Reproducibility Study - Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

George Boukouvalas
LIACS Leiden University
Leiden, Netherlands
s3645983@vuw.leidenuniv.nl

Srishankar Sundar
LIACS Leiden University
Leiden, Netherlands
s3151298@vuw.leidenuniv.nl

***Abstract*—This study reproduces two experiments from the paper 'Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing' by Zaharia et al., which proposes the use of Resilient Distributed Datasets (RDDs) as opposed to the erstwhile conventional MapReduce. To this end, we implement two such experiments on relatively more constrained hardware in an effort to reproduce the results obtained on Spark's computation across workload sizes and its fault-tolerance in the face of failure of individual nodes in the distributed system. Further, we use the results obtained to both confirm the results obtained in the original study and also drive our own insights on the two experiments.**

I. Introduction

'Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing' by Zaharia et al. [2] proposes Resilient Distributed Datasets (RDDs), a distributed data structure that facilitates in-memory computation. This is proposed as a challenger to the erstwhile champion computation framework, MapReduce.

In brief, one of the key difference between the aforementioned two paradigms is the in-memory aspect of RDDs using Spark, which the paper and further use-cases have shown to be several orders of magnitude faster when it comes to batch computations. MapReduce falls behind in many use-cases due to working primarily off the disk.

The paper goes on to compare other metrics between the two systems, namely fault tolerance and performance under various conditions, as well as study the performance of Spark alone in certain scenarios. Out of these experiments as a whole, we attempt to reproduce the results obtained in two specific experiments. These include the performance of RDDs under a workload comprised of interactive queries on datasets of various sizes, as well as an experiment that inspects iteration times for Spark when one worker is brought down in one of the iterations.

We use these results to derive our own insights, and to corroborate the findings by Zaharia et al.

The following sections of this study encompass an overview of the original paper and the results obtained by Zaharia et al. for the experiments selected, a description of the experimental setup and the datasets used, a presentation of the results and finally a discussion based on the same.

The setup steps taken, the code used and any auxiliary files can be found in the repository linked - *Github*

II. RDDs - Paper Overview

The primary problem of erstwhile computation frameworks is defined by Zaharia et al. as being inefficiency when it comes to iterative computations that reuse intermediate data across said computations. Examples of such computations are Machine Learning workloads, which are inherently iterative in nature. This is primarily due to the dependence on writing this intermediate data onto disk during each iteration when a framework such as MapReduce is used.

The authors present **four characteristics** of RDDs that double as benefits over another alternative paradigm, i.e that of Distributed Shared Memory (DSM), which by itself is in-memory but differs significantly from RDDs -

- **Coarse-grained Writes**: RDDs offer only 'coarse-grained' control, which we understand to be a higher level of control especially when it comes to memory locations. This design decision allows for higher fault-tolerance by taking control away from the user
- **Immutability**
- **Task Scheduling based on data locality**
- **Graceful degradation** in the face of a lack of memory

A. Experiments in the Paper and Experiments Selected

The authors present the benefits of RDDs through a series of comparisons on both performance and fault-tolerance. **The frameworks compared are Spark (RDDs), Hadoop and HadoopBinMem**, the latter of which is a modification of Hadoop which converts data into binary on the initial iterations in order to speed up parsing on subsequent iterations. The paper also presents isolated experiments on Spark in various scenarios. The experiments performed include:

1) Duration of first and later iterations for Logistic Regression (LR)
2) Duration of first and later iterations for K-Means clustering
3) Running time of later iterations of LR for various number of machines
4) Running time of later iterations of K-Means for various number of machines
5) Micro-benchmarks on a single machine against MapReduce on an in-memory File System
6) Iteration times for PageRank algorithm
7) **Iteration times for Spark for K-Means clustering in the presence of a single machine failure**
8) Performance of Logistic Regression with varying amounts of data in-memory
9) Running times of real user applications using Spark on various number of machines
10) **Response times for Spark for interactive queries on datasets of various sizes**

B. *Experiments for the Reproducibility Study*

We select two experiments from the list above to replicate in this reproducibility study - (7) Iteration times for K-Means in the presence of a single machine failure and (10) Response time of Spark for interactive queries on various dataset sizes.

