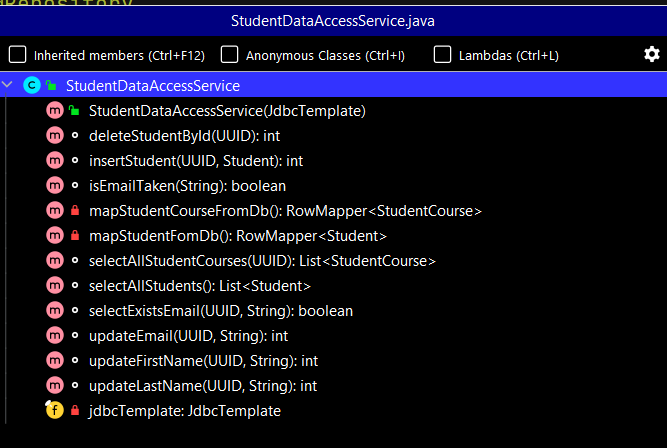
**---------------------------------intelliJ Short cut-----------------------------------------------**

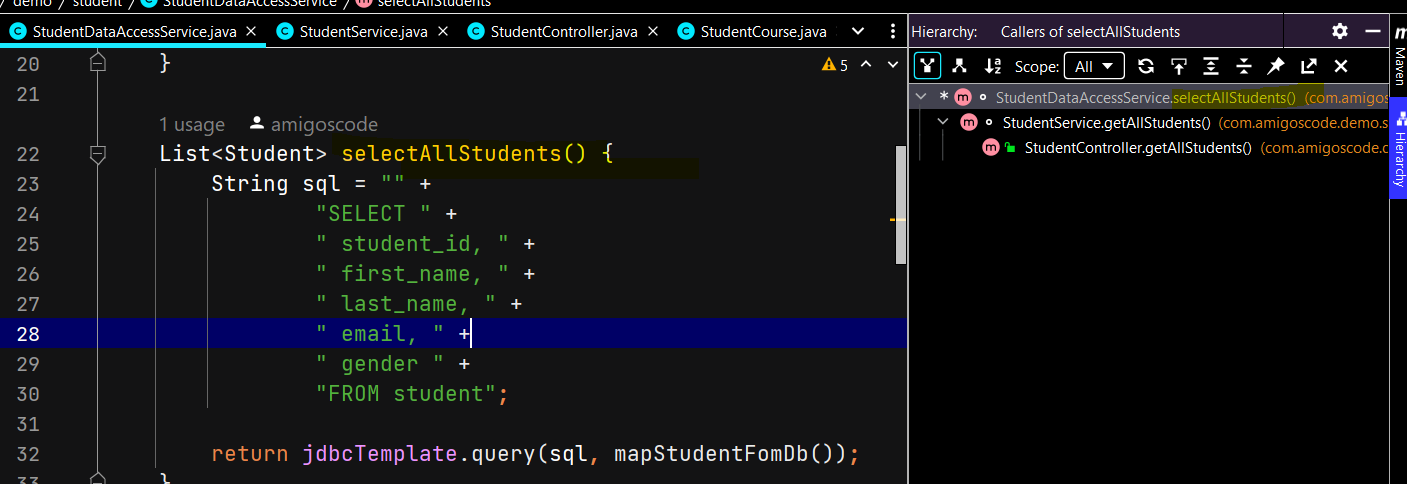
1. Ctrl+Shift+R or ctrl+shift+N for file search
2. Press shift button double
3. Ctrl+` for switch
4. Ctrl+F12 to see java file structure (all methods and functions)

The below screen shows the file structure of the class and here we can see all the method, Anonymous classess, Lambda used in the methods etc.

****

**Searching Call Hierarchy**

1. Select the particular method > goto Navigation Menu> Call Hierarchy
2. Short cut: Slect the method and ctrl+alt+H

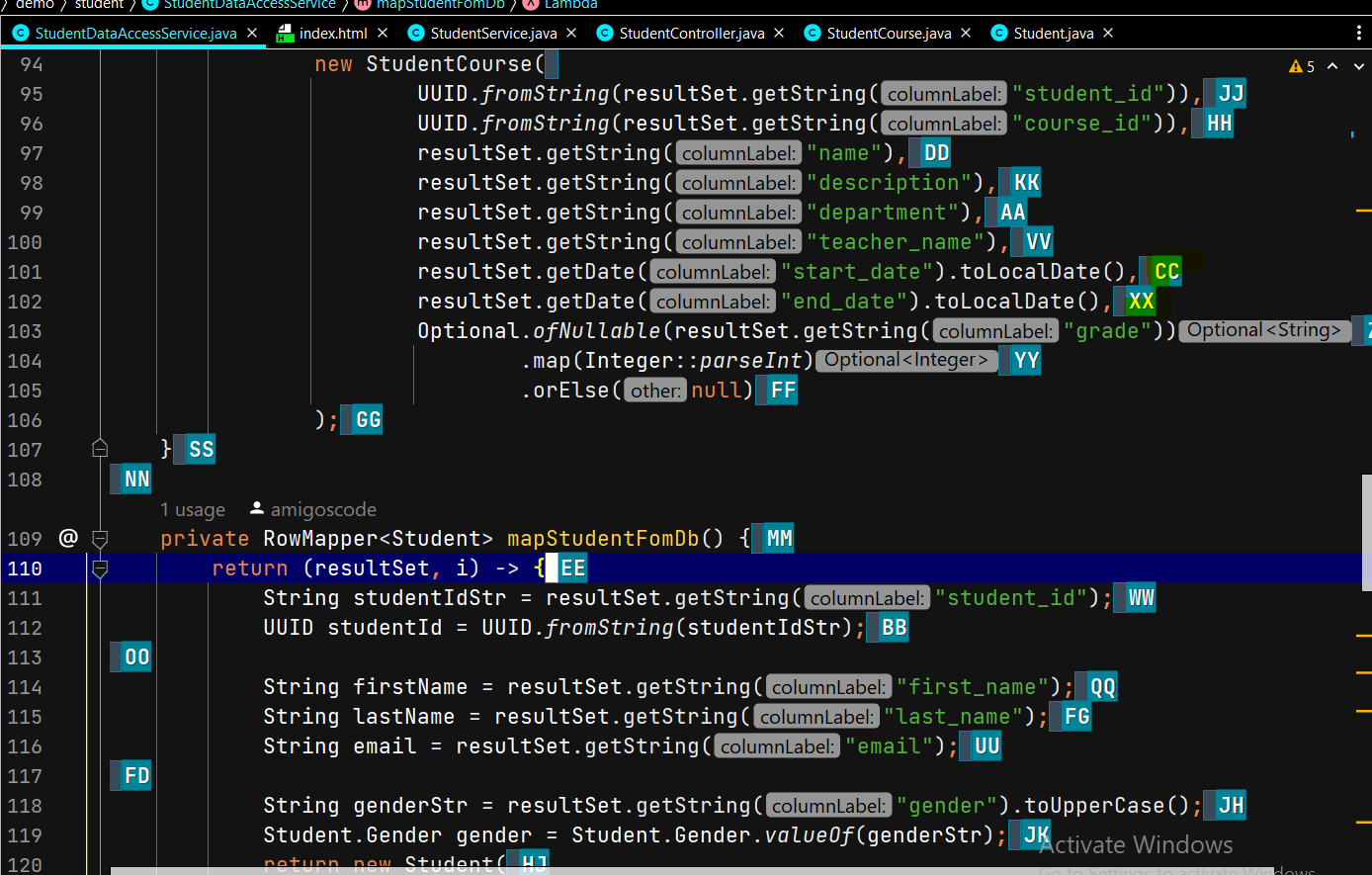
****

**Use of find usages in Intellij:** It just tells the exact number of usages and line of the method in the class, call hierarchy is more powerful than find usages.

**Right click on method > find usages**

**Note1-: We can use several shortcut mentions in the Intellij itself: like – Alt+1 for project explorer, Alt+2 for class structure and so on**

**Note2- Install plugin AceJump in the Intellij and go to the class > press ctrl+ ; > press left or right arrow > and then it will populate the later> press the latter to go to that location**

****

**For example to any line just press that particular letter and cursor will reach there.**

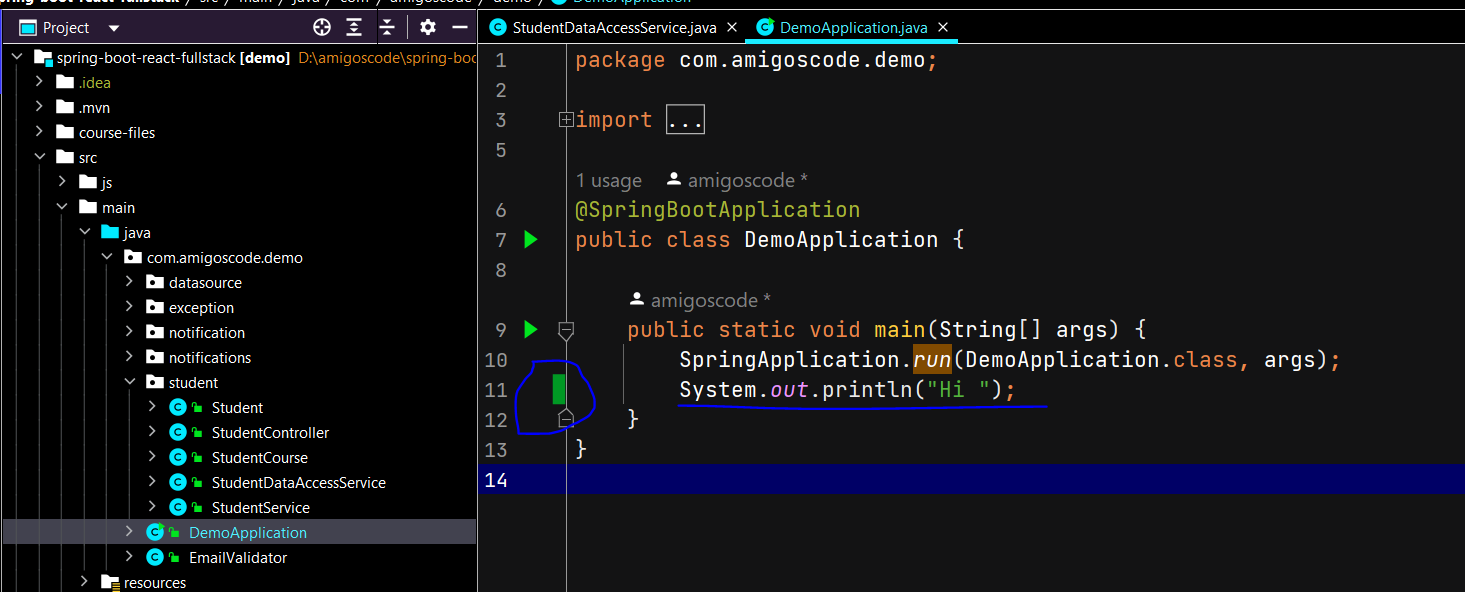
**Generating Code in Intellij Idea:**

1. Creating pojos of the model class: ctrl+insert or right click > generate

Git Integration: from Intellij

1. How to create a new branch
2. How to commit and push your changes

Note1- The newly created file will be shown in green and modified file will be shown in other colors and if we go inside the file then it will show the modified location in green color.



