# CS744 Big Data Systems - Assignment 1

Group 12

Changho Shin Hokeun Cha Christine Lee

## 1 Introduction

In this report, we present our observations and analysis of assignment 1. Each section contains a description of our experiments and a discussion of our results. In Section 2, we describe the hardware configurations and detailed steps of our software configurations. This section includes *Part 0: Mounting Disks* and *Part 1: Software Deployment* of the assignment. Section 3 describes the implementation of a Spark program to sort data. This section includes *Part 2: A simple Spark Application* of the assignment. Section 4 includes the implementation of the PageRank algorithm with evaluation of our implementation under various conditions. This section includes *Part 3: PageRank* and four tasks: 1) Write a Scala/Python/Java Spark application that implements the PageRank algorithm; 2) Add appropriate custom DataFrame partitioning and see what changes; 3) Persist the appropriate DataFrame as in-memory objects and see what changes; 4) Kill a Worker process and see the changes. In Section 5, we conclude our report.

## 2 Environment Setup (Part 0, 1)

### 2.1 Hardware Configuration

We conducted all experiments on three Cloudlab nodes that share homogeneous hardware architecture. Each node contains an Intel Xeon E5-2630 v3 CPU which runs at 2.4 GHz. The CPU has 5 cores with 64 KB of L1 cache, 256 KB of L2 cache, and 20 MB of L3 cache. Each machine is equipped with 32 GB of DRAM. We mounted an extra disk with `ext4` file system as Cloudlab only provides 16 GB for local storage.

### 2.2 Software Configuration

In this section, we describe the details of our software setup. We first set up Apache Hadoop. The IP address (i.e., 10.10.1.1) and data path were set for the master node, along with the datanodes. We share all the necessary configurations with our own script file below. Figure 1 shows the overview of our Hadoop setup.

Listing 1: Hadoop Configuration

```
#!/bin/bash
cur=$PWD

## update/install basic dependencies
sudo apt update
sudo apt install openjdk-8-jdk

## mount disk
hadoop_data_path=/mnt/data/hdfs
spark_data_path=/mnt/data/spark
sudo mkfs.ext4 /dev/xvda4
sudo mkdir -p /mnt/data
sudo mount /dev/xvda4 /mnt/data
sudo mkdir ${hadoop_data_path} ${hadoop_data_path}/datanode ${hadoop_data_path}/namenode
sudo mkdir ${spark_data_path} ${spark_data_path}/logs
sudo chown -R hcha /mnt/data
sudo chmod -R 777 /mnt/data

## install hadoop
wget https://dlcdn.apache.org/hadoop/common/hadoop-3.2.2/hadoop-3.2.2.tar.gz
tar zvxf hadoop-3.2.2.tar.gz

## set hadoop path
hadoop_home=${cur}/hadoop-3.2.2
echo "PATH=$(hadoop_home)/bin:${hadoop_home}/sbin:\$PATH" >> ~/.profile
echo "export HADOOP_HOME=${hadoop_home}" >> ~/.bashrc
echo "export PATH=\${PATH}:\${HADOOP_HOME}/bin:\${HADOOP_HOME}/sbin" >> ~/.bashrc
source ~/. profile
source ~/.bashrc

## set java path
echo "export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/jre" >> ${hadoop_home}/etc/hadoop/hadoop-env.sh

## set node info
echo "10.10.1.2" > ${hadoop_home}/etc/hadoop/workers
echo "10.10.1.3" >> ${hadoop_home}/etc/hadoop/workers

## set namenode IP address
sed -i 's!<configuration>!<configuration>\n<property>\n<name>fs.default.name</name>\n<value>hdfs://10.10.1.1:9000
    </value>\n</property>!' $(hadoop_home)/etc/hadoop/core-site.xml;

## set data path for namenode and datanodes
namenode_dir=${hadoop_data_path}/namenode
datanode_dir=${hadoop_data_path}/datanode
sed -i 's!<configuration>!<configuration>\n<property>\n<name>dfs.namenode.name.dir</name>\n<value>/users/hcha/'$namenode_dir'
    /</value>\n</property>\n<property>\n<name>dfs.datanode.data.dir</name>\n<value>/users/hcha/'$datanode_dir'/</value>\n
    </property>!' $(hadoop_home)/etc/hadoop/hdfs-site.xml;
```

