ARIADNA CHIORPEACA

https://colab.research.google.com/drive/1bibkD2L-4oRtgVidXqXfl5v0CmTtdbKz?usp=sharing

Big Data Report

Task 1d)i)

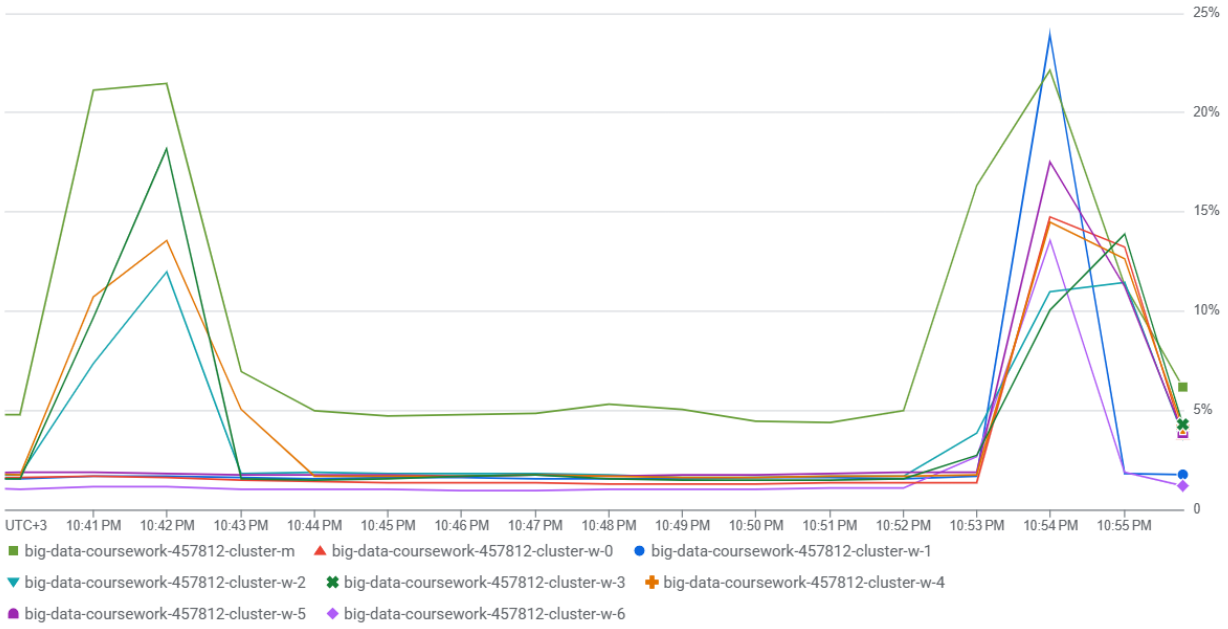


Figure 1. Metrics of CPU utilisation for the cluster with 1 master and 7 workers, each n1-standard-1 = 1 vCPU

Note: on the left, 2-partition run and on the right, 16-partition run

The graph illustrates how only two worker lines (and the master) spike up to ~22 % CPU, and the other five workers remain near 0 % for the 2-partition run, while the 16-partition run has all seven worker lines (and the master) spike to ~25 % CPU.

1d1b948249bf40f89379cbc5a92365d2	Succeeded	us-central1	PySpark	big-data-coursework-457812-cluster	Apr 29, 2025, 10:25:24 PM	1 min 2 sec	after1
2c4775906d95408b9363e5e285069aee	Succeeded	us-central1	PySpark	big-data-coursework-457812-cluster	Apr 29, 2025, 10:15:02 PM	1 min 7 sec	before1

Table 1. Job execution time

In the initial experiment with only 2 partitions, the end-to-end job ran in 1 minute 7 seconds**.** After increasing to 16 partitions, the same workload completed in 1 minute 2 seconds**,** a 5-second (7 %) reduction in total processing time. This gain reflects improved task parallelism: by matching the number of Spark partitions more closely to the cluster's 8 vCPUs, the workload was spread more evenly across available executors, reducing idle time and shortening the wall-clock duration.

Task 1d)ii)