Note2- If we want to the person name committed this code then right click on the anywhere near the number and click on annotate and it will show the person name.

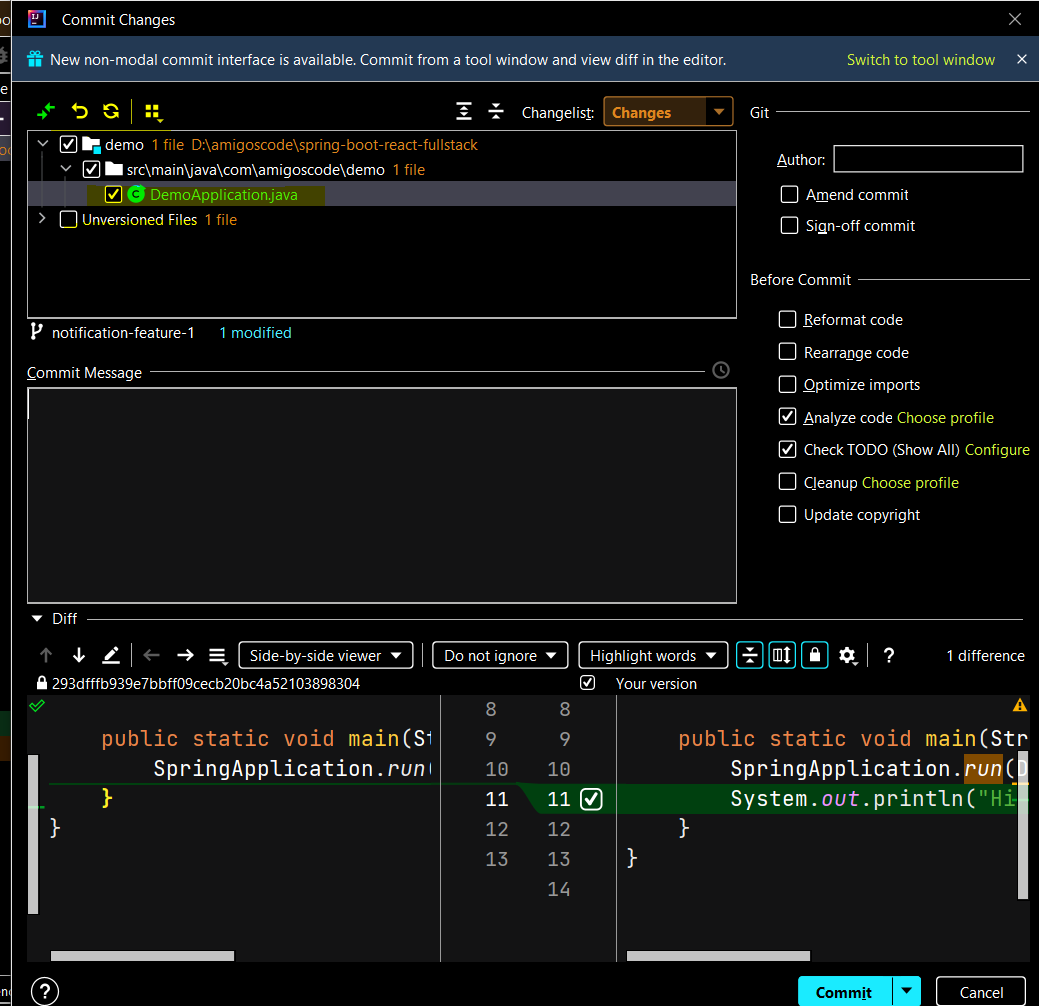
|  |  |
| --- | --- |
|  |  |

Use of Git or VCS menu in the Intellij:

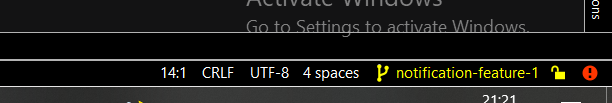
|  |  |
| --- | --- |
|  | 1. We can clone the project 2. We can create the new branch of the project from the current cloned branch (like Master branch) 3. We can push , pull the code 4. Use VCS operation for committing the code in the GIT |
|  | 1. We can commit the file 2. We can see the difference 3. We can see the history we can push the code into git 4. We can rollback the changes |

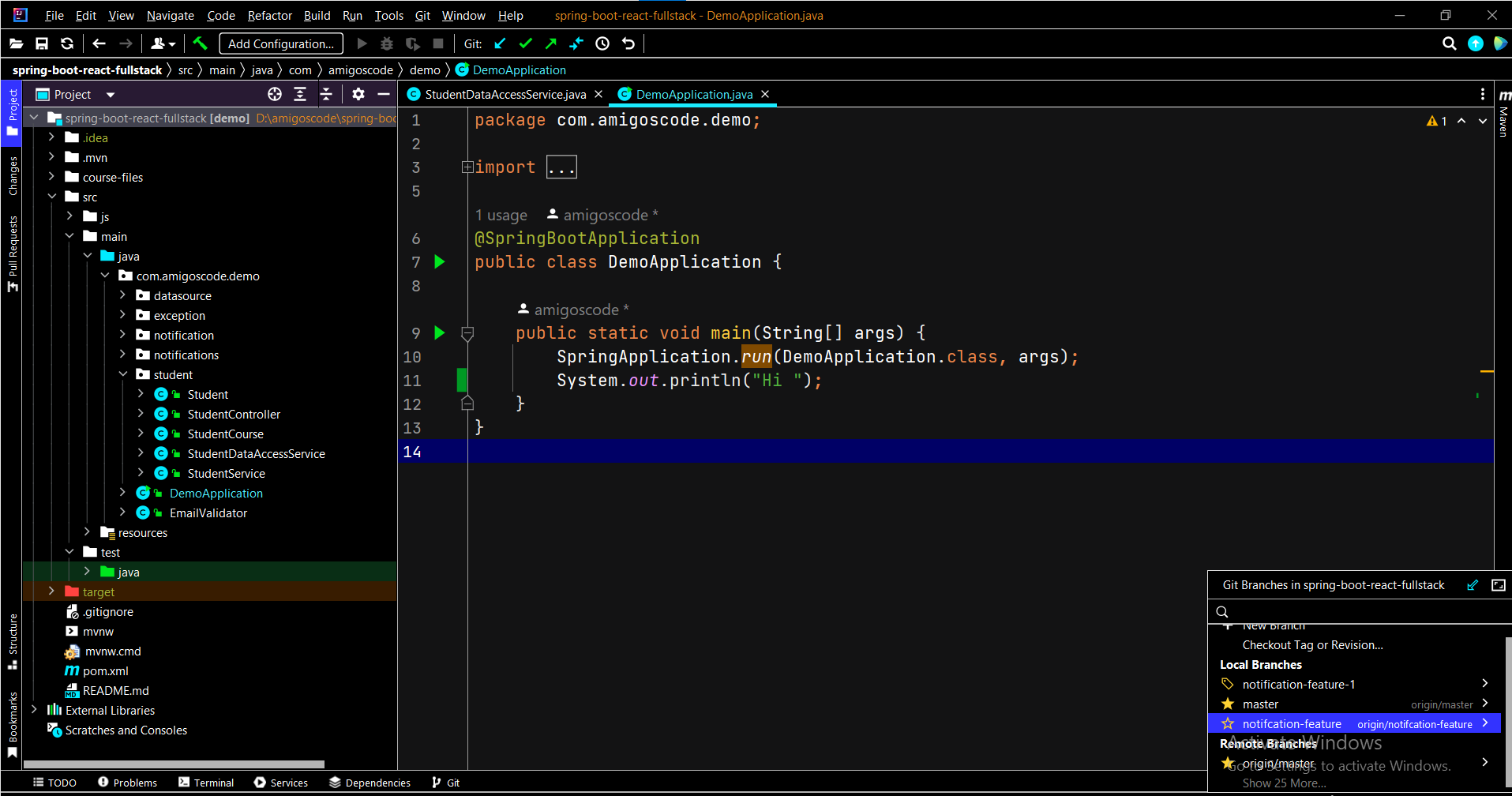
When we click on commit or commit file then we will get window to commit the code.

We can see the difference and can see the latest changes to commit.

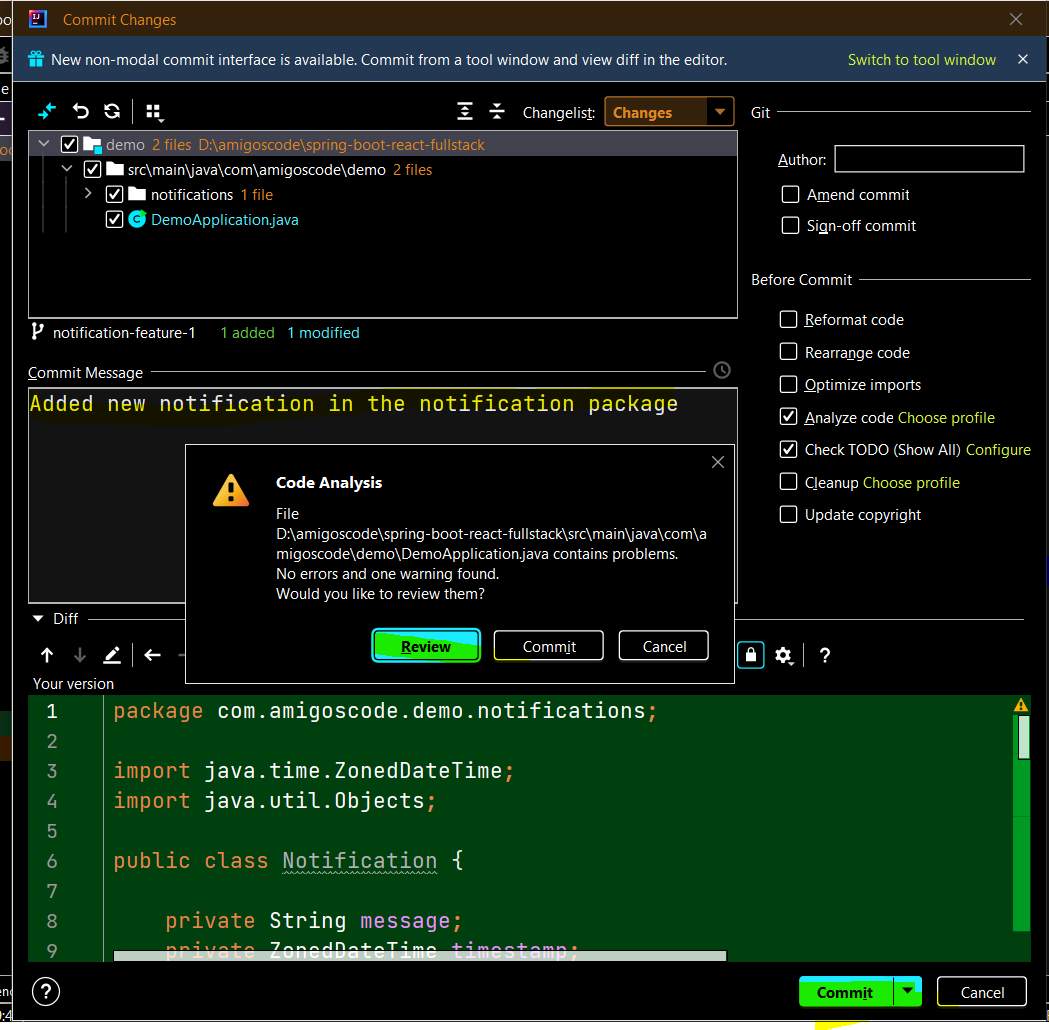


To so the code in the current branch just go to RHS bottom corner and we will see the branch name if we click then