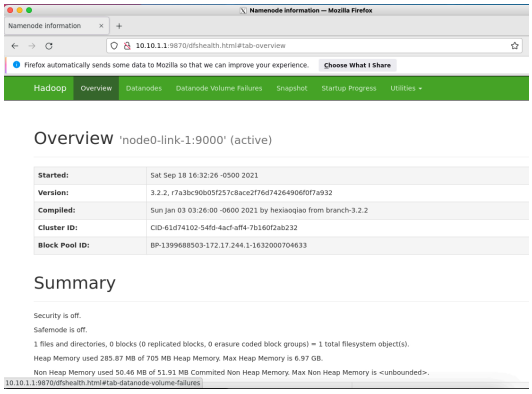



Figure 1: Hadoop Overview

Then, we installed Spark after setting the correct IP addresses for the Hadoop datanodes. Likewise, we provide our configuration script below. Figure 2 denotes the overview of our Spark setup.

Listing 2: Spark Configuration

```
#!/bin/bash

## install spark
wget https://dlcdn.apache.org/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2.tgz
tar zvxf spark-3.1.2-bin-hadoop3.2.tgz

## set Worker IPs
echo "10.10.1.2" > spark-3.1.2-bin-hadoop3.2/conf/workers.template
echo "10.10.1.3" >> spark-3.1.2-bin-hadoop3.2/conf/workers.template

## set spark local directory path
SPARK_CONF_DIR=spark-3.1.2-bin-hadoop3.2/conf
SPARK_LOCAL_DIR=/mnt/data/spark
HDFS_LOCAL_DIR=/mnt/data/hdfs
echo "spark.local.dirs ${SPARK_LOCAL_DIR}" >> ${SPARK_CONF_DIR}/spark-defaults.conf.template
echo "SPARK_LOCAL_DIRS=\"${SPARK_LOCAL_DIR}\"" >> ${SPARK_CONF_DIR}/spark-env.sh.template
echo "SPARK_PID_DIR=\"${SPARK_LOCAL_DIR}" >> ${SPARK_CONF_DIR}/spark-env.sh.template
echo "export SPARK_LOCAL_DIRS" >> ${SPARK_CONF_DIR}/spark-env.sh.template
echo "export SPARK_PID_DIR" >> ${SPARK_CONF_DIR}/spark-env.sh.template
echo "SPARK_JAVA_OPTS+=\" -Dspark.local.dir=${SPARK_LOCAL_DIR} -Dhadoop.tmp.dir=${HDFS_LOCAL_DIR}" >>
    ${SPARK_CONF_DIR}/spark-env.sh.template
echo "export SPARK_JAVA_OPTS" >> ${SPARK_CONF_DIR}/spark-env.sh.template
mv ${SPARK_CONF_DIR}/spark-env.sh.template ${SPARK_CONF_DIR}/spark-env.sh
sh ${SPARK_CONF_DIR}/spark-env.sh

echo "export SPARK_LOCAL_DIRS=${SPARK_LOCAL_DIR}" >> ~/.bashrc
source ~/.bashrc
```

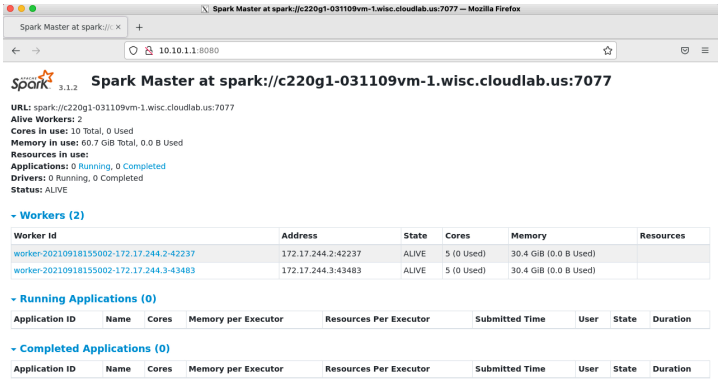



Figure 2: Spark Overview

# 3 A Simple Spark Application (Part 2)

In this section, we explain our simple spark application. The application was implemented using Python, which aimed to result a data output sorted firstly by country code followed by timestamp. The application first initializes the spark session, reads the input file, and writes the output after sorting. Figure 3 shows the first 10 lines of unsorted input data, and Figure 4 shows the first 10 lines of the sorted output after the execution of Listing 4. We do not evaluate its performance as it simply

