**What is Pair RDD?**

Pair RDDs can be created by running a map() function that returns key or value pairs.

val pairs = lines.map(x => (x.split(" ")(0), x))

Actions Available on Pair RDDs are as follows :

**countByKey() :** Count the number of elements for each key pair.

**collectAsMap() :** Collect the result outputs as a map to provide easy lookup.

**lookup(key) :** Return all values associated with the provided key pair.

**Pandas vs Spark DataFrame?**

**What is difference between Driver and executer in Spark?**

* Spark driver is the master and the Spark executor is the slave. Each of these components runs as an independent process on a Spark cluster. A Spark application consists of one and only one Spark driver and one or more Spark executors.
* Each Spark executor is a JVM process and is exclusively allocated to a specific Spark application. This was a conscious design decision to avoid sharing a Spark executor between multiple Spark applications in order to isolate them from each other so one badly behaving Spark application wouldn’t affect other Spark applications. The lifetime of a Spark executor is the duration of a Spark application, which could run for a few minutes or for a few days.
* Since Spark applications are running in separate Spark executors, sharing data between them will require writing the data to an external storage system like HDFS.
* The Spark *driver* is the central coordinator of a Spark application, and it interacts with a cluster manager to figure out which machines to run the data processing logic on. For each one of those machines, the Spark driver requests that the cluster manager launch a process called the Spark *executor*.
* Another important job of the Spark driver is to manage and distribute Spark tasks onto each executor on behalf of the application.
* If the data processing logic requires the Spark driver to display the computed results to a user, then it will coordinate with each Spark executor to collect the computed result and merge them together. The entry point into a Spark application is through a class called SparkSession, which provides facilities for setting up configurations as well as APIs for expressing data processing logic.
* Playing the slave role, each Spark executor does what it is told, which is to execute the data processing logic in the form of tasks. Each task is executed on a separate CPU core. This is how Spark can speed-up the processing of a large amount of data by processing it in parallel. In addition to executing assigned tasks, each Spark executor has the responsibility of caching a portion of the data in memory and/or on disk when it is told to do so by the application logic.

**How to decide number of Executer and memory in Spark?**

Let’s assume we have 10 machines with 16 cores so we have 160 cores. Each machine has 64 GB RAM.  
We can have Smallest size executer, Biggest size executer and right approach.

1. Smallest:
   * 1 core and 4 GB ram per executer.
   * 16 executers on each machine since there are 16 cores.
   * This is not a good approach because we are not using parallelization capability of executer.
2. Largest:
   * 10 Executer (since we have 10 machine, executer will take all cores on that machine)
   * 64 GB RAM per executer.
   * 16 core per executer (all 16 cores are given to one executer on that machine).
   * IO Contention is a problem as all executer Cores will be competing with each other to read partition from HDFS.
   * No resource left for OS
   * No memory overhead for YARN
3. Best:
   * We should leave one core on each machine for OS processes
   * That means 160 -10 = 150 Cores can be assigned to executers.
   * 150/10 = 15 => on every machine we have 15 cores.
   * No. of executer core = 5, as more cores per executer results in IO contention.
   * No. of executer per machine = 15/5 = 3.
   * Memory: Give 1 GB to OS. So now we have 63 GB ram per machine
   * Per machine we have 3 Executer that means Ram per executer is 63/3 = 21GB is available
   * But we also have to consider for YARN overhead (2GB).
   * Per executer RAM = 19GB.

**What happens if my data doesn’t fit in memory in Spark?**

It is spilled to disk allowing Spark to run on any size of data. Likewise, cached data that do not fit in memory are either spilled to disk or recomputed on the fly when needed.

**What is difference between Executer and Executer core?**

Executer is jvm process, while executer core is number of parallel tasks that are related to jvm process called executer.

Executer is Yarn container while Executer core is thread of that process.