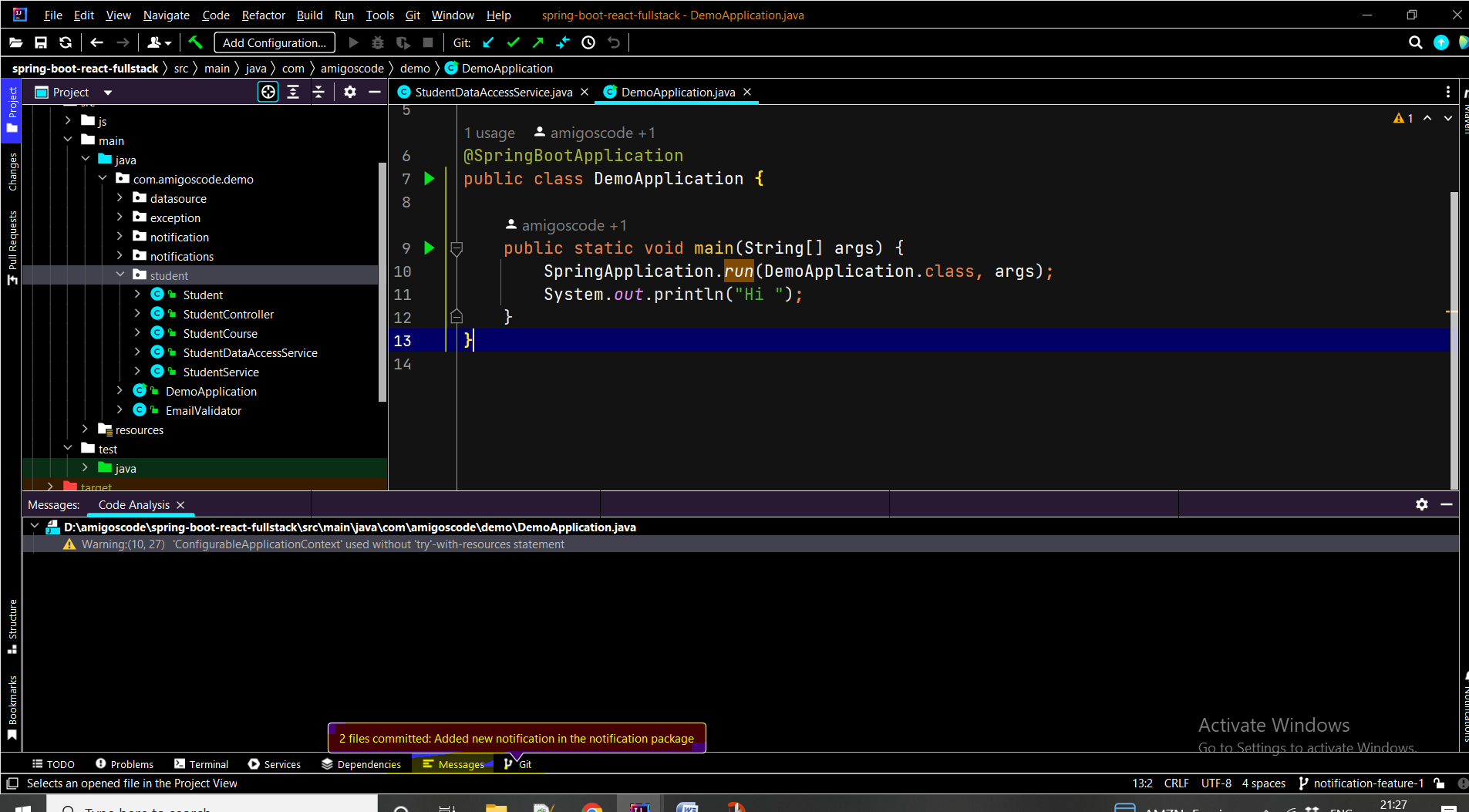




Here we can create the new feature branch from the current branch cloned in the Intellij and the latest changes we can commit in the newly created branch. Now let’s commit the code.



Once we click on commit then we will see the above popup. Click on commit again



And finally we committed the code in new branch with one popup message and now file looks normal like other existing. We can select commit and push from the commit drop down.

**If you have trouble downloading Hortonworks Data Platform...**

In the next lecture, we're going to install a complete "big data" framework - Hortonworks Data Platform - on your own desktop!

However, the download process on the Cloudera website seems to work unreliably at times. If you find you cannot download the HDP 2.6.5 sandbox image for Virtualbox as shown, try going directly to this page:

<https://www.cloudera.com/downloads/hortonworks-sandbox/hdp.html>

Choose "Virtualbox" for your installation type, provide the information they request, and then select HDP 2.6.5 from the "older versions."

If that also fails, you can try a direct link to their HDP 2.6.5 download:

<https://archive.cloudera.com/hwx-sandbox/hdp/hdp-2.6.5/HDP_2.6.5_virtualbox_180626.ova>

Or, if you need 2.5 instead:

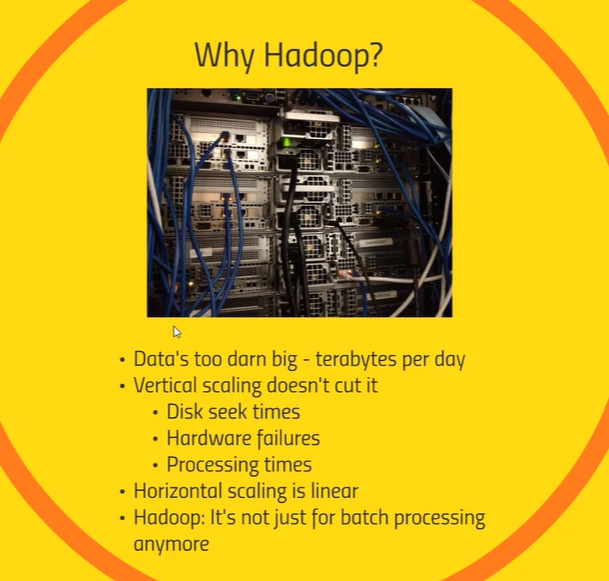
<https://archive.cloudera.com/hwx-sandbox/hdp/hdp-2.5.0/HDP_2.5_virtualbox.ova>

Remember, you need at least 8GB of free RAM on your system to run HDP. If you don't have that much memory, don't worry - you can still learn by just watching the videos.

Onward!

|  |  |
| --- | --- |
|  |  |





<https://www.youtube.com/watch?v=9SgS3TFCBzw>

**Section 4- Programming Hadoop with Spark**



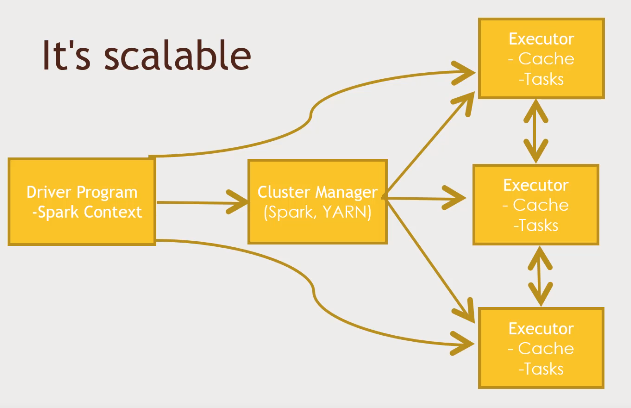
1. Spark is a memory based solution
2. We can write the programming using java, scala and python to do complex manipulation and transformations analysis of the data
3. PIG is build on the top of the spark framework

So Spark is parallel execution framework which enables us to perform parallel operation on big dataset in a highly parallel fashion. As we already know that Hadoop is a parallel execution framework which uses the MapReduce framework to crunch big dataset it’s a very revolutionary framework but it has some cons as well.

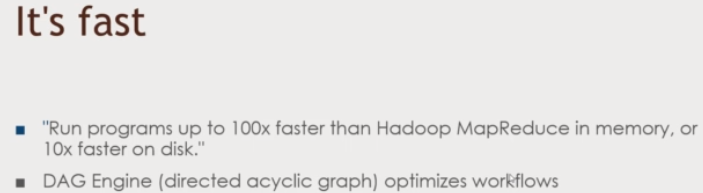
1. It’s a very rigid model, you first do a map and then reduce process which is powerful but may be not suitable for all the requirements
2. If you have complex requirements then you need to change together map-reduce job and once one map-reduce executes then the result have to sit in disk and have to reloaded to next map task and due to this it increase the cost of performance.

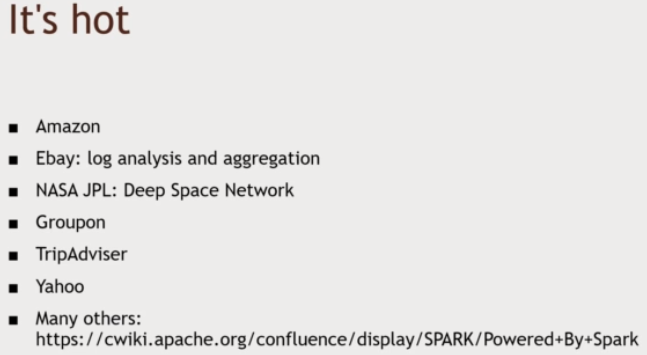
And that is why the spark framework says it is 100x time faster than Hadoop MapReduce in memory or 10x faster on the disk.

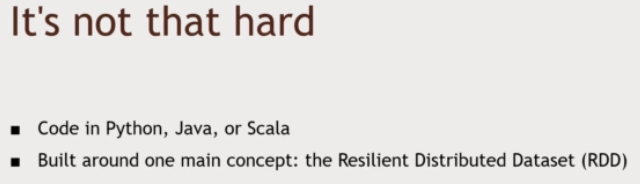
Spark instead of running one task after another, it actually builds an execution plan where we don’t need to care about its execution spark is smart enough to execute the tasks in parallel even if one task is dependent on other.