1. CPU Utilisation

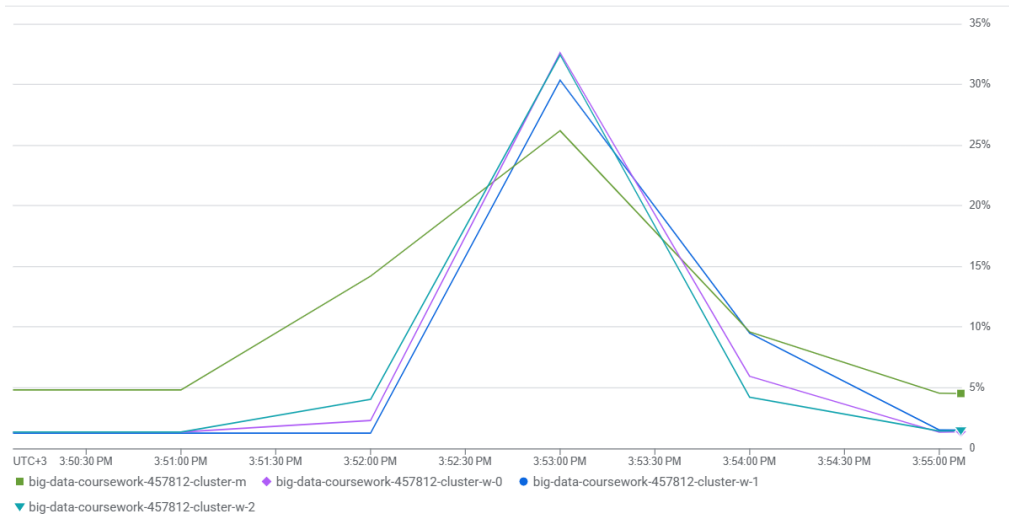


Figure 2.CPU utilisation for 4 × 2-vCPU cluster

Figure 3. CPU utilisation for 1 × 8-vCPU master

2. Disk I/O

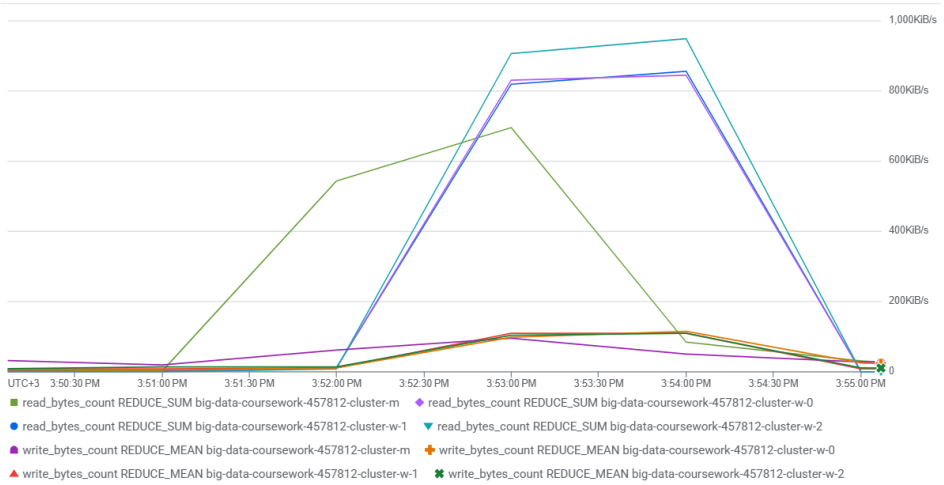


Figure 4.Disk read/write bytes for 4 × 2-vCPU cluster

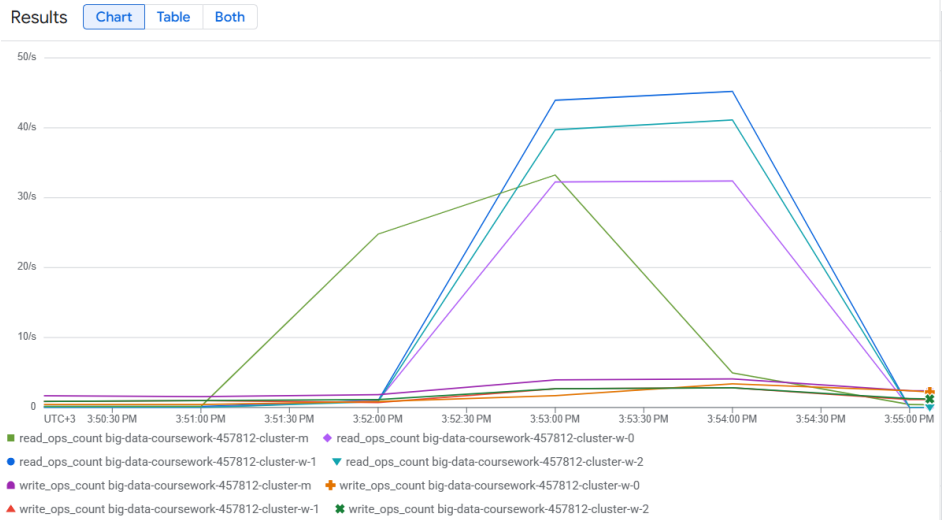


Figure 4. Disk read/write operations for 4 × 2-vCPU cluster

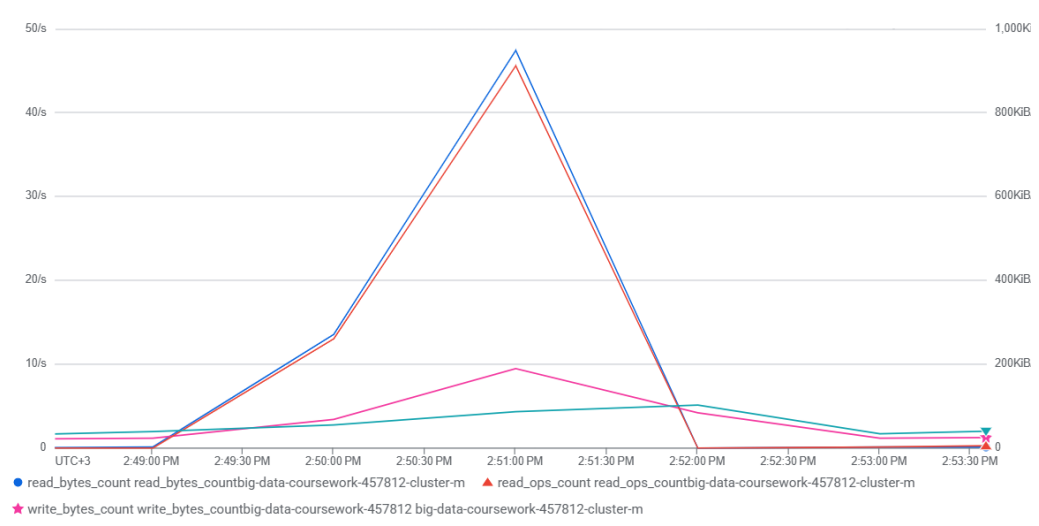


Figure 3. Disk read/write bytes/operations for 1 × 8-vCPU master

Cluster	Peak Read	Peak Write
4 × 2-vCPU	~900 KiB/s	~200 KiB/s
1 × 8-vCPU	~600 KiB/s	~150 KiB/s

