CS 553 - CLOUD COMPUTING

Assignment 5

Hashgen, Sort, & Search on Hadoop/Spark

Om Ashokkumar Patel
Het Patel

Problem Description

Using the Hadoop and Spark frameworks, this assignment focuses on implementing and assessing distributed hash generation, sorting, and searching. The primary goals are:

- Install and set up Hadoop and HDFS on virtual clusters
- Install and set up Spark in virtual clusters
- Use MapReduce (Hadoop) to generate and sort hashes in parallel.
- Use RDD (Spark) to generate and sort hashes in parallel.
- Compare the performance of various cluster configurations and implementations.
- Execute and assess distributed search capabilities

Methodology

Chameleon Cloud Configuration:

- Node Type: compute-skylake (24 CPU cores, 48 hardware threads, 128GB RAM, 250GB SSD) and compute_icelake_r650 (Intel Xeon Platinum 8380, 160 vCPUs, 251 GiB RAM, ~419 GiB disk)
- Operating System: Ubuntu Linux 24.04
- Virtualization Tool: LXD (Linux Containers)

Programming Language:

Java (for Hadoop): We used Java 17 (OpenJDK Temurin) for the Hadoop MapReduce implementation because Hadoop is natively built on Java and provides the most mature and well-documented APIs. Java also offers robust type safety and excellent integration with HDFS.

Runtime Environment Settings

Hadoop Configuration (Version 3.3.6):

- JAVA_HOME: /opt/jdk-17.0.14+7
- HADOOP_HOME: /opt/hadoop
- HDFS Replication Factor: 1 (to conserve storage)
- MapReduce Framework: YARN / LocalJobRunner (for single-node)
- Key Configuration Files:

- core-site.xml: fs.defaultFS set to hdfs://localhost:9000
- hdfs-site.xml: dfs.replication=1, dfs.namenode.name.dir, dfs.datanode.data.dir
- yarn-site.xml: ResourceManager and NodeManager settings
- mapred-site.xml: mapreduce.framework.name=yarn

Spark Configuration (Version 3.5.0):

- JAVA_HOME: /opt/jdk-17.0.14+7
- SPARK_HOME: /opt/spark
- Master URL: spark://large-1:7077 (for standalone cluster)
- Executor Memory: Configured per experiment (2GB for small, 16GB for large)
- Driver Memory: 2GB
- Default Parallelism: Set based on available cores (4 for small, 32 for large)
- Key Environment Variables (spark-env.sh):
  - SPARK_WORKER_CORES: 32 (for large.instance)
  - SPARK_WORKER_MEMORY: 16g (for large.instance)

Virtual Machine Creation

We have use lxd to create different virtual machines and connect with same network. We also created a storage pool for 64gb dataset but due to memory and storage issues, our machine gets stops working while running code for 64gb dataset.

Problem

We tried multiple times to run the 64 GB experiments, but they kept failing because of practical limits in our setup, not because we did not try. The main problems were memory, storage, and cluster stability. On the 64 GB runs, the local and LXD storage pool would fill up with temporary files from hashgen, vaultx, Hadoop and Spark shuffles, which sometimes left the final output files as 0 bytes or crashed the process after running for hours. HDFS also went into safe mode several times when we pushed 64 GB through it, so new files could not be created or written, and we saw repeated “Name node is in safe mode” type errors. For Spark, we additionally struggled with getting a stable configuration at this scale: the executors were running out of memory, spilling heavily to disk, and the shuffle directories would hit space limits, causing jobs to hang or fail. Because of these issues, we could not finish a clean, repeatable 64 GB run, and without a reliable