**Which language to choose for Spark? Scala or Python?**

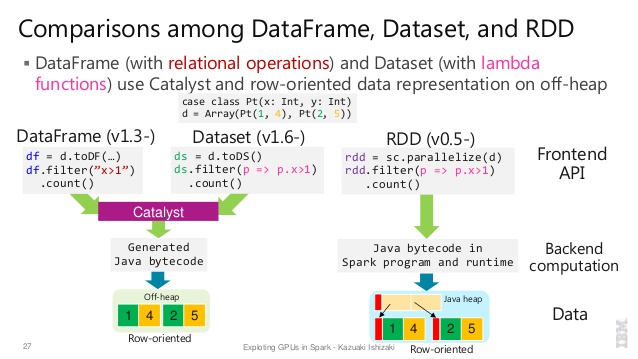
Performance, Enterprise Acceptance, Type Safe: Scala

Usability, Learning Curve, ML, Libraries, Viz: Python

**What is difference between RDD, DataFrame and DataSet?**

|  |  |  |
| --- | --- | --- |
| **RDD** | **DataFrame** | **DataSet** |
| RDD,is **R**esilient **D**istributed **D**ataset that is more of a blackbox of data that cannot be optimized as the operations that can be performed against it, are not as constrained.  RDD is a fault-tolerant collection of elements that can be operated on in parallel. | DataFrame has additional metadata due to its tabular format, which allows Spark to run certain optimizations on the finalized query.  it is recommended to use a DataFrame where possible due to the built in query optimization. | 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**. |
|  | DataFrame to an RDD via its rdd method, and from an RDD to a DataFrame (if the RDD is in a tabular format) via the toDF method | **val** rdd = sc.textFile("data.txt")  **val** ds = spark.createDataset(rdd) |
| * Your data is unstructured, for example, binary (media) streams or text streams * you want to control your dataset and use low-level transformations and actions * your data types cannot be serialized with Encoders (an optimized approach that uses runtime code generation to build custom bytecode for serialization and deserialization) * you are ok to miss optimizations for DataFrames and Datasets for structured and semi-structured data that are available out of the box * you don’t care about the schema, columnar format and ready to use functional programming constructs | * your data is structured (RDBMS input) or semi-structured (json, csv) * you want to get the best performance gained from SQL’s optimized execution engine (Catalystoptimizer and Tungsten’s efficient code generation) * you need to run hive queries * you appreciate domain specific language API (.groupBy, .agg, .orderBy) * you are using R or Python | * your data is structured or semi-structured * you appreciate type-safety at a compile time and a strong-typed API * you need good performance (mostly greater than RDD), but not the best one (usually lower than DataFrames)   Spark RDD to DataFrame |

**Why dataset is type safe?**



**What is dynamic resource Allocation?**

By Default, it is disabled.

Dynamic allocation allows Spark to, dynamically scale the cluster resources allocated for the Spark application. When dynamic allocation is enabled, if there are backlog of pending tasks for a Spark application, it can request for new executors. When the application becomes idle, its executors are released and the same can be acquired by other spark applications. To enable Dynamic allocation for Spark, following steps could be used:

1. In Ambari Spark-Configs, edit the Custom spark-defaults section and add the following parameters:

1. spark.dynamicAllocation.enabled = true
2. spark.shuffle.service.enabled = true
3. spark.dynamicAllocation.initialExecutors = 3 (Initial number of executors to run if dynamic allocation is enabled, this is same as "spark.dynamicAllocation.minExecutors")
4. spark.dynamicAllocation.minExecutors = 3 (executors number will come to this number if executors are not in use, after 60 sec(default), controlled by "spark.dynamicAllocation. executorIdleTimeout")
5. spark.dynamicAllocation.maxExecutors = 30 (maximum executors that job can request)

2. Restart Spark services using Ambari

3. Validate Dynamic allocation by running a sample job, for example,

1. pyspark --master yarn

Since no executor specification is specified at run time, the job would start with the settings completed in Ambari. As defined in our example setting, it would start with 3 executors. If the job needs more executor, it would request for the same and the following messages would be seen on the console:

1. 6/05/23 09:39:47 INFO ExecutorAllocationManager: Requesting 2 new executors because tasks are backlogged (new desired total will be 4)
2. 16/05/23 09:39:48 INFO ExecutorAllocationManager: Requesting 1 new executor because tasks are backlogged (new desired total will be 5)

If executors are specified while running the job, Dynamic allocation would be disabled. For example,

pyspark --master yarn --num-executors 50 --executor-memory 3G

The above program would display the following warning on the console:

1. 16/05/23 09:18:54 WARN SparkContext: Dynamic Allocation and num executors both set, thus dynamic allocation disabled.

