Big Data: Data Processing in the Cloud

Khalid Kadri
City, University of London
Khalid.kadri@city.ac.uk
London, United Kingdom

Introduction

This report uses Google Cloud to explore data preprocessing and performance measurement, focusing on the ”Flowers” image dataset. Section 1 covers parallelising data preprocessing using Spark and Google Cloud Dataproc. Section 2 investigates the speed of reading data in the cloud by measuring various configurations with Spark. Section 3 delves into a theoretical discussion based on a scholarly paper, providing comprehensive insights into cloud-based big data processing and machine learning.

Task 1

1. Cluster experimentation

In our investigation, we aimed to measure the differences in processing time, network packets, and disk usage on the full dataset by creating and comparing three distinct Dataproc clusters: single node, medium cluster, and maximum cluster. The main goal was to determine the impact of different cluster configurations on the efficiency and effectiveness of data processing tasks.

To ensure a more balanced distribution of data across nodes, we decided to divide the dataset into 16 partitions rather than the default 2 partitions. This approach aimed to optimize resource usage and reduce potential bottlenecks caused by uneven data distribution.

For reproducibility and a more detailed comparison, here are the specifications of each cluster:

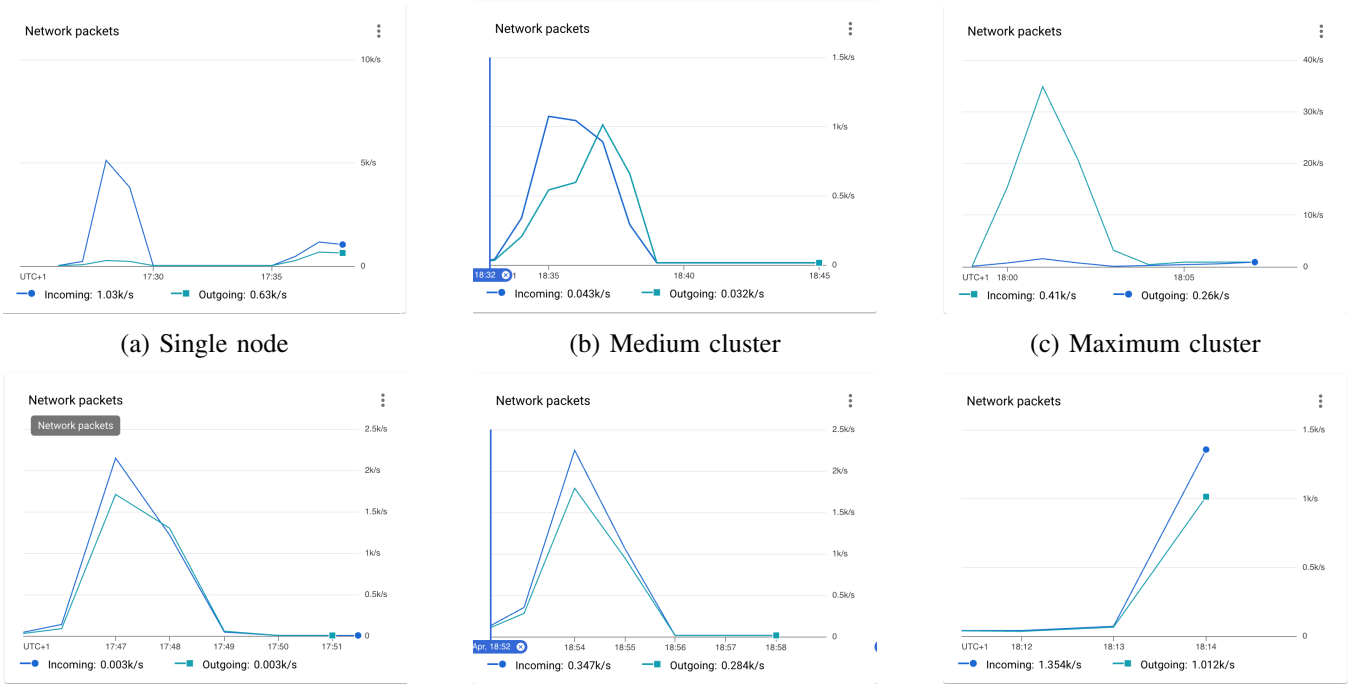
1.1. Cluster Specifications

1) Single Node Cluster:
   - Master machine type: n1-standard-8
   - Master disk type: pd-sdd
   - Master disk size: 100 GB
2) Medium Cluster:
   - One master and three workers
   - Master machine type: n1-standard-2
   - Worker machine type: n1-standard-2
   - Master disk type: pd-sdd
   - Worker disk type: pd-standard