sorts a small dataset. The implementation details are included in our assignment submission.

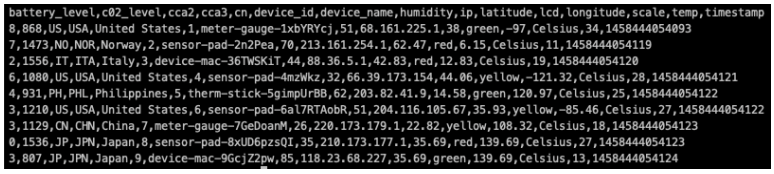



Figure 3: First 10 lines of input (unsorted)

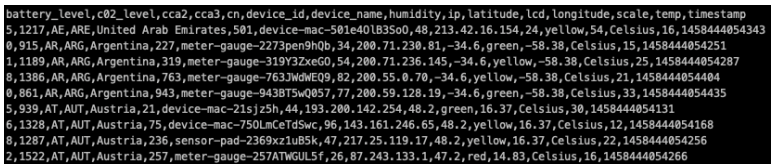



Figure 4: First 10 lines of output (sorted)

Listing 3: A Simple Spark Application Implementation

```
from pyspark.sql import SparkSession
import sys

def simple_application(input_path, output_path):
    # initialize spark sql instance
    spark = (SparkSession
             .builder
             .appName("part2")
             .getOrCreate())

    # load input data
    df = spark.read.option("header", True).csv(input_path)

    # sort the input data firstly by country code and then by timetstamp
    df_sorted = df.sort(['cca2', 'timestamp'], ascending=[True, True])

    # write output
    df_sorted.write.csv(output_path, header=True)

if __name__ == "__main__":
    print("Arguments", sys.argv)
    simple_application(sys.argv[1], sys.argv[2])
```

# 4 PageRank (Part 3)

PageRank is an algorithm developed by Google to measure the importance of website pages and rank those pages in search engine results. This algorithm considers the number and quality of links to a page to estimate the importance of each website. As more important websites are assumed to have more links with other websites, a recursive measure is used to define the importance of each website by calculating the links to other websites. In this section, we describe the PageRank algorithm, discuss variable factors that affect performance, and provide our evaluation results. Note that we conducted all the experiments with two worker nodes, each running 5 cores.

### 4.1 Baseline Algorithm (Task 1)

We implemented our baseline PageRank algorithm as shown in Listing 4. The algorithm first initializes the spark session and reads input files while mapping pages with their neighbors. Then based on the contributions given to the rank of other pages and neighbors, the algorithm calculates and updates the page ranks. After ten iterations, the application completes the procedure by writing the output.

Listing 4: PageRank Baseline

```
import time
import re
import sys
from operator import add
from pyspark.sql import SparkSession

# This code mainly referred to https://github.com/apache/spark/blob/master/examples/src/main/python/pagerank.py

def compute_probs(pages, rank):
    num_pages = len(pages)
    for page in pages:
        yield (page, rank / num_pages)

def parse_neighbors(pages):
    neighbors = re.split(r'\s+', pages)
    return neighbors[0], neighbors[1]

def run_pagerank(input_path, output_path, num_iters=10):
    spark = (SparkSession.
             builder.
             appName("PageRank").
             getOrCreate())

    # Loads input files
    lines = spark.read.text(input_path).rdd.map(lambda r: r[0])

    # Read pages in input files and initialize their neighbors
    links = lines.map(lambda pages: parse_neighbors(pages)).distinct().groupByKey()

    # Initialize ranks
    ranks = links.map(lambda page_neighbors: (page_neighbors[0], 1.0))

    # Calculates and updates page ranks up to num_iters
    for iteration in range(num_iters):
        # Calculates page contributions to the rank of other pages
        probs = links.join(ranks).flatMap(
                lambda page_pages_rank: compute_probs(page_pages_rank[1][0], page_pages_rank[1][1]))

        # Re-calculates page ranks based on neighbor contributions.
        ranks = probs.reduceByKey(add).mapValues(lambda rank: rank * 0.85 + 0.15)

    # Write output
    ranks.saveAsTextFile(output_path)
    spark.stop()

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print("Usage: pagerank.py <input> <output>")
        sys.exit(-1)

    run_pagerank(sys.argv[1], sys.argv[2])
```

