**Spark Core Programming**

**Observations of MapReduce Programming**

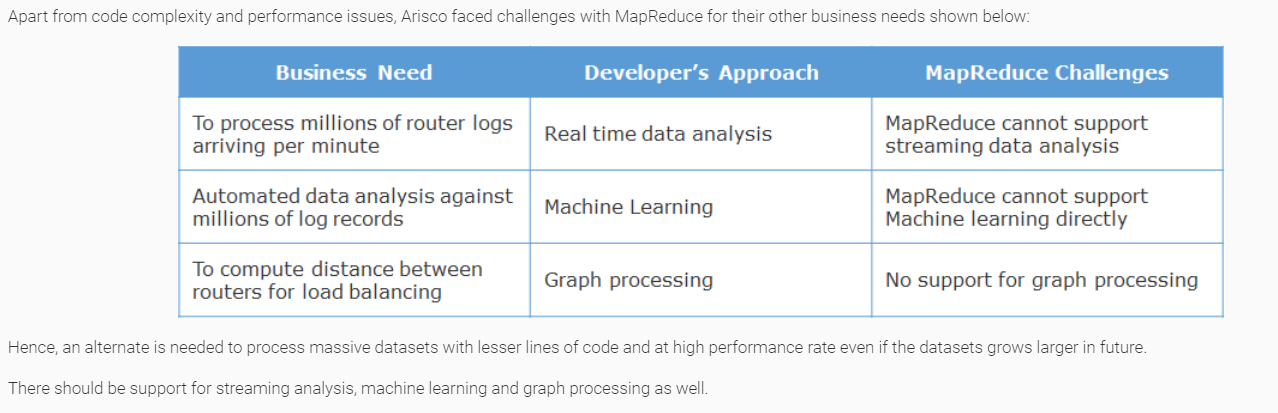
Just to compute a 500 MB file for a single requirement of finding number of occurrence of each record,

* Mapreduce requires almost **40+ lines** which looks highly complex
* It took approximately **3.23 minutes**

     What would be the time taken to run a **500 TB**or a **5 PB** file, or a **5 EB** file in year 2030?

* MR would lead to latency issues to complete the jobs when the dataset grows bigger.

The above observations shows that MR is not a good solution to compute big dataset. Let us see what are the other challenges of MR in data analysis.



# Why Spark?

Spark is a computational tool in Hadoop stack which can replace MapReduce to perform high speed computation with lesser and easier lines of code.

Below code is for the same Arisco’s requirement to compute the number of duplicate occurrences of records in a dataset using Spark (Java 8):

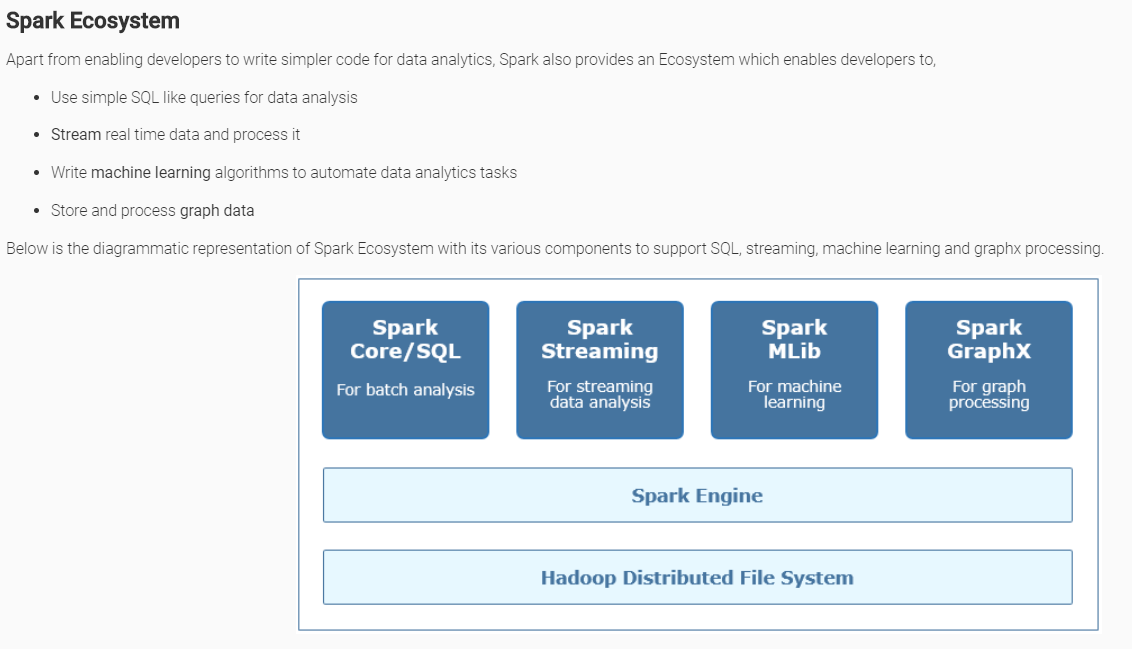
It took just **15 lines of simple elegant code** compared to**40+ lines**of Mapreduce logic

The same code using Spark Scala 2.11.8 would contain just 3 lines as shown below:

val lines = sc.textFile("data.txt")  
val lineLengths = lines.map(s => (s,1)).reduceByKey(\_+\_)  
lineLengths.saveAsTextFile("hdfs://chnbdamcty1:9000/jaihdfs/jop");

When the above code is run as a Spark job it completes the execution with in 10 seconds as shown below,

This shows that Spark offers better performance compared to MapReduce with lesser lines of code.



# Spark - Overview

Spark is a lightening fast, in-memory(RAM) computation tool to process big data files stored in Hadoop's HDFS, NoSQL DBs or on local file system. It was introduced by Apache Software Foundation and has rich APIs available in **Scala, Java and Python** for data computation.  
  
It is based on Hadoop's MapReduce and extends MapReduce model for quick computations. Unlike MapReduce, Spark processes intermediate data of your computation in RAM which is faster compared to MapReduce.

In case of MapReduce, the intermediate data of mappers and reducers is persisted in disk and processed which leads to time latency.

# What if the RAM size is not enough?

Spark holds processing data(programmatic objects) in disk and does computations when the RAM size is limited and still works 10 times faster than MapReduce.

