Table of Contents

SPARK INTRO

1. INTRO'S INTRO

1.1. Purpose

The purpose of this document is to provide you with a single starting point for the Apache Spark

1.2. Document status

This document is work in progress, should it achieve a minimal level of completeness according to it's purpose the status should be changed to ready.

1.3. Audience

The Guide is aimed mainly for Spark developers , yet it might be of interest for the following roles as well:
- busines operational executives
- developers
- devops operators
- integration specialists

1.4. Master storage and storage format

The master storage of this document is the following md file in the intro project's doc directory:
<<todo: add link>>

1.5. Version control

The version control is implemented in the following GitProject:
<<todo:add link>>.
Should you want to add,delete,update more content to it simply inform the owner of the GitHub repository you are accessing this document from.

2. BIG DATA GENERALLY

2.1. What is BigData ?!

"Big Data" is a common term to refer to:
- large datasets - tens of terabytes of data or MORE !!!
- the category of computing strategies and technologies that are used to handle those large datasets.

Big Data Systems differ from the non-big data system by:
- volume - the volume of data is larger
- variety - the variety is greater
- velocity - the velocity is greater

2.2. The scalability problem

Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.
source: apache.org

2.3. The fault tollerance problem

Hardware will brake, network connections will go down , differrent nodes will fail for one reason or another ...

2.4. Hadoop

Framework that allows distributed processing of large data sets across clusters of computers.
Hadoop is an open source distributed processing framework that manages data processing and storage for big data applications running in clustered systems. It is at the center of a growing ecosystem of big data technologies that are primarily used to support advanced analytics initiatives, including predictive analytics, data mining and machine learning applications. Hadoop can handle various forms of structured and unstructured data, giving users more flexibility for collecting, processing and analyzing data than relational databases and data warehouses provide.
It all started with Google's "BigTable paper" ...
http://hadoop.apache.org/

2.4.1. MapReduce

MapReduce is a programming model for processing and generating large data sets with a parallel, distributed algorithm on a cluster.

MapReduce is hard !!!

3. SPARK DEFINITION

Definition from the Spark project's page:
Apache Spark is a fast and general-purpose cluster computing system.
Definition from Wikipedia
"Apache Spark is an open-source cluster-computing framework. Apache Spark provides programmers with an application programming interface centered on a data structure called the Resilient Distributed Dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way."
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-overview.html

3.1. A distributed computing framework

3.2. An Apache open-source project

4. SPARK UNIFIED STACK

The Spark project currently comprised of Spark Core and four libraries that are optimized to cater four different types of use cases. An application typically using Spark Code and at least one of these libraries:
- Spark SQL
- Streaming
- Mlib
- GraphX
https://spark.apache.org/

4.1. Spark SQL

Spark SQL lets you query structured data inside Spark programs, using either SQL or a familiar DataFrame API.

4.2. Spark Streaming

Spark Streaming is a framework for large scale data stream processing.
- IOT data
- gaming data
- near real-time data processing requirements
- Scales to 100s of nodes
- Can achieve second scale latencies
- Integrates with Spark’s batch and interactive processing
- Provides a simple batch-like API for implementing complex algorithm
- Can absorb live data streams from Kafka, Flume, ZeroMQ, etc.

4.3. Spark Mlib

Spark’s graph computational framework.

4.4. GraphsX

For graph data processing.

5. THE SPARK DATA MODEL API

5.1. The RDD API

Centered on a data structure called the Resilient Distributed Dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way

5.2. The Datasets API

A Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL’s optimized execution engine. A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.).

5.3. The Dataframes

A DataFrame is a Dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.

5.4. Actions

Actions cause calculations to be performed.
reduce(func): Aggregate the elements of the dataset using a function func (which takes two arguments and returns one).

The function should be commutative and associative so that it can be computed correctly in parallel.
collect(): Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count(): Return the number of elements in the dataset.

