

# Spark Fundamentals - II

*Using RDD persistence and memory tuning*

---

# Contents

**USING RDD PERSISTENCE AND MEMORY TUNING ..... 3**

1.1 USING RDD PERSISTENCE AND MEMORY TUNING. .... 4

SUMMARY ..... 9

---

## Using RDD persistence and memory tuning

This lab will cover caching and memory tuning. You will start by caching the trips and stations RDDs. Then compare the different storage levels including serialized vs not, and replication.

After completing this hands-on lab, you should be able to:

- Use RDD persistence to speed up operations.
- Understand different storage levels
- Use Kryo vs Java serialization

Allow 15 minutes to complete this section of lab.

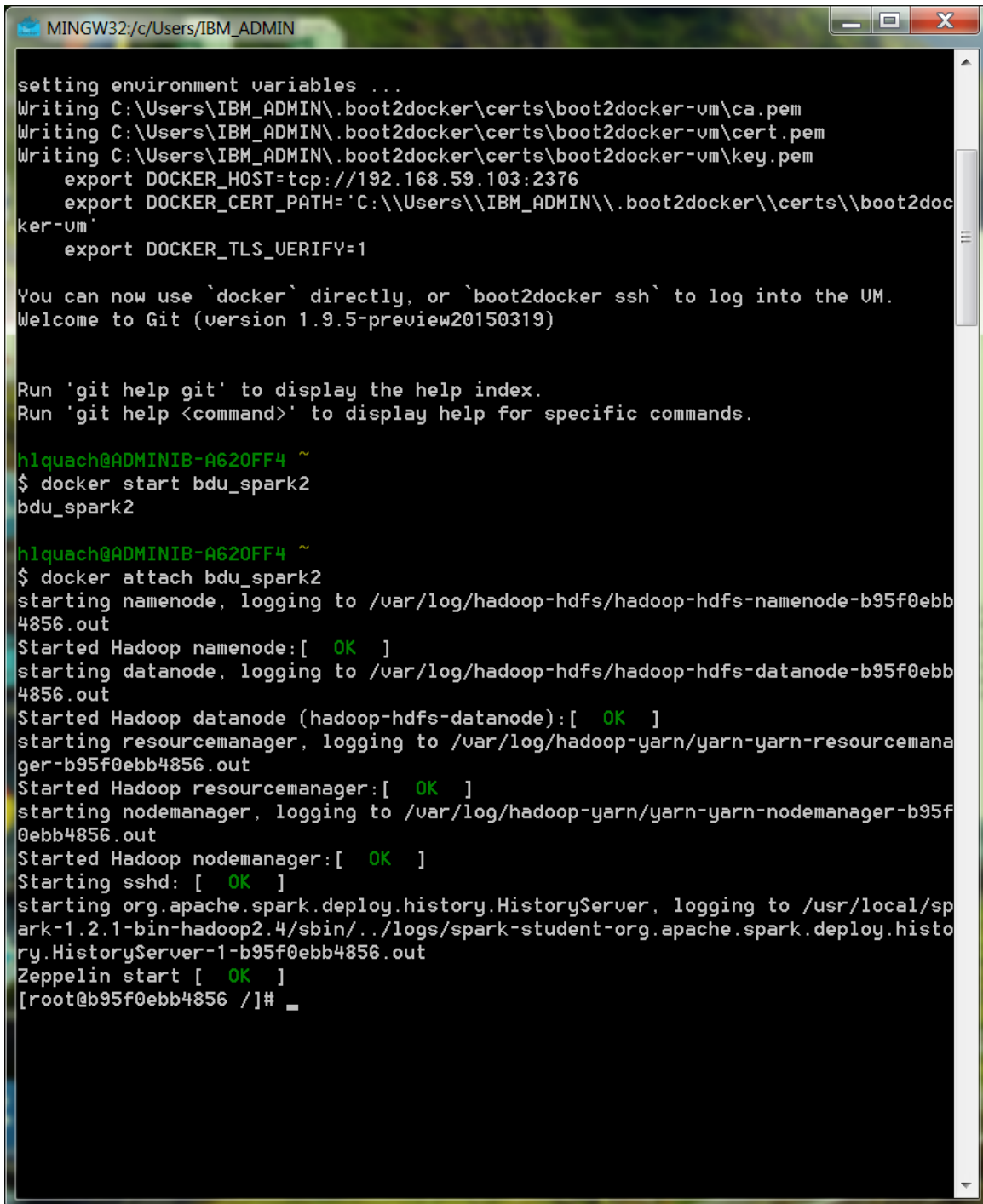
## 1.1 Using RDD persistence and memory tuning.

