Cloud Storage Performance Evaluation

Aidan Gray
University of California, Santa Cruz
Santa Cruz, California
aitgray@ucsc.edu

Charles Alders
University of California, Santa Cruz
Santa Cruz, California
calders@ucsc.edu

***Abstract*—Evaluating the performance of modern database systems is a crucial step in designing scalable and cost efficient cloud architectures. While the Yahoo Cloud Serving Benchmark (YCSB) provides a standardized method for comparing database performance, it assumes the target system is already deployed, configured, and network-accessible. This limitation makes it difficult for organizations to perform rapid and reproducible evaluations.**

This work presents an automated benchmarking framework which integrates cloud provisioning, containerized deployment, and YCSB workload execution into a single extensible system. We evaluate and compare two widely used database technologies, PostgreSQL and ScyllaDB, across multiple workloads. In addition, we collect system level metrics like CPU utilization, disk throughput, IOPS, and network traffic to compare application level performance with underlying resource behavior.

1. Introduction

Selecting a database for large-scale systems is a major decision with both performance and financial implications. Modern applications handle billions of read and write operations, across many diverse regions. Choosing a suboptimal data store can compound over years of operations, leading to high latency and unpredictability [1]. The goal of this project is to develop an evaluation framework capable of provisioning, deploying, and benchmarking multiple databases in both a standard and reproducible manner.

1.1. Motivation

Organizations often lack a lightweight and reproducible method to evaluate data performance under workload-specific conditions. A benchmark that can automatically deploy, populate, and access a database, can enable said organizations to avoid expensive architecture missteps.

1.2. Importance

Database selection is one of the primary factors in the scalability and longevity of distributed systems. A relevant example can be seen in Discord's evolution from MongoDB, to Cassandra, and eventually to ScyllaDB. Earlier performance insights may have prevented the need for intermediate migrations, reducing the overhead of operation. A unified framework for data driven comparisons can reduce risk and accelerate the decision making process.

1.3. Gap Addressed

While the Yahoo Cloud Serving Benchmark (YCSB) has been adopted widely for cross-database performance measurement, it assumes that the target system is already both deployed and configured. No existing tool provides a seamless pipeline that handles the provisioning of cloud resources, deployment of databases, execution of benchmarks, and the aggregation of those results. Our system addresses this gap through an extensible framework, which incorporates Terraform, Docker, and YCSB with support for PostgreSQL and ScyllaDB.

2. Background and State-of-the-Art

Benchmarking database systems has been a longstanding challenge, due to the variety of data models, storage engines, and concurrency strategies used in modern data platforms. As distributed systems continue to scale, performance evaluation has shifted from simple throughput measurements to more complex assessments measuring resource utilization, latency, and long term scalability.

2.1. Database Benchmarking Tools

YCSB is the most widely adopted framework for comparing NoSQL and SQL databases [2]. Introduced by Yahoo in 2010, YCSB standardizes benchmarking practices, enabling consistent cross-database comparisons. It's modular architecture provides bindings for dozens of databases including MongoDB, Cassandra, DynamoDB, and PostgreSQL.

While YCSB remains the de facto standard, it has a number of limitations. Most notably, it doesn't provision databases, manage cloud resources, or configure deployment environments. Users must manually install, tune, an expose each database before running any workloads. As a result, comparisons can suffer from inconsistencies relating to configurations, hardware selection, or deployment methodology.

2.2. Cloud Provisioning and Infrastructure Automation

Infrastructure-as-code (IaC) tools such as Terraform have made it possible to reliably deploy cloud resources with deterministic configurations. Terraform enables users to orchestrate compute instances, networks, and storage with minimal intervention [3]. While powerful, Terraform doesn't address the remaining steps needed to benchmark a database.

Containerization frameworks like Docker Compose provide an alternative, universal approach for benchmarking systems. These tools enable rapid deployment of standard setups, but lack the ability to provision cloud infrastructure.