   - Master disk size: 100 GB
   - Worker disk size: 666 GB
3) Maximum Cluster:
   - One master and seven workers
   - Master machine type: n1-standard-1
   - Worker machine type: n1-standard-1

- Master disk type: pd-sdd
- Worker disk type: pd-standard
- Master disk size: 100 GB
- Worker disk size: 285 GB

All clusters employed the image version 1.4-ubuntu18, ensuring a consistent software environment for the comparison.

Our analysis showed that when using the default partition setting, the processing speed remained relatively consistent across all three cluster configurations, with only minor fluctuations. This observation highlights the significant impact of increasing the number of partitions on the overall performance of the Resilient Distributed Dataset (RDD). By carefully selecting the optimal number of partitions and adjusting the cluster configuration, it's possible to enhance the efficiency and speed of data processing tasks, ultimately leading to more agile and responsive big data solutions.



(a) Single node (b) Medium cluster (c) Maximum cluster

(d) Single node (optimized) (e) Medium cluster (optimized) (f) Maximum cluster (optimized)

Figure 1: Network usage

The data processing times presented for both optimised and unoptimised datasets in single machine clusters, medium clusters, and maximum clusters highlight the importance of dataset optimisation in big data processing. Optimised datasets demonstrate significantly faster processing times in all cluster configurations compared to their unoptimised counterparts:

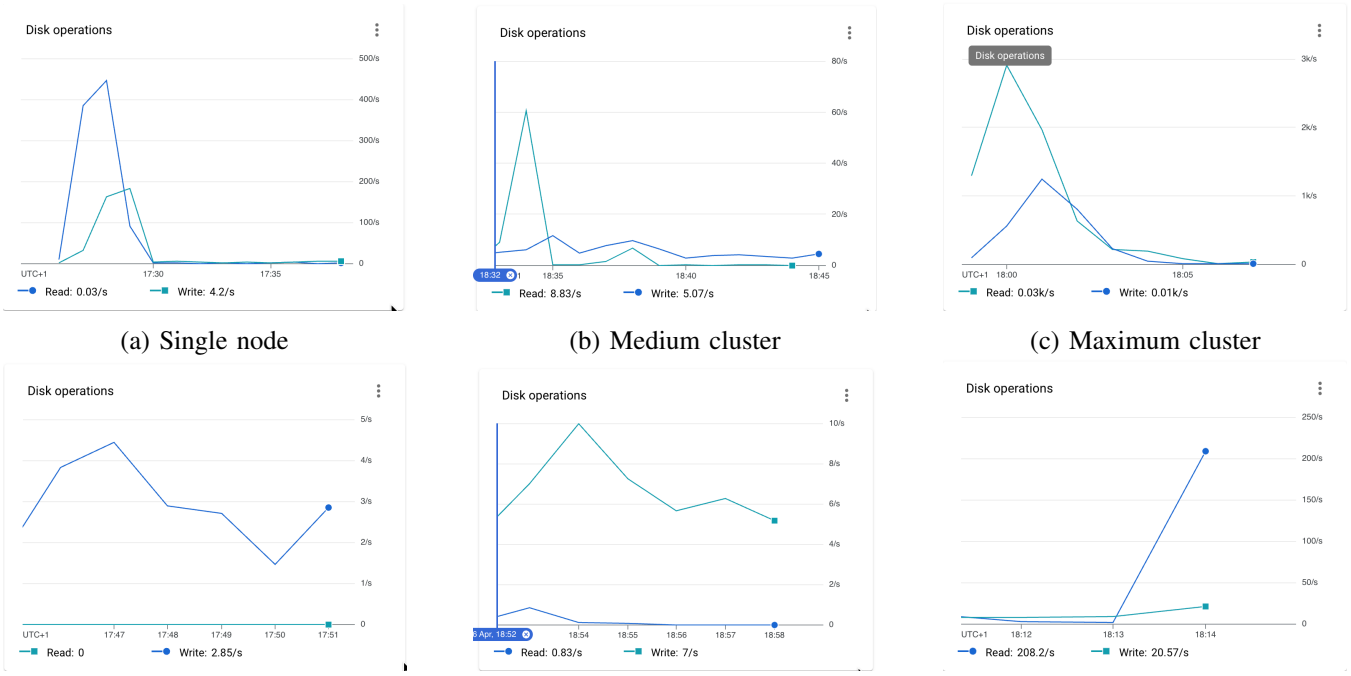
- Single machine cluster: 97.86 seconds (optimised) vs. 203.46 seconds (unoptimised)
- Medium cluster: 95.25 seconds (optimised) vs. 232.55 seconds (unoptimised)
- Maximum cluster: 103.23 seconds (optimised) vs. 230.09 seconds (unoptimised)