**What are benefits of External Shuffle Service?**

ExternalShuffleService is an **external shuffle service** that serves shuffle blocks from outside an [Executor](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-Executor.html) process. It runs as a standalone application and manages shuffle output files so they are available for executors at all time. As the shuffle output files are managed externally to the executors it offers an uninterrupted access to the shuffle output files regardless of executors being killed or down.

You start ExternalShuffleService using [start-shuffle-service.sh shell script](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-ExternalShuffleService.html#start-script) and enable its use by the driver and executors using [spark.shuffle.service.enabled](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-ExternalShuffleService.html#spark.shuffle.service.enabled).

**What is difference between Yarn Client Mode and Cluster mode?** [how to choose which one to use?](https://stackoverflow.com/questions/41124428/spark-yarn-cluster-vs-client-how-to-choose-which-one-to-use)

In YARN client mode driver runs on the machine from where job was submitted. While in Yarn Cluster mode, driver will run on a data node on the cluster.

A common deployment strategy is to submit your application from a gateway machine that is physically co-located with your worker machines (e.g. Master node in a standalone EC2 cluster). In this setup, client mode is appropriate. In client mode, the driver is launched directly within the spark-submit process which acts as a client to the cluster. The input and output of the application is attached to the console. Thus, this mode is especially suitable for applications that involve the REPL (e.g. Spark shell).

Alternatively, if your application is submitted from a machine far from the worker machines (e.g. locally on your laptop), it is common to usecluster mode to minimize network latency between the drivers and the executors. Note that cluster mode is currently not supported for Mesos clusters. Currently only YARN supports cluster mode for Python applications."

**What Optimization techniques you have used in Spark?**

* First filter then Join

**What happens if memory cache size is full and I try to cache data?**

Data is spilled on disk

**What is Lineage in Spark?**

When a transformation (map or filter etc) is called, it is not executed by Spark immediately, instead a lineage is created for each transformation. A lineage will keep track of what all transformations has to be applied on that RDD, including the location from where it has to read the data.

For example, consider the following example

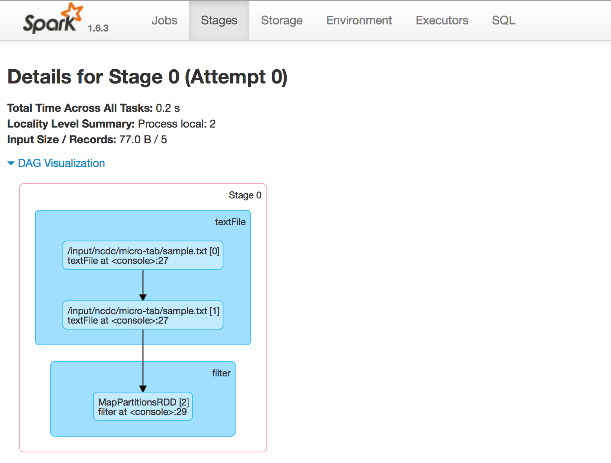
val myRdd = sc.textFile("spam.txt")

val filteredRdd = myRdd.filter(line => line.contains("wonder"))

filteredRdd.count()

sc.textFile() and myRdd.filter() do not get executed immediately, it will be executed only when an Action is called on the RDD - here filteredRdd.count().

An Action is used to either save result to some location or to display it. RDD lineage information can also be printed by using the command filteredRdd.toDebugString(filteredRdd is the RDD here). Also, DAG Visualization shows the complete graph in a very intuitive manner as follows:



**What is role of SparkContext?**

**SparkContext** is the entry gate of [**Apache Spark**](http://data-flair.training/blogs/apache-spark-tutorial-quickstart-introduction/) functionality. The most important step of any Spark driver application is to generate SparkContext. To learn all the functions of SparkContext, first we will know the brief introduction SparkContext.

