DPS Assignment 1

Reproducibility Report of Spark for K-means and word count

Stephan v.d. Putten
s1528459
Leiden University

Tristan Kattenberg
s2508907
Leiden University

1 INTRODUCTION

Distributed processing system are the backbone of many modern systems. Such systems are build with the help of clusters. Clusters are a combination of servers, usually called nodes, that are orchestrated using systems such as Hadoop [9] or Spark [10] to run applications on. The systems are usually tested in confined testing spaces creating a mismatch between real performance and measured performance [7]. Providing consistent performance is not a given due to the high variability in systems [5]. Furthermore the lack of standard benchmark methodology of parallel computing systems makes reproducibility and accurate measuring a challenge on its own [4]. In this paper we will challenge ourselves by reproducing the performance of the Spark system and report on the claims given by this system.

Our paper is organised as follows. In Section 2 we take a closer look at the original research paper of Sparks and give the necessary background knowledge. In Section 3 we explain the authors methodology and explain what and why we made adjustment to the authors methodology for our duplication. We report our results in Section 4. We conclude with Section 5, 6 containing the discussion and conclusion.

2 BACKGROUND

In this section we will give the general background. This section will start with high level introduction of the Hadoop Distributed File System (HDFS) and Resilient Distributed Datasets (RDDs) move onto what applications RDDs are better suited for then HDFS system and the reasons why RDDs are a better fit.

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

"Resilient Distributed Datasets"[10] introduces RDDs arguing this outperform the existing HDFS[6] on application which utilises coarse-grained transformations and easy data recovery using lineage. One claim is that the iterative machine learning applications in Spark outperforms Hadoop 1.9× to 3.2× using K-means.

This is the results of using RDDs. When data is reused it does not require the reloading of data into memory from local storage or network bandwidth. Data is held in memory instead of having to be written into a distributed file system then copied to different nodes and read again from one of these nodes. This removal of I/O overhead decreases the time necessary to perform computational tasks.

2.2 Introduction to RDDs and HDFS

HDFS In order to store massive amount of data in a redundant way the HDFS replicates data into many location on a network. The replication reduces the risk of failure. When data is lost on one location the data is still accessible using a replicated node. Distributed computing systems are designed to allow the building of complex application on top of the HDFS system through the use of MapReduce[2]. When using MapReduce the only way to store output of a operation is to copy the output back into a HDFS system. For each MapReduce call you must first load data into memory run a operation and then write the memory back into a HDFS system , as shown in figure 1. This type of read, iteration, write loop is not necessary for applications "[which] reuse intermittent results across multiple computations" [10]. This is where the authors proposes Resilient Distributed Datasets (RDDs).

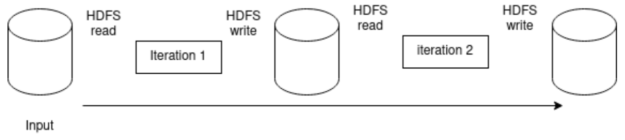

Figure 1: Demonstration of iterative calls performed on HDFS file system

Resilient Distributed Datasets (RDDs) RDDs are "read only, partitioned collection of records."[10]. We can only create a RDD from data in a stable storage or from other RDDs, e.g. using map, filter or join. RDDs are able to reconstruct themselves after failure since they retain enough information on how it was created from the previous computed data. A program may also indicate where a RDD partitions in memory and where the RDDs elements are partitioned across machines based on keys.
In Figure 1 we have multiple iterations where the data is stored in memory.

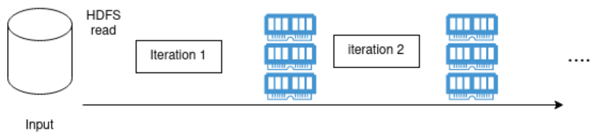

Figure 2: Demonstration of iterative calls performed on RDDS with input from HDFS followed by storage in memory after each iteration

2.3 Advantages of RDDs compared to HDFS

RDDs reduces the need of I/O operations since RDDs allow data to be stored in memory. As such this makes Spark faster than Hadoop where after each iteration data must be written back to the HDFS. Compared to only using HDFS, RDDs have lineage which allows is to efficiently reconstruct itself while HDFS is more error prone. Furthermore the nature of RDDs offer fault-tolerance with speed being no issue as as only the lost data has to be recomputed compared to a redo of the complete task computation. However this does require I/O operations to be performed but does not require the whole data set to be recalled from disk storage.