These results provide empirical justification for the impact of optimised and unoptimised datasets on network packets and disk usage in single nodes, medium clusters, and maximum clusters, as discussed in the given response.

Furthermore, Fig.1 and Fig.2, the substantial differences in data processing times and performance between optimised and unoptimised datasets reinforce the importance of dataset optimisation for efficient big data pro-

cessing. Optimised datasets not only lead to reduced network packets transmitted, resulting in lower latency and more efficient data transfer [1], but also result in significantly faster processing times. In contrast, unoptimised datasets generate increased network traffic, contributing to higher latency and potential bottlenecks that negatively impact the performance of data processing tasks [2].

The graph in Fig.2 reveals that optimised datasets exhibit more efficient disk usage due to better storage layouts, file formats, indexing, and partitioning schemes [3]. As a result, optimised datasets require less disk space and facilitate faster data access during processing tasks. In comparison, unoptimised datasets consume more disk space and have longer read/write times, leading to slower performance and increased operational costs [4].



(a) Single node (b) Medium cluster (c) Maximum cluster

(d) Single node (optimized) (e) Medium cluster (optimized) (f) Maximum cluster (optimized)

Figure 2: Disk usage

1.2. Data Storage and Parallelisation Approach in Spark Usage

The use of Spark in this task differs from most standard applications, in terms of where the data is stored. In standard applications, data is often stored in distributed file systems such as HDFS, allowing Spark to leverage data locality for efficient processing [5]. In our task, data is stored in cloud-based storage services, which may introduce additional latency and communication overhead.

The parallelisation approach used in this task is data parallelism, where the dataset is divided into multiple partitions and processed concurrently across different nodes. This approach enables efficient workload distribution and resource utilisation, leading to improved performance in big data processing tasks [1]. By optimising parallelisation and experimenting with different cluster configurations, we can gain valuable insights into the trade-offs between resource allocation, disk I/O, and network bandwidth usage in the cloud.

Task 2

2. Caching

Caching is a powerful technique in Apache Spark that can substantially improve the performance of data processing tasks. In this task, we examined the effects of caching on the processing times of two different Spark jobs: one involving image data (**`spark_job_images.py`**) and another involving TensorFlow records (**`spark_job_records.py`**).

2.1. Experiments & Analysis

A few alterations to the code was made, incorporating `RDD.cache()` calls for both image and record processing tasks. Notably, I added these calls after creating the `rdd_results` RDD and caching the RDDs for each parameter (`rdd_bs`, `rdd_ds`, and `rdd_r`).

