CS553 Homework #5

Hashgen, Sort, & Search on Hadoop/Spark

Instructions:

- *Assigned date: Monday November 10th, 2025*
- *Due date: 11:59PM on Tuesday November 25th, 2025*
- *Maximum Points: 100*
- *This assignment can be done in groups of 2 students*
- *Please post questions on Canvas*
- *Only a softcopy submission is required through GIT: push changes to GIT repository*
- *Late submission will be penalized at 20%; late submissions beyond 24 hours will receive 0 points*

1. Introduction

The goal of this programming assignment is to enable you to gain experience programming with:

- The Hadoop framework (http://hadoop.apache.org/)
- The Spark framework (http://spark.apache.org/)

In Homework #4, you implemented a hash generator that sorted the hashes. You will now expand into implementing parallel hash generation and parallel sort with Hadoop and with Spark.

2. Your Assignment

This programming assignment covers hashgen, sort, and search through Hadoop and Spark on multiple nodes. You must use a Chameleon node using Bare Metal Provisioning (https://www.chameleoncloud.org). You must deploy Ubuntu Linux 24.04 using "compute-skylake" nodes; other instance types can be used if no Skylake nodes are available. Once you create a lease (up to 7 days are allowed), and start your 1 physical node, and Linux boots, you will find yourself with a physical node with 24 CPU cores, 48 hardware threads, 128GB of memory, and 250GB SSD hard drive. You will install your favorite virtualization tools (e.g. virtualbox, LXD/KVM, qemu), and use it to deploy three different type of VMs with the following sizes: tiny.instance (4-cores, 4GB ram, 10GB disk), small.instance (4-cores, 4GB ram, 30GB disk), and large.instance (32-cores, 32GB ram, 240GB disk).

This assignment will be broken down into several parts, as outlined below:

Hadoop File System and Hadoop Install: Download, install, configure, and start the HDFS system (that is part of Hadoop, https://hadoop.apache.org) on a virtual cluster with 1 large.instance + 1 tiny.instance, and then again on a virtual cluster with 8 small.instances + 1 tiny.instance. You must set replication to 1 (instead of the default 3), or you won't have enough storage capacity to conduct your experiments on the 64GB dataset.

Datasets: Once HDFS is operational, you must generate your dataset. Since ; you will create 3 workloads: data-16GB.bin, data-32GB.bin, and data-64GB.bin, for K=30, K=31, and K=32. You may not have enough room to store them all, and run your compute workloads. Make sure to cleanup after each run. Remember that you will typically need 3X the storage, as you have the original input data (1x), temporary data (1x), and output data (1x). Configure Hadoop to run on the virtual cluster, on 1 large.instance + 1 tiny.instance as well as the separate installation on 6 small.instances + 1 tiny.instance. The tiny.instance will run parts of Hadoop (e.g. name node, scheduler, etc).

Spark Install: Download, install, configure, and start Spark (https://spark.apache.org). Note that you will need the HDFS installation for Hadoop to work from and to.

Hashgen: Run the hashgen you implemented in HW4 on the small.instance and large.instance defined above on the 16GB, 32GB, and 64 GB datasets respectively. If you don't have a working version, write a paragraph about why you could not complete the hashgen implementation, even a simple version that sequentially generated the hashes and then sequentially sorted them. A naïve implementation of hashgen should have been possible in 100 lines of code or less with a few hours of work. Detail your issues with why you could not accomplish this.

Vault: Run the vault provided in your repo on the small.instance and large.instance defined above on the 16GB, 32GB, and 64 GB datasets respectively. You can use the following commands:

16GB dataset with 2GB RAM:

```
./vaultx -t 32 -i 1 -m 2048 -k 30 -g data-16GB.tmp -f data-16GB.bin
```

64GB dataset with 16GB RAM:

```
./vaultx -t 32 -i 1 -m 16384 -k 32 -g data-64GB.tmp -f data-64GB.bin
```

You can verify the data was generated correctly with:

```
./vaultx -f data-64GB.bin -v true
```

Hadoop Hashgen/Sort: Implement the HadoopSort application (you can use Java, Python, or SCALA). You must generate Blake3 hashes to be stored in HDFS. You can use any Blake3 libraries for this assignment. Here are some good libraries for Java and Python. You can use Class Blake3 in the package org.apache.commons.codec.digest; more information can be found at: https://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/digest/Blake3.html. For Python, you can use the Python bindings for the Rust blake3 crate at https://pypi.org/project/blake3/. For SCALA, you can use the optimized blake3 implementation for scala, scala-js and scala-native: https://index.scala-lang.org/catap/scala-blake3. You must retrieve a 10-byte hash from Blake3 using a 6-byte NONCE input (generate random for each invocation). The final value to write into HDFS should be a 16-byte value (10-byte hash followed by 6-byte NONCE). You must specify the number of reducers to ensure you use all the resources of the 6-node cluster. You must generate the data, store it on HDFS, then read it back from HDFS, sort it, and then store the sorted data in HDFS and validate it. Measure the time from beginning to end, including hash generation, writing to HDFS, reading from HDFS, sorting, and writing the final result to HDFS; do not include the time to verify the data has been sorted correctly.