Lets understand Spark RDD processing with a scenario,

# Scenario

Let us consider a 200 MB of an HDFS file split and stored as 4 blocks in HDFS. 4 blocks of the file is processed via Spark within an RDD collection as shown below,

 Block1 is loaded in to an rdd[] of Machine1 and processed using Spark

 Block2 is loaded in to an rdd[] of Machine2 and processed using Spark

 Block3 is loaded in to an rdd[] of Machine3 and processed using Spark

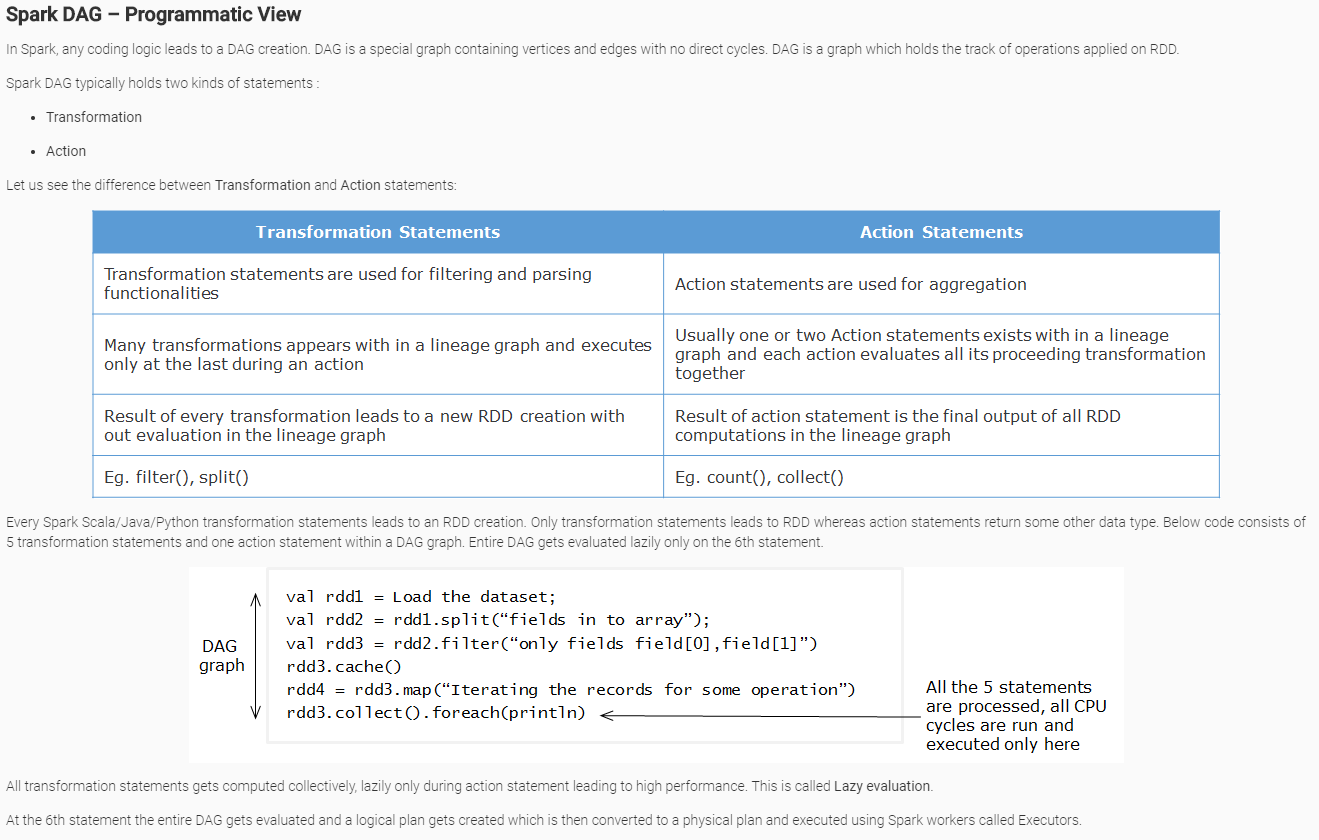
 Block4 is loaded in to an rdd[] of Machine4 and processed using Spark

In the above scenario,

* rdd[] is a programmatic collection which lives in RAM of every data node holding the data blocks
* RDD lives in memory through out the life cycle of the Spark job
* When an RDD is lost in the middle of processing it gets recreated automatically from Lineage graph

# Lineage graph:

The entire set of Spark statements reside inside a Lineage graph with every statement's RDD holding the pointer reference to its previous statement's RDD. Hence when an RDD is lost it gets recreated from its dependencies in Lineage graph.

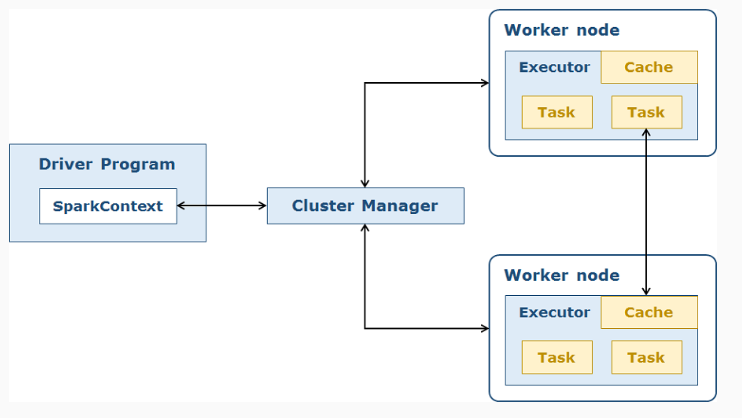


# Spark Architecture

Spark applications run as different sets of independent processes on Spark Cluster, where all processes are coordinated by **SparkContext**object created in the driver program. **SparkContext** object acts as an entry point for all operations in Spark application.

Driver program is where,

* SparkContext gets created
* Program gets converted into lineage execution graph
* Tasks get scheduled and monitored

 As shown in above diagram,

* **SparkContext** connects to the respective cluster manager (YARN in hadoop 2.x, Mesos etc) where the cluster manager in turn allocates resources to Spark job
* Spark creates **executors** on slave nodes in the cluster. Executors are processes meant to run computations against the data blocks spread across HDFS
* **SparkContext**then sends across the tasks to all the executors to run it in the respective nodes. Task is a unit of execution that computes the data in each Worker node.
* **Worker nodes** are the servers where actual data to be processed is stored

**Note:**Spark applications can be built in Scala, Java or Python programming languages. In this course all demos are written in Scala.

As of now we have got a fair idea on Spark Core architecture and its components. Let us now understand the concepts of RDD and lineage graph in detail.

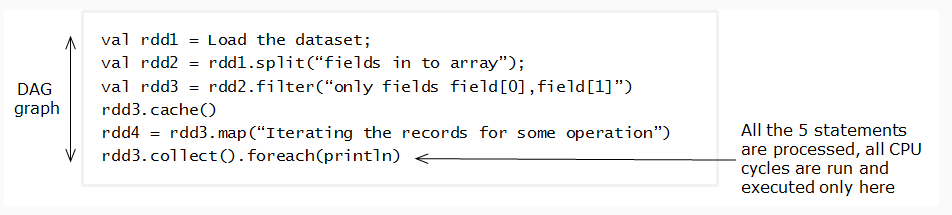
# Resilient Distributed Dataset (RDD)

* Spark's fundamental data structure used for in-memory computations
* Immutable collection of records distributed in the form of partitions across the cluster nodes
* Computations happen in parallel on these partitions

# RDD Lineage graph

* In Spark, any application execution leads to a **D**irected **A**cyclic **G**raph (DAG) creation
* DAG  is a special graph containing vertices and edges with no direct cycles. In RDD lineage graph, vertices represent RDDs and edges are the operations applied on RDDs
* DAG indicates what transformations to be executed once an action statement is encountered
* Every Spark transformation statement leads to an RDD creation where as an action statement returns a final value

Below code consists of 5 transformation statements and one action statement within a DAG graph.