Make sure your docker and the Zeppelin is running. If not, get it started before continuing with this lab. Use the instructions here:

<https://registry.hub.docker.com/u/bigdatauniversity/spark2/>

We are using the same two datasets from the last lab.

- \_\_\_1. Here I show how you would restart the container if you need to. Otherwise, continue from the previous instance of the boot2docker terminal from lab 3.



```

setting environment variables ...
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\ca.pem
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\cert.pem
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\key.pem
export DOCKER_HOST=tcp://192.168.59.103:2376
export DOCKER_CERT_PATH='C:\\Users\\IBM_ADMIN\\.boot2docker\\certs\\boot2docker-vm'
export DOCKER_TLS_VERIFY=1

You can now use `docker` directly, or `boot2docker ssh` to log into the VM.
Welcome to Git (version 1.9.5-preview20150319)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

hlquach@ADMINIB-A620FF4 ~
$ docker start bdu_spark2
bdu_spark2

hlquach@ADMINIB-A620FF4 ~
$ docker attach bdu_spark2
starting namenode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-namenode-b95f0ebb4856.out
Started Hadoop namenode:[ OK ]
starting datanode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-datanode-b95f0ebb4856.out
Started Hadoop datanode (hadoop-hdfs-datanode):[ OK ]
starting resourcemanager, logging to /var/log/hadoop-yarn/yarn-yarn-resourcemanager-b95f0ebb4856.out
Started Hadoop resourcemanager:[ OK ]
starting nodemanager, logging to /var/log/hadoop-yarn/yarn-yarn-nodemanager-b95f0ebb4856.out
Started Hadoop nodemanager:[ OK ]
Starting sshd: [ OK ]
starting org.apache.spark.deploy.history.HistoryServer, logging to /usr/local/spark-1.2.1-bin-hadoop2.4/sbin/../logs/spark-student-org.apache.spark.deploy.history.HistoryServer-1-b95f0ebb4856.out
Zeppelin start [ OK ]
[root@b95f0ebb4856 /]#

```

- \_\_\_2. Open up **Lab 4** in the Zeppelin notebook.
- \_\_\_3. The first task you have seen before -- it is just loading and parsing the data from the CSV files. Go ahead and run it.
- \_\_\_4. Next calculate the average duration of trips both by start and end terminals in the trips RDD. Add in the code in the first panel. There should not be any surprises here.

```

val durationsByStart = trips.keyBy(_.startTerminal).mapValues(_.duration)
val durationsByEnd = trips.keyBy(_.endTerminal).mapValues(_.duration)

val resultsStart = durationsByStart.aggregateByKey((0, 0))((acc, value) => (acc._1 + value, acc._2 + 1), (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val avgStart = resultsStart.mapValues(i => i._1 / i._2)

val resultsEnd = durationsByEnd.aggregateByKey((0, 0))((acc, value) => (acc._1 + value, acc._2 + 1), (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val avgEnd = resultsEnd.mapValues(i => i._1 / i._2)

avgStart.collect()
avgEnd.collect()

```

```

val durationsByStart = trips.keyBy(_.startTerminal).mapValues(_.duration)
val durationsByEnd = trips.keyBy(_.endTerminal).mapValues(_.duration)

val resultsStart = durationsByStart.aggregateByKey((0, 0))((acc, value) =>
(acc._1 + value, acc._2 + 1), (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 +
acc2._2))
val avgStart = resultsStart.mapValues(i => i._1 / i._2)

val resultsEnd = durationsByEnd.aggregateByKey((0, 0))((acc, value) => (acc._1
+ value, acc._2 + 1), (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val avgEnd = resultsEnd.mapValues(i => i._1 / i._2)

avgStart.collect()
avgEnd.collect()

```

- \_\_5. Run the panel to get the average duration. This is done without any caching.
- \_\_6. Next, call **cache()** or **persist(StorageLevel.MEMORY\_ONLY)**. The **cache()** function actually calls that very same **persist** function underneath.


```
trips.persist(StorageLevel.MEMORY_ONLY)
```

Run the panel with that new line of code right after the stations RDD.