Hadoop Verify: Implement a verification program that reads the data from HDFS and verifies that it has been sorted. This can be a simple Java/Python program that can read HDFS and sequentially verify the data is sorted.

Spark Sort: Implement the SparkSort application. Make sure to use RDD to speed up the sort through in-memory computing. You must generate the hashes, write them to HDFS, read the data back from HDFS, make use of RDD, sort, and write the sorted data back to HDFS, and finally validate the sorted data with valsort. Measure the time from hash generation all the way to writing the sorted data back to; do not include the time to validate the sorted data.

Performance: Compare the performance of your own hashgen (from HW4), the vaultx provided by the professor, Hadoop Sort, and Spark Sort on a 8-node cluster with the 16GB, 32GB, and 64GB datasets. Fill in the table below, and then derive new tables or figures (if needed) to explain the results. Your time should be reported in seconds.

Complete Table 1 outlined below. Perform the experiments outlined above, and complete the following table:

Table 1: Performance evaluation (measured in seconds); each instance needs a tiny.instance for the name node

Experiment	hashgen	vaultx	Hadoop Sort	Spark Sort
1 small.instance, 16GB *dataset, 2GB RAM*				
1 small.instance, 32GB *dataset, 2GB RAM*				

1 small.instance, 64GB *dataset, 2GB RAM*	N/A	N/A		
1 large.instance, 16GB *dataset, 16GB RAM*				
1 large.instance, 32GB *dataset, 16GB RAM*				
1 large.instance, 64GB *dataset, 16GB RAM*				
8 small.instances, 16GB *dataset*	N/A	N/A		
8 small.instances, 32GB *dataset*	N/A	N/A		
8 small.instances, 64GB *dataset*	N/A	N/A		

Some of the things that will be interesting to explain are: how many threads, mappers, reducers, you used in each experiment; how many times did you have to read and write the dataset for each experiment; what speedup and efficiency did you achieve?

For the 64GB workload, monitor the disk I/O speed (in MB/sec), memory utilization (GB), and processor utilization (%) as a function of time, and generate a plot for the entire experiment. Here is an example of a plot that has cpu utilization and memory utilization (https://i.stack.imgur.com/dmYAB.png), plot a similar looking graph but with the disk I/O data as well as a 3rd line. Do this for both shared memory benchmark (your code) and for the Linux Sort. You might find some online info useful on how to monitor this type of information (https://unix.stackexchange.com/questions/554/how-to-monitor-cpu-memory-usage-of-a-single-process). For multiple instances, you will need to combine your monitor data to get an aggregate view of resource usage. Do this for all four versions of your sort. After you have all six graphs (2 system configurations [small.instance and large.instance] and 4 different sort techniques), discuss the differences you see, which might explain the difference in performance you get between the two implementations. Make sure your data is not cached in the OS memory before you run your experiments.

Note that you can set the memory limit to be 2GB for the small.instance and 16GB for the large.instance. What conclusions can you draw? Which seems to be best at 1 node scale (1 large.instance)? Is there a difference between 1 small.instance and 1 large.instance? How about 8 nodes (8 small.instance)? What speedup do you achieve with strong scaling between 1 to 8 nodes? What speedup do you achieve with weak scaling between 1 to 8 nodes (you may need to run K=29 on 1 small instance, and compare to K=32 on 8 small instances)? How many small.instances do you need with Hadoop to achieve the same level of performance as your hashgen or vaultx programs? How about how many small.instances do you need with Spark to achieve the same level of performance as you did with your hashgen/vault? Does Spark seem to offer any advantages over Hadoop for this application? Can you predict which would be best if you had 100 small.instances? How about 1000?