All transformation statements gets computed collectively, lazily only during action statement leading to high performance. This is called **Lazy evaluation**.

**Note:**  Detailed explanation of **Transformations** and **Actions**is provided later in this course.

As shown in above DAG,

* Four RDDs (rdd1,rdd2,rdd3,rdd4) get created with in a DAG sequence based on the transformations applied
* DAG helps to process data in fault tolerant manner. When any RDD or a partition is lost in the middle of execution, it gets recreated.
* The entire graph gets executed only at the last line **rdd3.collect().foreach(println)** where an action is encountered

 Let us understand Lineage graph through a scenario:

Sam writes 30 lines of Spark code with each line creating a new RDD. First 10 processing statements were successful executed leading to 10 RDDs creation. 11th  statement produced a cluster issue/machine failure.

# What happens here?

Spark operation continues further because Spark can still traverse through the lineage (DAG) graph to the 10th RDD which was successful and can continue from there.

This is how RDDs are fault tolerant as well.

Shane works for a retail client and writes a Spark program to perform data analysis. Program contains 18 transformation statements followed by 19th statement which is an action. At which line all RDDs actually get created?

Ans :: Line 19

We are aware that RDD is an immutable collection of records distributed across the memory of the machines in the form of partitions, let us now look at different operations that can be applied on RDDs.

# Transformations

* RDDs are processed across machines using set of operations called as **transformations**. It is a function that processes existing RDD and produces new RDD as an output. Input RDD can not be modified.
* Applying transformations will build RDD lineage graph which is used while executing the application
* Lazy in nature and get executed only when an action statement is encountered in the program

Below is the list of few transformations that can be applied on RDDs

**map(function) -** Creates a new RDD by passing each record in the source RDD through a user defined function

**flatMap(function) -** Similar operation as map() but each input item can be mapped to 0, 1 or more output items

**filter(function) -** Creates a new RDD by filtering the records of the source RDD on which function returns true

**repartition(new number of Partitions) -** Modifies the degree of parallelism by changing the number of partitions

**union(other RDD) -** New RDD will be created by union of two RDDs

**reduceByKey(function) -** When called on a RDD of (K, V) pairs, returns a new RDD of (K, V) pairs where the values for each key are aggregated using the given reduce function

**join(other RDD, [numTasks]) -** When called on two RDDs of (K, V) and (K, W) pairs, returns a new RDD of (K, (V, W)) pairs with all pairs of elements for each key

**Note:** These transformations are used and implemented in Demos section.

# Actions

* Actions are RDD operations which does not produce RDD as an output. These operations finally produce a value which can be saved on the disk.
* Spark follows lazy evaluation pattern while executing the application. Until an action is encountered in the program, transformations are not executed.
* These operations can really create all the RDDs in lineage graph. Final output is sent back by executors to the driver program.

Below is the list of few action operations:

**reduce(func):**aggregates the elements of the dataset based on the function logic provided. This operation can be applied only on commutative and associative operations.

**collect():** return all the elements of RDD as an array at the driver program.

**count():** returns the number of elements in RDD

**first():** returns the first element of RDD

**take(n):** returns first n elements of RDD as an array

**saveAsTextFile(path):** saves RDD content as a text file in a provided path in local file system or HDFS

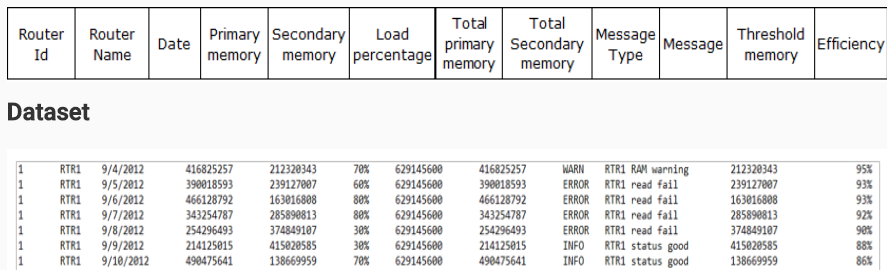
**Working with RDD’s in Spark:**

# Business scenario

Arisco Networks is a router manufacturing firm. Arisco's routers generate log every minute leading to a massive dataset stored in Hadoop’s HDFS(Hadoop distributed file system) of their hadoop cluster.

Below is a sample of the massive dataset of Arisco Networks which holds 2 million records:

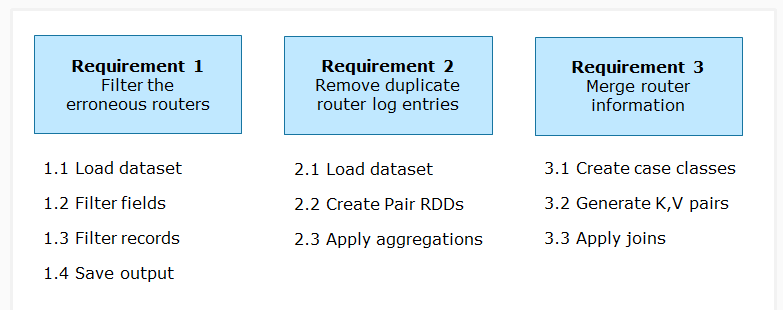
**Schema**



**Arisco has a requirement to process the above dataset to count the number of unique occurence of every record. This is needed to eliminate duplicate record entries.**

**They are using MapReduce (MR) for this analysis. Let us have a look at their MR code.**

Below is the complete snapshot of ArisCCNetwork's requirement



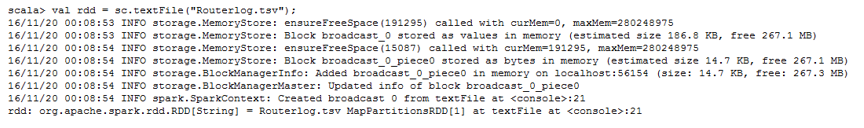
# Requirement 1: Filter the erroneous routers

**Load dataset in to RDD**

**Note:**All the demos in Spark modules are built using **Scala** programming language.

In this step ArisCCNetwork’s log dataset would be loaded in to a Spark RDD for analysis.