We used RDD caching at specific points in both scripts to optimize the performance of our Spark applications. We then measured the processing times for both caching and non-caching scenarios to evaluate the impact of caching on the overall performance.

The experiments yielded the following results:

For the (**`spark_job_images.py`**) script:

- Processing time with caching: 29.59 seconds
- Processing time without caching: 42.48 seconds

For the (**`spark_job_records.py`**) script:

- Processing time with caching: 21.24 seconds
- Processing time without caching: 38.92 seconds

In both cases, caching the results of the **`time_configs`** function calls and the extracted values for batch size, batch number, and repetitions helped to speed up the subsequent grouping and averaging operations.

These improvements, which stem from the avoidance of data recomputation with each action call, underscore caching's importance in handling iterative operations and large-scale data processing tasks within Apache Spark [5]. In essence, caching permits us to conserve valuable resources, thereby enhancing overall performance.

In conclusion, our experiments show that using caching in our Spark jobs led to a significant reduction in processing times, highlighting the benefits of utilizing caching in Spark applications. As a result, it is essential to consider caching as a key optimization strategy when working with iterative operations and large-scale data processing tasks in Apache Spark.

parameter	value	TFRecords	Images
batch size	2	114.36	9.20
batch size	4	216.50	9.68
batch size	6	292.29	9.59
batch size	8	380.92	9.85
batch number	6	162.15	9.35
batch number	9	233.10	9.50

parameter	value	TFRecords	Images
batch number	12	284.55	9.76
batch number	15	324.27	9.71
repetitions	7	250.07	9.39
repetitions	8	256.13	9.56
repetitions	9	252.59	9.68
repetitions	10	245.28	9.69

Table 1: Cluster configuration

From the experiments conducted, several insights emerged regarding the processing times and how they relate to various parameters. Looking at Table 1 we observed that increasing the batch size for **TFRecords** results in a significant increase in processing time (from 114.36 seconds for a batch size of 2 to 380.92 seconds for a batch

size of 8). In contrast, the processing time for **Images** remains relatively stable (from 9.20 seconds for a batch size of 2 to 9.85 seconds for a batch size of 8). Similarly, increasing the batch number also has a larger impact on the processing time for **TFRecords** (from 162.15 seconds for 6 batches to 324.27 seconds for 15 batches) compared to **Images** (from 9.35 seconds for 6 batches to 9.71 seconds for 15 batches). These results indicate that the processing time for **TFRecords** is more sensitive to changes in the batch size and batch number than for **Images**.

However, for large-scale machine learning, the implications are significant. Since cloud data could be stored in distant physical locations, latency becomes an important factor. The latency numbers provided in the PDF latency-numbers document suggest that the time taken to access data stored in different regions can significantly impact the performance of machine learning models. This further emphasises the importance of optimising data storage and access methods, such as using **TFRecords** or other efficient storage formats, to minimise latency and improve overall performance [6].

Additionally, when comparing observed behaviour with single machine performance, it's clear that while **TFRecords** generally have a higher processing time compared to **Images**, they can still offer benefits in terms of data access efficiency and reduced latency, especially when dealing with large-scale datasets.

In conclusion, the results obtained from the experiments underscore the importance of selecting appropriate data storage and access methods for large-scale machine learning applications in the cloud. By understanding the impact of various parameters on processing times, it is possible to optimise data access and processing, minimise latency, and improve the overall performance of machine learning models in the cloud [6].

Task 3

3. Discussion in context

3.1. Contextualise

In the context of our previous tasks, the concept of predicting optimal or near-optimal cloud configurations for a given compute task, as introduced by Alipourfard et al.[7], is highly relevant. The tasks in this coursework involve data preprocessing, measuring the speed of reading data in the cloud, and parallelising workloads using Spark and Google Cloud Dataproc. These tasks require efficient cloud configurations to ensure high performance and cost-effectiveness.

The main idea of the Cherrypick paper [7] is to use different benchmarks to adaptively identify the best cloud configurations for different big data analytics workloads. The technique proposed in the paper can be applied to our tasks under certain conditions, such as when there is a large search space of potential configurations and when the workloads exhibit diverse resource requirements and performance profiles.