1) Query Response Times of Spark for Interactive Queries: In the original study, this experiment presents 9 datapoints spread across 3 types of queries and 3 datasets of varying sizes (**100GB, 500GB, 1TB**), run on a **100-node** cluster. with each node having **8 cores** and **68GB of RAM** each. The experiment aims to drive insights on how RDDs scale across dataset sizes, and does this through different queries.

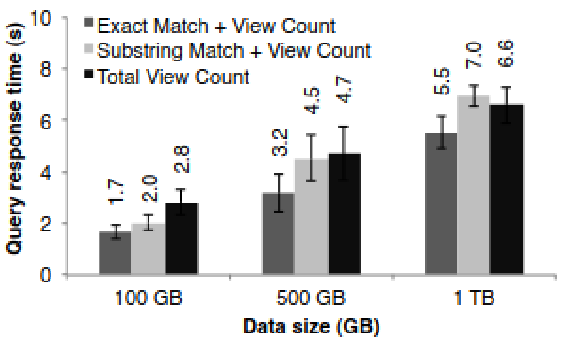


Fig. 1. Results presented by Zaharia et al. for interactive query workloads on various dataset sizes

The three queries used, in order of increasing estimated computation are that of (1) an exact match on a column and subsequently retrieving the count, (2) A partial (substring) match on a column and retrieving the count and (3) A group by and a subsequent count.

From the figure above, we see that RDDs perform well with increasing dataset sizes, with the increase in query response time being much lower than the proportionate increase in dataset size and consequently expected computation. For example we see a 5 times increase in dataset size between 100 to 500GB, while the query response times increase by only 2 to 3 times. We attempt several variations of this experiment in order to confirm the insights presented and draw some of our own. This is further elaborated in the coming sections.

2) Iteration Times in the Presence of a Single-Machine Failure: This experiment, running on a **75-node** cluster (with similar specifications as mentioned in the subsection above) on **100GB of data** tests how Spark copes with the loss of a machine one the 6th iteration of an iterative task, K-Means clustering in this case. The challenge here is that when a worker node goes down, so does the data and tasks it was running.

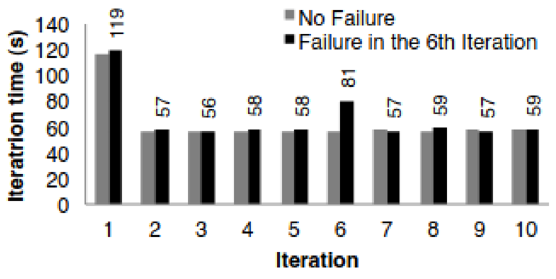


Fig. 2. Results presented by Zaharia et al. to show how RDDs perform on failure of one of the machines in the cluster workloads

The figure above shows an increase in query response time on the 6th iteration, which the authors attribute to the overhead required for Spark to re-run this computation in parallel on another worker, after which the subsequent iterations revert back to the normal execution times. The authors make note of the fact that this is due to Spark using **lineage graphs** instead of something like a checkpointing system. The latter would require more computation to run when the system has to re-run computations and would also require the entire dataset to be persisted on disk or stored continuously in-memory, which is unwieldy. We replicate this experiment on a different task and discuss our results in the coming sections.

III. Experiment Setup

A. *Cluster*

- **CPU**: 32 cores
- **Memory**: 64GB RAM with a cache/buffer of 26.67GB out of the 64GB

- **Nodes**: 3 or 4 nodes including the master, depending on the experiment/variation

Spark clusters involve one or more worker nodes and a single master node to which the scripts have been submitted using *spark-submit*. Thus when we refer to *three worker nodes*, for example, it indicates three worker nodes and one master node.