3. Position of This Work

No current system provides a one-stop-shop solution which provisions cloud resources, deploys databases, executes benchmarks, collects metrics, and aggregates the results into reproducible summaries. This work closes that gap, combining the previously listed technologies into a unified framework, automating all stages of the setup and benchmarking process. Our system enables rapid comparisons across SQL and NoSQL databases, beginning with PostgreSQL and ScyllaDB, and can be expanded to additional databases with minimal modification.

This approach represents a step towards standardized, automated cloud benchmarking, and provides a practical foundation for academic research and industry evaluations.

4. Approach / Hypothesis

4.1. Architecture

The core of our project is three Google Cloud Platform (GCP) virtual machines (VMs). We provision one VM for each database and a separate VM to run the benchmark queries. Our first tests ran the benchmarks from our own devices, which we quickly realized was far from ideal due to network latency. By provisioning a separate benchmarking VM in the same datacenter, we are able to run queries over LAN which removes the network latency bottleneck and more realistically emulates how a production application would behave.

Our baseline VMs run on the `n2d-standard-2` machine type, which corresponds to 2 vCPUs and 8GB of RAM. For our database VMs, we also attach a 375GB local NVME SSD. A local SSD is physically connected directly to the same rack as the VMs, meaning much higher throughput compared to other network-attached storage. This is the recommended configuration for ScyllaDB on GCP. Additionally, all virtual machines run Google's Container-Optimized OS [4]. Container-Optimized OS is a lightweight image that is incredibly quick to spin up and serve a containerized application. We will mention the downsides of this OS image later in this paper.

All of this configuration is managed and provisioned by our main Terraform file. Terraform is an infrastructure-as-code tool built to both provision and manage cloud resources via a configuration language. This tool enables us to easily spin up new cloud services as necessary, automating many of the steps required to set up a database.

This managed provisioning allows us to automatically run scripts on VM startup, removing any manual database setup steps. Both database VMs must first create respective data storage directories to mount with Docker, find the local SSD, make a new ext4 filesystem, and finally mount the drive. This initial step ensures all data is stored on the fastest storage available on our machines. After mounting the drive, each VM will wait for the database to listen for connections, then execute the required initialization queries provided in the YCSB documentation. For example, the Scylla VM must create a new keyspace and a new table within the keyspace. This step must be completed before benchmarking.

These setup steps will vary between databases, but this Terraform template can easily be used in the future to benchmark to other databases without much hassle.

4.1.1. Databases. Our initial evaluation focuses on two representative databases: PostgreSQL and ScyllaDB. We selected these two because we wanted to compare a tried and true SQL-based database with a newer cutting-edge database built with modern database architecture in mind. Postgres maintains the traditional relational (SQL) paradigm: a fixed schema, ACID compliance, and mature query optimization engines. In contrast, Scylla leverages a shard-per-core, horizontally scalable architecture designed for high concurrency and large-scale distributed workloads. By benchmarking both under identical workloads, deployment mechanisms, and orchestration paths, we aim to compare their performance trade-offs under fair and reproducible conditions. Specifically, we test against Postgres version 9.6 and ScyllaDB version 2025.2 [5], [6], [7].

4.2. Deployment

As mentioned above, we primarily use Terraform to handle our deployment to Google Cloud Platform. Our primary README.md file includes straightforward instructions to install the necessary dependencies: Google Cloud's CLI tool and Terraform. After installation, GCloud CLI requires the user to authenticate with a guided process. This installation and authentication are only necessary once.

Once the prerequisites steps are finished, our infrastructure can be easily deployed in a single command. From the root of our project, execute `cd terraform && terraform apply`. Terraform will ask for review, then the resources will begin provisioning.