**Can I run Spark without Hadoop?**

Yes, Spark can run without Hadoop and also on Hadoop.

**Why Spark is used for Machine Learning and other Low Latency Applications?**

In Memory very fast data access and iterations, that is core of Machine Learning.

**What is the default level of parallelism in Spark?**

select

**Spark Partition Tuning**

Let us first decide the number of partitions based on the input dataset size. The rule of thumb to decide the partition size while working with HDFS is **128 MB**. As our input dataset size is about 1.5 GB (1500 MB) and going with 128 MB per partition, the number of partitions will be:

Total input dataset size / partition size => 1500 / 128 = 11.71 =**~12 partitions.**

This is equal to the Spark default parallelism (spark.default.parallelism) value. The metrics based on default parallelism are shown in the above section.

Now, let us perform a test by reducing the partition size and increasing the number of partitions.

./bin/spark-submit

--name FireServiceCallAnalysisDataFramePartitionTest

--master yarn

--deploy-mode cluster

--executor-memory 2g

--executor-cores 2

--num-executors 2

--conf spark.sql.shuffle.partitions=23

--conf spark.default.parallelism=23

--class com.treselle.fscalls.analysis.FireServiceCallAnalysisDF /data/SFFireServiceCall/SFFireServiceCallAnalysis.jar /user/tsldp/FireServiceCallDataSet/Fire\_Department\_Calls\_for\_Service.csv

**Spark Lineage vs DAG?**

Lineage Graph

Lineage is the graph of how all the parent RDD’s are connected to the derived RDD’s. It hence is a graph of how each RDD is dependent on the other and how the transformations are applied to each RDD. Each RDD points to one or more parent along with the metadata about the type of relationship with the parent.

For example: val result=data.map(), where result keeps a reference of the RDD data, that’s a lineage. This RDD lineage is used to recompute the data if there are any faults as it contains the pattern of the computation. toDebugString() method is used to display the RDD lineage.

DAG in [Apache Spark](https://data-flair.training/blogs/apache-spark-for-beginners/) is a combination of Vertices as well as Edges. In DAG vertices represent the RDDs and the edges represent the Operation to be applied on RDD. Every edge in DAG is directed from earlier to later in a sequence. When we call an [Action](https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/), the created DAG is submitted to DAG Scheduler which further splits the graph into the stages of the task.

**What are Partition and Coalesce operations?**

Spark splits data into partitions and executes computations on the partitions in parallel. You should understand how data is partitioned and when you need to manually adjust the partitioning to keep your Spark computations running efficiently.

Lest’s create a DataFrame of numbers to illustrate how data is partitioned:

val x = (1 to 10).toList  
val numbersDf = x.toDF(“number”)

On my machine, the numbersDf is split into four partitions:

numbersDf.rdd.partitions.size // => 4

Each partition is a separate CSV file when you write a DataFrame to disc.

numbersDf.write.csv(“/Users/powers/Desktop/spark\_output/numbers”)

Here is how the data is separated on the different partitions.

Partition A: 1, 2  
Partition B: 3, 4, 5  
Partition C: 6, 7  
Partition D: 8, 9, 10

coalesce

The coalesce method reduces the number of partitions in a DataFrame. Here’s how to consolidate the data in two partitions:

val numbersDf2 = numbersDf.coalesce(2)

Increasing partitions: The coalesce algorithm changes the number of nodes by moving data from some partitions to existing partitions. This algorithm obviously cannot increase the number of partitions.