B. Environment

The experiments are conducted on specific Spark, Java and Python versions, listed below:

- **Java Version**: Java 8
- **Python Version**: Python 2.7
- **Spark Version**: Spark 2.4.0

We started by fixing the Python version to 2.7 and writing our PySpark scripts on the same. We then picked a Spark version compatible with Python 2.7, which lead us to Spark 2.4.0, and finally picked the Java version (between 8 and 11) as being Java 8, which is the one compatible with Spark 2.4.0

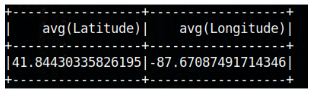
C. Datasets and Experiment Design

1) Query Response Times of Spark for Interactive Queries: Details pertaining to the dataset are given below -

- For the interactive querying section, we use the *Crimes - 2001 to Present* dataset from the Chicago Data Portal
- The dataset is **1.8GB** in its original form. In order to simulate higher data sizes for this specific experiment, we simply duplicate and append the dataset to itself to make two further variants - that of **3.6GB** and **5.4GB** of data
- The dataset contains records of **crimes in the city of Chicago** from 2001 to the present day (November 3rd 2022 at the time of this study), with about **7.6M rows** and **22 columns**

Details pertaining to the experiment design are given below:

- The first experiment performed has **3 basic queries**. This is carried out on a cluster with **3 worker nodes**, in the first experiment. We do this to check if the scaling Spark exhibits in the original study is retained in the much smaller scale experiment conducted.
  - The *Location Description* column, which presents the location where the crime was performed is queried for an **exact match** on 'STREET', and the resulting number of rows are counted, which relays to us the number of crimes that have taken place in the streets. Other such locations include 'APARTMENT' and 'RESIDENCE', for example
  - The *Date* column, which timestamps the crime, is queried for a **partial match** on 'AM', with the count of the resulting rows being taken. This relays to us how many crimes were performed past midnight and up to afternoon
  - The *Latitude* and the *Longitude* columns are aggregated with **averages** calculated on both, to glean insights on what the possible center of the crimes could be. The figure below shows an example *stdout* from the Spark script



```
+----------------+-----------------+
|   avg(Latitude)|   avg(Longitude)|
+----------------+-----------------+
|41.84430335826195|-87.67087491714346|
+----------------+-----------------+
```

Fig. 3. Output for a *show* action after the average aggregations

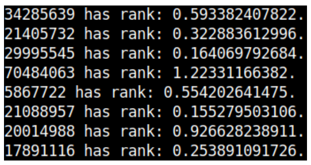
- The second variant of the experiment runs the same queries but on a cluster with **2 worker nodes**. We do this to see if Spark still exhibits the same scaling as shown in the experiment above
- We perform one final variation of the experiment, where we run **all 3 queries sequentially** in one script on a cluster with 3 worker nodes to check if the consequent operations after the first query would run faster than if it was executed individually
- For recording the query response times, we run each query on each dataset thrice and take an average across all **three repetitions**. This is accounted for in the code used. Time is calculated using Python's *time()* method, part of the *time* Python package. The times before the query and after the *show()* action are recorded as start and end respectively, and are subtracted to calculate the query response time.
- It is to be noted that this could possibly include some overhead with actually printing the data to console, however since the idea of the experiment is to calculate relative increases/decreases in execution times this is deemed to be within reason

2) Iteration times in the Presence of a Single-Machine Failure: Details pertaining to the dataset are given below:

- While the original experiment uses K-Means clustering as the workload, **we instead use PageRank**, as the script written for this provides more fine-grained control over tracking iteration times as compared to using a library for performing K-Means clustering
- To this end, we use the *Social Circles: Twitter* dataset from the Stanford Large Network Dataset Collection [1], which is an adjacency list depicting social circles on Twitter, with about **80,000 nodes** (users) and **1.7M edges** (connections, such as following)

Details pertaining to the experiment design are given below:

- The PageRank algorithm, being iterative by nature, is such that the PageRank, which is a measure of relevance of a single node, is calculated for all nodes in the network in each iteration. Thus, within each iteration, we trigger an action on the data structure, which *collects* all the nodes and their associated PageRanks. An example of an output at one such iteration is shown below:



```
34285639 has rank: 0.593382407822.
21405732 has rank: 0.322883612996.
29995545 has rank: 0.164069792684.
70484063 has rank: 1.22331166382.
5867722 has rank: 0.554202641475.
21088957 has rank: 0.155279503106.
20014988 has rank: 0.926628238911.
17891116 has rank: 0.253891091726.
```

Fig. 4. Part of output for the PageRank script after 1 iteration - the numbers on the left indicate the node IDs

- We hardcode a fixed number of iterations (10), and on the **6th iteration**, the script issues a shell command to terminate a hard-coded *reservation ID* on DAS-5
- We set up the cluster such that two of the worker nodes are under one reservation ID, and one remaining worker node is under a separate reservation ID on DAS-5. The latter is the node that is brought down during the experiment. We hardcode the reservation ID into the script. As such this is a limitation and is discussed in the Limitations section
- The time calculation, conducted using Python's *time()* method is done by recording the time at the start of the iteration and the time at the end of the iteration after the *collect* action. Thus the overhead time taken to issue the shell command and bring down the node is not factored into the recorded time.
- Before the time is recorded at the start of the iteration, we check for the iteration being the 6th in number, in which case we send the shell commands to decommission the reservation for the lone node
- The experiment has been **repeated thrice** and the times depicted in the Results and Discussion section are an average across these repetitions. However we note here that repetitions are not implemented in code (unlike the previous experiment) and are instead recorded and averaged manually

IV. Results and Discussion

A. Query Response Times of Spark for Interactive Queries

From figure 5, we see that the **query response times between the 1.8 and 3.6GB datasets are fairly similar for a 3-node cluster**, with an increase in response time for the 5.4GB dataset. This shows that Spark performance scales across datasets of these sizes, with the increase in response time being far lesser than the proportional increase in dataset size.

We repeat this experiment for a cluster with **2 worker nodes** to investigate whether this trend holds when fewer nodes handle the same amount of data. This is depicted in figure 6.

The 2-node cluster shows a **relatively more uniform increase in response times** across the queries. The quantum of increase is also higher for two worker nodes, which is interesting to note in comparison with a cluster with 3 worker nodes. We note that the execution times themselves are not much higher than those on 3 worker nodes, however the cluster with 2 nodes scales relatively worse. This can be rationalized by the idea that with 3 workers, Spark is able to use the extra node to offload extra computation more effectively.