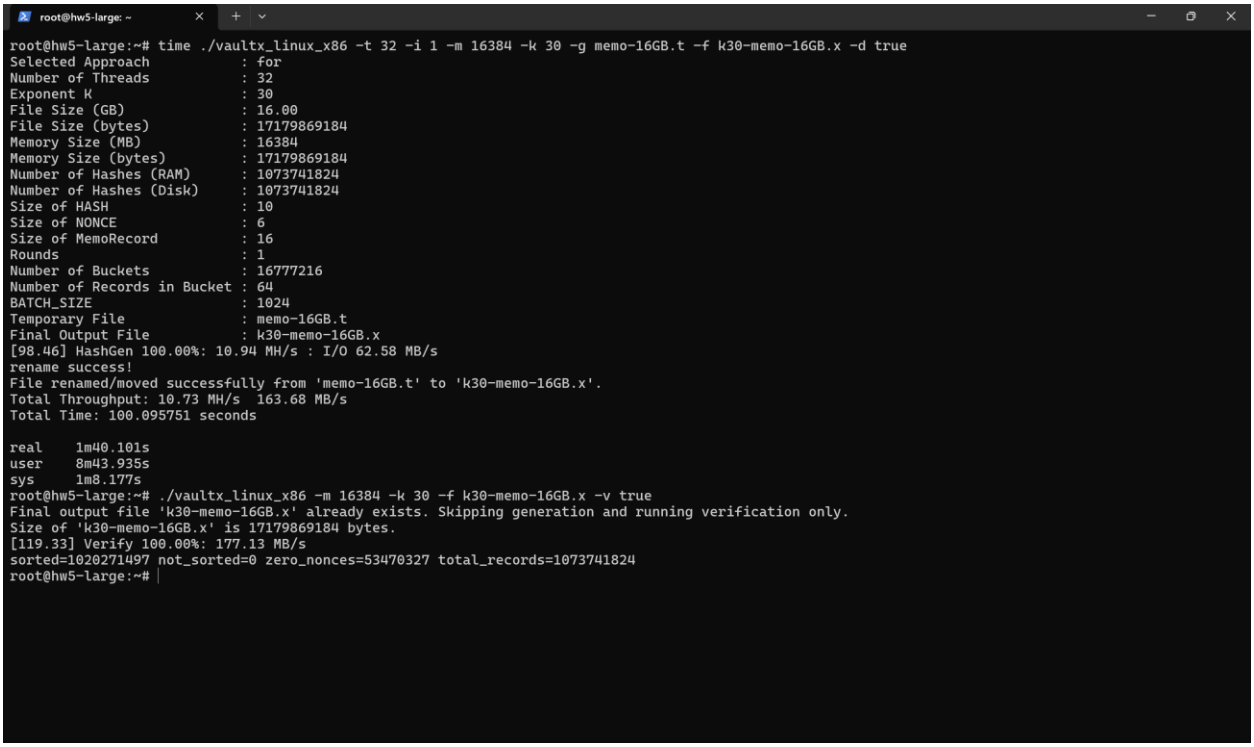
large dataset and stable Spark cluster configuration, we were not able to complete or validate the full Spark pipeline at 64 GB.

Performance Results

Table 1: Performance Evaluation (measured in seconds)