Below is the Scala syntax to load the data into RDD using **sc.textFile()** method.



In the above snippet, **sc** refers to built-in **SparkContext** object available in Scala Spark shell mode.

**Note:** In case we are creating Spark application in scala using IDE, SparkContext object has to be created explicitly.

The code in a Scala application with in a Scala object would look like below:

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext.\_

import org.apache.spark.SparkConf

object SparkWordCount {

def main(args: Array[String])

{

// create Spark context with Spark configuration

val sc = new SparkContext(new SparkConf().setAppName("Spark Scala Application"))

val rdd = sc.textFile("HDFS file path"); }

# What if a dataset in array need to be loaded in to RDD?

Spark offers another method called **parallelize(input data,no. of partitions)** to load arrays or lists in to RDDs.

# Example

1. val myarray = Array(1,2,3,4,5);
2. val rdd = sc.parallelize(myarray,5); *//Load array in to an RDD*

where,

* **myarray**indicates array with data
* **5**  indicates number of partitions. i.e. RDD (created out of **myarray)**gets divided into 5 partitions to enable parallel computation

Note: **textFile()** method also take number of partitions as a second parameter. sc.textFile("File Path", no of partitions)

After loading the dataset, let us filter specific fields from an RDD.

Message and Message-type fields of ArisCCNetwork need to be filtered as these fields holds the information like Errors, Warnings, Info etc.

Shown below is the code snippet to filter the fields:

val rdd = sc.textFile(“file path”);

//Filtering Message and Message Type fields

//For every record split the fields and concatenate Message Type(r(8)) and Message (r(9)) fields with “,”

rdd.map(\_.split("\t")).map(r => r(8)+ ","+r(9));

Functions used:

textFile(): Used for creating RDD from a dataset in file system

map(func): Transformation function which iterates every record in the dataset and applies func

.\_split(delimiter): Transformation function which splits every record with delimiter

After choosing specific message fields, in the next step you will learn to filter the records starting with ERRORS, and apply aggregation like count() to count the number of errors produced by each router.

val rdd = sc.textFile(“filename”);

val rdd2 = map(\_.split("\t")).map(r => (r(8)+ ","+r(9)))

//Filtering the ERROR records and count the router specific errors

val rdd3 = rdd2. filter(\_.startsWith(“ERROR”)) //Filter records starting with errors

val R1Errs = rdd3.map(\_.contains(“RTR1”)).count() //Count Rtr1 records

val R2Errs = rdd3.map(\_.contains(“RTR2”)).count() // Count Rtr2 records

val outarray = Array(“Router 1 contains ”+R1Errs,”Router 2 contains ”+R2Errs); //Collate all router output to an array

Functions used:

filter(func):

\_.startsWith(value): Transformation function for filtering the records starting with value

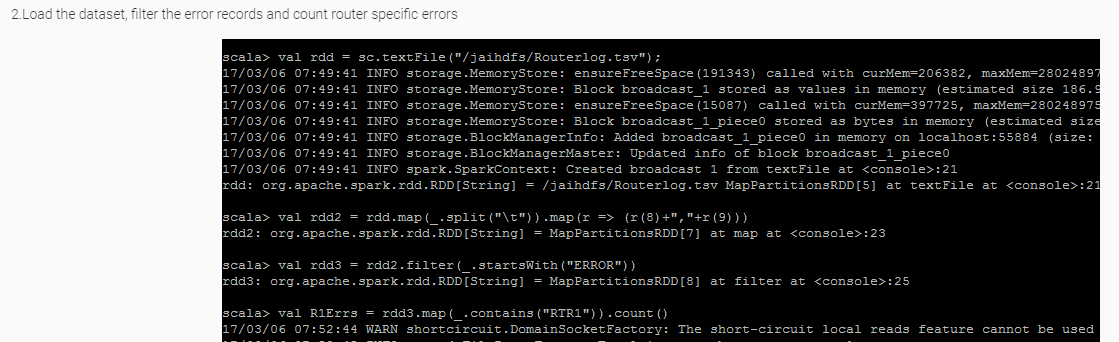
\_.contains(value): Transformation function for filtering only the records containing value in them

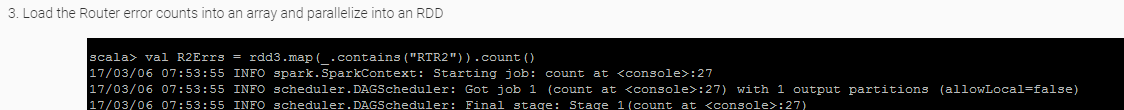
count(): Action function for counting the number of records in RDD

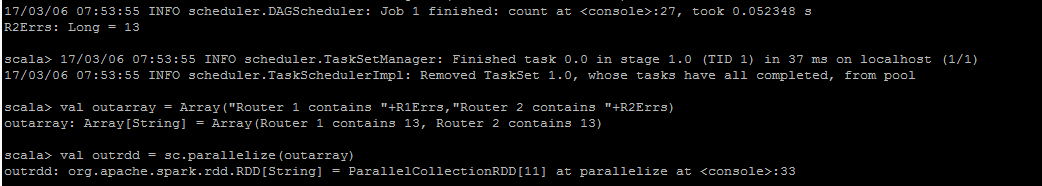
# Demosteps

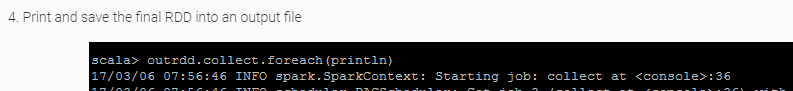
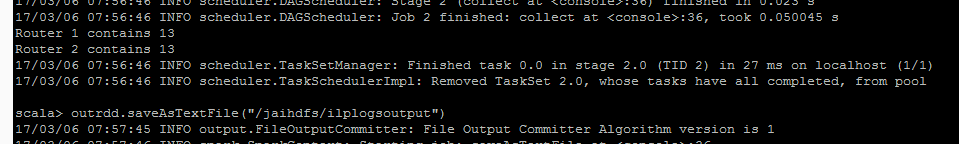
In this demo, you will load the dataset, compute and find the erroneous routers using Spark scala

1. Type in **spark-shell** to open the scala shell of Spark







Which of the below Spark Scala API can be used to load retail.csv file and create RDD?

Ans :: val retailRDD = sc.textFile("/HDFSPATH/retail.csv")

Shane works in data analytics project and needs to process Users event data (UserLogs.csv file). Which of the below code snippet can be used to split the fields with "," as a delimiter and fetch only first two fields from it?

Ans :: val logsRDD = sc.textFile("/HDFSPATH/UserLogs.csv"); val FieldsRDD = logsRDD.map(\_.split(",")).map(r => (r(0),r(1)))

**PairedRDD’s: Key-Value Paired RDD**

Next, Let us understand and solve **requirement 2**

* ArisCCNetwork needs to generate Key-Value pairs and apply aggregations on values against the key
* This helps them to find the number of occurrence of a record in the router log dataset
* This analysis is used to remove duplicate records in the dataset

Similar to requirement 1, in this step we will load the dataset using textFile() method.