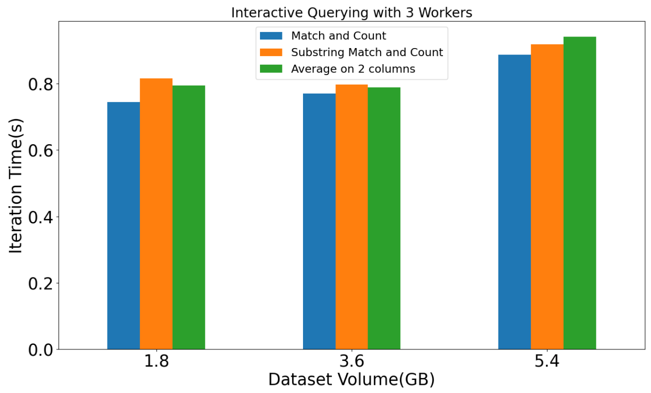


Fig. 5. Query response times on a cluster with 3 worker nodes

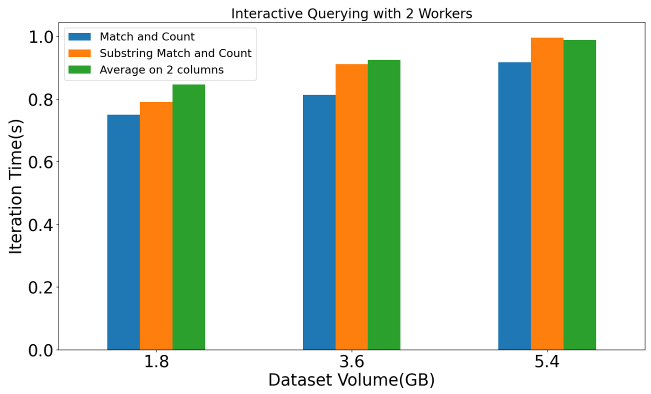


Fig. 6. Query response times on a cluster with 2 worker nodes

We can conclude from these two experiments that we are able to reproduce the results from the original study to a certain extent. We note that **the third query, that of the average aggregation is significantly different from the third query performed in the original study**, which is that of a Group By and is inherently heavier than the first two queries.

Further, we run one final experiment where the **queries are run sequentially within a single script** on a cluster with 3 worker nodes. We do this to check if the subsequent queries after the first query show faster response times. This is depicted in Figure 7. We see that the first query runs with response times similar to the previous experiments, however we note a significant difference in response times of the subsequent queries for the 1.8 and 3.6GB datasets. However, this trend is broken for the 5.4GB dataset.

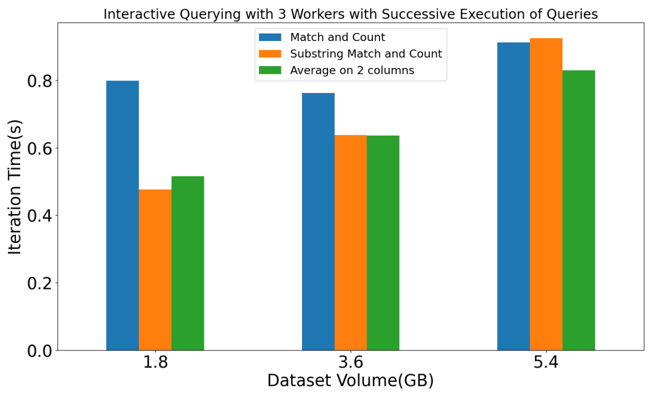


Fig. 7. Successive query response times on a cluster with 3 worker nodes

Firstly, the subsequent queries executing faster could be attributed to Spark serializing the RDDs for the first query and not repeating this step for the subsequent queries thus saving time. Secondly, it is unclear why this doesn't show the same effectivity for the 5.4GB dataset. One reason for this could be that the computations required for the 5.4GB dataset are heavy enough to push the 3 worker nodes to their limit and thus canceling out any gains. This could also have been due to a single outlier in one of the repetitions large enough to sway the average. We discuss this briefly in the Limitations section.

B. Iteration times in the Presence of a Single-Node Failure

This experiment investigates how Spark performs when **one of the worker nodes is brought down in the middle of a computation**. We recall that, in the original study, the iteration where the worker is brought down takes longer to respond, following which the response times fall back to the norm in subsequent iterations.

To reproduce this experiment, we run an iterative PageRank computation on a cluster with 3 worker nodes. We perform two runs, where a worker is brought down on the 6th iteration on one of the runs. The results of the experiment are depicted in figure 8.

We see here that we are unable to replicate the effect of the node being brought down in the original study, where the 6th iteration had a spike in response time. The possible reason for this is that the computation load is light enough for the overhead of Spark re-runnning the failed jobs on the remaining nodes to be light enough that it doesn't show up as an increase in time.

We do however notice a phenomenon not observed in the original study, in that in our experiment **we see a gradual increase in computation time on iterations after the 6th**. It is clear from our figure that iterations 2,3,4 and 5 take lesser time than iterations 6 to 10. Thus, while the iteration on a node is brought down it does not experience a spike in response time, the subsequent iterations take longer to run on the remaining nodes. We can however state that the increase is gradual and well within reasonable range.