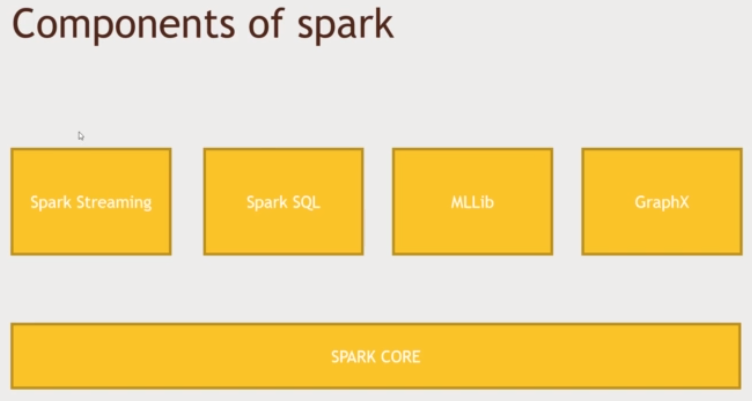
* Spark is just like other Hadoop based technology it is scalable
* Here we have one drive program in the spark which controls what is going to happen in your job
* Which intern goes to Cluster manager (Spark Yarn), it distributes your jobs to different computer commodities so that it could be processed in parallel
* Each Executor process will have some sort of Cache and Tasks

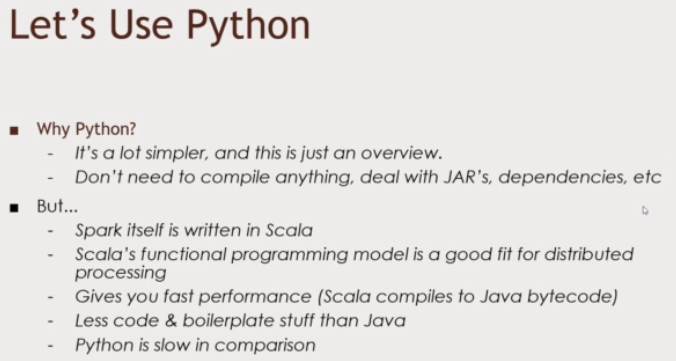


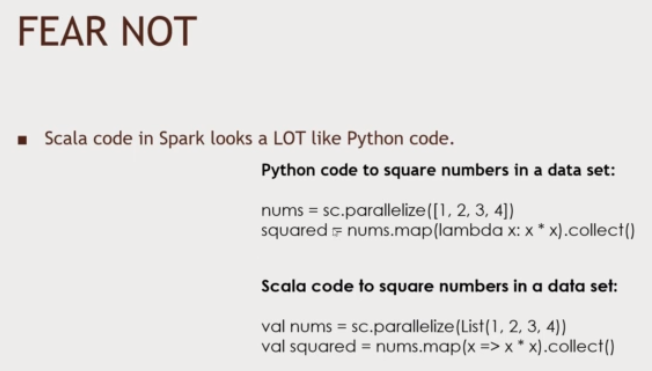




RDD is nothing but an object which represents the distributed dataset







**Note**: Here nums or val nums is an object which is nothing but RDD.

**The Resilient Distributed Dataset (RDD)**

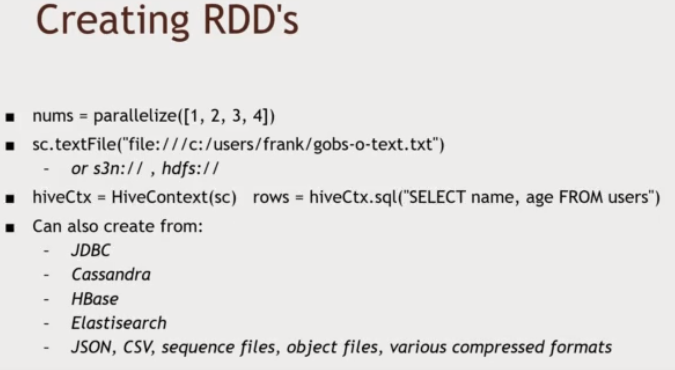
* Resilient
* Distritubuted
* Dataset

It is nothing the object of data set which store the object in the form of keys and values which can do the right thing on the cluster.

**How do we make the RDD? SparkContext:**



Basically our Driver program will create SparkContext environment which make the RDD run.



Here in the above, we have created RDD from different ways.

1. nums represents a RDD having list of numbers
2. Created a RDD from local file or Amazon or from hdfs, which in-turn return a line of text from the File to be processed further
3. We can created RDD from HiveContext
4. We can create RDD from any database like (JDBC, Casandra, HBase, Elasticsearch etc)

**MapReducer Concept in Spark ( Transformed RDD’s)**

|  |  |
| --- | --- |
|  |  |
| We can apply the below function on the RDD and can transform(Mapped) it according to our need and create a new RDD | For example here we have created on RDD (rdd) having list value 1,2,3,4 and we are transforming it using map function and getting new transformed (mapped) RDD (squaredRDD). Here each row of rdd is mapped and transformed to new squaredRDD. |

**MapReducer Concept (Collector RDD’s)**

|  |  |
| --- | --- |
|  |  |

**----------------------------------------------------------------------------------------------------------------------------------------**

**Warning - Java 9+ is not supported by Spark 2. You can optionally use Spark 3.**

Spark 3 was released in June 2020, and the only real difference (it's a big one!) between that and Spark 2 is that **Spark 3 supports Java 9+.**

On the course, you can choose:

**If you have Java 8 and are happy to use it:**

* Great! Just use the course files as provided. Everything should work just fine.

**If you don't have Java 8 but you're happy to work with Spark 2:**

* No problem - you can easily download a Java, use the instructions at the bottom of this page. You won't be missing out using Spark 2, the new features in 3 are very advanced and don't affect anything.

**If you need to use Java 9+:**

* Just download the updated pom.xml file (attached to this lecture) and use it instead. This simply has the spark dependencies bumped up to 3.x. Make sure you run eclipse:eclipse as described in the video, and you'll now be working with Spark 3.

If you are working with Spark 3 and a modern Java, you'll get the following warning:

1. WARNING: An illegal reflective access operation has occurred
2. WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/C:/Users/Richard/.m2/repository/org/apache/spark/spark-unsafe\_2.12/3.1.0/spark-unsafe\_2.12-3.1.0.jar) to constructor java.nio.DirectByteBuffer(long,int)
3. WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
4. WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
5. WARNING: All illegal access operations will be denied in a future release

**You can safely ignore this**(assuming that the rest of your code is running!). This is really a warning to the Spark devs for the future.

**How to Download a Java 8 JDK:**

At the time of writing, you can download the Oracle JDK from here: <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

However, Oracle are very likely to remove this link at some point in the future, so as a backup you can use the OpenJDK from here: https://adoptopenjdk.net/  (choose OpenJDK 8 LTS)

To install a JDK in Eclipse:

Window -> Preferences -> expand Java tab -> Installed JREs -> Add -> Standard VM -> Directory...

Navigate to where you installed the JDK (Default in windows for Oracle version is c:/Program Files/Java/jdk 1.8.xxx) -> OK

Click Finish.

(**important step**): Click the checkbox for the new JDK. Notice that the old JDK is now unticked automatically.

Once done you can apply the dialog box, and all should be well.

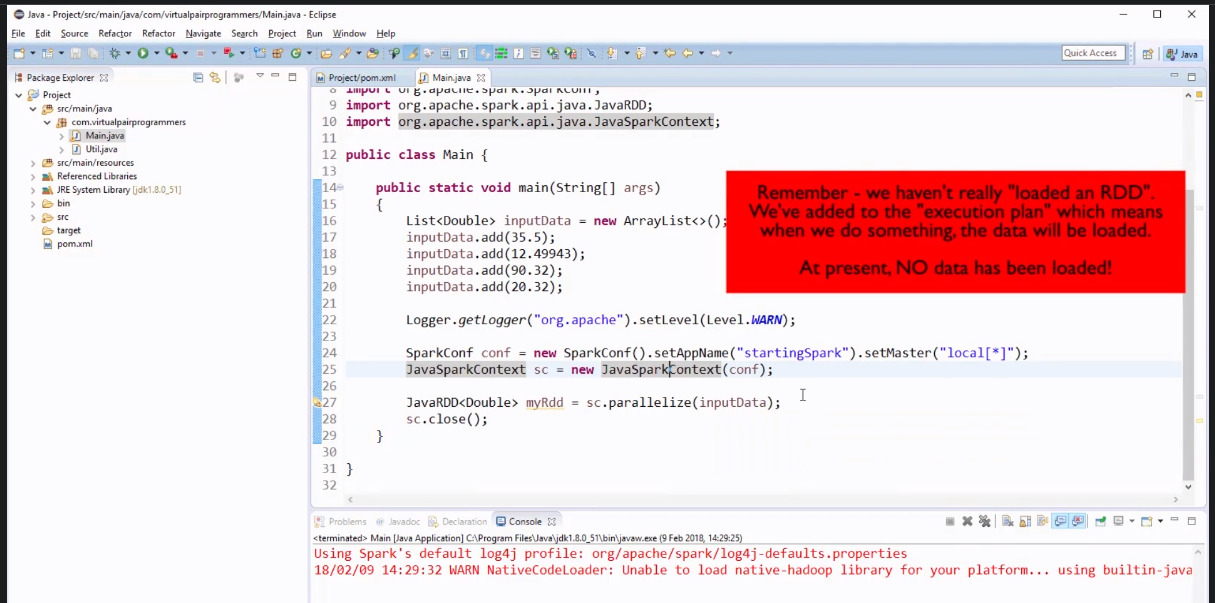
**Installing Spark 2.0 (Local dev env setup) Mapping on RDD**

* Spark 2.0 support java 8
* To start with spark we just need the below dependencies

|  |
| --- |
| **<dependencies>**  **<dependency>**  **<groupId>org.apache.spark</groupId>**  **<artifactId>spark-core\_2.10</artifactId>**  **<version>2.0.0</version>**  **</dependency>**  **<dependency>**  **<groupId>org.apache.spark</groupId>**  **<artifactId>spark-sql\_2.10</artifactId>**  **<version>2.0.0</version>**  **</dependency>**  **<dependency>**  **<groupId>org.apache.hadoop</groupId>**  **<artifactId>hadoop-hdfs</artifactId>**  **<version>2.2.0</version>**  **</dependency>**  **</dependencies>** |
| **package** com.virtualpairprogrammers;  **import** java.util.ArrayList;  **import** java.util.List;  **import** org.apache.spark.SparkConf;  **import** org.apache.spark.api.java.JavaRDD;  **import** org.apache.spark.api.java.JavaSparkContext;  **public** **class** Main {  **public** **static** **void** main(String[] args)  {    List<Double> inputData= **new** ArrayList<>();  inputData.add(35.5);  inputData.add(12.49);  inputData.add(90.42);  inputData.add(20.32);    System.***out***.println("Spark1111-----");  //1- First create the Spark config for all the cluster  SparkConf config= **new** SparkConf().setAppName("StartingSpark").setMaster("local[\*]");  // Here the above line local[\*] means run the spark program in all the available thread and if we write only local then spark program will run on a single thread  //2- Now create the spark context using the spark config which create a connection to our spark cluster  JavaSparkContext sc= **new** JavaSparkContext(config);  // JavaSparkContext- It represents a connection to our spark cluster which help our RDD to be distributed in all the cluster automatically.  //3- Now loads the file to create the RDD with the help of SparkContext, here we are loading the list instead of file in the cluster to create RDD  JavaRDD<Double> listRDD = sc.parallelize(inputData);// this parallelize() method take the collections and convert it into RDD i.e. it returns a RDD  // Note: Spark is implemented in Scala,so here we have written JavaRDD using scala RDD , i.e. this JavaRDD communicates with Scala RDD.  // i.e. JavaRDD is a wrapper  //4- Now after communications is finished we need to stop spark context  sc.close();    }  } |