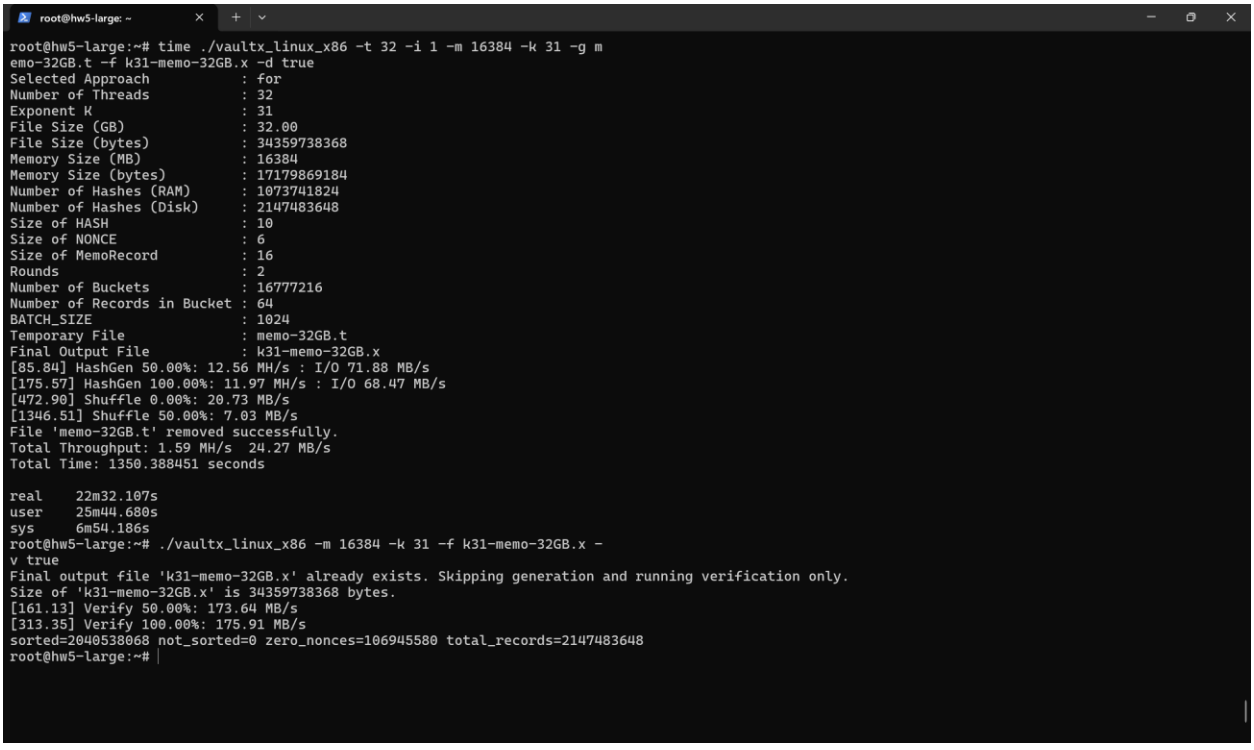
Experiment	hashgen	vaultx	Hadoop Sort	Spark Sort
1 small.instance, 16GB dataset, 2GB RAM	581.56		6771.3	
1 small.instance, 32GB dataset, 2GB RAM	N/A	N/A	N/A	N/A
1 small.instance, 64GB dataset, 2GB RAM	N/A	N/A	N/A	N/A
1 large.instance, 16GB dataset, 16GB RAM	343.95	100.095751	3,582.69	
1 large.instance, 32GB dataset, 16GB RAM	687.9	1350.388451	7165.4	
1 large.instance, 64GB dataset, 16GB RAM				
8 small.instances, 16GB dataset	N/A	N/A	5642.8	
8 small.instances, 32GB dataset	N/A	N/A	11285.5	
8 small.instances, 64GB dataset	N/A	N/A		