**Observation** Figure 5 shows the total execution time with the two datasets: 1) web-BerkStan 2) enwiki-pages-articles. As the raw data size of the enwiki-pages-articles dataset is much larger than the web-BerkStan dataset, it takes a significant amount of additional time to complete execution.

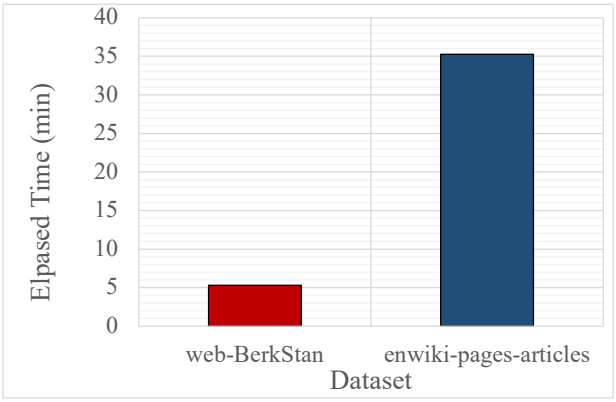



Figure 5: PageRank baseline execution on two datasets

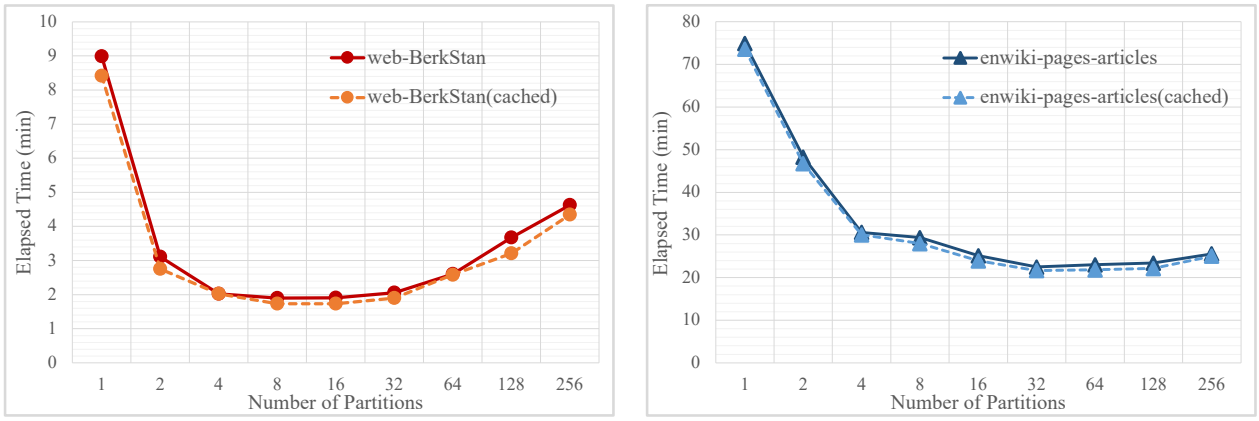
The baseline performance of the two datsets are shown to provide reference for analyzing the results in our next experiments.