5.5. Transformations

Transformations just set things up (that is using lazy evaluation)
map(func): Return a new distributed dataset formed by passing each element of the source through a function func.
filter(func): Return a new dataset formed by selecting those elements of the source on which func returns true
union(otherDataset): Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(otherDataset): Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([numTasks])): Return a new dataset that contains the distinct elements of the source dataset
join(otherDataset, [numTasks]): 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. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

6. SPARK FEATURES THE IT ARCHITECT PERSPECTIVE

6.1. Scalable architecture from the very beginning

Runs on JVM. Designed to scale and used in installations with tens of thousands of nodes.

6.2. Works on commodity hardware

Compare to buying an starting from 1 Million Exadata rack ... A spark big data environment deployed and maintained properly might save hundreds of thousands of $$ from hardware and licences expences.

6.2.1. Excellent support for HDFS

Spark can read and write data to HDFS providing similar API's as for local file systems operations.
Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, Amazon S3, etc. Spark supports text files, SequenceFiles, and any other Hadoop InputFormat.

6.3. Universal connectivity via JDBC and Hive

You could connect to any existing RDBMS and NON-SQL via

6.4. Expandability

Could utilize libraries for scientific computing, linear algebra and random number generation

7. SPARK FEATURES - THE DEVELOPER'S PERSPECTIVE

Spark is scalable. Period.

7.1. A powerfull combination for Analytics

Combine streaming, graph, machine learning and sql analytics on a single platform, yet supports external libraries.

7.2. Simple programming model

Internal details like partitioning of data, creation of threads, etc. IS be hidden.

7.3. Multi-language support

Spark supports the following languages:
- Scala (Spark is written in Scala)
- Java
- Python
- R (partially supported)

7.4. High level abstraction for parallel computing

As soon as you as a Developer understand how the Spark Execution Control Flow is implemented and obey couple of simple but unavoidable rules, you become a powerfull parallel computing developer without any need to implement any concepts related to parallelism, forks, semaphors , threads , networking ect.

7.5. The Integration specialist's perspective

Rapid access and storage to Big Data in almost any format.

7.6. ETL for batch and interactive processing

Quick extract transform, load batch and / or ad-hoc queries can be run on variety of data sources - examples:

```
val textFile = spark.read.textFile("/path/to/file.txt")
textFile.filter(line => line.contains("Spark")).count() // How many lines contain "Spark"?
```

7.7. Easy parallel computing without having to know much about parallelism

Parallel programs are hard

8. SPARK FEATURES - THE DEVOPS PERSPECTIVE

8.1. Easy deployability

For the DevOps

8.2. Simple programming model

Spark supports the following languages:
- Scala (Spark is written in Scala)
- Java
- Python
- R (partially supported)
You as Developer do not have to deal with the all the parallelism and fault tollerance issues, you could concentrate on your application logic implementation - spark takes care of the rest.

8.3. Easy scalability

Running on cheap commondity hardware or cloud deployments.

8.4. High availability

Provided by design from the architecture. Of course you would have to ensure that your end-to-end implementation is alighed with this one ...

8.5. Applications' monitoring

Any full Spark installation provides rich set of UI's for application monitoring:
https://spark.apache.org/docs/latest/monitoring.html

8.6. Capability to run Spark in common cluster manager

Besides standalone mode Spark can be run on a common cluster manager such as Mesos or Hadoop Yarn.

8.7. Integrations related featues

Rapid access and storage to Big Data in almost any format.

8.8. Easy monitoring via web interface and API

Spark provides web UI for monitoring of the :
The SPARK UI:
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-webui-SparkUI.html
- Jobs
- Stages
- Tasks
- Environment
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-webui-executors.html

9. SPARK EXECUTION CONTROL FLOW

9.1. The entry point

This phase from the control flow is often overlooked, yet it simply represents the interface or component initiating the control flow - it could be the spark shell in Spark Standalone installation , it could be Apache Zeppelin note , etc.