3 METHODOLOGY/DESIGN

This section explains our experiments and motivate our approach. We start by explaining the infrastructure. Secondly we extract the relevant experiments from the paper on RDDs and Spark. In the experiments we explain how to original author proposed the setup and how we tweaked the setup to fit our system. Next we propose our setup for the same experiment. The goal is to duplicate and verify the result or the original paper. Our experiments are publicly available on Github [1].

3.1 DAS-5 Cluster System

The Distributed ASCI Supercomputer 5 (DAS-5) is a distributed system used for research by computer scientists [1]. Using SLURM resource management system[2] we can allocate different amounts of nodes for our experiments. DAS-5 has multiple clusters. We use the Leiden University cluster which has access to 24 cpu nodes, each consisting of: dual 8-cores with 2.4Ghz speed, 64GB Memory, 128 TB storage, 2*4TB local HDD's and using network IB and GBe.

Cluster setup Hadoop en Spark support multiple cluster managers. The cluster manager types use different job scheduling and resource managing. In this experiment we are only working with 1 type of cluster manager, however will briefly state other relevant systems below. Standalone is the default for Spark and Hadoop has YARN built in. We use JDK 8u271 provided by oracle as our java version. Our experiments are run using one driver + *n* workers/nodes depending on the experiment. We store the data on our HDFS for the cluster and use local storage devices for the workers. Our OS is CentOS 7.

- Standalone: Works on FIFO principle only used by SPARK
- Apache Mesos: More general can work with both Spark and Hadoop MapReduce
- Hadoop YARN: Hadoops MapReduce resource manager [8]

Mesos is used in the original paper, we state it briefly here to give a more holistic overview however we will primarily use YARN. We use Spark 2.4.7 and Hadoop 2.10.1.

3.2 Experiments

In this section we explain what experiments we are running, how they are relevant, what benchmarks we use and go into more detail about the experiments we run.

Experiment choice Evaluating how the authors ran their experiments we choose to run a K-means and word count test. This was mainly due to our benchmark system HiBench including a K-means and word count algorithm for both Hadoop and Spark.

K-means Original The original author ran their experiments on Amazon EC2 using the m1.xlarge EC2 node. As mentioned before in section 2.1 Spark outperforms Hadoop with K-Means. The authors shows a K-means compared to Logistic Regression, though no implementation is shown of K-means. 10 iterations were run on 100GB of data using 25-100 nodes. Each node is set with 4 cores and 15GB of RAM, HDFS is used for storage with block sizes of 256MB, instead of the default 128MB. The author states that Spark shared the resources of HDFS using the Mesos cluster manager [3]. In case of memory usage the author states that it computes as much as possible in the RAM for Spark.

For practical reasons we decreased the node size to match a setup of 5, 10 and 15 nodes plus 1 master in each setup. The cluster manager is changed to YARN to support shared HDFS usage for both Hadoop and Spark. See section 3.1 for remaining details. The data set is changed to HiBench's default setting of large, see section 3.4.

K-means Our approach To improve on the original paper, we choose to increase the amount of RAM to 40GB and increase the amount of cores to 16. Individual servers have improved over time, more CPU's and RAM, and to see if the increased capability of nodes are meaningful for distributed systems we tune these specifics shown in Table 1. As the author states K-means is a algorithm that is computationally heavy we expect increasing the computational power should give an increase in performance. We leave remaining parameters the same.

Interactive Data Mining Original In the original paper they analyse different sizes of data sets using 100 nodes m2.x4large EC2 instances with 8 cores and 68GB of RAM each. They perform different queries on data sizes ranging from 100GB to 1TB.

In our setup we change the cluster size to 14 nodes and perform word counts instead which can also be seen as a form of query. Furthermore we use 50GB of RAM and perform the queries on a tenfold increase of data sets of size 0.32GB, 3.2GB and 32GB. As the original author we will report on the speed of the queries.

Interactive Data Mining Our approach As our current setup deviations enough from the original paper we do not have to tweak much from the original parameters setup by the author. Similarly to our previous approach we will increase the amount of cores from 8 to 16 cores and run the benchmark in Spark again. Spark is run with YARN. Table 1 show the changed settings.

Table 1: Experiment settings in the original and new approach for the nodes in K-means.