**Now when we run the above program, then in the console**

|  |
| --- |
| Spark1111-----  Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties  22/04/19 22:10:25 INFO SparkContext: Running Spark version 2.0.0  22/04/19 22:10:26 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable  22/04/19 22:10:26 INFO SecurityManager: Changing view acls to: saraswati  22/04/19 22:10:26 INFO SecurityManager: Changing modify acls to: saraswati  22/04/19 22:10:26 INFO SecurityManager: Changing view acls groups to:  22/04/19 22:10:26 INFO SecurityManager: Changing modify acls groups to:  22/04/19 22:10:26 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(saraswati); groups with view permissions: Set(); users with modify permissions: Set(saraswati); groups with modify permissions: Set()  22/04/19 22:10:28 INFO Utils: Successfully started service 'sparkDriver' on port 61224.  22/04/19 22:10:28 INFO SparkEnv: Registering MapOutputTracker  22/04/19 22:10:28 INFO SparkEnv: Registering BlockManagerMaster  22/04/19 22:10:28 INFO DiskBlockManager: Created local directory at C:\Users\saraswati\AppData\Local\Temp\blockmgr-930a45ca-4df3-4c81-8328-504c246f50f7  22/04/19 22:10:28 INFO MemoryStore: MemoryStore started with capacity 896.4 MB  22/04/19 22:10:28 INFO SparkEnv: Registering OutputCommitCoordinator  22/04/19 22:10:28 INFO Utils: Successfully started service 'SparkUI' on port 4040.  22/04/19 22:10:28 INFO SparkUI: Bound SparkUI to 0.0.0.0, and started at http://192.168.56.1:4040  22/04/19 22:10:28 INFO Executor: Starting executor ID driver on host localhost  22/04/19 22:10:28 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 61245.  22/04/19 22:10:28 INFO NettyBlockTransferService: Server created on 192.168.56.1:61245  22/04/19 22:10:28 INFO BlockManagerMaster: Registering BlockManager BlockManagerId(driver, 192.168.56.1, 61245)  22/04/19 22:10:28 INFO BlockManagerMasterEndpoint: Registering block manager 192.168.56.1:61245 with 896.4 MB RAM, BlockManagerId(driver, 192.168.56.1, 61245)  22/04/19 22:10:28 INFO BlockManagerMaster: Registered BlockManager BlockManagerId(driver, 192.168.56.1, 61245)  22/04/19 22:10:29 INFO SparkUI: Stopped Spark web UI at http://192.168.56.1:4040  22/04/19 22:10:29 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!  22/04/19 22:10:29 INFO MemoryStore: MemoryStore cleared  22/04/19 22:10:29 INFO BlockManager: BlockManager stopped  22/04/19 22:10:29 INFO BlockManagerMaster: BlockManagerMaster stopped  22/04/19 22:10:29 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!  22/04/19 22:10:29 INFO SparkContext: Successfully stopped SparkContext  22/04/19 22:10:29 INFO ShutdownHookManager: Shutdown hook called  22/04/19 22:10:29 INFO ShutdownHookManager: Deleting directory C:\Users\saraswati\AppData\Local\Temp\spark-7f5833f9-5b8b-4006-afe3-8794542f4fd5 |
| **Note: We have just created the RDD but still not yet loaded** |

****

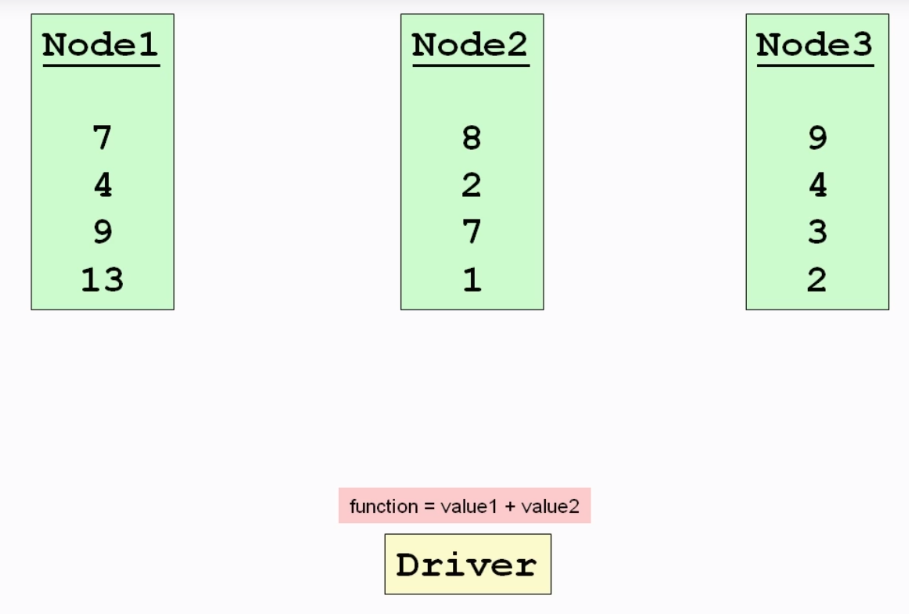
**Note:**  In the above program we have just mapped the RDD. Now we have Reduce the RDD.

**7. Reduces on RDDs**

1- Reduce will be used when we want perform the operation against the RDD , which going to transform a big data set (like here we have list of double value) into a single answer (result) i.e. using Reduce we are crunching down a big data into a single answer.

For example, let’s say we want to sum up the above list values then it means we are going to sum all the values reside in RDD.

2- In the normal java collection we can sum-up the above list values using the loop so it looks very easy, but as we know that we are going to work with the massive data-set, so our data of dataset could be stored in the node in the multiple clusters. As shown in the below screen shot



So now we can’t sum-up these list values using loop because these dataset is spread in the multiple physical computer (JVM).

1. So Now to sum-up these values spread in the multiple cluster, we will create one Driver program where we will write a method or function to add these values, this is nothing but called Reduce function. The Driver program is smart enough which will apply this function to across all the nodes

|  |  |
| --- | --- |
|  | |
|  |  |
|  |  |
|  | 33 |

|  |  |
| --- | --- |
|  | |
| So this way it will add first the values present at each nodes and once it complet sum-up then all the Reduced node values will be gathered together on a single node |  |

And again the same function will be applied on this node as shown above and we will get the final reduce result.



So we have understood that this RDD values could be spread on different physical computer or node.

So the sum-up these node value we will use reduce () function which itself will accept one function to sum-up these values.

|  |
| --- |
| **public** **static** **void** main(String[] args) {    List<Double> inputData= **new** ArrayList<>();  inputData.add(35.5);  inputData.add(12.49);  inputData.add(90.42);  inputData.add(20.32);    System.***out***.println("Spark1111-----");    //1- First create the Spark config for all the cluster  SparkConf config= **new** SparkConf().setAppName("StartingSpark").setMaster("local[\*]");  // Here the above line local[\*] means run the spark program in all the available thread and if we write only local then spark program will run on a single thread    //2- Now create the spark context using the spark config which create a connection to our spark cluster  JavaSparkContext sc= **new** JavaSparkContext(config);  // JavaSparkContext- It represents a connection to our spark cluster which help our RDD to be distributed in all the cluster automatically.    //3- Now loads the file to create the RDD with the help of SparkContext, here we are loading the list instead of file in the cluster to create RDD    JavaRDD<Double> listRDD = sc.parallelize(inputData);  // this parallelize() method take the collections and convert it into RDD i.e. it returns a RDD  // Note: Spark is implemented in Scala,so here we have written JavaRDD using scala RDD , i.e. this JavaRDD communicates with Scala RDD i.e. JavaRDD is a wrapper    ***reduce* = listRDD.reduce((val1,val2)-> val1+val2);**  **System.*out*.println("Sum of RDD:- "+*reduce*);**    //4- Now after communications is finished we need to stop spark context  sc.close();  }  } |

|  |
| --- |
| **Spark1111-----**  Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties  22/04/19 23:23:11 INFO SparkContext: Running Spark version 2.0.0  22/04/19 23:23:11 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable  22/04/19 23:23:12 INFO SecurityManager: Changing view acls to: saraswati  22/04/19 23:23:12 INFO SecurityManager: Changing modify acls to: saraswati  22/04/19 23:23:12 INFO SecurityManager: Changing view acls groups to:  22/04/19 23:23:12 INFO SecurityManager: Changing modify acls groups to:  22/04/19 23:23:12 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(saraswati); groups with view permissions: Set(); users with modify permissions: Set(saraswati); groups with modify permissions: Set()  22/04/19 23:23:12 INFO Utils: Successfully started service 'sparkDriver' on port 57463.  22/04/19 23:23:12 INFO SparkEnv: Registering MapOutputTracker  22/04/19 23:23:12 INFO SparkEnv: Registering BlockManagerMaster  22/04/19 23:23:12 INFO DiskBlockManager: Created local directory at C:\Users\saraswati\AppData\Local\Temp\blockmgr-0a184119-05c8-4953-bb53-17473f07e308  22/04/19 23:23:12 INFO MemoryStore: MemoryStore started with capacity 896.4 MB  22/04/19 23:23:13 INFO SparkEnv: Registering OutputCommitCoordinator  22/04/19 23:23:13 INFO Utils: Successfully started service 'SparkUI' on port 4040.  22/04/19 23:23:13 INFO SparkUI: Bound SparkUI to 0.0.0.0, and started at http://192.168.56.1:4040  22/04/19 23:23:13 INFO Executor: Starting executor ID driver on host localhost  22/04/19 23:23:13 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 57484.  22/04/19 23:23:13 INFO NettyBlockTransferService: Server created on 192.168.56.1:57484  22/04/19 23:23:13 INFO BlockManagerMaster: Registering BlockManager BlockManagerId(driver, 192.168.56.1, 57484)  22/04/19 23:23:13 INFO BlockManagerMasterEndpoint: Registering block manager 192.168.56.1:57484 with 896.4 MB RAM, BlockManagerId(driver, 192.168.56.1, 57484)  22/04/19 23:23:13 INFO BlockManagerMaster: Registered BlockManager BlockManagerId(driver, 192.168.56.1, 57484)  22/04/19 23:23:14 INFO SparkContext: Starting job: reduce at Main.java:37  22/04/19 23:23:14 INFO DAGScheduler: Got job 0 (reduce at Main.java:37) with 4 output partitions  22/04/19 23:23:14 INFO DAGScheduler: Final stage: ResultStage 0 (reduce at Main.java:37)  22/04/19 23:23:14 INFO DAGScheduler: Parents of final stage: List()  22/04/19 23:23:14 INFO DAGScheduler: Missing parents: List()  22/04/19 23:23:14 INFO DAGScheduler: Submitting ResultStage 0 (ParallelCollectionRDD[0] at parallelize at Main.java:33), which has no missing parents  22/04/19 23:23:15 INFO MemoryStore: Block broadcast\_0 stored as values in memory (estimated size 2.2 KB, free 896.4 MB)  22/04/19 23:23:15 INFO MemoryStore: Block broadcast\_0\_piece0 stored as bytes in memory (estimated size 1432.0 B, free 896.4 MB)  22/04/19 23:23:15 INFO BlockManagerInfo: Added broadcast\_0\_piece0 in memory on 192.168.56.1:57484 (size: 1432.0 B, free: 896.4 MB)  22/04/19 23:23:15 INFO SparkContext: Created broadcast 0 from broadcast at DAGScheduler.scala:1012  22/04/19 23:23:15 INFO DAGScheduler: Submitting 4 missing tasks from ResultStage 0 (ParallelCollectionRDD[0] at parallelize at Main.java:33)  22/04/19 23:23:15 INFO TaskSchedulerImpl: Adding task set 0.0 with 4 tasks  22/04/19 23:23:15 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, localhost, partition 0, PROCESS\_LOCAL, 5378 bytes)  22/04/19 23:23:15 INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1, localhost, partition 1, PROCESS\_LOCAL, 5378 bytes)  22/04/19 23:23:15 INFO TaskSetManager: Starting task 2.0 in stage 0.0 (TID 2, localhost, partition 2, PROCESS\_LOCAL, 5378 bytes)  22/04/19 23:23:15 INFO TaskSetManager: Starting task 3.0 in stage 0.0 (TID 3, localhost, partition 3, PROCESS\_LOCAL, 5378 bytes)  22/04/19 23:23:15 INFO Executor: Running task 3.0 in stage 0.0 (TID 3)  22/04/19 23:23:15 INFO Executor: Running task 2.0 in stage 0.0 (TID 2)  22/04/19 23:23:15 INFO Executor: Running task 0.0 in stage 0.0 (TID 0)  22/04/19 23:23:15 INFO Executor: Running task 1.0 in stage 0.0 (TID 1)  22/04/19 23:23:15 INFO Executor: Finished task 1.0 in stage 0.0 (TID 1). 875 bytes result sent to driver  22/04/19 23:23:15 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 875 bytes result sent to driver  22/04/19 23:23:15 INFO Executor: Finished task 2.0 in stage 0.0 (TID 2). 875 bytes result sent to driver  22/04/19 23:23:15 INFO Executor: Finished task 3.0 in stage 0.0 (TID 3). 875 bytes result sent to driver  22/04/19 23:23:15 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 282 ms on localhost (1/4)  22/04/19 23:23:15 INFO TaskSetManager: Finished task 2.0 in stage 0.0 (TID 2) in 172 ms on localhost (2/4)  22/04/19 23:23:15 INFO TaskSetManager: Finished task 3.0 in stage 0.0 (TID 3) in 172 ms on localhost (3/4)  22/04/19 23:23:15 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1) in 188 ms on localhost (4/4)  22/04/19 23:23:15 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool  22/04/19 23:23:15 INFO DAGScheduler: ResultStage 0 (reduce at Main.java:37) finished in 0.359 s  22/04/19 23:23:15 INFO DAGScheduler: Job 0 finished: reduce at Main.java:37, took 1.443679 s  **Sum of RDD:- 158.73000000000002**  22/04/19 23:23:15 INFO SparkUI: Stopped Spark web UI at http://192.168.56.1:4040  22/04/19 23:23:16 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!  22/04/19 23:23:16 INFO MemoryStore: MemoryStore cleared  22/04/19 23:23:16 INFO BlockManager: BlockManager stopped  22/04/19 23:23:16 INFO BlockManagerMaster: BlockManagerMaster stopped  22/04/19 23:23:16 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!  22/04/19 23:23:16 INFO SparkContext: Successfully stopped SparkContext  22/04/19 23:23:16 INFO ShutdownHookManager: Shutdown hook called  22/04/19 23:23:16 INFO ShutdownHookManager: Deleting directory C:\Users\saraswati\AppData\Local\Temp\spark-de1f4a08-35b7-4414-87c2-8f2cc672eed4 |