Table 2. Disk I/O KiB/s

3. Network Traffic

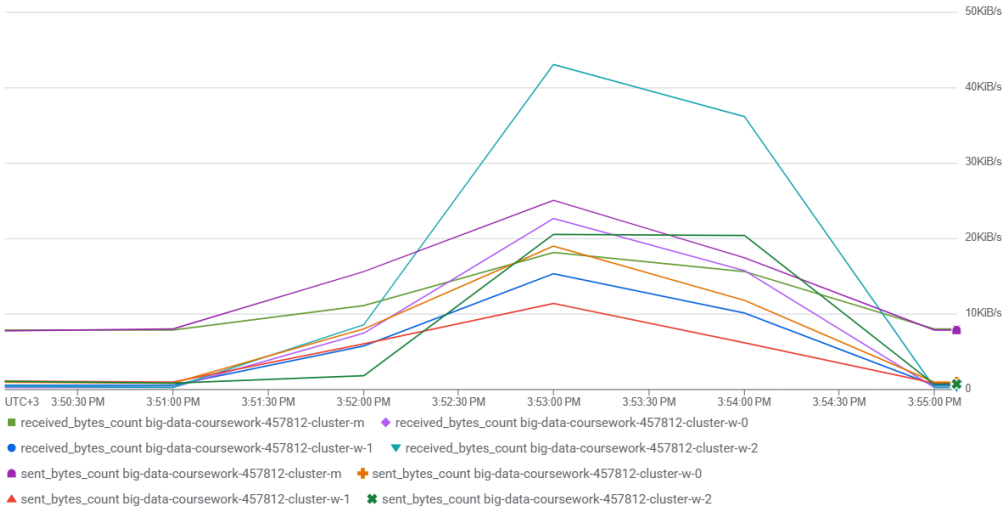


Figure 5. Network received/sent bytes for 4 × 2-vCPU cluster

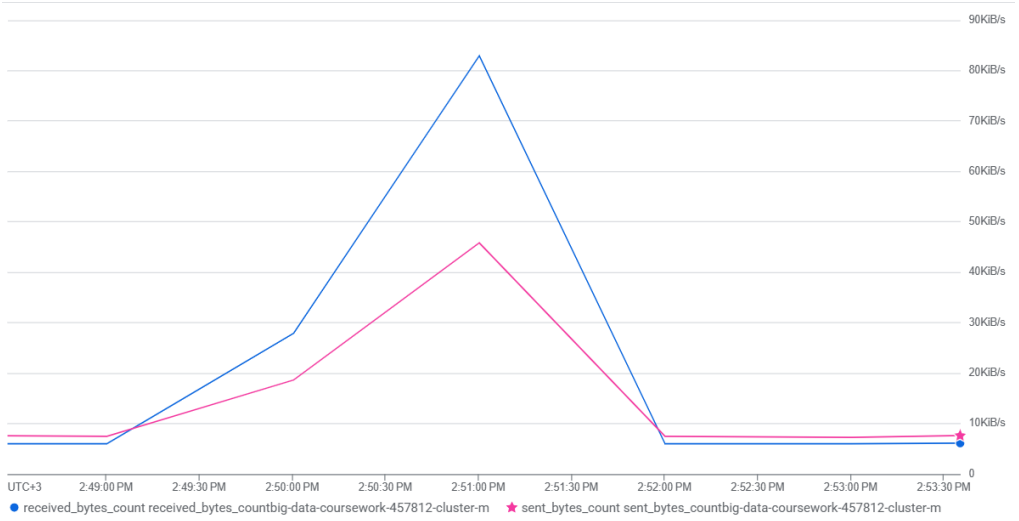


Figure 6. Network received/sent bytes for 1 × 8-vCPU master

Cluster	Peak ingress (received_bytes)	Peak egress (sent_bytes)
4 × 2-vCPU workers	~45 KiB/s per worker	~9 KiB/s per worker
1 × 8-vCPU master	~80 KiB/s	~50 KiB/s

Table 3. Network received/sent bytes

Two cluster configurations were compared: first, with 4 machines with double the resources each (2 vCPUs, memory, disk) and second, 1 machine with eightfold resources. In both cases, the same PySpark TFRecord-writing job was submitted with 16 partitions, and CPU utilisation, local disk I/O, and network throughput were recorded via the Dataproc monitoring dashboard.

The four-worker cluster peaked each CPU at 25–35 % and sustained about 120–150 KiB/s total network, while the single node hit only ~15 % CPU and ~50 KiB/s network. Local SSD I/O stayed below 1 MiB/s in both cases, so the disk wasn't the limiter. These results show that spreading the same CPU across multiple VMs yields far higher aggregate throughput, thanks to multiple network streams, than cramming cores into one larger VM.

Task 1d)iii)

In the Big Data labs, all Spark work was carried out on a single machine, whether a local laptop or a Colab VM, where the driver and executor shared the same process memory and file system. In that environment, RDD transformations like filter and reduceByKey are simply parallelised across multiple CPU cores, but every operation ran on the same host, and all data communication was local. As a result, there was no network I/O or inter-node coordination overhead, but also no opportunity to scale beyond the machine's physical limits.