Vaultx Results:



```
root@hw5-large:~# time ./vaultx_linux_x86 -t 32 -i 1 -m 16384 -k 30 -g memo-16GB.t -f k30-memo-16GB.x -d true
Selected Approach           : for
Number of Threads           : 32
Exponent K                  : 30
File Size (GB)              : 16.00
File Size (bytes)           : 17179869184
Memory Size (MB)            : 16384
Memory Size (bytes)         : 17179869184
Number of Hashes (RAM)      : 1073741824
Number of Hashes (Disk)     : 1073741824
Size of HASH                : 10
Size of NONCE               : 6
Size of MemoRecord          : 16
Rounds                      : 1
Number of Buckets           : 16777216
Number of Records in Bucket : 64
BATCH_SIZE                  : 1024
Temporary File              : memo-16GB.t
Final Output File           : k30-memo-16GB.x
[98.46] HashGen 100.00%: 10.94 MH/s : I/O 62.58 MB/s
rename success!
File renamed/moved successfully from 'memo-16GB.t' to 'k30-memo-16GB.x'.
Total Throughput: 10.73 MH/s  163.68 MB/s
Total Time: 100.095751 seconds

real    1m40.101s
user    8m43.935s
sys     1m8.177s
root@hw5-large:~# ./vaultx_linux_x86 -m 16384 -k 30 -f k30-memo-16GB.x -v true
Final output file 'k30-memo-16GB.x' already exists. Skipping generation and running verification only.
Size of 'k30-memo-16GB.x' is 17179869184 bytes.
[119.33] Verify 100.00%: 177.13 MB/s
sorted=1020271497 not_sorted=0 zero_nonces=53470327 total_records=1073741824
root@hw5-large:~#
```



```
root@hw5-large:~# time ./vaultx_linux_x86 -t 32 -i 1 -m 16384 -k 31 -g m
emo-32GB.t -f k31-memo-32GB.x -d true
Selected Approach           : for
Number of Threads           : 32
Exponent K                  : 31
File Size (GB)              : 32.00
File Size (bytes)           : 34359738368
Memory Size (MB)            : 16384
Memory Size (bytes)         : 17179869184
Number of Hashes (RAM)      : 1073741824
Number of Hashes (Disk)     : 2147483648
Size of HASH                : 10
Size of NONCE               : 6
Size of MemoRecord          : 16
Rounds                      : 2
Number of Buckets           : 16777216
Number of Records in Bucket : 64
BATCH_SIZE                  : 1024
Temporary File              : memo-32GB.t
Final Output File           : k31-memo-32GB.x
[85.84] HashGen 50.00%: 12.56 MH/s : I/O 71.88 MB/s
[175.57] HashGen 100.00%: 11.97 MH/s : I/O 68.47 MB/s
[472.90] Shuffle 0.00%: 20.73 MB/s
[1346.51] Shuffle 50.00%: 7.03 MB/s
File 'memo-32GB.t' removed successfully.
Total Throughput: 1.59 MH/s  24.27 MB/s
Total Time: 1350.388451 seconds

real    22m32.107s
user    25m44.680s
sys     6m54.186s
root@hw5-large:~# ./vaultx_linux_x86 -m 16384 -k 31 -f k31-memo-32GB.x -
v true
Final output file 'k31-memo-32GB.x' already exists. Skipping generation and running verification only.
Size of 'k31-memo-32GB.x' is 34359738368 bytes.
[161.13] Verify 50.00%: 173.64 MB/s
[313.35] Verify 100.00%: 175.91 MB/s
sorted=2040538068 not_sorted=0 zero_nonces=106945580 total_records=2147483648
root@hw5-large:~#
```

Additional Successful Measurements (1GB baseline tests):

Node Configuration	Dataset	Program	Time (seconds)	Notes
host ok (bare metal)	1 GB	vaultx	9.25	Correctly sorted, t=16
small-1 (4 vCPU, 4GB)	1 GB	vaultx	15.64	Correctly sorted, t=16

Attempted Experiments: For 64GB datasets, experiments were initiated but not completed successfully due to:

- **Vaultx 64GB:** Temporary files were created, but final timing measurements were not captured reliably. In some attempts, output .bin files ended up as 0 bytes due to process interruptions and insufficient monitoring during multi-hour runs.
- **Hadoop 64GB:** HDFS NameNode repeatedly entered safe mode, preventing file writes. Error messages included: "Cannot create file ... Name node is in safe mode". Additionally, data-64GB.bin local file corruption (0 bytes) and JAR path resolution issues prevented successful completion.