### 4.2 Partitions and Caching (Task 2, 3)

**Partitions (Task 2)** The performance of Spark applications depends a great extent on the method of partitioning and caching. Spark partitions act as a logical chunk of large distributed datasets, each storing RDDs in different cluster nodes. These partitions enable parallel data processing on a cluster. However, the number of partitions should be selected carefully, considering trade-offs of task scheduling and network traffic. If an RDD has too many partitions, task scheduling may take more time than the actual execution time, while some partitions may contain no data at all. Moreover, too many partitions cause increasingly frequent data traffic among worker nodes. On the other hand, having too little partitions may result in lower concurrency and inefficient resource utilization as some worker nodes would stay idle while the others are consistently busy.

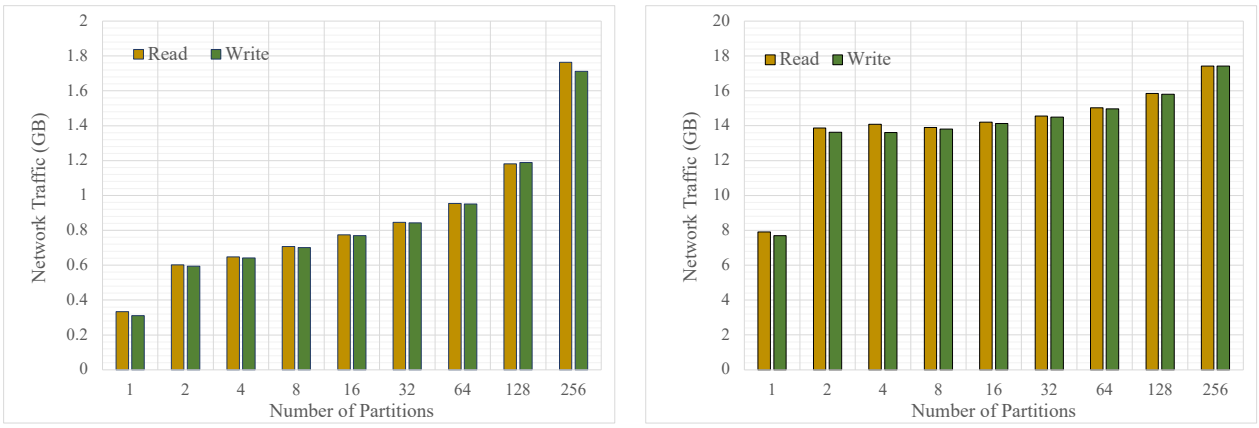
**Caching (Task 3)** Caching stores all the RDDs in-memory while using it across parallel operations. RDDs can be cached in the worker nodes for immediate reuse, resulting in faster iterations as applications can refer to data without reloading from disk to memory. This technique of saving intermediate results for data to be repetitively used upon request allows efficient time and memory management for interactive algorithms, reducing the computation overhead.

**Observation** Figure 6 shows the PageRank execution time corresponding to various numbers of partitions on two different datasets: 1) web-BerkStan 2) enwiki-pages-articles. The graph compares the execution time of two conditions, one with cache enabled and the other disabled. We first focus on the effects of the *number of partitions*. As we increase the number of partitions, their total execution time decreases because of the benefits that come from higher concurrency. However, as illustrated in Figure 6(a), the execution time rather increases with larger number of partitions. We attribute the




(a) web-BerkStan dataset (b) enwiki-pages-articles dataset

Figure 6: PageRank execution on different datasets with varying number of partitions