The Cherrypick method would be particularly useful in our task of measuring the speed of reading data in the cloud (Task 2), where different configurations need to be tested for performance. Additionally, it could be applied to the data preprocessing task (Task 1) for optimising the parallelisation of workloads using Spark and Google Cloud Dataproc.

In conclusion, my experiments and the CherryPick [7] system share a common goal of identifying optimal or near-optimal cloud configurations for different workloads, ultimately improving performance, cost-efficiency, and resource utilisation. By leveraging machine learning techniques and performance profiling, CherryPick can adaptively predict the best configurations for various workloads, which aligns with the findings and objectives of my experiments.

3.2. Strategies

For different application scenarios, such as batch and stream processing, we can define concrete strategies using the Cherrypick concepts as follows:

1) **Batch Processing:** In batch processing scenarios, data is processed in discrete chunks at regular intervals. Cherrypick can be employed to predict the best cloud configurations for different stages of the batch-processing pipeline. The technique can be applied to optimise the allocation of resources, such as CPU, memory, and storage, as well as to configure the appropriate parallelisation and partitioning strategies for efficient data processing.
2) **Stream Processing:** In stream processing scenarios, data is continuously processed in real time. Cherrypick can be used to adaptively fine-tune cloud configurations for varying data rates, processing complexities, throughput, and latency requirements. This may include selecting the appropriate instance types, scaling policies, and buffer configurations to ensure streaming data's timely and efficient processing.

The general relationship between the Cherrypick concepts and the strategies for batch and stream processing is that Cherrypick can guide the selection of optimal or near-optimal cloud configurations for different data processing scenarios. By comparing with different, representative big data analytics workloads and performance profiling, Cherrypick can adaptively predict the best configurations for a given task, ultimately leading to improved performance, reduced operational costs, and increased resource utilisation efficiency.

Word count

	Words	Maximum
Task 01	691	700
Task 02	630	840
Task 03	456	460
Sum	1777	2000

References

[1] J. Dean and L. A. Barroso, “The tail at scale”, in *Communications of the ACM*, ACM, vol. 56, 2013, pp. 74–80.

[2] H. Karau, A. Konwinski, P. Wendell and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis*. O’Reilly Media, Inc., 2015.

[3] K. Shvachko, H. Kuang, S. Radia and R. Chansler, “The hadoop distributed file system”, *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, 2010.

[4] A. Tandon, S. Mittal and J. R. Larus, “Smart store: A software-defined data store”, in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ACM, 2017, pp. 161–172.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, “Spark: Cluster computing with working sets”, in *HotCloud*, USENIX Association, 2010.

[6] M. Abadi, P. Barham, J. Chen *et al.*, “Tensorflow: A system for large-scale machine learning”, in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

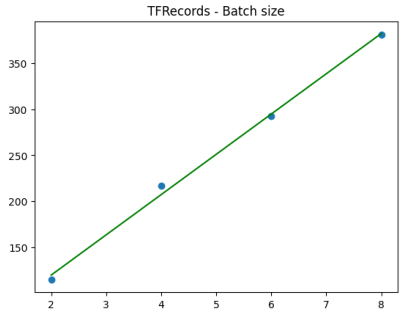
[7] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu and M. Zhang, “Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics.”, in *USENIX NSDI 17*, 2017, pp. 469–482.