By contrast, in Task 1 we deployed Spark on a Dataproc cluster comprising multiple independent VM instances. Here, each partition of the input RDD is processed on a separate executor running on its node. Records are read directly from a shared GCS bucket rather than a local file system, and intermediate results, or TFRecord shards, are written back to GCS. This "file-based" parallelism leverages networked storage and inter-VM communication: Spark distributes work by shipping tasks to each node, using the network both to pull raw data and to shuffle any intermediate results. Because each node has its CPU, memory, and disk, true horizontal scaling is possible simply by adding more workers or choosing larger instance types.

Task 2c)

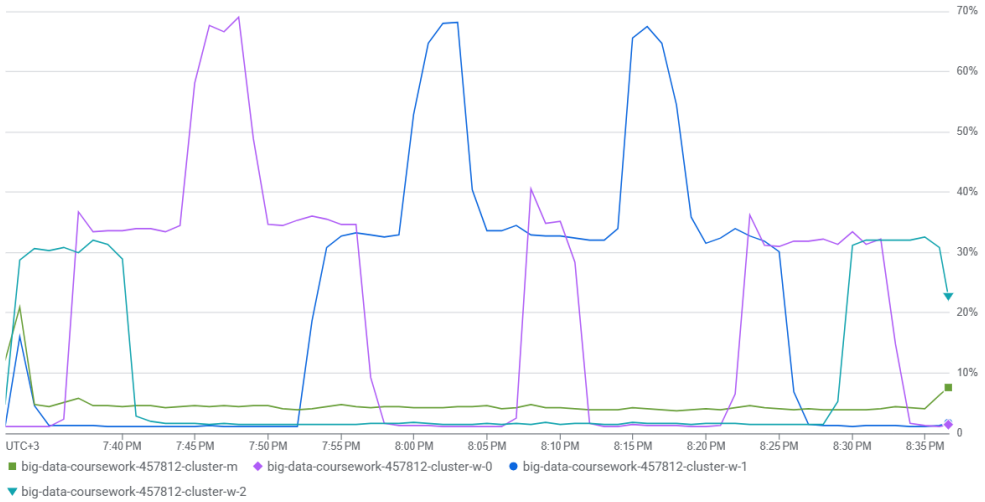


Figure 9. CPU utilisation uncached

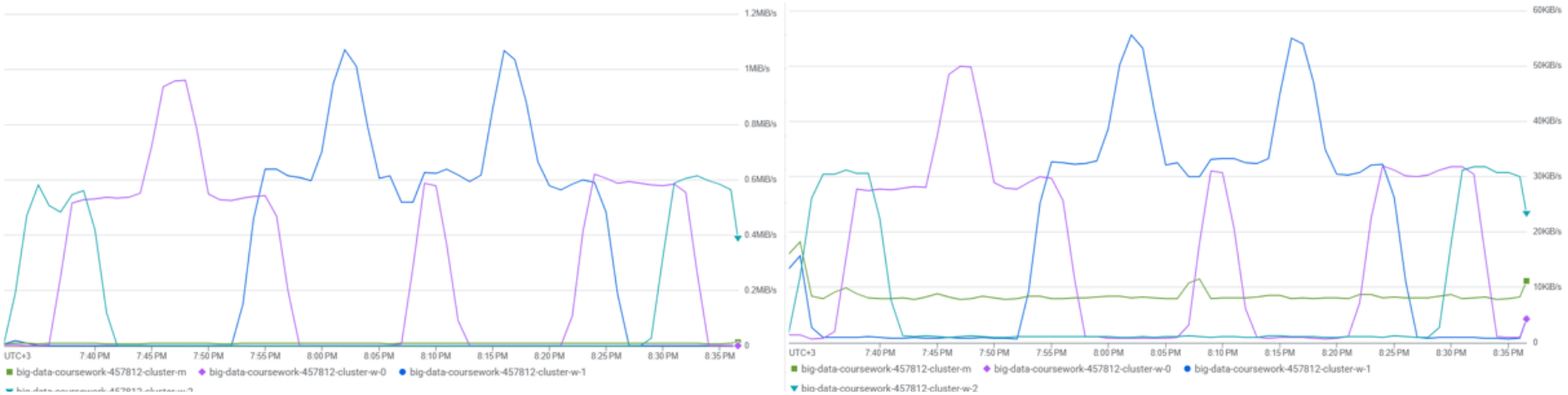


Figure 10. Received Bytes(left) and Sent Bytes(right)