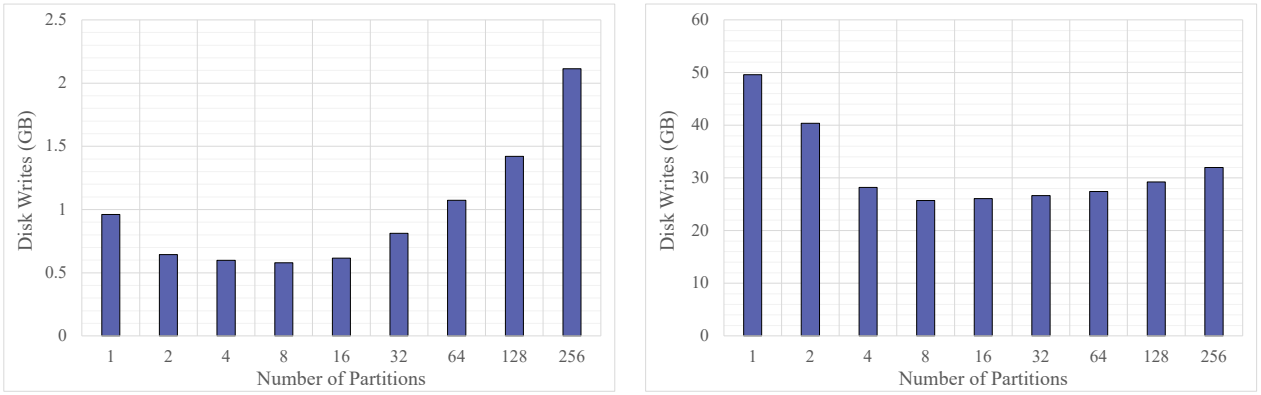

(a) web-BerkStan dataset (b) enwiki-pages-articles dataset

Figure 7: PageRank network traffic in workers on different datasets with varying number of partitions

reason to the larger overheads in task scheduling and network transmission.

Figure 7 elaborates the total network traffic in workers during the application execution on the two datasets with varying number of partitions. The results are based on the snapshots of the network statistics before and after each execution. As described earlier in Section 4.2, too many partitions results in excessive network traffic during shuffle phase, which causes a bottleneck in the network and slows down the overall performance. Although it is possible to achieve higher parallelism with a larger number of partitions, there exists a trade-off between concurrency and data transmission as the network bandwidth is limited. We note that the increase in runtime with a higher number of partitions in Figure 6(b) is less noticeable as it handles a larger amount of data.

Figure 8 reports the amount of disk writes in workers withing the two datasets (i.e., web-BerkStan and enwiki-pages-articles) when varying the number of partitions. The results of the figures show the




(a) web-BerkStan dataset

(b) enwiki-pages-articles dataset

Figure 8: PageRank disk writes in workers on different datasets with varying number of partitions

quantity of disk writes reducing as the number of partitions increase. This finding can be supported by the properties of advanced parallelism and high memory capacity, as these features enable intermediate data to be safely stored in memory. However, Figure 8(a) denotes an inflation of the amount of disk writes when the number of partitions become excessive. We attribute the reason to the granularity of disk writes, which is set to 512 bytes in our cluster configuration.

Although an increased number of partitions allow high parallelism, an inadequate size of data causes write amplification. This amplification occurs not only when spilling data, but also when storing intermediate data to guarantee fault tolerance. On the execution of a larger dataset, a slight increase of disk writes is observed with a larger number of partitions as shown in Figure 8(b). This observation is believed to occur as the size of the dataset is notably larger compared to the web-BerkStan dataset.

Unlike the effect of varying the number of partitions discussed above, *caching* did not bring significant performance improvement despite our expectations. We believe that it is due to large RDDs which cannot fit in memory. This overflow causes an inevitable spill to disk, hence not being able to leverage the advantage of caching.

### 4.3 Fault Tolerance (Task 4)

Spark ensures fault tolerance by logging coarse-grained transformations of RDDs, called *lineage*. By only storing the transformations rather than the actual data, it efficiently deals with worker failures without costly replication. Upon a node failure, the master reschedules data processing of the lost RDDs to the other workers, and they can be recovered using the lineage of the lost RDDs without having to roll back the whole program. The dependencies of RDDs are classified into two types: narrow dependencies, which refer to a partition of the parent RDD that is used by at most one partition of the child RDD, and wide dependencies which refer to a partition of the parent RDD that is used by multiple child RDDs. Narrow dependencies can be recovered with pipe-lined execution as