**repartition**

The repartition method can be used to either increase or decrease the number of partitions in a DataFrame.

val bartDf = numbersDf.repartition(6)  
bartDf.rdd.partitions.size // => 6

**How do you handle computing task failure in Spark?**

**What is Broadcast variable, give a real life example?**

scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))

broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> broadcastVar.value

res2: Array[Int] = Array(1, 2, 3)

**Unpersisting Broadcast Variable —**unpersist **Methods**

**Destroying Broadcast Variable —**destroy **Method**

**What is accumulator give a real life example?**

scala> val accum = sc.accumulator(0, "Accumulator Example")

accum: spark.Accumulator[Int] = 0

scala> sc.parallelize(Array(1, 2, 3)).foreach(x => accum += x)

scala> accum.value

res4: Int = 6

**What is difference between Spark Partitions and Bucketing?**

|  |  |
| --- | --- |
| **Partitioning** | **Bucketing** |
| **Partitioning** data is often used for distributing load horizontally, this has performance benefit, and helps in organizing data in a logical fashion  ***Example***: if we are dealing with a large employee table and often run queries with where clauses that restrict the results to a particular country or department. For a faster query response Hive table can be PARTITIONED BY (country STRING, DEPT STRING). Partitioning tables changes how Hive structures the data storage and Hive will now create subdirectories reflecting the partitioning structure like  .../employees/country=ABC/DEPT=XYZ.  If query limits for employee from country=ABC, it will only scan the contents of one directory country=ABC. This can dramatically improve query performance, but only if the partitioning scheme reflects common filtering. Partitioning feature is very useful in Hive, however, a design that creates too many partitions may optimize some queries, but be detrimental for other important queries. Other drawback is having too many partitions is the large number of Hadoop files and directories that are created unnecessarily and overhead to NameNode since it must keep all metadata for the file system in memory. | **Bucketing** is another technique for decomposing data sets into more manageable parts. For example, suppose a table using date as the top-level partition and employee\_id as the second-level partition leads to too many small partitions. Instead, if we bucket the employee table and use employee\_id as the bucketing column, the value of this column will be hashed by a user-defined number into buckets. Records with the same employee\_id will always be stored in the same bucket. Assuming the number of employee\_id is much greater than the number of buckets, each bucket will have many employee\_id. While creating table you can specify like CLUSTERED BY (employee\_id) INTO XX BUCKETS; where XX is the number of buckets . Bucketing has several advantages. The number of buckets is fixed so it does not fluctuate with data. If two tables are bucketed by employee\_id, Hive can create a logically correct sampling. Bucketing also aids in doing efficient map-side joins etc.  CREATE TABLE mytable (  name string,  city string,  employee\_id int )  PARTITIONED BY (year STRING, month STRING, day STRING)  CLUSTERED BY (employee\_id) INTO 256 BUCKETS |

[**Are failed tasks resubmitted in Apache Spark?**](https://stackoverflow.com/questions/26260006/are-failed-tasks-resubmitted-in-apache-spark)

Yes, but there is a parameter set for the max number of failures

spark.task.maxFailures 4 Number of individual task failures before giving up on the job. Should be greater than or equal to 1. Number of allowed retries = this value - 1.

[**map vs flatmap in Apache Spark?**](https://stackoverflow.com/questions/26260006/are-failed-tasks-resubmitted-in-apache-spark)

Map is one to one transformation while Flatmap is not so.

Here is an example of the difference, as a spark-shell session:

First, some data - two lines of text:

val rdd = sc.parallelize(Seq("Roses are red", "Violets are blue")) // lines

rdd.collect

res0: Array[String] = Array("Roses are red", "Violets are blue")

Now, map transforms an RDD of length N into another RDD of length N.

For example, it maps from two lines into two line-lengths:

rdd.map(\_.length).collect

res1: Array[Int] = Array(13, 16)

But flatMap (loosely speaking) transforms an RDD of length N into a collection of N collections, then flattens these into a single RDD of results.

rdd.flatMap(\_.split(" ")).collect

res2: Array[String] = Array("Roses", "are", "red", "Violets", "are", "blue")

We have multiple words per line, and multiple lines, but we end up with a single output array of words

Just to illustrate that, flatMapping from a collection of lines to a collection of words looks like:

["aa bb cc", "", "dd"] => [["aa","bb","cc"],[],["dd"]] => ["aa","bb","cc","dd"]

The input and output RDDs will therefore typically be of different sizes for flatMap.