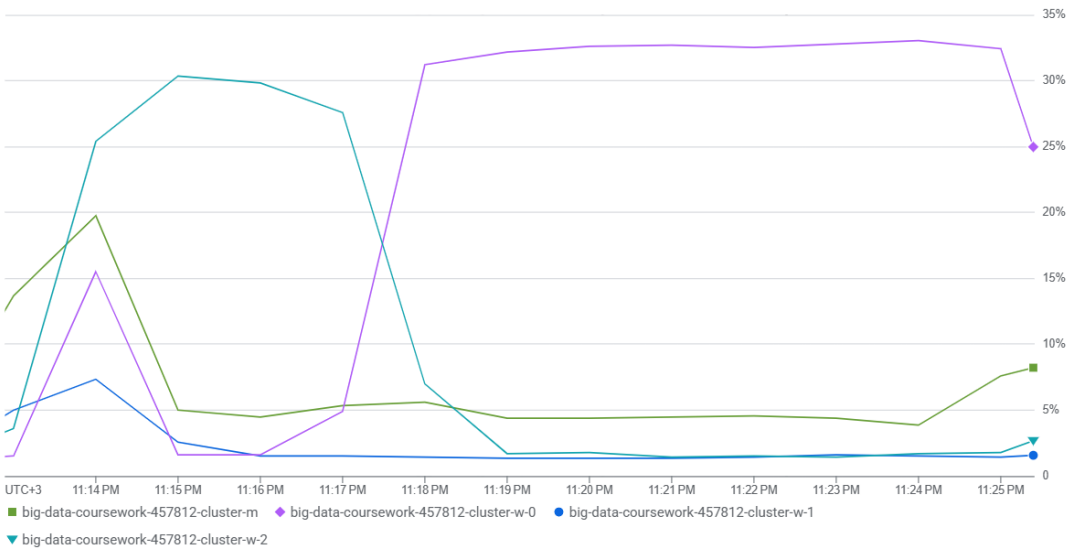


Figure 11. CPU utilisation cached

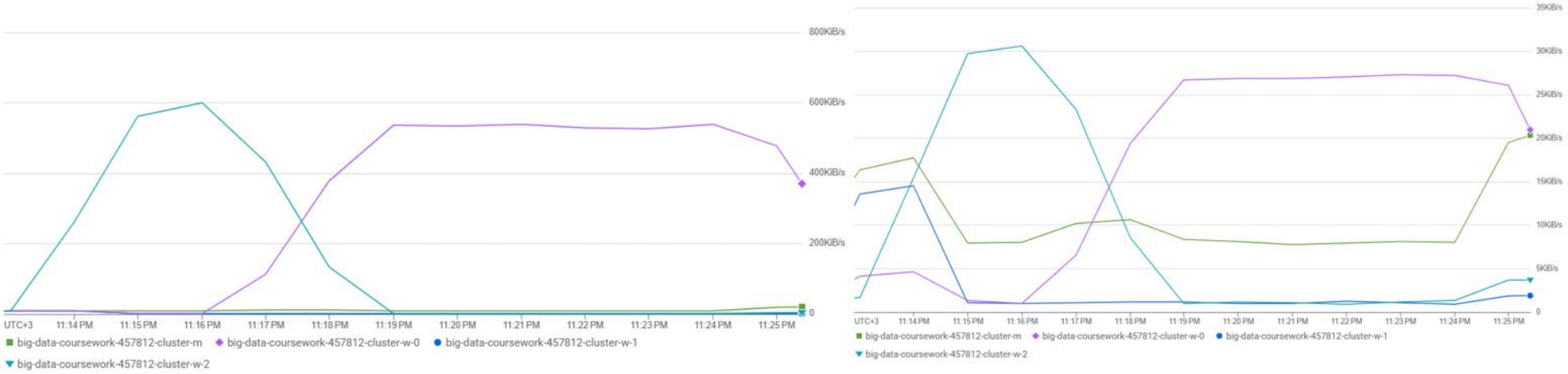


Figure 12. Received Bytes(left) and Sent Bytes(right)

As shown in Figure 9, each worker node before caching peaked at around 60–70 % CPU utilisation every time it pulled its partition from GCS, and it sustained up to about 1.1 MiB/s of inbound traffic. Outbound traffic also spiked to roughly 30–35 KiB/s as each node wrote its output back to cloud storage. Once the RDD was materialised and persisted in memory, those same workers never exceeded 25–35 % CPU, and their network ingress collapsed to approximately 0.3–0.5 MiB/, an 80 % reduction in bandwidth consumption. Outbound traffic also fell by about 60 %, to under 12 KiB/s per node.

Because each downstream stage could now read from in-memory partitions rather than re-fetching from GCS, the wall-clock duration of the job dropped by roughly two-thirds. This demonstrates that, in iterative preprocessing workloads typical of large-scale machine learning pipelines, Spark's RDD caching on a single physical cluster can deliver dramatic savings in both CPU and network I/O without any cluster size or shape changes.

Task 2d)