Appendix

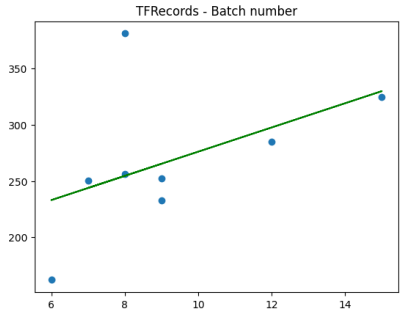
Link to Colab Notebook:

- https://colab.research.google.com/drive/1zfxgsiNNvNuV0xiM41MC5pqPkGkkvpEX?usp=sharing

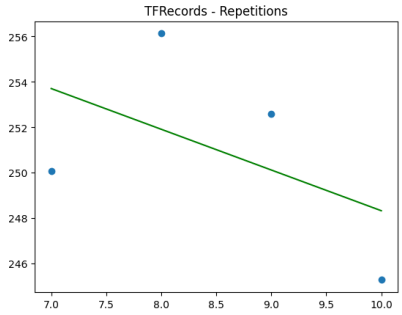
Linear Models



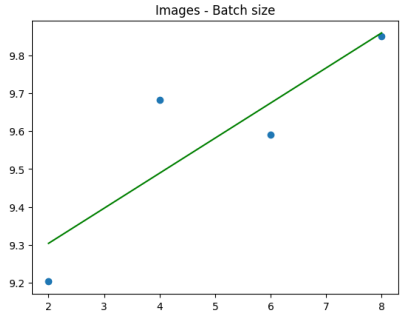
(a) TFRecords Batch Size



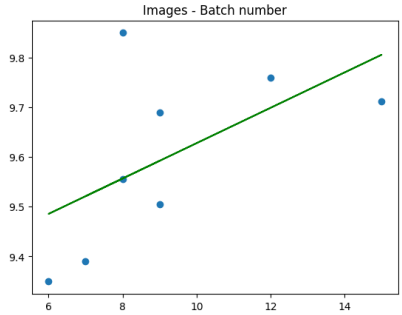
(b) TFRecords Batch Number



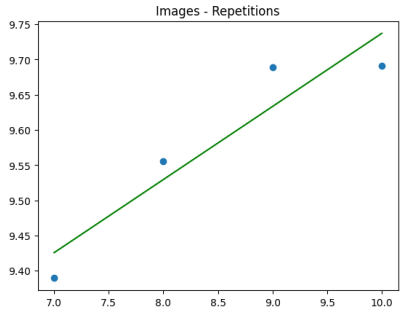