**8. Mapping Operations**

We have seen Reduce operation, now let’s look at Mapping operation. The Map method on RDD allows us to transform the structure of RDD from one form to another form.

|  |  |
| --- | --- |
|  |  |
| Let’s say we have a RDD with massive set of values and we want to transform those RDD values, for example below we have a set of RDD values and we need to find out square root.  As here we can see one function defined in the driver will be applied to all the RDD values. | Note: A RDD is always immutable, once we create we cannot change, but we can create new RDD using the previous RDD as show above in the screen shot. |

|  |
| --- |
| **package** com.virtualpairprogrammers;  **public** **class** MapRDD{  **public** **static** **void** main(String[] args) {    List<Integer> listData = **new** ArrayList<>();  listData.add(4);  listData.add(9);  listData.add(16);  listData.add(25);    Logger.*getLogger*("org.apache").setLevel(Level.***WARN***);    SparkConf config = **new** SparkConf().setAppName("SpartMap").setMaster("local[\*]");  JavaSparkContext sc = **new** JavaSparkContext(config);    JavaRDD<Integer> listRDD = sc.parallelize(listData);  // Reduce function  Integer sumRDD = listRDD.reduce((value1, value2) -> value1+value2);  System.***out***.println("Reduced new RDD::"+ sumRDD);    JavaRDD<Double> sqrtMapRDD = listRDD.map(value-> Math.*sqrt*(value));    sqrtMapRDD.foreach(value-> System.***out***.println(value));  Double reducedNewRDD = sqrtMapRDD.reduce((v1,v2)-> v1+v2);  System.***out***.println("sum of sqrt - reducedNewRDD:::"+reducedNewRDD);  }  } |
| **Output:**  [Stage 0:> (0 + 0) / 4]  Reduced new RDD::54  5.0  3.0  2.0  4.0  sum of sqrt - reducedNewRDD:::14.0 |

**10. Counting Big Data Items**

Q: Let’s count how element are there in sqrtMapRDD?

A: One simple answer is we can use count () method like sqrtMapRDD and it will return the number of value present in it.

|  |
| --- |
| System.***out***.println("Number of element in the RDD: "+sqrtMapRDD.count()); |

Q: But it’s a very simple example to it looks easy to count number of the element in the RDD, what if we have very big data set and sometime if we want to do interim count then how to do it?

A: In such cases we will have to do further mapping on RDD using map and reduce.

|  |  |  |
| --- | --- | --- |
|  |  |  |

So as above we can see that we have sqrtRDD and to count the element inside this RDD first we have mapped by 1 in each element of RDD and then reduced it using another function.

|  |
| --- |
| **package** com.virtualpairprogrammers;  **public** **class** MapRDD{  **public** **static** **void** main(String[] args) {    List<Integer> listData = **new** ArrayList<>();  listData.add(4);  listData.add(9);  listData.add(16);  listData.add(25);    Logger.*getLogger*("org.apache").setLevel(Level.***WARN***);    SparkConf config = **new** SparkConf().setAppName("SpartMap").setMaster("local[\*]");  JavaSparkContext sc = **new** JavaSparkContext(config);    JavaRDD<Integer> listRDD = sc.parallelize(listData);  // Reduce function  Integer sumRDD = listRDD.reduce((value1, value2) -> value1+value2);  System.***out***.println("Reduced new RDD::"+ sumRDD);    JavaRDD<Double> sqrtMapRDD = listRDD.map(value-> Math.*sqrt*(value));  sqrtMapRDD.foreach(value-> System.***out***.println(value));    Double reducedNewRDD = sqrtMapRDD.reduce((v1,v2)-> v1+v2);  System.***out***.println("sum of sqrt - reducedNewRDD:::"+reducedNewRDD);    System.***out***.println("Number of element in the RDD simple java method: "+sqrtMapRDD.count());  // map value 1 each sqrtRDD  JavaRDD<Integer> singleReducedRDD = sqrtMapRDD.map(value ->1);  // Now adding or reducing this using reduce  Integer reducedCountRDD = singleReducedRDD.reduce((v1,v2)-> v1+v2);    System.***out***.println("Number of element using MapReduce method::"+ reducedCountRDD);  }  } |

|  |
| --- |
| Output:  Reduced new RDD::54  3.0  2.0  5.0  4.0  sum of sqrt - reducedNewRDD:::14.0  Number of element in the RDD simple java method: 4  Number of element using MapReduce method::4 |

So here the Spark first will map 1 against each element of RDD and then finally it will add all the 1s, in this way using map-reduce we can count the number of element of RDD.

**11. If you've had a "NotSerializableException" in Spark**

**Note: in above we have used below line to print each element of RDD.**

|  |
| --- |
| **sqrtMapRDD.foreach(value-> System.*out*.println(value));** |

Here if we use the below line of code then we will get NotSerializableException ;

|  |
| --- |
| sqrtMapRDD.foreach(System.***out***::println); |

**Reason**: Actually this println method specifically works at the same JVM but in as we know that the RDD is spread across different physical computer or node who has different JVM and in this case the above (System.***out***::println) gets failed. To resolve this exception we will use the below line of code.

|  |
| --- |
| sqrtMapRDD.collect().forEach(System.***out***::println); |

So here first we are converting RDD into collection (List) using collect() and now in this case all the spread element of RDD on different nodes or computer are gets collected on one physical computer (JVM) and then in this case we will not get any exception.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Section5: Tuples:**

As in the above example we have two RDD (1- OriginalListRDD and 2- sqrtMapRDD). It is of no use to keep these two data set separately. So if we want to group these two dataset i.e. we want to keep these two dataset in one single RDD

|  |  |
| --- | --- |
|  |  |

Now the above Single RDD is looks like java map or Database table. This second table is more useful to further anywhere because all the dataset is available in one RDD.

Q: Now question comes, how we can do that?

A: Option 1:

* 1. In this option we will create on class having these to variable (originalListRDD and sqrtMapRDD).
  2. And then enclose these two variable within the class and hence we will get a single RDD having these two RDD

|  |
| --- |
| **public** **static** **void** main(String[] args) {    List<Integer> listData = **new** ArrayList<>();  listData.add(4);  listData.add(9);  listData.add(16);  listData.add(25);    Logger.*getLogger*("org.apache").setLevel(Level.***WARN***);  //Creating spark config and spark context  SparkConf config = **new** SparkConf().setAppName("SpartMap").setMaster("local[\*]");  JavaSparkContext sc = **new** JavaSparkContext(config);    // Creating RDD  JavaRDD<Integer> originalsListRDD = sc.parallelize(listData);  // Calculating sqrt or transforming or mapping RDD into another RDD using map() method  JavaRDD<Double> sqrtMapRDD = originalsListRDD.map(value-> Math.*sqrt*(value));    // So here now we have two RDD, 1- OriginalListRDD and 2- sqrtMapRDD. So let comment the above line and create the square root as given below  JavaRDD<IntegerWithSqrt> tupleRDD = originalsListRDD.map(value->(**new** IntegerWithSqrt(value)));  //tupleRDD.collect().forEach(System.out::println);;  tupleRDD.foreach(value->System.***out***.println(value));  } |
| **package** com.virtualpairprogrammers;  **public** **class** IntegerWithSqrt {  **private** **int** originalsListRDD ;  **private** **double** squareRoot;    **public** IntegerWithSqrt(**int** originalsListRDD) {  **super**();  **this**.originalsListRDD = originalsListRDD;  **this**.squareRoot = Math.*sqrt*(originalsListRDD);  }  } |

**Note: The above option is very complex to make a single RDD.**

**Option2- Using Scala concept called Tuples:**

|  |  |
| --- | --- |
|  |  |

* 1. Scala Tuples is nothing but the collection of values. For example (64, 8) is a two element tuple and tuple can have many items inside it and it can be any type as well.
  2. (“cat”, “Goose”, “dog) is a three element tuple
  3. We can have the object inside the tuple e.g. (object1, object2, object3, object4, object5)

**In the above screen shot the first image Is nothing but the different example of Tuples where as in the second box its show how we can declare the Scala Tuples. It is very simple to declare. For example the first one denotes the Tuple of coordinates and second denotes the Tuples of animals**