After loading the dataset into RDD, use map() operation to create a Key-Value(K,V) paired RDD where complete record is taken as Key and integer 1 is taken as Value . This Value is required for aggregating and identifying duplicate records.

Code snippet:

val lines = sc.textFile(“routerLog.tsv ")

// Creating Pair RDDs

val pairs = lines.map(record => (record, 1)) // Use map() to generate pair RDD (record,1) for every record

Paired RDDs : RDDs with <Key, Value> data structure

Building blocks in most of the data analytic projects, Provide several operations which act on each key, group and aggregate the values in parallel.

Data distributed across the nodes can be regrouped and processed easily using Paired RDDs

After creating (K,V) paired RDD, next let us aggregate values of all same keys in Paired RDD (record,1).

Aggregation is applied using reduceByKey(function) transformation that executes the logic defined by function against values of each key.

In this scenario (record,1) is the (K,V) pair. For each and every record, values of record gets added as shown below:

Code snippet:

val lines = sc.textFile(“routerLog.tsv ")

val pairs = lines.map(record => (record, 1))

//Aggregating the Key-Value pair

val counts = pairs.reduceByKey((a, b) => a + b) //reduceByKey() used for aggregating sum of values for each key in a paired RDD

counts.saveAsTextFile(“/hdfspath/myuniquedir”)

**Demo steps:**

In this demo, Load the data as RDD, compute and find the number of unique occurrence of each record in the dataset

Type in spark-shell command to open the Spark scala shell.

Type in the below code line by line in the shell

val lines = sc.textFile(“/hdfspath/routerLog.tsv ")

val pairs = lines.map(record => (record, 1))

val counts = pairs.reduceByKey((a, b) => a + b)

counts.collect.foreach(println)

counts.collect.saveAsTextFile(“/hdfspath/myuniquedirectory”)

1. You can find the output as (record, number of occurrences)

Consider a retail scenario where a paired RDD exists with data **(Productname, Price)** as shown below. **Price** value has to be reduced by 500 as a discount offer to the customers. Which paired RDD function in spark could be used for this requirement?

Ans :: mapValues()

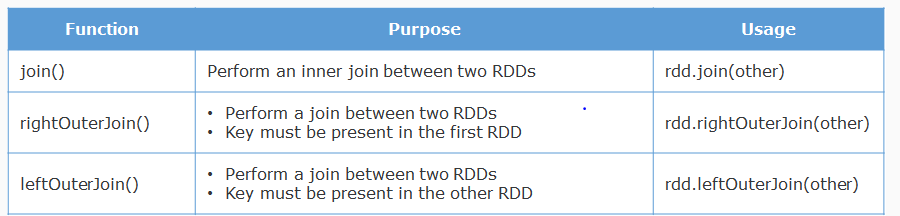
Consider a banking scenario where credit card transaction logs need to be processed. Log contains **CustomerID, CustomerName, CreditCard Number, TransactionAmount.**Which of the below code snippet creates a paired RDD **<CustomerID, TransactionAmount>**?

Ans :: val logsRDD = sc.textFile("/HDFSPath/Logs.txt"); val LogsPairedRDD = logsRDD.map(\_.split(",")).map(r => (r(0),r(3).toInt))

# Spark Joins

Spark joins are used to join one or more datasets. Spark provides **join()** function which can be used to join two paired RDDs based on the same key.

Various Spark join functions available:



Requirement

Let us consider two different datasets of ArisCCNetwork RouterLocationInfo.tsv and RouterPurchaseInfo.tsv.

Schema:

RouterLocationInfo.tsv: <RouterID, Name, Location>

RouterPurchaseInfo.tsv: <RouterID, Date, PrimaryMemory, SecondaryMemory, Cost>

Join these two datasets to fetch Routers Location, Cost and Memory details into a single RDD.

Implementation steps to join

Step 1: Create Case classes representing datasets

Create two case classes representing schema of each dataset.

// Case class representing RouterLocationInfo.tsv schema

case class RouterLocation(rid:Int,name:String,location:String);

// Case class representing RouterPurchaseInfo.tsv schema

case class RouterPurchase(rid:Int,date:String,pmemory:Long,smemory:Long,cost:Float);

Step 2: Generate K,V pairs using case class object

In this step,datasets are loaded as RDDs

Paired RDDs (K, V) are created where K = common column in both RDDs, V = Case class object.

//Load RouterLocation dataset and generate Rid(common field),RouterLocation object

val locRDD = sc.textFile(“RouterLocationInfo.tsv").map(\_.split("\t")).map(r => (r(0), RouterLocation(r(0).toInt,r(1),r(2)))

//Load RouterPurchase dataset and generate Rid(common field),RouterLocation object

val purRDD = sc.textFile(“RouterPurchaseInfo.tsv").map(\_.split("\t")).map(r => (r(0), RouterPurchase(r(0).toInt,r(1),r(2).toLong,r(3).toLong,r(4).toFloat))

Step 3: Apply join() function

In this step, Spark join is applied against the grouped fields of locRDD and purRDD from the previous step.

//Join locRDD with purRDD using join()

locRDD.join(purRDD).collect()

**Demosteps:**

Type in the below code line by line in the shell

scala>case class RouterLocation(rid:Int,name:String,location:String);

scala>case class RouterPurchase(rid:Int,date:String,pmemory:Long,smemory:Long,cost:Float);

scala>val locdata = sc.textFile(“RouterLocationInfo.tsv").map(\_.split("\t"));

scala>val locRDD = locadata.map(r => (r(0), RouterLocation(r(0).toInt,r(1),r(2)));

scala>val purRDD = sc.textFile(“RouterPurchaseInfo.tsv").map(\_.split("\t")).map(r => (r(0), RouterPurchase(r(0).toInt, r(1), r(2).toLong, r(3).toLong, r(4).toFloat));

scala>locRDD.join(purRDD).collect()

2. Load the output in to a text file using saveAsTextFile() as discussed in previous demos

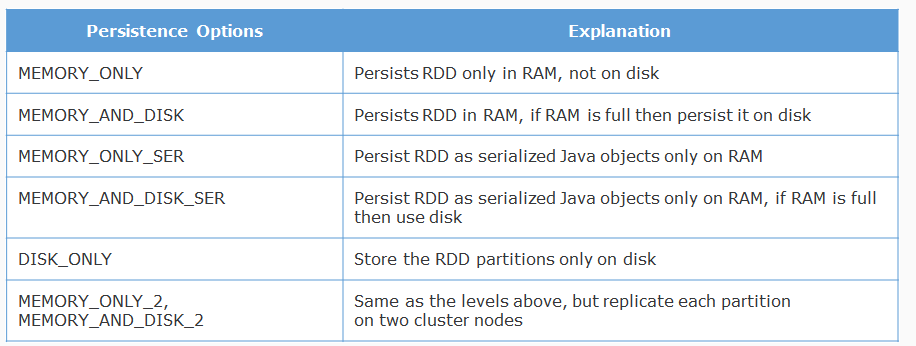
# RDD Persistence

RDDs can be cached in memory via **cache()** or **persist()** method leading to performance optimization. Cached RDDs are fault tolerant in nature.