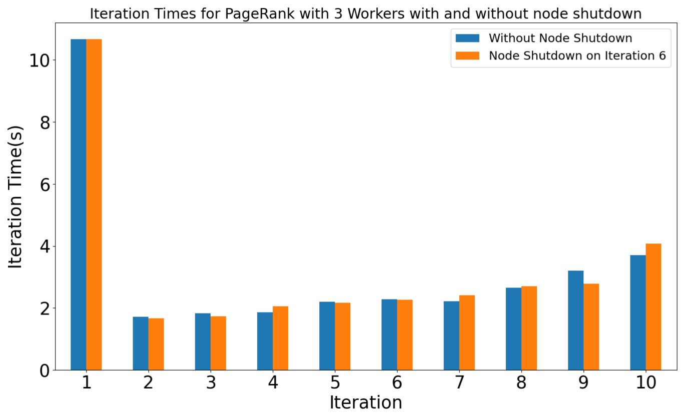


Fig. 8. Iteration times for PageRank with and without a node failure on a cluster with 3 worker nodes

V. Limitations and Further Improvements to the Reproducibility Study

- The primary limitation of the way in which the reproducibility study has been conducted is the **reproducibility of the experiments** themselves. Due to using specific Spark, Java and Python versions, the ideal way to facilitate reproducibility on the experiments we conducted would have been to wrap the installation of the environment (Java 8, Python 2.7 and Spark 2.4.0) into a singular script along with deployment of Spark on the cluster. However, the current implementation only performs the latter while the dependencies are listed as requirements
- Our study **could have performed more variations of the experiments on a higher range of datasets**. The study is currently limited in this sense. This is evidenced in the node failure experiment, where we fail to reproduce the exact pattern exhibited in the original study. We note that we attempted the usage of larger graphs in this case, however they resulted in Out-of-Memory errors even on larger clusters of 7 to 10 workers. At the same time, increasing the number of workers while retaining the same dataset reduced the response times to an extent where we could not arrive at a significant comparison. Thus further exploration of more suitable datasets would have been ideal
- **Hard-coding** reservation ID for the node failure experiment is a glaring limitation of the node failure experiment. The more direct approach here would have been to shut a random individual worker down using a script/a

Spark-specific command, however we had to defer to the current implementation in the interest of time

- The second experiment, that of iteration times on node failures **does not factor in repetitions within the script**. Instead, this has been run three times and the execution times have been averaged extrinsically. This was also done in the interest of time and is not an ideal design decision
- As evidenced by the slight ambiguity in the results of the interactive queries experiment, where we study the queries running one after the other on the same script, **more repetitions for each experiment** could have helped determine the possibility of this ambiguity simply being an outlier

VI. Conclusion

Over the course of this reproducibility study we replicated two experiments on Spark RDDs - that of response times on interactive queries and that of iteration times in the event of a node failure. We are able to reproduce the results of both experiments to an extent on smaller datasets and resources. With response times on interactive queries, we conduct a few additional variations on the original experiment. Through these, we arrive at the conclusion that Spark exhibits similar scaling on different workloads even with a reduced number of resources, as well as observing that if the queries are executed sequentially within the same script there is a noticeable speedup in execution of the subsequent queries. With iteration times in the event of a failure, we notice that the experiments we conduct do not exhibit the spike in iteration time in the event of a failure that is seen in the original study, however, we do notice a gradual average increase in iteration times. We also note certain limitations of the reproducibility study conducted, with possible improvements being use of reproducible scripting for the environment setup, further variations, repetitions and in the experiment design, and the removal of any hard-coding to facilitate ease of reproducibility.

Appendix

A. Tasks and Roles

Tasks	**Person***	**Time Taken****
Environment Setup	George, Srishankar	6 days
Scripts for Experiments	Srishankar	3 days
Debugging, Code cleanup	George	1 day
Report Writing	Srishankar	3 days
Visualizations	George	0.5 days

* Persons:
George - George Boukouvalas
Srishankar - Srishankar Sundar

** Time Taken: 1 day represents 1 working day. This is a rough estimate.

References

[1] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[2] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.