Parameters	**K-means**		**Word count**	
	Original	New	Original	New
Max Ram	15GB	40GB	50GB	50GB
Cores per node	4	16	8	16

3.3 HiBench

HiBench is a benchmark suite created by Intel to evaluate big data frameworks [3]. Instead of creating a custom set-up we belief using an open-source popular benchmark increases the validity of our experiments. HiBench uses Scala 2.11 and is built using maven

[1]https://github.com/Stvdputten/Distributed-Data-Processing-Systems-A1
[2]https://slurm.schedmd.com/documentation.html
[3]https://github.com/Intel-bigdata/HiBench

3.6.3. We wrote a script that runs on DAS-5 to select the nodes and benchmarks and outputs the results.

3.4 Dataset

One of the main advantages of HiBench is that it allows us to use random generated data with fixed sizes and stores the dataset on our storage device of HDFS. HiBench settings are set to 'large' K-means setup, the parameters can be seen in Table 2. For obvious reason we won't include the computation time of the dataset generator as that is not the goal of this experiment. The samples are created using a Gaussian distribution with the initial centroids setup by a Uniform distribution. The data size is around 3.7GB.

For the Data Mining we can use the auto generated word documents that are described using zipf's distribution. The dataset sizes are small, large, huge; respectively 320MB, 3.2GB and 32GB.

Table 2: Parameter setup of the K-means.

Parameters	K-means
Number of clusters	5
Size of dimensions	20
Number of samples	20000000
Samples per file	4000000
Max iterations	5
K	5

3.5 Experiment 1

Experiment 1 consist of running K-means with the settings mentioned in Table 2. To reduce the variability our experiment are run 20 times in both Hadoop and Spark using HiBench. With the parameters tuned to be as similar to the original author. The cluster size is 5, 10 or 15 workers and 1 driver node. The experimental setup is shown below. We run the data generation at the start and run the K-means algorithm *N* times.

We change the compute power of our nodes by increasing the RAM and CPU power per node/worker. The experiment will be run 20 times for both Hadoop en Spark. The goal is to increase the individual work node computational speed to further reduce the computational time it takes to calculate the K-means. The remaining experimental setup is unchanged.

(1) Generate data points and centroids 1 time
(2) Run Hadoop K-means run
(3) Run Spark K-means run
(4) Save results and go back to step (2) until N runs are done
(5) Output results

3.6 Experiment 2

Experiment 2 consists of running a simple word count benchmark. We only use Spark to query on the datasets generated for a size *S*, which can be either small, large or huge. We will report on the computation time and report on the effect of increased data set sizes for Spark. Below we show a brief overview of the approach to our experiments.

(1) Generate a dataset with words of size *S*
(2) Run Spark word count
(3) Save results and go back to step (2) until *N* runs are done
(4) Output results

We use box plots and bar plots to represent the results. We define outliers as the larger or small with the usual definition of 1.5xIQR from the 1st quartile and added to the 3th quartile.

4 RESULTS

In this section we present the results from the paper using K-means; comparing these results with our results. In the discussion we will report on the notable results and give our explanation.

4.1 Experiment 1: K-means

In Figure 3 the results are shown with the setup of the original author reproduced with our infrastructure for K-means. The box plots are for both Hadoop and Spark with the computation time on the y-axis and number of nodes 5, 10 or 15 on the x-axis. We can see that the increase of nodes decreases the computation time for Hadoop and increases for Spark. The box plots show little variance for both Spark and Hadoop. Spark always has a lower computation time and seems to be between 2× and 3× faster.

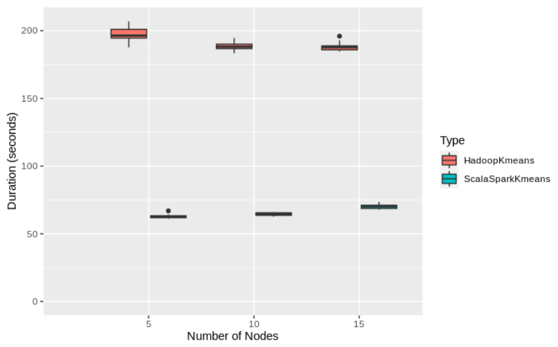

Figure 3: Hadoop and Spark execution time box plot for 20 runs on K-means with the setup of the original author.

In Figure 4 we show the results of our second experimental setup. The experiments are run with increased RAM and CPU's. The results are similar to the original authors setup. The same trends reoccurs as our previous setup as we see that Hadoop has a decreasing median trend with the increase of nodes. Spark has a slight increase of the median speed. The Hadoop 10 node setup does show a noticeable increase of variance. Furthermore the 15 node setup of Hadoop shows a few noticeable outliers that have a computational speed around 125 seconds.

In Figure 5 we show a bar plot to compare the results of the original article and our setup. The figure shows similar mean results for the different node setups. The results show that both setups have varying performance with no clear winner.