Let us discuss this with respect to ArisCCNetwork's scenario.

# ArisCCNetwork Network’s caching requirement

ArisCCNetwork can cache their frequently used RDDs to improve performance. While persisting RDD, various persistence options are available as shown below. Persistent options can be set as a parameter in persist() method



It is the developer's responsibility to pick the best persistence level suiting their requirement.

**Note:**RDD.cache() method is equivalent to RDD.persist(MEMORY\_ONLY).

Unpersisting RDD

* Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) manner
* To manually remove an RDD, use **RDD.unpersist()** method

At this point of time you had different RDD persistence options available in Spark. But as per Spark architecture, variables defined in driver code will lose its scope as executors runs the code across machines separately.

Solution : Create Shared variables.

Shared variables retain its scope across executors running Spark tasks. Shared variables inside driver code solves the issue by keeping variables on stack rather on heap

Types of shared variables:

Accumulators for aggregation, Broadcast variable to share large values in distributed environment

Accumulators: Accumulators provide simple way of aggregating values in slave nodes and sending back to driver program.

Used to count events that occur during job execution and debugging purpose, Used to implement counters (as in MapReduce).

Limitations: If the executor failed for any reason, you will see an inconsistent value for count or sum as it executes from beginning again.

Example : Below code is used to identify and count empty lines in dataset using Accumulator.

val rdd = sc.textFile("/HDFSPath");

//Create and initialize accumulator to count blank lines in the file.

val blankLinesCounter = sc.accumulator(0);

//verify each line and increment the counter

rdd.foreach { line =>

if(line.length() == 0) blankLinesCounter += 1

}

println("Empty Lines: " + blankLinesCounter )

Broadcast variable: Broadcast variable is used to send large values over the cluster compared to accumulator

Cached on each machine rather than moving a copy of it with tasks, Broadcast variables are read-only in nature

Create and Access Broadcast variable: Broadcast variables can be created and retrieved as shown below,

//Creating a broadcast variable with simple array of integers

val broadcastVar = sc.broadcast(Array(100, 200, 300))

//Retrieving the value of a broadcast variable

broadcastVar.value

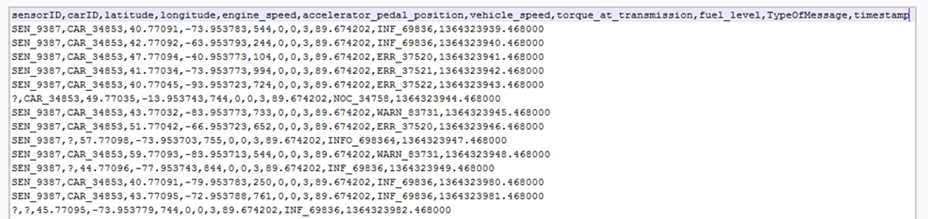
**Data Analysis Using Spark-SQL**

# Business Scenario

Arisconn Cars provides rental car service across the globe. To improve their customer service, the client wants to periodically analyze each car’s sensor data to repair faults and problems in the car.

Sensor data from cars are streamed through Events Hub (data ingestion tool) into Hadoop's HDFS (distributed file system), and analyzed using Spark Core programming to find out cars generating maximum errors. This analysis would help Arisconn to send the service team to repair the cars even before they fail.

Below is a sample of the big dataset of Arisconn Cars which holds 10 million records approximately.



Below is the Spark code to implement the first three requirements.

//Loading a text file in to an RDD

val Car\_Info = sc.textFile("/HDFSPath/ArisconnCars.txt");

//Referring the header of the file

val header = Car\_Info.first();

//Removing header and splitting records with ',' as delimiter and fetching relevant fields

val Car\_temp = Car\_Info.filter(record => record!=header).map(\_.split(",")).map(c =>(c(0),c(1),c(2).toDouble,c(3).toDouble,c(6).toInt,c(9)));

//Filtering only valid records(records not starting with '?'), and \_.\_1 refers to first field (sensorid)

val Car\_Eng\_Specs = Car\_temp.filter(!\_.\_1.startsWith("?"));

//Filtering records holding only error messages and \_.\_6 refers to 6th field (Typeofmessage)

val Car\_Error\_logs = Car\_Eng\_Specs.filter(\_.\_6.startsWith("ERR"));

In the above code,

Arisconn's dataset is loaded into RDD (Car\_Info)

The header of the dataset is removed and only fields (sensorid,carid,latitude,longitude,vehiclespeed,TypeOfMessage) are filtered. Refer to RDD Car\_temp

Records starting with '?' are removed. Refer to RDD Car\_Eng\_Specs.

Records containing TypeOfMessage = "ERR" get filtered

===

In the previous code, below are few observations and challenges of Spark Core programming

No schema or table like structure for performing advanced aggregations

Use of indices instead of field names is more complex to understand. For example, \_.\_1 is used to refer first field(sensor id)

Code customization will be highly difficult when the dataset's schema changes

Also, implementation of the fourth aggregate requirement is another challenge as shown below

Code to compute the number of errors produced by every car

//Loading a text file in to an RDD

val Car\_Info = sc.textFile("/HDFSPath/ArisconnCars.txt");

//Splitting records with ',' as delimitter and fetching relevant fields

val Car\_temp = Car\_Info.map(\_.split(",")).map(c =>(c(0), c(1), c(2).toDouble, c(3).toDouble, c(6).toInt, c(9)));

// Filtering only valid records not starting with '?'

val Car\_Eng\_Specs = Car\_temp.filter(!\_.\_1.startsWith("?"));

val Car\_Error\_logs = Car\_Eng\_Specs.filter(\_.\_5.startsWith("ERR"));

//Filtering records holding only error messages

val Car1Errors = Car\_Error\_logs.filter(\_.\_1.startsWith("CAR\_34853")).count();

//Filtering car1 records and counting the number of occurences

val Car2Errors = Car\_Error\_logs.filter(\_.\_1.startsWith("CAR\_34854")).count();

//Filtering car2 records and counting the number of occurences

.

.

//Lines of code would increase as per more number of cars.

//Combining and aggregating all the cars errors and printing them would be highly challenging

We observed that count() in the above code is applied individually on each car. With more number of cars, more number of statements would be required.