Explanation of Results

1. **How many threads, mappers, reducers used in each experiment?**
   In hashgen and vaultx, on small instance there is 4 threads and on large instance there is 32 threads.
   In Hadoop sort, there is 4 (+4 In tiny instance) threads in small instance, 32 (+4 In tiny instance) threads in large instance, and 32 (+4 In tiny instance) threads in 8 small instances.
   In spark sort, there is 4 (+4 In tiny instance) threads in small instance, 32 (+4 In tiny instance) threads in large instance, and 32 (+4 In tiny instance) threads in 8 small instances.
2. **How many times did you have to read and write the dataset for each experiment?**
   In hashgen and vaultx, it writes the whole dataset two times as it writes every small chunk externally every time and for sorting. And it reads the whole dataset two times for sort and verification

   In Hadoop sort, it writes the whole dataset two times as it first writes the unsorted hashes to HDFS from hashgen and then again writes the sorted result back to HDFS after the map reduce sort. And it reads the whole dataset two times, one time when the sort job reads the unsorted data from HDFS and one time when the verify program reads the sorted file

to check it. We only count the full reads and writes of the dataset here, not all the many temporary files and spills that Hadoop is doing inside on local disks, because those happen many times but they are just internal steps not new full passes over the dataset.

In Spark sort, it also writes the whole dataset two times as it writes the unsorted hashes to HDFS when generating them and then writes the sorted data to HDFS after the Spark sort is done. And it reads the whole dataset two times, one time when Spark loads the unsorted data from HDFS into RDD and one time when valsort (or similar) reads the sorted output to verify. Same as Hadoop, we only count the full dataset read/write and not all the temporary shuffle and spill data Spark creates many times inside, because those are internal operations and not extra full scans of the dataset.

3. **What speedup and efficiency did you achieve?**
   On the 16 GB dataset on 1 large.instance, vaultx is about 36 times faster than Hadoop sort, and hashgen is about 10 times faster than Hadoop sort. On 1 small.instance with 16 GB, hashgen is around 11 to 12 times faster than Hadoop sort. This shows the native code (hashgen/vaultx) is using CPU and memory much more efficiently than Hadoop, which spends a lot of time in HDFS I/O, job startup and shuffle overhead. Spark in other studies is often 3 to 10 times faster than Hadoop for sort workloads, but it usually still cannot beat a highly optimized single-node native program like vaultx.

Resource Monitoring for 64GB Dataset

As we are not able to execute 64gb dataset using our code. So we don't have visualization graphs for this.

Analysis Questions

1. **Which seems to be best at 1 node scale (1 large.instance)?**
   At 1 large.instance, vaultx is clearly the best. For 16 GB, vaultx finishes in about 100 seconds, hashgen in about 344 seconds, and Hadoop sort takes around 3583 seconds. Spark would normally be faster than Hadoop on the same node because it keeps data in memory and avoids some HDFS writes, but even then it is unlikely to beat vaultx on this shared-memory machine.

2. **Is there a difference between 1 small.instance and 1 large.instance?**
   Yes, a big difference. For the 1 GB vaultx test, the small.instance took about 15.64 seconds while the bare-metal host took 9.25 seconds, so the small.instance was clearly

slower. For 16 GB Hadoop sort, going from 1 small.instance to 1 large.instance almost halves the runtime. The large.instance has more cores and more RAM, so it can run more tasks in parallel and spill less, but Hadoop still does not get close to linear speedup because it is limited by I/O and framework overhead.

3. **How about 8 nodes (8 small.instance)?**
   For Hadoop sort, 8 small.instances did not outperform 1 large.instance. For 16 GB, 1 large.instance finished in about 3583 seconds, while 8 small.instances took about 5643 seconds. For 32 GB, the same pattern holds: 1 large.instance is faster than 8 small.instances. The network, HDFS metadata, shuffle overhead and slower per-node hardware make the 8-node cluster less efficient, even though it has a similar total core count. Hashgen and vaultx as implemented only use 1 node, so they do not benefit from 8 nodes at all.
4. **What speedup do you achieve with strong scaling between 1 to 8 nodes?**
   In our actual data for Hadoop sort, going from a single powerful large.instance to 8 small.instances does not give speedup, it actually slows things down. The 8-node cluster takes roughly 1.6 times longer than the single large.instance for both 16 GB and 32 GB. So the measured strong scaling efficiency is very low here. In theory Spark would scale better than Hadoop in strong scaling, but we did not complete reliable 8-node Spark runs to show that numerically.