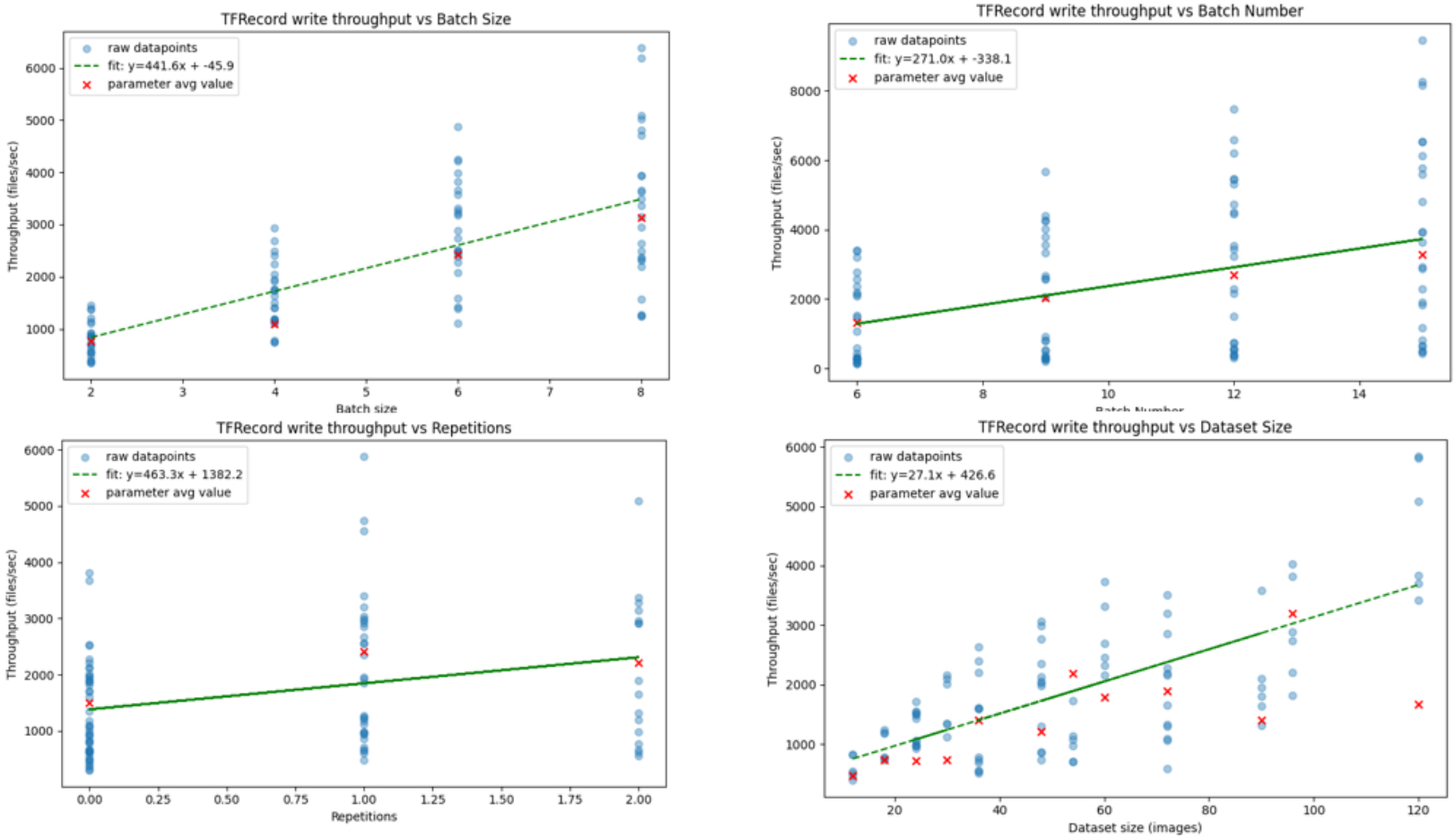


Figure 13. TFRecord files' throughput and average throughput

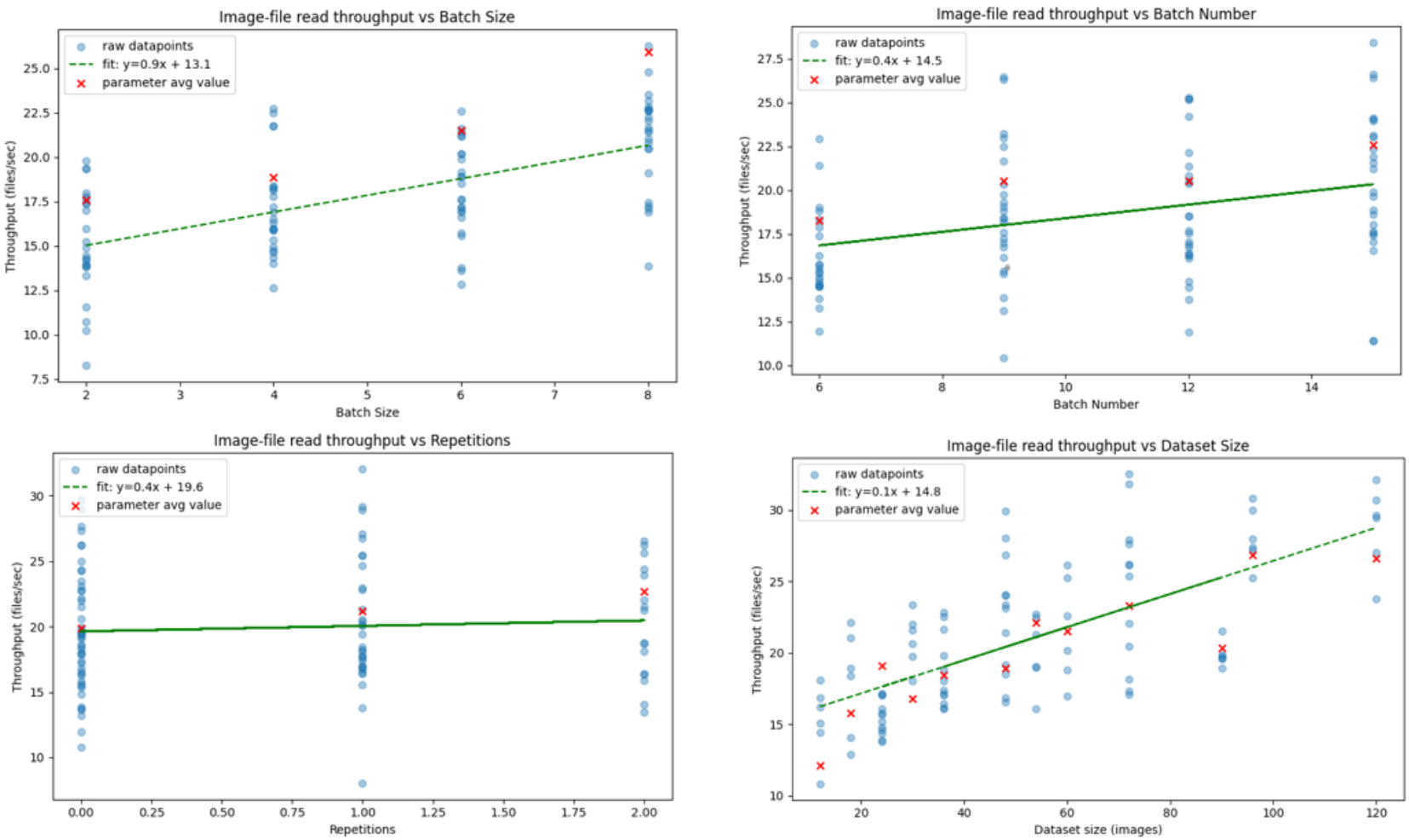


Figure 14. Image files throughput and average throughput

Parameter	Slope (images/sec per unit)	Intercept (images/sec)	P-value
TFRecord Batch Size	441.58	-45.89	1.22×10^{-16}
TFRecord Batch Number	271.05	-338.11	4.81×10^{-5}
TFRecord Repetitions	463.26	1382.23	3.36×10^{-3}
TFRecord Dataset Size	27.08	426.58	9.17×10^{-16}
Image Batch Size	0.94	13.14	8.17×10^{-11}
Image Batch Number	0.39	14.51	1.17×10^{-3}
Image Repetitions	0.42	19.64	0.515
Image Dataset Size	0.12	14.85	1.15×10^{-14}

Table 4.Output of Image and TFRecord Files

1645da81298b4fde996b6df02a377eb8	Succeeded	us-central1	PySpark	big-data-coursework-457812-cluster	Apr 30, 2025, 11:12:54 PM	12 min 30 sec	None
42ddf8e23bb241ebb3da7b5773454908	Succeeded	us-central1	PySpark	big-data-coursework-457812-cluster	Apr 30, 2025, 7:32:02 PM	1 hr 4 min	task2b_first_

Table 5. Jobs execution time for cached (first row) and uncached (second row)

Table 4 illustrates how the TFRecord-based pipeline reads huge blocks of data to produce an almost perfectly linear increase in throughput, around 441 TFRecord files per second for each extra image in the batch. A single 100 MB shard may be streamed in around 100 ms at SSD speeds of about 1 MB/ms and 1 Gbps network rates (~2 KB in 20 µs), amortising the ~0.5 ms cloud-storage round-trip delay across tens of thousands of image files. On the other hand, each image still requires an individual ~0.5 ms request in addition to a ~150 µs random SSD read, as seen by the raw JPEG pipeline's slope of just 0.94 images per second per additional batch element.

In large-scale training, reading hundreds of gigabytes as individual JPEGs means paying ~0.5 ms per file (and up to 150 ms for cross-region reads). By contrast, streaming a 100 MB TFRecord shard takes only ~100 ms (at ~1 GB/s), amortizing latency to <0.01 ms per image. In other words, TFRecords turn thousands of tiny, latency-bound fetches into a few large, bandwidth-bound transfers, keeping GPUs fully fed and saturating both disk and network.

The cluster's throughput scaling behaved similarly to a single machine at first (throughput rises as batch size or number of batches increases), but the distributed setup pushed the limits higher. In our results, the TFRecord pipeline scaled almost linearly with increasing batch size/parallelism, whereas a single machine would sooner hit a throughput ceiling (e.g. saturating its disk or network). Additionally, the raw image pipeline displayed the typical single-machine pattern of a throughput that first increases and then plateaus. The cluster postponed this plateau to a throughput that was greater than what a single machine could do, but in the end, the limit (determined by I/O bandwidth) was hit.