- \_\_7. Now re-run, but with a different storage level:

```
trips.persist(StorageLevel.MEMORY_ONLY_SER)
```

- \_\_8. Well, how do you tell if anything different happened? You can check out the Spark UI (on port 4040). Open a new tab and put in this URL: <http://192.168.59.103:4040/>. Then click on the Storage tab to see the last two serialized RDDs.



1.2.1

Jobs

Stages

Storage

Environment

Executors

Zeppelin application UI

## Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
204	Memory Serialized 1x Replicated	2	100%	68.7 MB	0.0 B	0.0 B
186	Memory Deserialized 1x Replicated	2	100%	164.3 MB	0.0 B	0.0 B

You'll noticed the top one (recent) uses less memory, why?



**Note:** If, for some reason, port 4040 is not the right port for the Spark UI in your docker container, you can find out what it is by executing **docker ps** in the boot2docker terminal to find the port mappings. For example, this was the mapping from my container:

```
h1quach@ADMINIB-A620FF4 ~
$ docker ps
CONTAINER ID        IMAGE               PORTS              COMMAND              CRE
ATED              STATUS              NAMES
b95f0ebb4856      bigdatauniversity/spark2:latest  0.0.0.0:32768->4040/tcp, 0.0.0.0:32773->4041/tcp, 0.0.0.0:32770->8042/tcp, 0.0.0.0:32771->8088/tcp, 0.0.0.0:32775->18080/tcp, 0.0.0.0:32769->50070/tcp, 0.0.0.0:32774->50075/tcp  bdu_spark2
Up 6 hours
```

Port 4041 uses Kryo serialization as you will see in the next part of Lab 4.

- \_\_9. Finally, the last task is to use the Fastutil collection library. Run the code as it is now, to use the standard Scala collection and note the time it takes to complete. Then run the code using the fast util version and note the time. The Fastutil collection is more efficient in terms smaller memory footprint. Check out <http://fastutil.di.unimi.it/> for more details.
- \_\_10. Lab 4 isn't completed yet. There is an alternate Kryo serialization lab. Open the Zeppelin notebook for **Lab 4 - With Kryo**
- \_\_11. This lab illustrates the difference using Kryo vs Java. The notebook here is running with a different instance of the Zeppelin Spark that is configured to use with the Kryo registrar. Remember that in order to use Kryo, your classes must be registered. This is one of the requirements.
- \_\_12. You should be familiar with the code in the first panel. Run it if you wish, otherwise, add in the code to calculate the average durations (same as the first part of Lab 4). Run it without any caching.
- \_\_13. Now run it with the default cache() function.
- \_\_14. Run it with persist(StorageLevel.MEMORY\_ONLY\_SER)
- \_\_15. Open up the Spark UI. In this case, it is on port 4041. <http://192.168.59.103:4041>

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
<a href="#">40</a>	Memory Deserialized 1x Replicated	2	100%	159.4 MB	0.0 B	0.0 B
<a href="#">22</a>	Memory Serialized 1x Replicated	2	100%	34.1 MB	0.0 B	0.0 B



You can see that the serialized version of Kryo uses half the memory than the Java one that you did earlier.

- \_\_16. The rest of the lab shows how you would configure Kryo serialization. Note that at the bottom of the panel, the classes are registered:

```
class Lab4KryoRegistrator extends KryoRegistrator {  
  
  override def registerClasses(kryo: Kryo): Unit = {  
    kryo.register(classOf[Trip])  
    kryo.register(classOf[Station])  
  }  
}
```

## Summary

Having completed this exercise, you should know how to use cache and persist to speed up the jobs. In fact, if you plan to reuse a RDD more than once, it is recommended to cache it, especially if that RDD was built from a long lineage of transformations. Otherwise, Spark will re-compute the same transformations each time to get to the RDD that you are using. It is also good practice to unpersist() the RDD after you are done to save storage.

[illegible]

## NOTES



---

© Copyright IBM Corporation 2015.

The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. This information is based on current IBM product plans and strategy, which are subject to change by IBM without notice. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way.

IBM, the IBM logo and [ibm.com](http://ibm.com) are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).



Please Recycle

---