If we had tried to use map with our split function, we'd have ended up with nested structures (an RDD of arrays of words, with type RDD[Array[String]]) because we have to have exactly one result per input:

rdd.map(\_.split(" ")).collect

res3: Array[Array[String]] = Array(

Array(Roses, are, red),

Array(Violets, are, blue)

)

Finally, one useful special case is mapping with a function which might not return an answer, and so returns an Option. We can use flatMap to filter out the elements that return None and extract the values from those that return a Some:

val rdd = sc.parallelize(Seq(1,2,3,4))

def myfn(x: Int): Option[Int] = if (x <= 2) Some(x \* 10) else None

rdd.flatMap(myfn).collect

res3: Array[Int] = Array(10,20)

(noting here that an Option behaves rather like a list that has either one element, or zero elements)

**What is Dstream?**

**What are properties of RDD?**

**How to create RDD in Spark?**

**Parallelized collection (parallelizing)**

1. val data=spark.sparkContext.parallelize(Seq(("maths",52),("english",75),("science",82), ("computer",65),("maths",85)))
2. val sorted = data.sortByKey()
3. sorted.**foreach**(**print**ln)
4. val rdd1 = spark.sparkContext.parallelize(**Array**("jan","feb","mar","april","may","jun"),3)
5. val result = rdd1.coalesce(2)
6. result.**foreach**(**print**ln)

**External Datasets (Referencing a dataset)**

1. import org.apache.spark.sql.SparkSession
2. def main(args: **Array**[String]):Unit = {
3. object Data**For**mat {
4. val spark = SparkSession.builder.appName("AvgAnsTime").master("local").getOrCreate()
5. val dataRDD = spark.read.csv("path/of/csv/file").rdd

val dataRDD = spark.read.json("path/of/json/file").rdd

val dataRDD = spark.read.textFile("path/of/text/file").rdd

**Creating RDD from existing RDD**

1. val words=spark.sparkContext.parallelize(Seq("the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"))
2. val wordPair = words.map(w => (w.charAt(0), w))
3. wordPair.**foreach**(**print**ln)

**What are real time applications of Apacke Spark?**

• Customer intelligence applications

• Data warehouse solutions

• Real-time streaming solutions

• Recommendation engines

• Log processing

• User-facing services

• Fraud detection

**How to Launch Spark Shell?**

To start a Spark Scala shell, enter this command in the Spark directory:

./bin/spark-shell

To start up a Spark Python shell, enter this command in the Spark directory:

./bin/pyspark

**How to launch Spark UI?**

Spark context Web UI available at <http://192.168.1.73:4042>

[**What is Lazy Evaluation?**](https://stackoverflow.com/questions/26260006/are-failed-tasks-resubmitted-in-apache-spark)

It is not desirable to immediately trigger an evaluation of every single filtering operation when a dataset is large in size. The typical end goal of a data processing task is to write the result out to some external storage system or to see how many records there are. This is when it makes sense to evaluate all the previously specified computational logic. One important optimization technique behind the lazy evaluation concept is the ability to collapse or combine similar transformations into a single operation during execution time.

**When should we use Flatmap and when Map?**

Flatmap when we want output to be single flattened set. While map when we want output to be applied on input set. That means if input set has 2 array output will also have 2 arrays in case of map.

**What is Data Shuffling?**

Certain key/value transformations and actions require moving data from multiple partitions to other partitions, meaning across executors and machines. This process is known as the *shuffle*, which is quite important to be familiar with because it is an expensive operation. During the shuffling process, the data needs to be serialized, written to disk, transferred over the network, and finally deserialized. It is not possible to completely avoid the shuffling, but there are techniques or best practices to minimize the need to shuffle the data. Shuffling data will add latency to the completion of the data processing in your Spark jobs.