**Q: The above declaration works in scala but how to declare or create Tuples in java**

**A:**

|  |
| --- |
| **import** scala.Tuple2;  **import** scala.Tuple3;  **import** scala.Tuple4;  **Tuple2<Integer, Double> twoEleTouple = new Tuple2<Integer, Double>(9,3.0);**  **Tuple3<Integer, Integer, Integer> threeEleTouple= new Tuple3<Integer, Integer, Integer>(1,2,3);**  **Tuple4<Integer, Integer, Integer,Integer> fourEleTouple= new Tuple4<Integer, Integer, Integer,Integer>(1,2,3,4);** |

**And so on according the number of element we will import the Tuples and create the Tuples accordingly.**

**The above example using the scala Tuple can be written as given below.**

|  |
| --- |
| **import** scala.Tuple2;  **import** scala.Tuple3;  **import** scala.Tuple4;  **import** scala.Tuple8;  **import** scala.Tuple9;  **import** scala.Tuple10;  **public** **class** Tuples {  **public** **static** **void** main(String[] args) {    List<Integer> listData = **new** ArrayList<>();  listData.add(4);  listData.add(9);  listData.add(16);  listData.add(25);    Logger.*getLogger*("org.apache").setLevel(Level.***WARN***);  //Creating spark config and spark context  SparkConf config = **new** SparkConf().setAppName("SpartMap").setMaster("local[\*]");  JavaSparkContext sc = **new** JavaSparkContext(config);    // Creating RDD  JavaRDD<Integer> originalsListRDD = sc.parallelize(listData);  // Calculating sqrt or transforming or mapping RDD into another RDD using map() method  //JavaRDD<Double> sqrtMapRDD = originalsListRDD.map(value-> Math.sqrt(value));    // So here now we have two RDD, 1- OriginalListRDD and 2- sqrtMapRDD. So let comment the above line and create the square root as given below  //JavaRDD<IntegerWithSqrt> tupleRDD = originalsListRDD.map(value->(new IntegerWithSqrt(value)));  // Tuple2<Integer, Double> twoEleTouple = new Tuple2<Integer, Double>(9,3.0);  //Tuple3<Integer, Integer, Integer> threeEleTouple= new Tuple3<Integer, Integer, Integer>(1,2,3);  // Tuple4<Integer, Integer, Integer,Integer> fourEleTouple= new Tuple4<Integer, Integer, Integer,Integer>(1,2,3,4);  // Tuple8<Integer, Integer, Integer,Integer,Integer, Integer, Integer,Integer> eightleTouple= new Tuple8<Integer, Integer, Integer,Integer,Integer, Integer, Integer,Integer>(1,2,3,4,1,2,3,4);    JavaRDD<Tuple2<Integer, Double>> map = originalsListRDD.map(value-> **new** Tuple2<Integer, Double>(value,Math.*sqrt*(value)));    } |
| **In- a clear view without any comments** |
| **public** **static** **void** main(String[] args) {    List<Integer> listData = **new** ArrayList<>();  listData.add(4);  listData.add(9);  listData.add(16);  listData.add(25);  Logger.*getLogger*("org.apache").setLevel(Level.***WARN***);  SparkConf config = **new** SparkConf().setAppName("SpartMap").setMaster("local[\*]");  JavaSparkContext sc = **new** JavaSparkContext(config);  // Creating RDD  JavaRDD<Integer> originalsListRDD = sc.parallelize(listData);    **JavaRDD<Tuple2<Integer, Double>> map = originalsListRDD.map(value-> new Tuple2<Integer, Double>(value,Math.*sqrt*(value)));**    } |

**So this is the way we can create the Scala Tuples in java:**

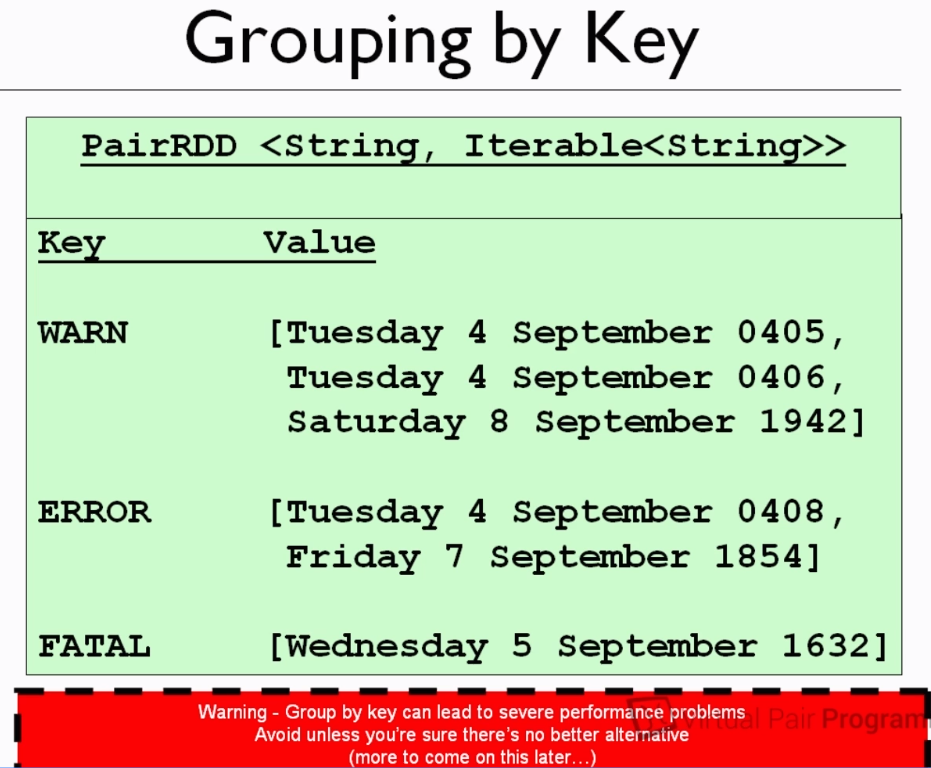
**Overview of PairRDDs**

* 1. Generally in most of the requirement we have two RDD, that is why we have the concept of PairRDD.
  2. These are stored in the form of Key and value, this concept gives an extra method in RDD which allows us to operation like grouping by Key.

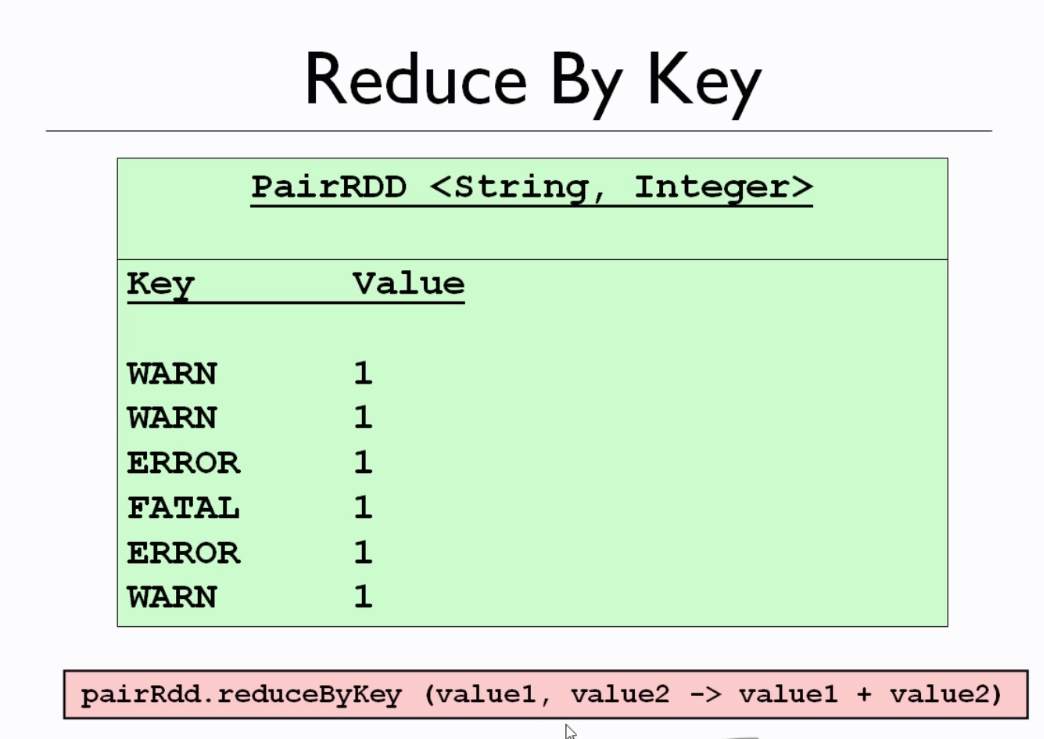
For example processing log file in the server.

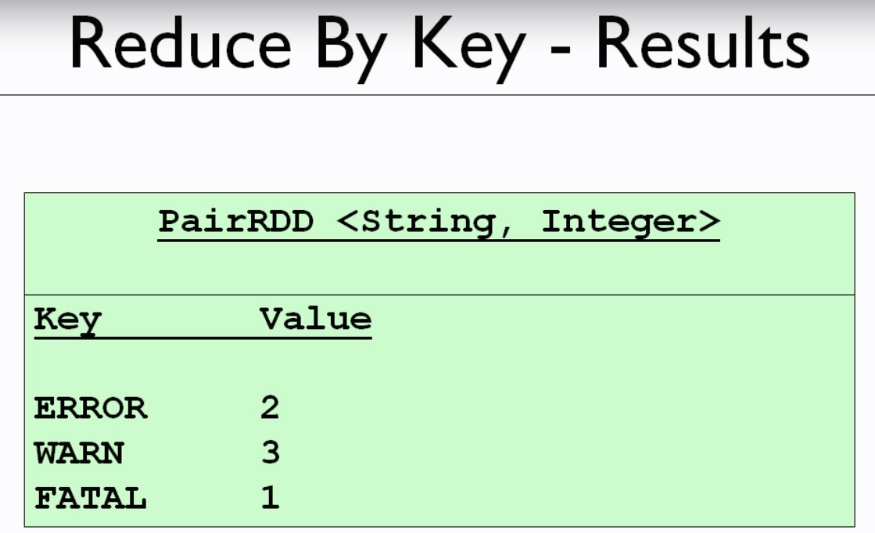
|  |
| --- |
|  |
| Here we have different kind of log level: Date and time stamp. Here we are going to cont the number of warning, errors and fatel error. So first of all we are goining to convert these into RDD. Here we are goinig to convert the above log level and its value into PairRDD where the log level will be as a key and its value will be as value as we can see in the blow image. |
|  |
| Here in the above image we can see that we have a pair RDD in the form of key and value pair just like a table having two colums. This PairRDD contains several methods which can be orperate on these keys. |

**Building a PairRDD:**



So as we can see that when we apply Grouping By Key then all the log level will be grouped by its key and its all value will be stored in the form of collection, now it is very to count the number of log level. But this leads to performance problems and avoid unless we don’t have any alternate solutions.





So here in the another alternative we are going to use Reduce By Key method and mapping one (1L) against each key (loglevel) and finally we can calculate the values.