9.2. The Driver

A Spark driver (aka an application’s driver process) is a JVM process that hosts SparkContext for a Spark application. It is the master node in a Spark application.
It is the cockpit of jobs and tasks execution (using DAGScheduler and Task Scheduler). It hosts Web UI for the environment.
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-driver.html

9.2.1. The Driver settings

Read trough the driver settings in the Settings section:

https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-driver.html

9.3. The Cluster Manager

Could be :
- Standalone
- Mesos
- Yarn

9.4. The Worker nodes in the cluster

Run executor processes executing different tasks.
Executor is a distributed agent that is responsible for executing tasks.
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-Executor.html

9.4.1. The spark executor settings

Check the settings section in the following link:
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-Executor.html

10. GENERAL ABOUT THE SCALA APPLICATION PROJECT

This section contains small receipts and code snippets, which should save you a lot of time googing or simply grasping a new concepts by just applying the bellow listed code snippets into your project.

10.1. Usefull build.sbt settings

The following settings have been proved usefull. Read the comments above each setting.

```
// add some stats for each test
testOptions in Test += Tests.Argument(TestFrameworks.ScalaTest, "-oD")
// define minimalistic log level
logLevel in test := Level.Info
// might or might not suits your reqs
fork in Test := true
// might or might not suit your reqs
parallelExecution in Test := false
//
logLevel in test := Level.Info
// do not print success msgs
showSuccess := false
// allocate more mem to the JVM
javaOptions ++= Seq("-Xms1G", "-Xmx4G", "-XX:+CMSClassUnloadingEnabled")
// print each test's score times ..
testOptions in Test += Tests.Argument(TestFrameworks.ScalaTest, "-oD")
```

10.2. Use code formatter

# use code formatters - scala fmt - forexample - reconfigure on accepted coding rules !!!

10.3. Use pureconfig for configuration building

Use strong-typed configuration to case classes:
https://github.com/pureconfig/pureconfig

10.4. Dependencies management !!!

Dependency management in dev,tst,qas and prod !!!

11. HOW-TO CODE SNIPPETS

This section contains small receipts and code snippets, which should save you a lot of time googing or simply grasping a new concepts by just applying the bellow listed code snippets into your project.

11.1. How-to copy files in Hadoop FS

There is more than one way to achieve this , but this is one example ...

```
import org.apache.hadoop.fs.{FileAlreadyExistsException, FileSystem, FileUtil, Path}
  val srcFileSystem: FileSystem = FileSystemUtil
    .apply(spark.sparkContext.hadoopConfiguration)
    .getFileSystem(sourceFile)
  val dstFileSystem: FileSystem = FileSystemUtil
```

```
    .apply(spark.sparkContext.hadoopConfiguration)
    .getFileSystem(sourceFile)
  FileUtil.copy(
    srcFileSystem,
    new Path(new URI(sourceFile)),
    dstFileSystem,
    new Path(new URI(targetFile)),
    true,
    spark.sparkContext.hadoopConfiguration)
```

11.2. Small HDFS path utils

This example demonstrates how you must have the spark session available in order to use any of the HDFS spark API's.

```
mport org.apache.hadoop.fs.{FileSystem, Path}
trait HavingSparkSession {
 implicit val spark: SparkSession = SparkSession.builder().getOrCreate()
}
object HdfsPathUtils extends HavingSparkSession {
 def pathExists(path: String): Boolean = {
  val spark: SparkSession = SparkSession.builder().getOrCreate()
  val conf = spark.sparkContext.hadoopConfiguration
  val fs = FileSystem.get(conf)
  fs.exists(new Path(path))
 }
 def getFullPath(path:String): String = {
  val conf = spark.sparkContext.hadoopConfiguration
  val fs = FileSystem.get(conf)
  fs.getFileStatus(new Path(path)).getPath().toString
 }
 def getAllFiles(path:String): Seq[String] = {
  val conf = spark.sparkContext.hadoopConfiguration
  val fs = FileSystem.get(conf)
  val files = fs.listStatus(new Path(path))
  files.map(_.getPath().toString)
 }
}
```