To confirm if the mean duration is effected by both numbers and nodes and the system we run a ANOVA test. We found a statistically-significant different in average duration by both node number (f(2)=9.01e-05, p < 0.001) and system type (f(1)= <2e-16, p < 0.001). The interaction between number of nodes and system type was also significant (f(2)=2.40e-10 , p < 0.001). We can conclude running a K-means algorithm has with Spark and Hadoop have statically significantly difference in average duration. Further

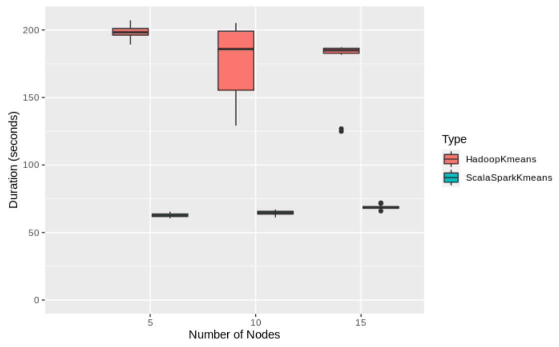

Figure 4: Hadoop and Spark execution time box plot for 20 runs on K-means with the new setup of increased RAM and cores.

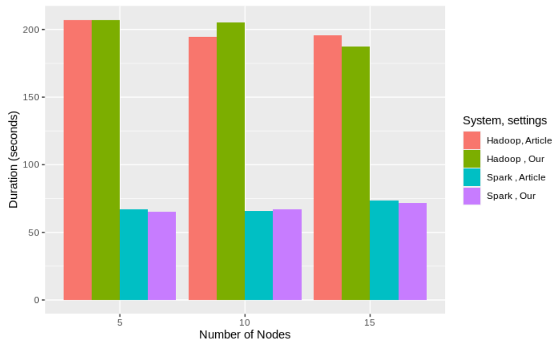

Figure 5: Comparison of Duration in second between Hadoop and Spark using our setting and new settings.

by examining both 5 and 4 we can conclude Spark has a shorter duration regardless of number of nodes or our settings.

We continue by examining the effect of the number of nodes on the duration. We find mean duration is statistically-significantly different (f(2) = 0.000399 < 0.001) depending on the number of nodes. A Tukey post-host test revealed significant pairwise different between node numbers 5 and 10 (+7.090538 seconds) at a P < 0.001 level. The difference between 15 and 10 (+3.245800 seconds) and 5 and 15 (+3.844738) are both not statistically significant at the .001 level. We believe these extra nodes computational power do not out weight their increase in overhead demand. This leads to a decrease in mean duration but a large enough decrease to be statistically significant.

To examine whether our settings cause a statistically significant difference in mean duration. We run a ANOVA test to see the effect of setting on mean duration while adding system type and number of nodes as blocking variables. We find a statistically signification difference (f(2)=<2e-16, p < 0.001). Our setting do have a statistically signification effect of duration but not in the downward direction. By examining figure 5 we see this effect.

4.2 Experiment 2: Word count

In Figure 6, 7 are results are shown for the word count application used to represent the interactive data mining. Figure 6 with the original setup the increase of the data size show a linear increase of the median computation speed. The 0.32GB and 3.2GB data set has little variance shown by the tight inter quartile range. The largest data set of 32GB shows an increase of variance however no outliers are shown.

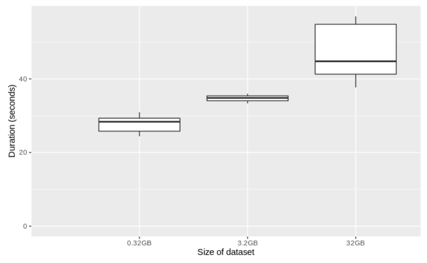

Figure 6: Word count Spark execution time box plot for 20 runs with the tweaked setup of the original author.

In Figure 7 we show our setup with the increased CPU count. In comparison to our previous figure we see a similar increasing trend of computation time with the increase of data set size. The variance is consistent between the data size increase. There are several outliers which all are in the upper quartile. Running an Anova test to see if mean duration is significantly different we the data set size does change mean duration(f(2) = 2e-16, p = 0.001). We again find our setting are not significant at either the 0.001 or 0.05 p-value levels.

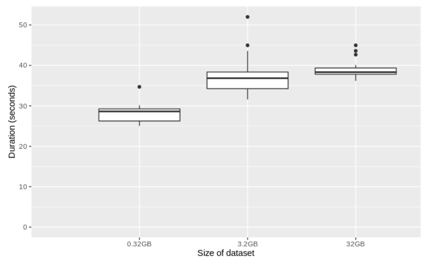

Figure 7: Hadoop and Spark execution time box plot for 20 runs on K-means with the new setup

The setup of the original author and our setup perform similar based on the median for all the sizes, 0.32GB, 3.2GB and 32GB.