Same applies to other complex aggregate functions such as AVG, SUM, MIN, MAX.

Solution

Spark SQL provides a simple way to create schema for the dataset and write SQL queries to perform data analytics at high speed.

Why Spark SQL?

Let us look at how a Spark SQL code looks like for the same set of analysis against Arisconn's dataset.

//SQL Context object creation in Spark SQL

val sqlContext = new org.apache.spark.sql.SQLContext(sc);

//Loading a text file in to an RDD

val Car\_Info = sc.textFile("hdfs://vimsmys-42:9000/user/jai\_trng/jaihdfs/ArisconnCars.txt");

// Creating a case class mapping the fields in the dataset

case class Cars(sensorid:String, carid:String, latitude:Double, longitude:Double, engine\_speed:Int, accelerator\_pedal\_position:Int, vehicle\_speed:Int, torque\_at\_transmission:Int, fuel\_level:Double, TypeOfMessage:String, timestamp:Double);

val header = Car\_Info.first();

// Creating a Spark SQL DataFrame

val DF = Car\_Info.filter(c => c!=header).map(\_.split(",")).map(c => Cars(c(0), c(1), c(2).toDouble, c(3).toDouble, c(4).toInt, c(5).toInt, c(6).toInt, c(7).toInt, c(8).toDouble, c(9), c(10).toDouble)).toDF();

//Registering the DataFrame as a temporary table

DF.registerTempTable("cars");

// A simple query to implement all 4 requirements of Arisconn cars

val errors= sqlContext.sql("SELECT sensorid,carid,latitude,longitude,vehicle\_speed,count(TypeOfMessage) FROM cars WHERE sensorID!='sensorID' AND carid!='?' AND TypeOfMessage Like 'ERR.\*' group by carid");

Observe the above code:

Contains a single elegant Spark SQL query for all 4 requirements decreasing the number of lines of code

For future client analysis requirements, new query statements shall be added making code maintainability easier

How does the code work?

In Spark SQL, data is loaded in to a special component called DataFrame which is a programmatic component meant for loading and applying schema to datasets

Temporary table 'cars' is created on DataFrame, SQL Query will run against it

In the above code, 'DF'(DataFrame) applies schema to the dataset via the 'Cars' case class.

We will see Spark SQL and DataFrames in detail as we move further.

What is Spark-SQL ?

We have practically seen the need of Spark SQL for data analysis. Let us understand what is Spark SQL, features and its programmatic components.

Spark SQL is one of the Spark ecosystem component used for structured data processing. It provides a structure/schema for the dataset and helps to perform computations using SQL queries.

No additional configuration is required to use Spark SQL. Developers can write Spark SQL queries along with their existing Spark Core's code in the Spark scala shell.

# Features

* Ability to write queries for data analysis
* Ability to analyze data available in Hive tables
* Spark SQL queries run nearly 100 times faster than Hive queries
* Spark SQL supports different file formats like Parquet, JSON, XML, Text, CSV/TSV etc

**Spark SQL - Programmatic Components**

Below are the Spark SQL programmatic components which provides schema to the dataset and run queries on top of it

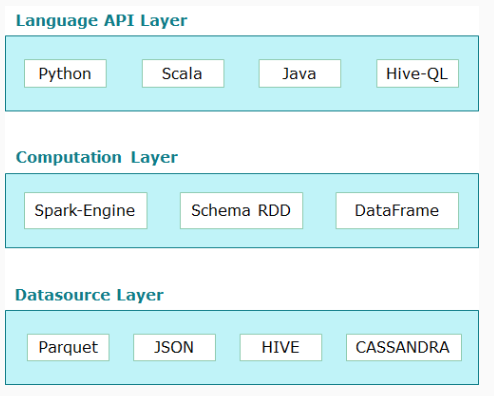
* DataFrame (Available from Spark 1.3 version)
* Dataset  (Available from Spark 1.6 version)

**Note:** We will learn more about Dataframes and Datasets in later sections.

There are few challenges in the above code and even the fourth requirement is too complex to implement in Spark Core.

# Spark SQL Architecture

Spark SQL contains multiple layers in its architectural view as shown below:

 **Data Source Layer**

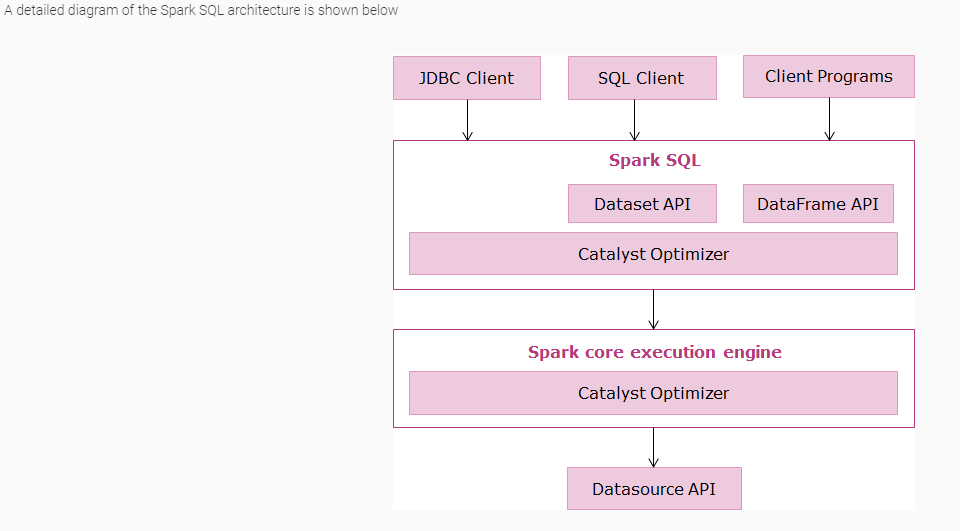
This layer refers to input file formats like Parquet, JSON, CSV, Hadoop files and JDBC data sources for analysis.

**Computation layer**

This layer provides SchemaRDD (older version), DataFrame and Dataset API for structuring and querying data. Spark SQL query gets parsed and executed in this layer.

**Language API Layer**

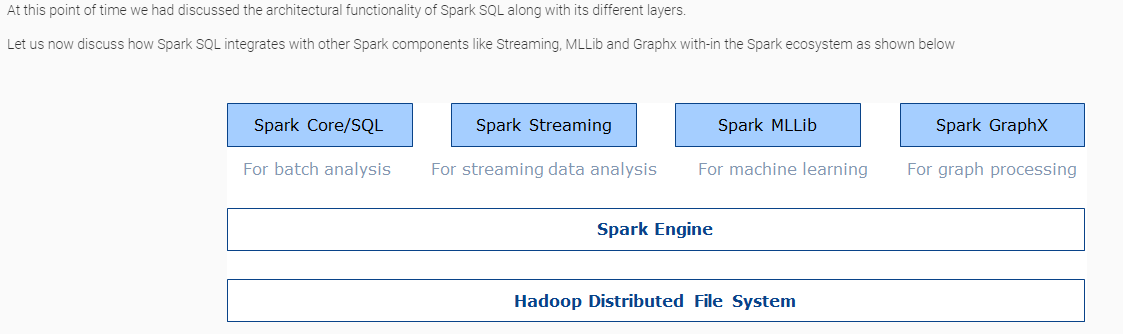
This layer lists out different programming languages to access data in DataFrames and Datasets via JDBC/ODBC and Console clients. Spark SQL applications can be built using Scala/Java/Python languages. HiveQL queries can be written directly in Spark SQL environment.