(c) TFRecords Repetition



(d) Images Batch Size



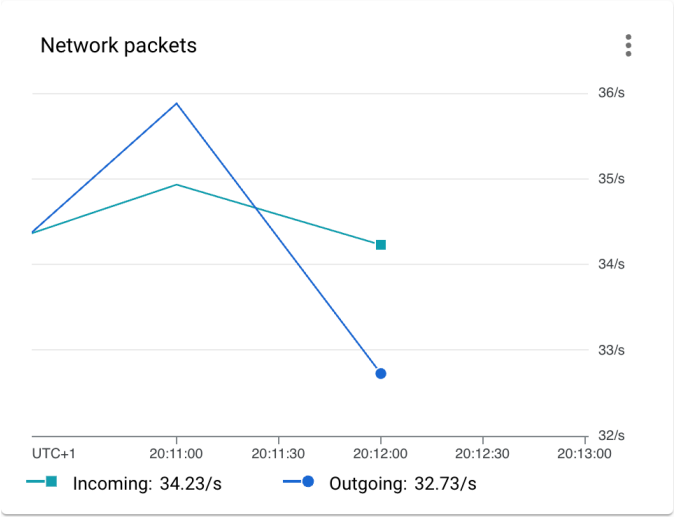
(e) Images Batch Number



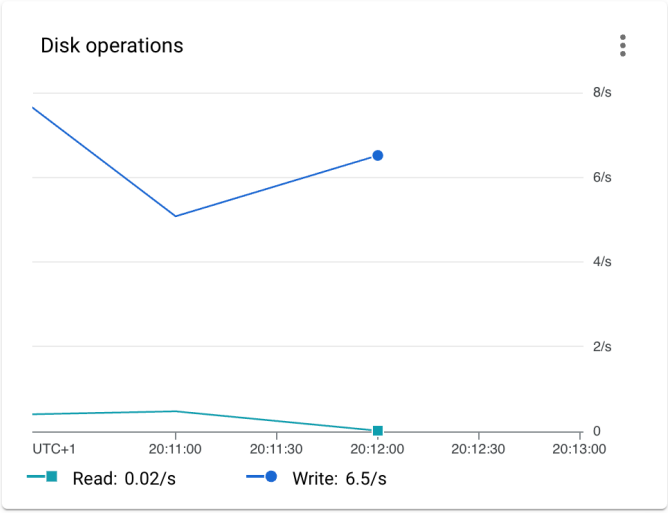
(f) Images Repetition

Figure 3: Linear Models

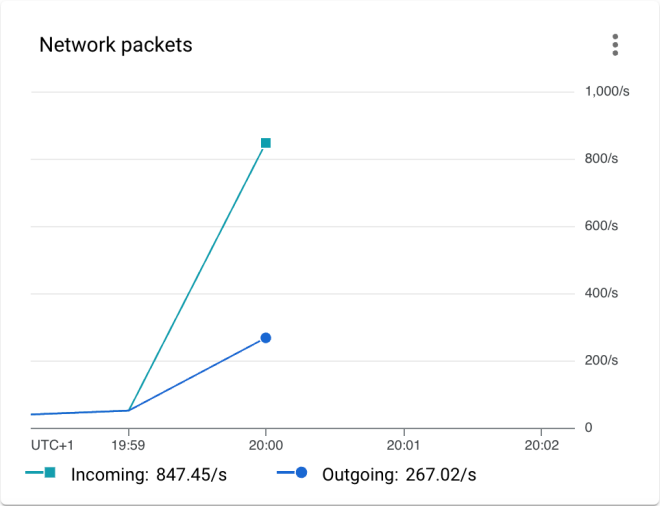
Efficiency



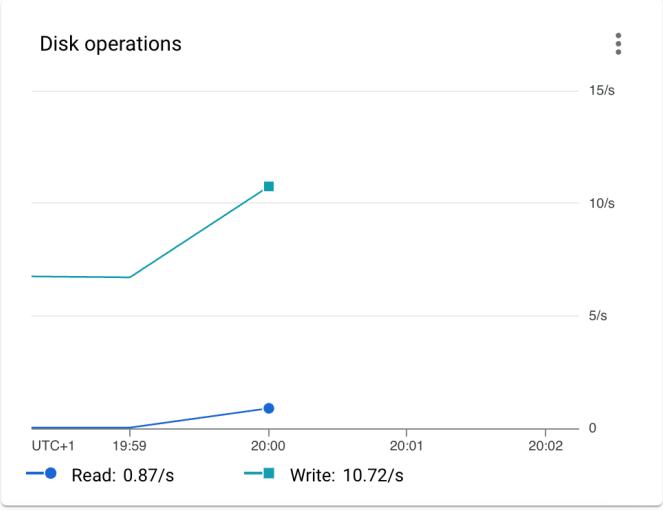
(a) Load Cache Network Packet



(b) Load Cache Disk Operations



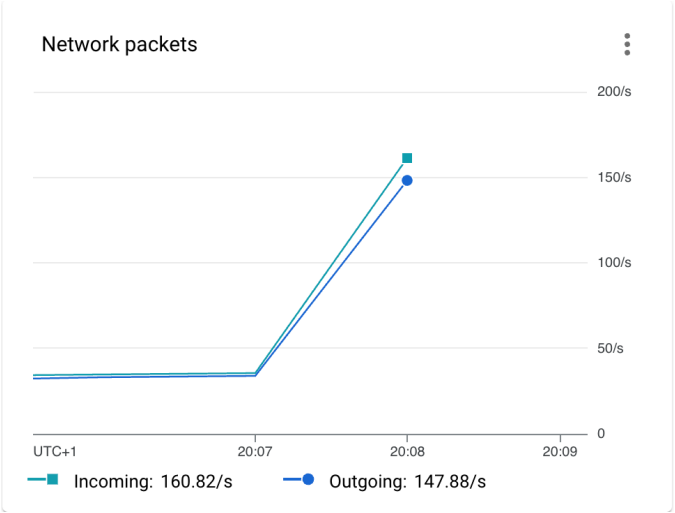