Search: Once you have everything done, we want to search through the file. You are going to generate random search queries based on the difficulty, and execute the search in a distributed fashion, and report statistics at the end of the search.

```
cc@hw4-raicu-skylake:~/vault$ ./vaultx -k 26 -f k26-memo.x -s 10 -q 3 -d true
searches=10 difficulty=3
Parsed k                     : 26
Nonce Size                   : 6
Record Size                  : 16
Hash Size                    : 10
On-disk Record Size          : 16
Number of Buckets            : 16777216
Number of Records in Bucket  : 4
Number of Hashes       : 67108864
File Size to be read (bytes) : 1073741824
File Size to be read (GB)    : 1.000000
```

```
Actual file size on disk     : 1073741824 bytes
[0] 1213ae MATCH 1213aef913a754c0aee2 185081113280512 time=0.053 ms comps=3
MATCH 1213aea8365ef5c396d8 153924027875328 time=0.053 ms comps=3
MATCH 1213aeb822e61a7e3ff7 161018189774848 time=0.053 ms comps=3
[1] c0f629 MATCH c0f629767847ab1875da 144034999828480 time=0.042 ms comps=2
MATCH c0f629d980132fa9c1a0 214554336493568 time=0.042 ms comps=2
[2] 08259a MATCH 08259a407625ea40f530 77072651452416 time=0.041 ms comps=4
MATCH 08259a3f70d4a742e072 58336628244480 time=0.041 ms comps=4
MATCH 08259a99379ef1a9efb3 48278586654720 time=0.041 ms comps=4
MATCH 08259a94723745fba6ec 172800476184576 time=0.041 ms comps=4
[3] deb0c7 MATCH deb0c7cfefbe66aa729b 52254786715648 time=0.040 ms comps=4
MATCH deb0c765426831f044be 10813519757312 time=0.040 ms comps=4
MATCH deb0c754426e38fb0e46 74989642514432 time=0.040 ms comps=4
MATCH deb0c7694ed820fe4941 29447235436544 time=0.040 ms comps=4
[4] 6ede11 MATCH 6ede11e8594021311f93 252420261216256 time=0.041 ms comps=4
MATCH 6ede11973a3e379fa70a 245398073638912 time=0.041 ms comps=4
MATCH 6ede11665a9f2e53cdbb 271868913254400 time=0.041 ms comps=4
MATCH 6ede11a889620aa92f47 98897779359744 time=0.041 ms comps=4
[5] 51f748 MATCH 51f748283b73e213e1b9 56298481778688 time=0.040 ms comps=4
MATCH 51f74873c2839b4e9bca 4112498425856 time=0.040 ms comps=4
MATCH 51f74820d16012dffda9 169079474421760 time=0.040 ms comps=4
MATCH 51f7489f406b08a47b26 104413758947328 time=0.040 ms comps=4
[6] 329b1f MATCH 329b1f159c3ba5023832 226377760571392 time=0.040 ms comps=4
MATCH 329b1fc9247f229b3c3f 256692545650688 time=0.040 ms comps=4
MATCH 329b1ff9ca18daa70a40 122168046321664 time=0.040 ms comps=4
MATCH 329b1ff34468c61ba528 127185390272512 time=0.040 ms comps=4
[7] 21aefa MATCH 21aefa03676de73cc537 180669460381696 time=0.041 ms comps=1
[8] 9e19a3 MATCH 9e19a3c2a0ac1cba1ef0 55166271422464 time=0.053 ms comps=3
MATCH 9e19a3ac1dc534aa8820 201843733102592 time=0.053 ms comps=3
MATCH 9e19a3e15fe2834bea19 79994940424192 time=0.053 ms comps=3
[9] 66a446 MATCH 66a44638447bf016491c 234814452203520 time=0.041 ms comps=4
MATCH 66a4468c151addf14d5d 162249603874816 time=0.041 ms comps=4
MATCH 66a446f61984400b77bb 262952477392896 time=0.041 ms comps=4
MATCH 66a4461ea8513780f333 142229553283072 time=0.041 ms comps=4
Search  Summary:  requested=10  performed=10  found_queries=10  total_matches=33  notfound=0
total_time=0.000433    s    avg_ms=0.043    ms    searches/sec=23081.842    total_seeks=10
avg_seeks_per_search=1.000            total_comps=33            avg_comps_per_search=3.300
avg_matches_per_found=3.300


cc@hw4-raicu-skylake:~/vault$ ./vaultx -k 26 -f k26-memo.x -s 10 -q 4 -d true
searches=10 difficulty=4
Parsed k                     : 26
Nonce Size                   : 6
Record Size                  : 16
Hash Size                    : 10
On-disk Record Size          : 16
Number of Buckets            : 16777216
Number of Records in Bucket  : 4
Number of Hashes      : 67108864
File Size to be read (bytes) : 1073741824
File Size to be read (GB)    : 1.000000
Actual file size on disk     : 1073741824 bytes
[0] 3267c542 NOTFOUND time=0.052 ms comps=0
[1] 26992e1a NOTFOUND time=0.042 ms comps=4
[2] 6caaeaad NOTFOUND time=0.041 ms comps=4
[3] 5353ac49 NOTFOUND time=0.041 ms comps=4
```

```
[4] fb623cf5 NOTFOUND time=0.041 ms comps=4
[5] b7c255f2 NOTFOUND time=0.041 ms comps=2
[6] acbbc8a2 NOTFOUND time=0.040 ms comps=3
[7] 5c5f7153 NOTFOUND time=0.040 ms comps=4
[8] bf143a9a NOTFOUND time=0.041 ms comps=2
[9] ffd3869c NOTFOUND time=0.040 ms comps=3
Search  Summary:  requested=10  performed=10  found_queries=0  total_matches=0  notfound=10
total_time=0.000419    s    avg_ms=0.042    ms    searches/sec=23893.378    total_seeks=10
avg_seeks_per_search=1.000           total_comps=30            avg_comps_per_search=3.000
avg_matches_per_found=0.000
```