Below steps indicates the architectural flow of Spark SQL execution,

1. JDBC/ODBC and Console clients execute a Spark SQL query using DataFrame/Dataset API
2. Data in DataFrame/Dataset gets processed and optimized using catalyst optimizer
3. Catalyst optimizer executes the query to a physical plan, optimizes and compiles parts of queries to Java byte code
4. The optimized byte code constructs RDDs as per the query plan which eventually gets executed with-in Spark engine as a DAG
5. The output will be persisted as a file format supported by Datasource API

In the above flow, Catalyst optimizer takes care of the optimization part by constructing a logical plan for the query which eventually gets converted into a physical plan.



**Structured Streaming**

* Scalable and fault tolerant stream processing engine built on top of Spark SQL engine available from Spark 2.3.0 version
* Dataframe/Dataset APIs in Scala, Java, Python and R shall be used to run streaming data analysis apart from batch processing

**Spark SQL with Machine Learning**

* Spark's machine learning package (spark.mllib from 2.3.0 version) includes Spark SQL DataFrame API
* MLLib provides support for both the RDD-based API and the DataFrame-based API. DataFrame API is more user friendly than RDD API
* Features of Spark SQL like DataFrame queries, catalyst optimizer can be leveraged in Spark machine learning algorithms as well

**Spark SQL with Graphx**

* GraphFrames is a new effort to integrate pattern matching and graph algorithms with Spark SQL
* GraphFrames contains graph-aware query planner which can speed up queries
* Apart from processing HDFS data, Spark SQL integrates and perform analysis on NoSQL Databases like HBase and Cassandra

Similar to SparkContext object which acts as an entry point to all Spark Core operations, Spark SQL also needs a special object called as **SQLContext**to carry all SQL operations.

# SQLContext

* Entry point to all Spark SQL operations
* Created on top of SparkContext object
* Provides various methods to create DataFrame from different file formats
* Allows to write SQL queries on top of a DataFrame

**Below is the code snippet to create SQLContext object:**

//create SQLContext object on top of SparkContext object sc

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// this library is used to access methods which implicitly convert an RDD to a DataFrame.

import sqlContext.implicits.\_

Which of the below component plays a key role in Spark SQL query optimization?

Ans : Catalyst Optimizer

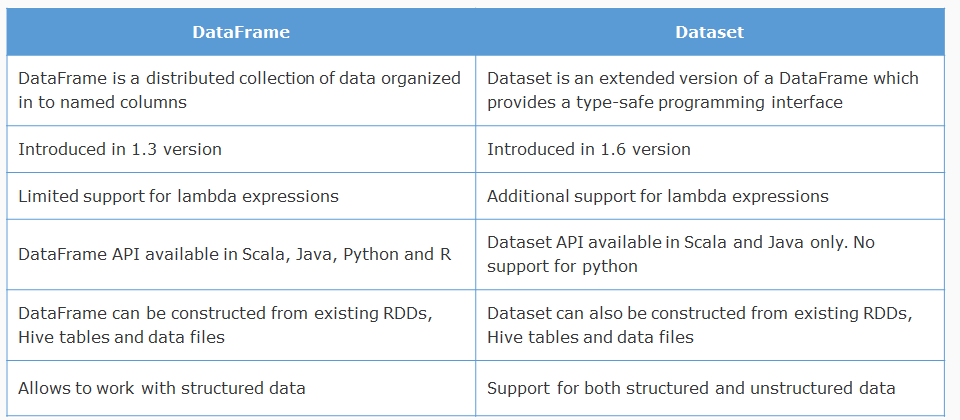
# Spark SQL Programming components

The first step in data processing using Spark SQL is to load a normal Spark RDD into either a **DataFrame**or **Dataset**.

Both DataFrame and Dataset are special API classes of Spark SQL and termed as Spark SQL's programmatic components.

DataFrame and Dataset helps to load a Spark RDD, provide a schema to it with the help of a case class, then register it as a table and further execute Spark SQL queries against the table.

Let us see a quick comparison between DataFrame and Dataset:



Requirement

ArisconnCars dataset has few corrupted records which needs to be filtered.

Solution:

1. Create a SparkContext object and load the dataset using the SparkContext object's textFile() method

//File provided in the path is loaded in to Spark RDD using SparkContext object 'sc'

val Car\_Info = sc.textFile("/HDFSpath/ArisconnCars.txt");

//dataset's first record contains schema details which needs to be ignored while processing

val header = Car\_Info.first()

2. Create a SQLContext object using SQLContext class

//Creation of a SQLContext object in Spark SQL

val sqlContext = new org.apache.spark.sql.SQLContext(sc);

3. Create a DataFrame and load a normal RDD in to the DataFrame using Case class

//Creation of a case class mapping the fields in the dataset

case class Cars(sensorid: String, carid: String, latitude:Double, longitude:Double, engine\_speed:Int, accelerator\_pedal\_position:Int, vehicle\_speed:Int, torque\_at\_transmission:Int, fuel\_level:Double, TypeOfMessage:String, timestamp:Double)

/\*Creation of a Spark SQL DataFrame by delimiting the fields, and loading them as case class properties. Note: "toInt", "toDouble" are Scala methods for converting text fields in to numerical\*/

val DF = Car\_Info.filter(c => c!=header).map(\_.split(",")).map(c => Cars(c(0), c(1), c(2).toDouble, c(3).toDouble, c(4).toInt, c(5).toInt, c(6).toInt, c(7).toInt, c(8).toDouble, c(9), c(10).toDouble)).toDF()

4. Register the DataFrame as a normal table using registerTempTable() method

//Registering the DataFrame as a temporary table

DF.registerTempTable("cars");

5. Write and execute Spark SQL queries against the table using SQLContext object's sql() method

/\* Query using sql method to filter corrupted records and consider only valid data for further analysis \*/

val valrecords= sqlContext.sql("SELECT sensorid,carid,latitude,longitude,vehicle\_speed,TypeOfMessage FROM cars WHERE sensorid Like 'SEN\_%' AND carid Like 'CAR\_%' AND sensorid!='sensorID'");

//display the dataframe records on console

valrecords.show()