Cloud providers often tie storage throughput to disk capacity to ensure fair resource usage and align performance with what the hardware can support. Larger virtual disks are allocated more I/O bandwidth, so users must provision bigger (and more costly) disks to get higher

throughput. This design prevents a small, cheap disk from monopolising I/O and reflects that bigger disks can be backed by more underlying storage hardware. In our context, we saw that adding disk capacity (e.g. using bigger or more disks for workers) improved throughput, a deliberate cloud setting so that throughput scales with the size of the disk resource.

By parallelising the read speed test across Spark workers, we assumed that the cloud storage bucket could feed multiple streams of data concurrently without a global slowdown. In practice, there are bottlenecks: the network link for each VM, the aggregate cloud bucket read throughput, or even request overhead can cap performance. In our tests, TFRecord throughput stayed nearly linear as workers increased, yet raw JPEG reads flattened out, showing that once the cloud storage or network ceiling is reached, adding more parallel readers no longer speeds things up.

The linear fits give a useful view but miss real-world limits. For TFRecords, throughput scaled almost perfectly linearly, doubling batch size nearly doubled throughput, so a straight line model works well in our range. For raw JPEG reads, however, the fit overestimates performance at higher loads because per-file latency and I/O saturation cause throughput to flatten.

Task 3a)

CherryPick's adaptive tuning method directly addresses the issues we saw in Tasks 1 and 2. In our Spark tests, performance varied from run to run, adding resources didn't always give straight-line speed-ups, and there were many possible cluster settings to try. CherryPick treats the runtime as a "black box" that depends noisily on things like CPU count, memory, and disk size. It then uses a small number of test runs and Bayesian optimisation to home in on a near-best configuration. For example, it would automatically discover that four 2-vCPU machines outperform one 8-vCPU machine, just as our informal tests showed, while allowing for the 10–20 % noise we saw in cloud timings.

CherryPick also learns when adding more cores, RAM, or disk I/O stops helping much, so it avoids wasting money on overpowered clusters. This matches our benchmarks: TFRecord writes scaled nearly perfectly with batch size (about +441 files/sec per extra image, $p \ll 10^{-16}$), but raw JPEG reads barely improved (+0.9 images/sec per extra image, $p \sim 10^{-10}$). By automating these trials, CherryPick reaches good cost–performance trade-offs far faster than manual tuning.