**What are Data Shuffling transformations?**

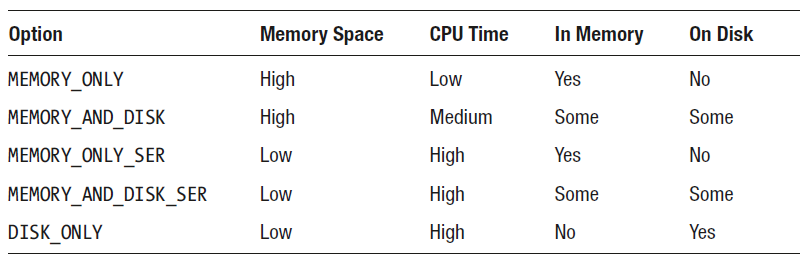
In general, any transformation or action that performs some sort of grouping, aggregating, or joining by key will need to perform data shuffling. Here is a subset of the transformations that fall into this category: groupByKey, reduceByKey, aggregateByKey, and join.

**How to Persist data in Spark?**

Persisting an RDD is extremely simple to do by calling the transformation persist() or cache(). Since they are transformations, only once a subsequent action is taken will the RDD be persisted in memory. By default, Spark will persist the dataset in memory.

cache() will use MEMORY\_ONLY. If you want to use something else, use persist(StorageLevel.<\*type\*>).

By default persist() will store the data in the JVM heap as unserialized objects.



**How do you read csv file in Spark in distributed manner?**

When you define the reading, the file would be divided to partitions based on your parallelism scheme and the instructions would be sent to the workers. Then the file is read directly by the workers from the filesystem (hence the need for a distributed filesystem available to all the nodes such as HDFS).

As a side note, it would be much better to read it to a dataframe using spark.read.csv and not in RDD. This would take less memory and would allow spark to optimize your queries.

**How to submit your Spark job for parallel processing?**

--class org.apache.spark.examples.SparkPi

--master yarn --deploy-mode client

--driver-memory 4g

--num-executors 2

--executor-memory 2g

--executor-cores 2 /opt/apps/spark-1.6.0-bin-hadoop2.6/lib/spark-examples\*.jar 10

-master yarn-client

--driver-memory 5g

–-num-executors 20

--executor-memory 4g

--executor-cores 4

**What are the differences between ORC, Avro and Parquet File Formats?**

|  |  |  |
| --- | --- | --- |
| AVRO | ORC | Parquet |
| * It is row major format. * Its primary design goal was schema evolution. * In the avro format, we store schema separately from data. Generally avro schema file (.avsc) is maintained. | Column oriented storage format.   * Originally it is Hive's Row Columnar file. Now improved as Optimized RC (ORC) * Schema is with the data, but as a part of footer. * Data is stored as row groups and stripes. * Each stripe maintains indexes and stats about data it stores. | * Similar to ORC. Based on google dremel * Schema stored in footer * Column oriented storage format * Has integrated compression and indexes |
| For DataFrame I'd go with Avro [data source directly](https://github.com/databricks/spark-avro):  Include spark-avro in packages list. For the latest version use:  com.databricks:spark-avro\_2.11:3.2.0  Load the file:   * val df = spark.read   .format("com.databricks.spark.avro")  .load(path) | val movies11 = spark.read.orc("<path>/book/chapter4/data/movies/movies.orc") | spark.read.parquet("<path>")  spark.read.format("parquet")  val movies9 = spark.read.load("<path>/book/chapter4/data/movies/movies.parquet")  val movies10 = spark.read.parquet("<path>/book/chapter4/data/movies/movies.parquet") |

In case of Parquet or any column based format, to access a row of data we need to query multiple nodes, as data file is split and distributed on nodes resulting in one record (columnar) to be distributed along multiple files or nodes.

**What are different ways to Improve Spark Performance?**

1. Tree reduce vs Reduce Function, or Tree Aggregate function. Here amount of data travelling from executer to driver is controlled.
2. User broadcast join wherever applicable.
3. Use Spark 2.0, It has optimized code generators and Spark Encoders.
4. In case of Spark 1.x use Kyro Serialization.

**How to load data in DataFrame in Spark?**

val auction = sc.textFile("ebay.csv").map(\_.split(",")).map(p =>

Auction(p(0),p(1).toFloat,p(2).toFloat,p(3),p(4).toInt,p(5).toFloat,p(6).toFloat,p(7),p(8).toInt )).toDF()