5. **What speedup do you achieve with weak scaling between 1 to 8 nodes?**
   We do not have clean 1-node vs 8-node results for a perfect weak-scaling pattern, so we cannot give exact numbers. From theory and published results, Hadoop and Spark usually show almost constant but slightly increasing runtime under weak scaling, because coordination, network and shuffle cost grow with the number of nodes. In our environment, with virtual machines and storage limits, it is reasonable to expect runtimes to grow more than ideal when going from 1 to 8 nodes, especially for Hadoop. Spark would usually be closer to constant time than Hadoop in a better tuned cluster.

6. **How many small.instances do you need with Hadoop to achieve the same level of performance as your hashgen or vaultx programs?**
   On 1 large.instance with 16 GB, vaultx is about 36 times faster than Hadoop. That means Hadoop would need a very large speedup to match it. Since our 8-node Hadoop cluster is actually slower than the single large.instance, real scaling is far from ideal. Based on known TeraSort results and the overhead we observed, Hadoop would probably need on the order of tens of small.instances, roughly in the 16 to 32 node range or even more on

this virtualized hardware, to get close to vaultx's single-node time at 16 GB, assuming very good tuning.

7. **How many small.instances do you need with Spark to achieve the same level of performance as you did with your hashgen/vaultx?**
   Spark is generally several times faster than Hadoop on sort workloads, often 3 to 10 times, because it keeps more data in memory and reduces HDFS writes. So Spark should need fewer nodes than Hadoop to hit the same target time as vaultx. If Hadoop needs something like 16 to 32 small.instances to approach vaultx's performance, a reasonable prediction is that Spark could reach a similar runtime with around 8 to 16 small.instances, given enough executor memory and a properly configured cluster.

8. **Does Spark seem to offer any advantages over Hadoop for this application?**
   Yes. Spark has clear advantages for this hash generation, sort and search style application. It keeps intermediate data in memory instead of always writing to HDFS, so there is less disk I/O and jobs finish faster. It can combine hash generation, sort, and search in one pipeline and reuse cached data for multiple searches. Spark also has lower job startup overhead and is easier to tune in terms of partitions and executor memory. Hadoop only looks competitive when memory is very small and everything spills to disk so Spark cannot use its in-memory advantage.

9. **Can you predict which would be best if you had 100 small.instances? How about 1000?**
   With about 100 small.instances, Spark sort would be the best choice for large datasets, because it can distribute work across many nodes and keep much of the data in memory while avoiding some of Hadoop's heavy disk usage. Hadoop will run, but it will still be slower because of extra HDFS and shuffle overhead. Hashgen and vaultx in their current single-node form would not scale at all to 100 nodes. At 1000 nodes, Spark should still be better than Hadoop, but network and coordination overhead become very large and the speedup will not grow much. For our 16 to 64 GB datasets, 1000 nodes is overkill, so both Hadoop and Spark would be under-utilized and not much faster than at around 100 nodes.

Search Performance Results

Table 2: Search Performance

Approach	K	Difficulty	Number of Searches	Total Time (s)	Time (ms) / search	Throughput (search/sec)	Searches Found	Searches Not Found
hashgen	30	3	1000	0.076333	0.076	13100.44	1000	0
hashgen	30	4	1000	0.076333	0.076	13100.44	421	578
hashgen	31	3	1000	0.079167	0.079	12631.58	1000	0
hashgen	31	4	1000	0.079167	0.079	12631.58	523	477
hashgen	32	3	1000					
hashgen	32	4	1000					
vaultx	30	3	1000	0.007100	0.710	1408.514	10	0
vaultx	30	4	1000	0.005747	0.575	1739.897	10	0
vaultx	31	3	1000	0.006521	0.652	1533.400	10	0

vaultx	31	4	1000	0.00647 6	0.648	1544.062	10	0
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					

Search Implementation Status

Status: Not Completed.