5 DISCUSSION

In this section we discuss the results reported in the previous section.

5.1 Experiment 1

Here we discuss a few of the note worthy points our results show. In general we note that similar to the original paper K-means in Spark outperforms Hadoop 2-3 times.

Increase of computation time Spark One of our first noticeable trends is the increase of the time when we increase the nodes. The trend holds for both our approach and the original approach. We think that the computation time necessary is actually small and throwing more computational power, a.k.a more nodes, increases computation time due to the overhead of distributing the systems resources.

Variability and Outliers In Figure 4 the 10 node have a higher variance. We think the variance could a byproduct of the higher server usage of the cluster during our experiment by other users. Otherwise the variance stays unexplainable, although there is variance there are almost no outliers which seem to indicate our results are reliable. In the 15 node setup we see two measurements that are 40 seconds faster than the remaining measurements. Locality of the data processing could be one of the reasons or the system has some caching implemented that increased the performance.

5.2 Experiment 2

In general we only have two points to discuss which are mentioned below, we do see that 10x fold increase of data does increase the amount of time necessary for Spark to compute, however the performance is better than we expected. The original paper shows similar results; as their the data size increases the time computations remain feasible.

Variance in original setup for the 32GB data set In contrast to all other results shown in both setups, we can see that Figure 6 has a higher variance with the bigger data set. The increase of size could be at the computational limit of the nodes as we can see in Figure 7 that increasing the thread count reduces the variance. The nodes are stuck in processing the jobs quick enough and need more cpu's.

Outliers in Our approach In Figure 7 outliers are measured in all sizes of the data. Our only explanation would be due to the increase of the number of cores the tasks are more spread out within nodes. The spread of the task causes more overhead due to communication of progress of each batch of tasks especially if the tasks are spread out more between nodes. Either communication is great or communication between nodes is increased due to low locality.

6 CONCLUSION

Distributed processing systems are the backbone of many moderns systems. Reprodicibility is one of the foundations of science. The mismatch between reported performance and actual performance with distributed systems can be noticeable. This bring us to evaluating Spark that uses in-memory calculations to speed up processing power compared to using local storage.

Our experiments in this paper ran on the DAS-5 system. We ran two experiments. First K-means which the author claimed performance of 2-3 times that of MapReduce with Hadoop. Secondly the author clams data mining on large datasets is very feasible and showing a linear performance with the increase of data. To reproduce the experiments we fit the original setup as much as possible to our current infrastructure and ran the experiments with our own improvements. The unexplained parameters of the original paper were left to system defaults.

Our final results for K-means show similar performance to the authors original claims both in the original setup and our setup.Increasing the amount of nodes does not improve computation time much for both Hadoop and Spark. Furthermore increasing hte computational power of individual nodes similarly does not signify an noticeable increase in performance. The interactive data mining showed varying outliers and variability in the measurements. This shows how measurements can have high unexplained variability even in closed cluster environments where variability is easier to explain. However even taken the unexplained variability we can still see that the claim of linear increase of computational time holds for the data mining.

In conclusion our paper reproduced the original authors experiments. Despite the ambiguity of specific experimental setups enough details was explained to reproduce the original results of the paper.

7 FUTURE WORK

In case of future work we would take a closer look at the in explainable variability of our current results. With sufficient time we would increase the sample size of our measurements. Furthermore we would perform our experiments on a computationally more expensive K-means as that current setup does not show an performance gained with the increase of nodes. Finally we would tweak the experimental code to perform the experiments with less tweaking.

REFERENCES

[1] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. 2016. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer* 49, 05 (may 2016), 54–63. https://doi.org/10.1109/MC.2016.127

[2] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design Implementation - Volume 6 (OSDI'04)*. USENIX Association, USA, 10.

[3] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*.

[4] T. Hoefler and R. Belli. 2015. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807644

[5] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 409–425. https://www.usenix.org/conference/osdi18/presentation/maricq

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[7] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. 2019. Is Big Data Performance Reproducible in Modern Cloud Networks? (2019). arXiv:cs.PF/1912.09256

[8] Vinod Vavilapalli, Arun Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: yet another resource negotiator. *Proceedings of the 4th Annual Symposium on Cloud Computing, SoCC 2013*. https://doi.org/10.1145/2523616.2523633

[9] Tom White. 2012. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.".

[10] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 15–28.

A TIMETABLE

In Table 3 we show the estimated time spent.

Table 3: Who did what and how much hours spent

Activity	**Stephan**	**Tristan**
Github	4	1.5
DAS-5	8	3
Q&A	0.5	0
Coding	40	25
Read Paper	4	3
Write Report	15	15
Experiments	15	4