11.3. How-to iterate over RDD rows

Remember each dataframe is RDD and you could easily convert from RDD to Dataframe in the spark 2.x.x versions.

```

  val rddRows: RDD[Row] =
   inDf.rdd.map(row => {

    val lstRow = row.toSeq.toList
    var lstRowNew = lstRow
    // do stuff on the new lstRow here
    Row.fromSeq(lstRowNew)
   })
  val dfOut = spark.createDataFrame(rddRows, inDf.schema)

```

11.4. The most simple example for factory design pattern

If you apply that for readers, writers and converters and have

```
 // all the readers must extend this trait
trait Reader {
 def read ( what: String ): DataFrame
}
object RdrFactory {
 // could pass also a custom reader type here ...
 def spownReader ( readerType: String , cnf: AppConfig) = {
  readerType match {
   case "db" => new RdrDb (cnf)
```

```
      case "file" => new RdrFile (cnf)
      case _ => throw new IllegalArgumentException( "unknown reader type !!!")
    }
  }
}
class RdrFile ( cnf: AppConfig )  extends Reader {
  def read ( what: String ): DataFrame = {
     throw new NotImplementedException ("todo:implement file reader !!!")
  }
}
class RdrDb ( cnf: AppConfig ) extends Reader {
  def read ( what: String ): DataFrame = {
     throw new NotImplementedException ("todo: implement db reader !!!")
  }
}
```

11.5. How-to register user defined function and call it

Note that you would have to register the UDF function in the sqlContext. Functions are very peaky about data types (use Java Primitive data types !!!)

```
// register and udf to the spark session
spark.sqlContext.udf.register(
     "getRandom64CharStr", () => scala.util.Random.nextString(64))
// or register and UDF accepting params
def getAboutTimeUDF = udf((ts: java.sql.Timestamp, timeFrame: String) => {
  import java.util.Calendar
  new java.sql.Timestamp(
    timeFrame match {
      case "%H" => DateUtils.round(ts, Calendar.HOUR).getTime
      case "%d"  => DateUtils.round(ts, Calendar.DATE).getTime
      case "%m" => DateUtils.round(ts, Calendar.MONTH).getTime
      case _        => throw new IllegalArgumentException()
    }
  )
})
// and call it by
val roundedTsDf = inDf
.withColumn("round_" + timeFrame,
            getAboutTimeUDF(col(timeFrame), lit(timeFrame))
```

11.6. How-to avoid if else bloat with scala

This is more of a Scala example , but just to illustrate that you could apply the match control flow operator when retrieving a DataFrame object as well ...

```
     val df: DataFrame = {
      filterType match {
        case FilterType.SqlLike => filterLikeSql(df)
        case FilterType.XlsLike => filterLikeXls(df)
        case _ => filterLikeSql(df)
      }
    }
```

11.7. How-to generate a dummy df with schema on the fly

This one is very often required for unit testing and could save you a lot of time and hardwiring ...

```
def genDummyDf(): DataFrame = {
  spark
  .createDataFrame(
    spark.sparkContext.parallelize(
      Seq(
        Row("foo")
      )),
    StructType(
      Seq(
      StructField("bar", StringType)
    ))
  )
}
```

11.8. How-to read a csv file

A really short , but probably a really often used one ...

```
  val df = spark.read
    .format("csv")
    .option("header", "true")
    .load(uri)
```

11.9. How-to create a dataframe with nullable columns

Nullable columns are huge pain .. , because if your do not know how the reflection works you will burn time as crazy because of it ...

```
   val spark = SparkSession.builder().getOrCreate()
  import spark.implicits._
  // format: off
  // obs first and second row hardcoded vals are for "teaching" schema !!!
  val ds = Seq(
    // foo comments
    (1,"foo",Some("bar"),Some(1850)) ,
    // bar comments
    (2,"foo",None,None)
  ).toDS()
  val df = ds.toDF(
    "id",
    "fooCol",
    "barCol",
  )
  // format: on
  println("START printing: df")
  println(df.show(false)) // false will not cut cols
  println("STOP printing: df")
```