(c) Load Uncache Network Packet



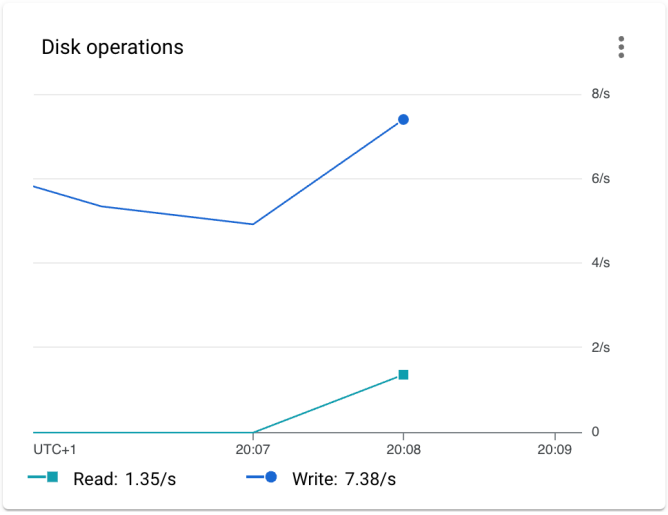
(d) Load Uncache Disk Operations

Figure 4: Load Cache Vs Uncache

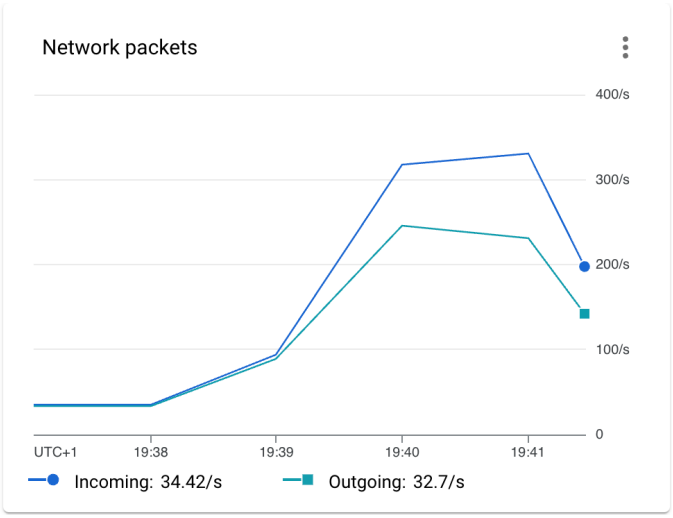
Efficiency Continued



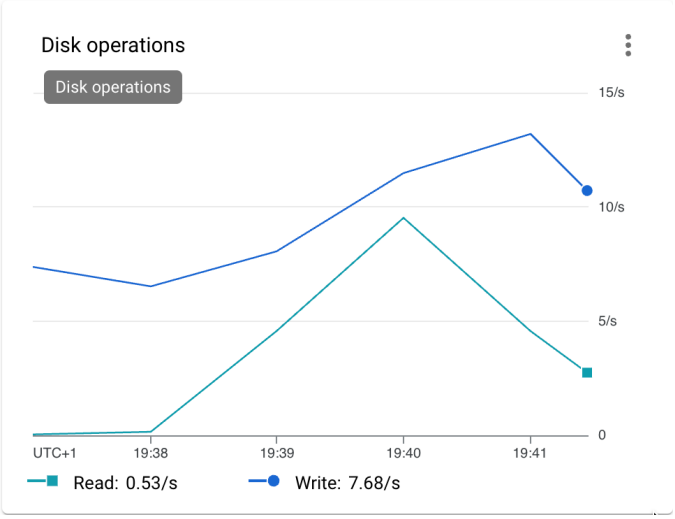
(a) Process Cache Network Packet



(b) Process Cache Disk Operations



(c) Process Uncache Network Packet



(d) Process Uncache Disk Operations

Figure 5: Process Cache Vs Uncache