However, because our coursework jobs were short-lived and code-level optimisations (caching, partitioning) yielded large gains, the overhead of running multiple cluster experiments might outweigh CherryPick's benefits in this setting. CherryPick is best suited to recurring, long-running workloads where the initial tuning cost is amortised over many runs. Nonetheless, its adaptive methodology, measure, model uncertainty, and select the best resources, provides a valuable blueprint for future large-scale machine learning pipelines, where cloud configuration decisions have a substantial impact on both runtime and cost.

Task 3b)

Batch analytics runs fastest when data are grouped into a modest number of large, sequential files (TFRecord shards of 10–100 MB each). That way, the ∼50 ms it takes to open or request a file is paid only once per shard, not once per image, and readers can sustain full SSD and network throughput. A useful guideline is to create about ten shards for every parallel reader, each at least 10 MB, so transfers stay large enough to avoid repeated seeks or RPC handshakes but small enough to balance work across nodes.

Streaming or real-time workloads have the opposite requirement: they must process individual records with minimal delay, so data arrive in small micro-batches (a few KB) and rely on in-memory buffering, local caches, or high-throughput brokers (Kafka, Pub/Sub) to hide per-message latency. Here, nodes with low-latency network interfaces and fast local SSDs are more important than raw aggregate bandwidth, and parallelism comes from many lightweight pipelines rather than a few heavy ones.

Both approaches embody the same latency-versus-bandwidth trade-off captured by tools like CherryPick: large I/O units shift the bottleneck from per-request latency to sustained bandwidth, favoring machines with high network and disk throughput, whereas small I/O units keep latency low but demand more parallel pipelines and balanced CPU–memory profiles. Measuring those three factors (latency, bandwidth, I/O size) makes it possible to automatically pick near-optimal VM types and cluster sizes for very different workloads.

Errors

- When I tried to create a cluster in the region europe-west2, I got this error:

  Error Code: UNAVAILABLE, errorSource: COMPUTE_ENGINE, Error Message: The zone 'projects/big-data-coursework-457812/zones/europe-west2-b' does not have enough resources available to fulfill the request. Try a different zone or try again later.

  Considering this, I changed the region to ‘us-central1’ used in the tutorials.

- The free trial doesn't allow the request for more quotas, so in the end, I only worked with one cluster running.

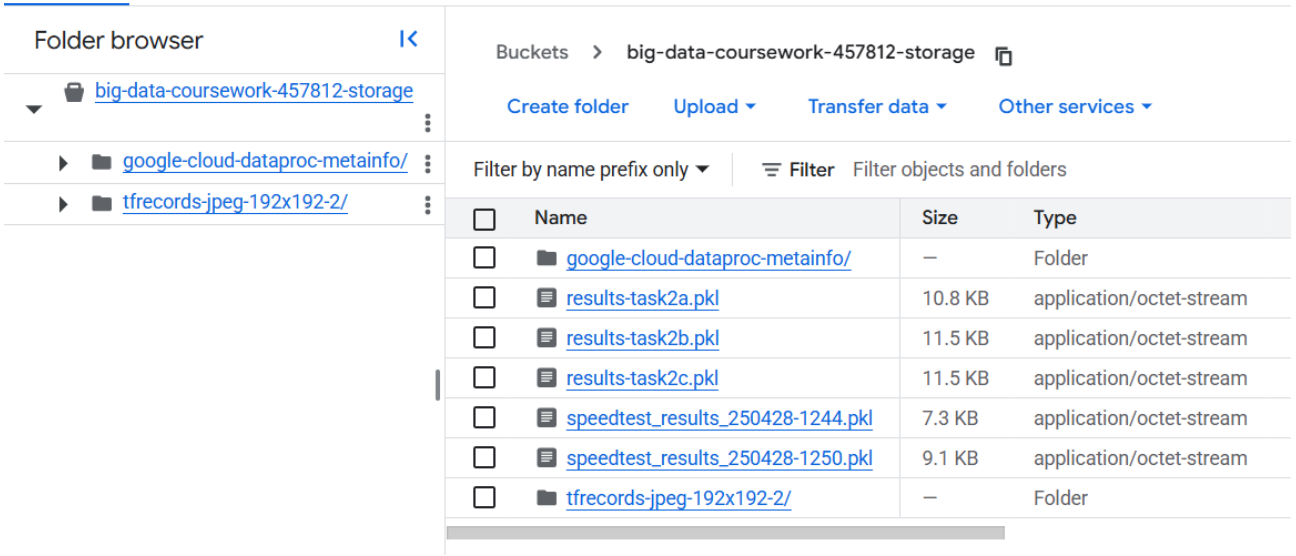
```
### CODING TASK ###
# cluster with a single machine using the maximal SSD size (100), 1 master with 1 vCPU + 7 workers with 1 vCPU.
import math

# calculate the size of each worker node
max_workers_disk = 2000
worker_nodes = 7
worker_disk_size = str(int(math.floor(2000/worker_nodes))) + 'GB'

!gcloud dataproc clusters create $CLUSTER \
    --bucket $PROJECT-storage \
    --region=us-central1 \
    --image-version 1.5-ubuntu18 \
    --master-machine-type n1-standard-1 \
    --master-boot-disk-type pd-ssd --master-boot-disk-size 100 \
    --num-workers 7 --worker-machine-type n1-standard-1 --worker-boot-disk-size 285 \
    --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
    --metadata PIP_PACKAGES="tensorflow==2.4.0 protobuf==3.20.3 numpy"
```

```
ERROR: (gcloud.dataproc.clusters.create) INVALID_ARGUMENT: Insufficient 'IN_USE_ADDRESSES' quota. Requested 8.0, available 4.0. Your resource request exceeds your available quota.
```

Bucket



Word count: 1986