You must run a number of search workloads to fill in the following table for each of your approaches (hashgen, vaultx, Hadoop, and Spark); hashgen and vaultx should run on 1 large.instance; Hadoop and Spark should be run on 8 small.instances; please compare and contrast the hashgen/vaultx search and the Hadoop/spark search performance, and why the results make sense:

Approach	K	Difficulty	Number of Searches	Total Time for all searches	Time (ms) / search	Throughput search/sec	Searches Found	Searches Not Found
hashgen	30	3	1000					
hashgen	30	4	1000					
hashgen	31	3	1000					
hashgen	31	4	1000					
hashgen	32	3	1000					
hashgen	32	4	1000					
vaultx	30	3	1000					
vaultx	30	4	1000					
vaultx	31	3	1000					
vaultx	31	4	1000					
vaultx	32	3	1000					
vaultx	32	4	1000					
Hadoop	30	3	1000					
Hadoop	30	4	1000					
Hadoop	31	3	1000					
Hadoop	31	4	1000					
Hadoop	32	3	1000					
Hadoop	32	4	1000					
Spark	30	3	1000					
Spark	30	4	1000					
Spark	31	3	1000					
Spark	31	4	1000					
Spark	32	3	1000					
Spark	32	4	1000					

3. What you will submit

The grading will be done according to the rubric below:

- Hadoop (installation/config) implementation/scripts: 20 points
- Spark (installation/config) implementation/scripts: 20 points
- Performance evaluation (data): 30 points
- Performance evaluation (Q&A and explanations): 30 points

The maximum score that will be allowed is 100 points.

You are to write a report (hw5_report.pdf). Add a brief description of the problem, methodology, and runtime environment settings. You are to fill in the table on the previous page. Please explain your results, and explain the difference in performance? Include logs from your application as well as the verification of the generated data that clearly shows the completion of the sort invocations with clear timing information and experiment details; include separate logs for hashgen, vault, Hadoop sort, and Spark sort, for the 64GB dataset. As part of your submission you need to upload to your private git repository your run scripts, build scripts, the source code for your implementation (Hadoop sort and spark sort), configuration files for both Hadoop and Spark, the report, a readme file (with how to build and use your code), and several log files. Make sure to answer all the questions posed from Section 2. Some of your answers might require a graph or table with data to substantiate your answer. Here are the specific files you need to submit:

- build.xml (Ant) / pom.xml (Maven)
- HadoopVault.java
- SparkVault.java
- Scripts
- Hw5_report.pdf
- readme.txt
- hashgen64GB.log
- vault64GB.log
- hadoop64GB.log
- spark64GB.log
- core-site.xml (Hadoop config file)
- hdfs-site.xml (Hadoop config file)
- yarn-site.xml (Hadoop config file)
- mapred-site.xml (Hadoop config file)
- spark-env.sh (Spark environment variables)

You will have to submit your solution to a private git repository created for you. The repository is create through GitHub Classroom and you will need to accept the assignment before you can clone it, by accessing this invitation link: https://classroom.github.com/a/I7Ayx-Ld. The first time you access an invitation link the system will ask you to identify yourself. Please select the identifier that contains your school email address. After that you can clone the repository. Then you will have to add or update your source code, documentation, and report. All repositories will be collected automatically at midnight on the day of the deadline, and then one last time 24 hours after the deadline. The timestamp on your git commits will be used to determine if the submission is on-time; do not make edits to your repository after the deadline. If you cannot access your repository, contact the TAs. You can find a git cheat sheet here: https://www.git-tower.com/blog/git-cheat-sheet/

Submit step: code/report through GIT.

Grades for late programs will be lowered 20%; no submission will be accepted more than 24 hours late