to also be commutative, so this code would be incorrect in Spark!

Resilient Distributed Datasets

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

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:
  `http://spark.apache.org/docs/latest/sql-programming-guide.html`

Spark Quick Start (cnt.)
Spark Tutorials

- 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
val textFile = sc.textFile("kalevala.txt")
textFile.count()
textFile.first()
```

Spark Quick Start (cnt.)

Spark Quick Start

- The log is:

```
Spark context available as sc.
SQL context available as sqlContext.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.5.2
      /_/

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")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at textFile at <consol

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

scala> textFile.first()
res1: String = Kalevala
```

Spark Quick Start (cnt.)

Spark Quick Start

- To find out how many lines contain the name “Louhi”:

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

- The log is:

```
scala> val textFile = sc.textFile("kalevala.txt")
textFile: org.apache.spark.rdd.RDD[String] =
MapPartitionsRDD[1] at textFile at <console>:24

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 or 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

Example Standalone Spark App

```
/* 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))
  }
}
```

RDD Transformations

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)

RDD Actions (cnt.)

Available RDD Actions:

- saveAsObjectFile(path)
- countByKey()
- foreach(func)
- ...

RDD Operations and Actions

RDD Operations and Actions:

- Many well known functional programming constructs 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 operations you write, not many automatic optimization can be done

Broadcast Variables

Broadcast Variables

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

```
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

- Accumulators allow a way to compute statistics done during operations:

```
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

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 can not be used to communicate back to the driver program
- Communication is done through RDDs

Buggy Code misusing Closures

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)

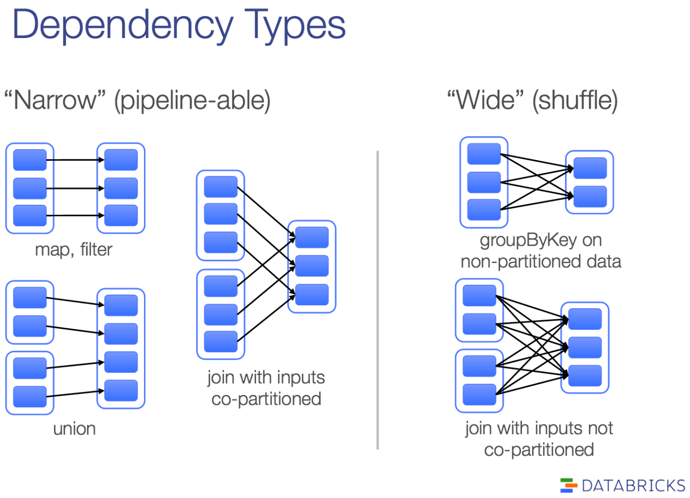
Lineage

Lineage can contain both narrow and wide dependencies:

- Figures in the following from “Databrics Advanced Spark Training by Reynold Xin”

Lineage (cnt.)

Lineage can contain both narrow and wide dependencies:



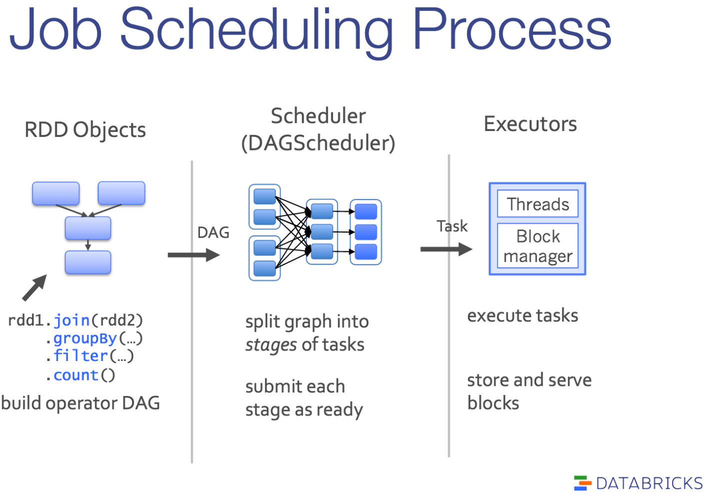
Narrow and Wide Dependencies

Narrow and Wide Dependencies

- In narrow dependencies, the data partition from an input RDD can be scheduled on the same physical machine as the out RDD. This is called “pipelining”
- Thus RDD operations with only narrow dependencies can scheduled to be done without any 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 the 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:



Pipelining into Stages

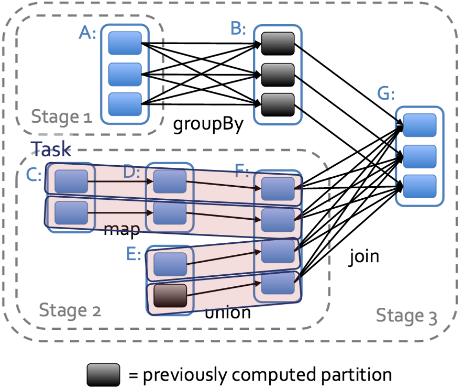
Scheduler pipelines work into stages separated by wide dependencies:

Scheduler Optimizations

Pipelines operations within a stage

Picks join algorithms based on partitioning (minimize shuffles)

Reuses previously cached data



Spark Motivation

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

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:
  Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, Fatma Özcan: Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. PVLDB 8(13): 2110-2121 (2015)
  `http://www.vldb.org/pvldb/vol8/p2110-shi.pdf`
  `http://www.slideshare.net/ssuser6bb12d/`
  `a-comparative-performance-evaluation-of-apache-flink`

Spark Extensions

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

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 a compiled fashion