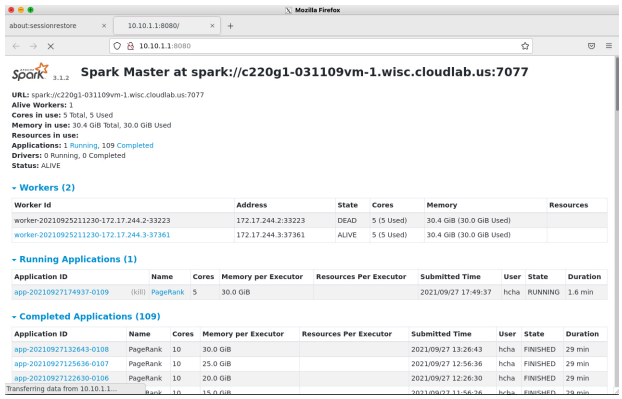
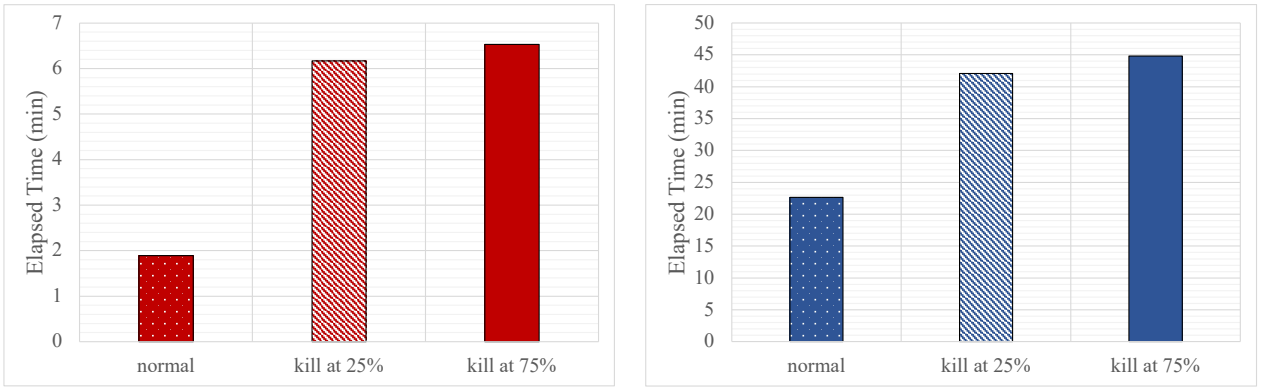



Figure 9: Spark health page with a dead worker

they are independent, while wide dependencies require more complicated recovery protocol depending on the complexity of the parent-child dependencies.



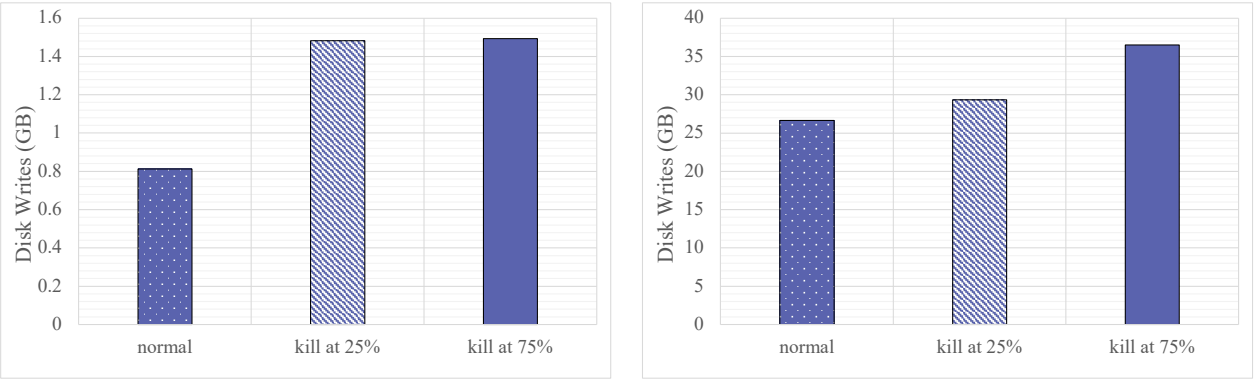

(a) web-BerkStan dataset

(b) enwiki-pages-articles dataset

Figure 10: PageRank execution on different datasets with worker failures

**Observation** Figure 10 shows the PageRank execution time on two datasets with three execution scenarios; normal completion, completions with a worker process killed at 25 % and 75 % of its lifetime. We observed a worker failure at 75 % causes longer execution time. As discussed in Section 4.3, failures at later times are prone to have more complex wide dependencies. Moreover, a node failure can cause the loss of some partitions from all the ancestors of an RDD which requires complete re-execution.