**Coding a ReduceByKey**

Step1- created a list and added log-level and datetime

Step2- created spark config object

Step3- Created SparkContext object using spark config object

Step4- using SparkContext perallelize() convert the list into single RDD

Step5- Now mapped this single RDD into key-value paired RDD using mapToPair() it returns the tuple

Step6- Mapped one long value(1L) against each key(log-level)

Step7- Add these values using reducedByKey

Step8- finally print the count of each log level

|  |
| --- |
| package com.virtualpairprogrammers.pairedrdd**;** import org.apache.log4j.Level**;** import org.apache.log4j.Logger**;** import org.apache.spark.SparkConf**;** import org.apache.spark.api.java.JavaPairRDD**;** import org.apache.spark.api.java.JavaRDD**;** import org.apache.spark.api.java.JavaSparkContext**;** import scala.Tuple2**;** import java.util.ArrayList**;** import java.util.List**;** public class DemoPairedRDD {  public static void main(String[] args) {  List<String> logList= new ArrayList<>()**;** logList.add("WARN: Tuesday 4 September 0405")**;** logList.add("ERROR: Tuesday 4 September 0408")**;** logList.add("FATAL: Wednesday 5 September 1632")**;** logList.add("ERROR: Friday 7 September 1854")**;** logList.add("WARN: Saturday 8 September 1942")**;** Logger.*getLogger*("org.apache").setLevel(Level.*WARN*)**;**  SparkConf sconfig = new SparkConf().setAppName("StartingSpark").setMaster("local[\*]")**;** JavaSparkContext sparkContext = new JavaSparkContext(sconfig)**;** // Now let's create RDD which is nothing but will be RowRDD (single)  JavaRDD<String> originalLogRDD = sparkContext.parallelize(logList)**;** // Now let's transform the above single RDD using Map and Reduce and we will transform  // in the structure in which we want(paird RDD) // In the previous program we have used Map to transform the RDD but here we want to transform  // this into two column RDD so Spark provide one mehot mapToPair()  JavaPairRDD<String**,** Long> objectObjectJavaPairRDD = originalLogRDD.mapToPair(rawValue ->  {  String[] columns = rawValue.split(":")**;** String loglevl = columns[**0**]**;** String dateTimeStamp = columns[**1**]**;** return new Tuple2<>(loglevl**,1L**)**;** })**;** JavaPairRDD<String**,** Long> sumRDD = objectObjectJavaPairRDD.reduceByKey((value1**,** value2) ->  value1 + value2)**;** // Adding the value based on its key and it will return a tuple  sumRDD.foreach(tuple->System.*out*.println(tuple.\_1 +" has "+ tuple.\_2+"instance"))**;** sparkContext.close()**;** } } |

Output:

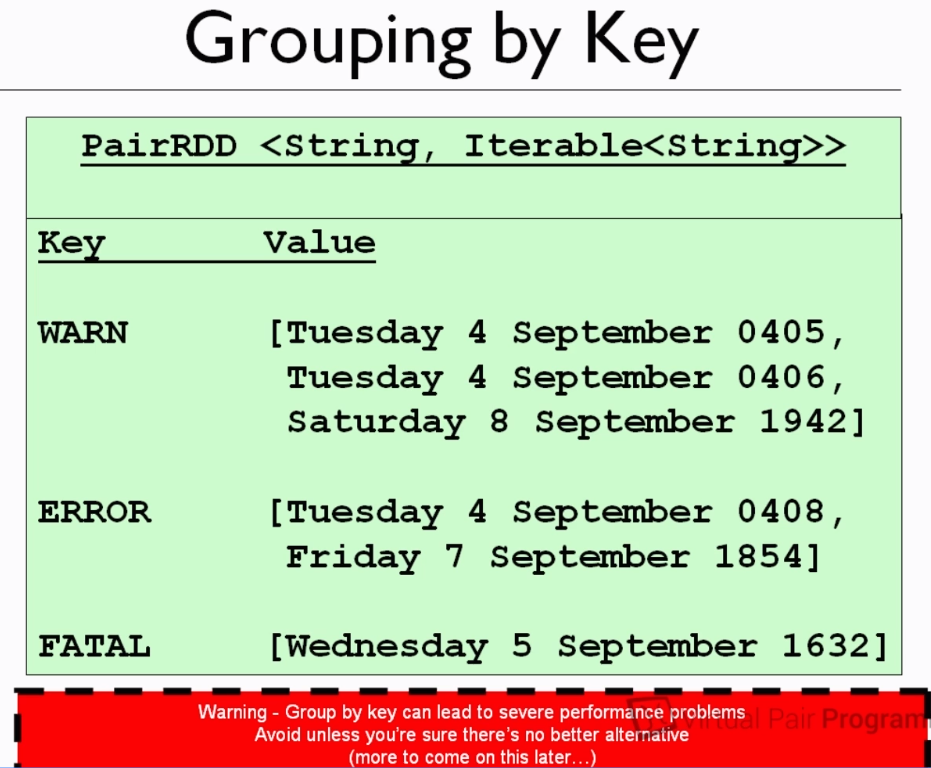
|  |
| --- |
| WARN has 2instance  ERROR has 2instance  FATAL has 1instance |

**17. Using the Fluent API**

The above line of code can be written in another way i.e. in a fluent way as given below

|  |
| --- |
| **sparkContext.parallelize(logList)// Return Single RDD  .mapToPair(rawValue-> new Tuple2<>(rawValue.split(":")[0],1L))// Retruns pairRDD**  **.reduceByKey((value1,value2)->value1+value2)**  .**foreach(tuple->System.*out*.println(tuple.\_1+" has "+ tuple.\_2+"instance"));** |

**18. Grouping By Key**

****

|  |
| --- |
| package com.virtualpairprogrammers.pairedrdd**;** import com.google.common.collect.Iterables**;** public class UsingGroupByKey { public static void main(String[] args) { List<String> logList = new ArrayList<>()**;** logList.add("WARN: Tuesday 4 September 0405")**;** logList.add("ERROR: Tuesday 4 September 0408")**;** logList.add("FATAL: Wednesday 5 September 1632")**;** logList.add("ERROR: Friday 7 September 1854")**;** logList.add("WARN: Saturday 8 September 1942")**;** Logger.*getLogger*("org.apache").setLevel(Level.*WARN*)**;** SparkConf sconfig = new SparkConf().setAppName("StartingSpark").setMaster("local[\*]")**;** JavaSparkContext sparkContext = new JavaSparkContext(sconfig)**;** System.*out*.println("another way in groupByKey way ")**;** sparkContext.parallelize(logList)  **.mapToPair(rawValue ->new Tuple2<>(rawValue.split(":")[0],1L))  .groupByKey()  .foreach(tuple->System.*out*.println(tuple.\_1 +" has "+ Iterables.*size*(tuple.\_2)));** } } |

So here the value is noting but the list of dateandtime values inside the paired tuple and here we have used Iterables.size(tuple.\_2) to get the side of each list against its key.

**Section-7 FlatMap and Filters:**

|  |  |
| --- | --- |
|  | As in the previous steps we already have seen that how we have used map() function to convert the RDD from one form to another form. For example in the previous example we using the sqrt() function we have created another RDD inside the map() function.  But here we have some restriction like; there must be one output value for every single input value.  But in case if want to have multiple output value(Or No output) on one single input value, then in that case FlatMap come in the picture. |
|  | Let’s we have an input RDD which contains a series of strings, here the first line contains the sentences and second line contains gap which will considered as an empty string and in the final line we have strings which contains three words.  Now the requirement is split this series of strings so that it could have one single word. So here Map() function would not work here because here we can see that in the first line we have four word to be transformed into new RDD. |
|  | So here we are going to apply some function which will transform into single string.  Similarly we will apply the same function on the empty line and it will return empty (i.e. will not return any string as output).  Again the same function will be applied on the third line and it will generate three single strings as shown in the image on LHS. |

So the above job will be done by FlatMap function. So here we are taking as single input value and it will generate multiple or no value.

|  |
| --- |
| public class UsingFlatMap { public static void main(String[] args) {  List<String> logList= new ArrayList<>();  logList.add("WARN: Tuesday 4 September 0405"); logList.add("ERROR: Tuesday 4 September 0408"); logList.add("FATAL: Wednesday 5 September 1632"); logList.add("ERROR: Friday 7 September 1854"); logList.add("WARN: Saturday 8 September 1942"); Logger.getLogger("org.apache").setLevel(Level.WARN); // SparkConf conf = new SparkConf().setAppName("Flatmap").setMaster("Local[\*]"); // JavaSparkContext sparkContext = new JavaSparkContext(conf); SparkConf sconfig = new SparkConf().setAppName("StartingSpark").setMaster("local[\*]"); JavaSparkContext sparkContext = new JavaSparkContext(sconfig); JavaRDD<String> originalRDD = sparkContext.parallelize(logList); // Create a new RDD JavaRDD<String> stringJavaRDD = originalRDD.flatMap(value -> Arrays.asList(value.split(" ")).iterator()); // Creating another new RDD just by splitting based on blank space stringJavaRDD.foreach(value->System.out.println(value)); } |

Output:

|  |  |  |
| --- | --- | --- |
| Tuesday  4  September  0408  WARN:  Tuesday  4  September  0405  FATAL:  Wednesday  5  September  1632  ERROR:  Friday  7  September  1854  WARN:  Saturday  8  September  1942 | 1. Created list of logList 2. Added only warning level log 3. Created Spark config for all the thread and clusters 4. Created spark context using spark config 5. Created first original RDD of the logList using sparkcontext’s method parallelize method 6. Now let’s use the flatMap function which take another function as an input and it will created another new RDD. Here we first we are converting the List into Arrays asList then using split method converting them into single string. 7. And finally we can see the output as a new RDD on LHS in the single words. | Output using Filter:  Wednesday  September  1632  ERROR:  Friday  September  1854  WARN:  Saturday  September  1942  WARN:  Tuesday  September  0405  ERROR:  Tuesday  September  0408 |

**Filters:**

This is very good function, which can be used to create a new RDD by removing the things which we don’t want. For example in the above example if want to remove the words having length 1 (like here we have numbers) i.e. we want to filter those out of the collections.

|  |
| --- |
| JavaRDD<String> originalRDD = sparkContext.parallelize(logList); // Create a new RDD  JavaRDD<String> stringJavaRDD = originalRDD.flatMap(value -> Arrays.asList(value.split(" ")).iterator());  // Creating another new RDD just by splitting based on blank space  JavaRDD<String> words = stringJavaRDD.filter(word->word.length()>1);  words.foreach(value->System.out.println(value)); |

In the above filter output we can see that we are getting the words length greater than 1.

The above line of code can be written as in short:

|  |
| --- |
| SparkConf sconfig = new SparkConf().setAppName("StartingSpark").setMaster("local[\*]");  JavaSparkContext sparkContext = new JavaSparkContext(sconfig);  sparkContext.parallelize(logList)  .flatMap(values-> Arrays.asList(values.split(" ")).iterator())  .filter(words->words.length()>1)  .foreach(value->System.out.println(value)); |

**21. Reading from Disk : i.e. reading file from window directories**

So far we have been working on hard code in the form of java collection. But now let’s start working on some real test data i.e. reading data from external file or disk.

1. While working hard code with java collection then we have used parallelize (pass value) of SparkContext but in case of reading data from file simply use method textFile (file Location) of SparkContext.
2. Here the path could be anything like file located in Amazon storage (S3 ://), or it could be HDFS file from the several cluster
3. Usually in Window while reading the file from any location then we will get one exception as given below



So to solve this winutils.exe exception we have to put this winutils.exe in the c: drive and will this exe path we will set as System property inside the application from where the file is being read.

|  |  |
| --- | --- |
|  | System.setProperty(“hadoop.home.dir”, “C:/Hadoop”); |