11.10. How-to join 2 dataframes , but on nullable columns

There is a way to join on nullable columns too ...

```
   val lstKeyCols = List("col1" , "col2" , "col3" )
dfLeft
    .join(
      dfRight,
          dfLeft("col1") <=> dfRight("col1_")
        && dfLeft("col2") <=> dfRight("col2_")
        && dfLeft("col3") <=> dfRight("col3_"),
      "fullouter"
    )
    .drop(lstKeyCols.map(_ + "_"): _*)
```

12. LESSONS LEARNED IN THE FIELD

12.1. At the end only the full END-TO-END result matters

You might have the best implementation, but if some configuration / hardware / OS / whatever issue occurrs vertically in the stack and affects negatively the end result, this IS what counts

12.2. Iterate on the team's cohesion

Write down the "rules " you define along the way in a versioned wiki - iterate each sprint !!!
Do small code reviews .. , iterate and get bigger along the way

12.3. Design, iterate and follow your software architecture

Which logical components could be identified in your future application ?

12.4. Drow your diagrams and keep them in version control

Drow your system components api diagrams and track them with version control
# as small pic changes matter a lot in the code !!!

12.5. Choose your partitition keys - optimize on the partition.

Wisely chosen partition keys could improve performance by orders of magnitude. Good understanding of your data model

12.6. Use broadcast variables

Usually when joining 2 or more larger datasets one of them is the "big brother" and the second one is the "look-up like " "small brother".

```
val thebroadcast Value = spark.sparkContext.broadcast(varToBroadCast)


val theGotBroadcastVar = thebroadcast.value
```

12.7. Spark session is peaky

Centralize as much as possible spark session handling.
Do you have a logical control flow ?!
Do not trow around spark session handling code snippets - you will get burned by realizing that the whole run-time might just shutdown on ??!! Condictions ...

12.8. Do not re-invent the wheel

Think you have a cool idea or design for a pattern how-to be implemented in Spark ?! Check first the docs , it has been probably implemented yet.

12.9. Do small changes

Do small changes - iterate faster - the more complex your environment the more easier you will fall into debugging timea eater traps

13. BIG NO-GO'S

13.1. Slowness in testing

Do your setup properly - you might loose 50% of the time in testing initializations etc. There is a huge overhead while using projects with huge dependencies ... Microservices ?!

13.2. If ain't tested with BigData it AIN'T BIG DATA

Really unit tests with increasing biz logic complexity without proper data loading and integration tests are waste of time ... Should do those on each commit or at least each merge into develop ...

13.3. No Vim and Emacs wars

# use IDEA and vim (optionally) , but not vim or emacs !!! - save time

13.4. KISS

Make simple things simple , but not simplier, first get it work , than improvize with fancy lang concepts ...

13.5. Too much hierarchies

# you probably are doing something wrong if you have hierarchy with more than 7 levels unless your data is deeply hierarchical ... both in directory structures , data structures etc.

13.6. Do not use the resource dirs for transformations - copy to tmp dirs

# if you are using data transformations in testing ALWAYS - copy from src input data into tmp dir and test there !!! because sbt does not clean the target dirs for you and even if you explicityly clear them it is buggy ...

14. DEMO

This small demo snippet demonstrates the reading of multiple files into dataframes and applying change to two of the columns based on the file names ...

15. RELATED PROJECTS & FUTHER DISCUSSIONS

Akka Actors, Spark Mlib.
E.g. Apache Spark, Apache Flink, Kafka + Samza, ScalaTion

15.1. Apache zeppelin

Apache Zeppelin
Web-based notebook that enables data-driven,
interactive data analytics and collaborative documents with SQL, Scala and more.
https://zeppelin.apache.org/

15.2. Log4j

The log4j and log4j2 .
Might seem trivial ... , yet great applications have a minimalistic and configurable logging functionality.

15.3. Akka

Combine with Akka for "concurrent" , "reactive" and "distributed" applications building:
https://akka.io/

15.4. Apache Kafka

Combine with Kafka for distributed stream processing:
https://kafka.apache.org/

15.5. Apache Flink

Combine with Flink for scalable batch and stream processing :
https://flink.apache.org/