Search was not finished for Hadoop or Spark. To run distributed search we first need completed and verified sorted datasets on all target sizes. Our 64 GB run did not complete reliably, and the 8 node cluster for Spark and Hadoop was unstable because of HDFS safe mode issues, storage limits and some Spark shuffle problems. Without a clean sorted baseline, it did not make sense to bolt a search layer on top.

Also, a proper distributed search in Hadoop and Spark would need extra work:

A custom partitioning scheme so that all candidate keys for a query stay in a known partition or bucket.

Binary search or similar logic on the reducer or executor side, working on sorted partitions.

A small aggregation step to collect and merge search results from all nodes.

Query generation that matches the vaultx format and difficulty levels from the assignment.

Given the time spent debugging HDFS safe mode, corrupted large files and Spark memory or spill problems, there was not enough time left to design, code and test this search pipeline before the deadline.

Expected Search Performance

For Hadoop and Spark, a realistic search job must go through HDFS or distributed memory and may involve starting a job, reading partitions and maybe doing a join. Their per query complexity is also logarithmic in partition size, but you pay extra network and framework overhead. On a single node Hadoop search will be much slower than vaultx, mostly because each search touches HDFS blocks and has higher startup time. In an 8 node or larger cluster, Hadoop or Spark can spread many queries over many nodes, so total cluster throughput can increase, but network and shuffle latency will still dominate for light queries.

1. **Compare and contrast hashgen/vaultx search and Hadoop/Spark search performance, and why the results make sense**

   Hashgen and vaultx implement search on a single node using a bucketed, sorted hash file. Each query does a quick bucket lookup and then a small in-memory search, so latency per search is extremely low and throughput can reach tens of thousands of queries per second when the dataset fits in RAM. Hadoop and Spark need to read from HDFS or from distributed memory, coordinate tasks and possibly shuffle data to run a search job. Per query latency is much higher, usually hundreds of milliseconds or seconds if you start a new job for each batch of queries. They only make sense when the dataset is so large that it cannot fit on one node and you care more about total throughput across the cluster than about single query latency. So it is expected that hashgen/vaultx search is much faster on one node, and Hadoop/Spark are only attractive when scaling to very large data and many nodes.

References

1. **Apache Hadoop Official Documentation**: https://hadoop.apache.org/docs/stable/
2. **HDFS Architecture Guide**: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
3. **MapReduce Tutorial**: https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html
4. **Apache Spark Official Documentation**: https://spark.apache.org/docs/latest/
5. **Spark Quick Start Guide**: https://spark.apache.org/docs/latest/quick-start.html
6. **Spark RDD Programming Guide**: https://spark.apache.org/docs/latest/rdd-programming-guide.html
7. **Blake3 Hash Function**: https://github.com/BLAKE3-team/BLAKE3
8. **Chameleon Cloud Documentation**: https://chameleoncloud.readthedocs.io/
9. **Linux Process Monitoring**: https://unix.stackexchange.com/questions/554/how-to-monitor-cpu-memory-usage-of-a-single-process
10. **Hadoop MapReduce Examples**: https://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm
11. **Databricks Blog "Apache Spark Sets New Sorting Record":** https://www.databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html
12. **phoenixNAP "Hadoop vs Spark Detailed Comparison":** https://phoenixnap.com/kb/hadoop-vs-spark
13. **IBM "Hadoop vs Spark What is the Difference":** https://www.ibm.com/think/insights/hadoop-vs-spark
14. **OpenLogic "Spark vs Hadoop Key Differences and Use Cases":** https://www.openlogic.com/blog/spark-vs-hadoop
15. **NetSolutions "Hadoop vs Spark Choosing the Right Big Data Framework":** https://www.netsolutions.com/insights/hadoop-vs-spark/
16. **Ksolves "A Comparison between Apache Spark and Apache Hadoop":** https://www.ksolves.com/blog/big-data/spark/a-comparison-between-spark-vs-hadoop