Figure 11 reports the amount of disk writes on the two datasets for the three execution scenarios. The increased amount of writes depicts the storage overhead of re-execution of dependent RDDs, each of which contributes to the delayed execution time.




(a) web-BerkStan dataset

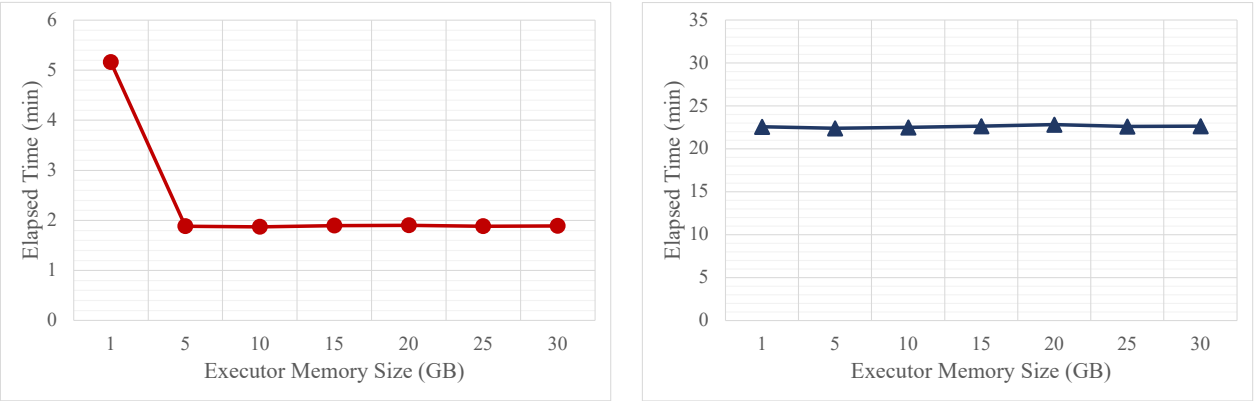
(b) enwiki-pages-articles dataset

Figure 11: PageRank disk writes on different datasets with worker failure

### 4.4 Memory Capacity

Spark applications are also sensitive to the memory capacity. In-memory data has to be spilled to disk when there is not enough memory, while tasks re-load the data from disk when necessary.

**Observation** We tested the performance through varying the memory size of the executor with 32 partitions. Improved performance can clearly be observed through increasing the memory size as shown in Figure 12(a). This observation can be explained by the nature of Spark keeping persistent RDDs in memory until the maximum capacity of the RAM, then spilling the overloads to the disk. When dealing with large datasets, this process can accumulate to the overload of performance. However, Figure 12(b) shows consistant performance regardless of available memory sizes. We believe it is because the execution time of a large dataset is determined less by the available memory but more by the amount of data getting shuffled over the network.




(a) web-BerkStan dataset

(b) enwiki-pages-articles dataset

Figure 12: PageRank execution on different datasets with varying executor memory sizes

## 5 Conclusion

Through this assignment, we learned how to store and process large datasets ranging in size and the use of applications to execute big data workloads. We explored the usage of distributed processing systems and the components to optimize the execution for best performance against large datasets. We specifically deployed and configured Apache Hadoop as the building block, upon where Apache Spark was built as the execution tool. We then implemented applications where Spark reads and writes from HDFS(Hadoop Distributed File System) data and interacts to enrich the processing capabilities. In order to process large datasets swiftly and effectively, a system should well balance data across tasks and minimize the possibility of high overhead and large delay. We explored these requirements through partitioning, caching in-memory, and fault tolerance. We learned how the flexibility in the number of partitions, in-memory storage, and failure recovery latency impacts the overall performance of big data systems.

## 6 Contribution

**Changho Shin** Experiment design, Part2 implementation, Part3 PageRank Implementation

**Hokeun Cha** Cluster setup, writing scripts, running experiments

**Christine Lee** Deliverable write-up