GCP-based virtual machines, while convenient and powerful, may not represent the optimal environment for every user. Therefore, we also provide a compose.yml configuration file to enable a broader range of use cases, from lightweight deployments to full-scale cloud development. This alternative enables the whole system to be deployed

both quickly and reproducibly on a single host or container platform. In effect, offering VM deployment via Terraform and container-based deployment through Docker [8] maximizes the flexibility of this tool, significantly lowering the barrier to entry for evaluation, experimentation, or development.

4.2.1. YCSB and Workloads. Our benchmarking VM runs a custom Docker image we built and published to Docker Hub. This image contains all of the steps to install YCSB and the dependencies needed to run tests against Postgres and Scylla. Notably, the YCSB Scylla benchmark requires JDK8, which can be a pain to install manually. This image also contains many pre-defined workload scripts that use YCSB as a base, but adjust parameters as needed for the different types of tests. We base all YCSB tests on "workloada", which tests a 50/50 split of reads and updates. We will discuss the parameters for each type of test later in this paper.

4.2.2. Additional Benchmarks. We also pre-load our benchmark VM with pgbench and cassandra-stress benchmarking utilities. While these additional benchmarks cannot compare the databases to each other, we felt that this addition is crucial when measuring the databases against themselves (vertical and horizontal scaling). Again, we will discuss the specifics of each test below.

4.3. Testing

The evaluation suite collects latency and throughput metrics across repeated benchmark iterations to ensure statistical reliability. Variables such as workload type, request distribution, and concurrency level can be adjusted. The following primary metrics are recorded:

Metrics measured (benchmarks):

- Overall run time (ms)
- Overall throughput (ops/sec)
- Average Latency(μs)
- Minimum Latency(μs)
- Maximum Latency(μs)
- 95th Percentile Latency(μs)
- 99th Percentile Latency(μs)

In addition to these metrics, we also measure hardware metrics using GCS monitoring services, focusing primarily on:

- CPU Utilization
- Memory Utilization
- Disk Throughput
- Disk IOPS
- Network Traffic

Utilizing these metrics, direct comparisons can be made between various workloads, demonstrating how the databases perform under certain conditions. These metrics are integral to determine the scalability and resilience of each database, enabling users to determine the optimal database for their use case. By providing automated graphing tools, these metrics can be compared with relative ease,

4.3.1. Hypothesis. We hypothesize that ScyllaDB, a massively parallelization append-optimized NoSQL system [9], will outperform PostgreSQL under high write concurrency and large dataset sizes. Conversely, we expect PostgreSQL will maintain a significantly better read latency for smaller workloads, due to its mature query optimizer and B-tree structure [10], [11]. These hypothesis guide the experimental design and interpretation of benchmarking results.

5. Experimental Results

We first compare the end-to-end performance of Postgres and Scylla across the 4 workloads using throughput metrics.

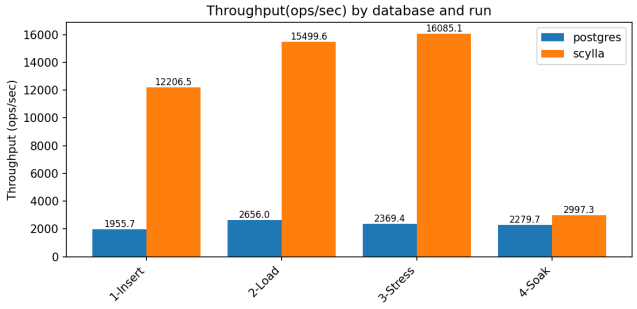

Figure 1: Throughput for each workload and database.

There's a clear-cut difference between the two databases in terms of throughput. In every scenario Scylla performed significantly better than Postgres, due in part to it's aggressive caching strategy. It's important to note; however, that the version of Postgres used is outdated. As discussed further in the limitations section, this constraint arises from restrictions imposed by YCSB and it's available bindings, as well as some flawed design decisions. The observed performance gap should not be interpreted as a definitive statement on the relative performance of contemporary versions of Postgres, but instead as proof of the functionality of the tool, which leverages a flawed benchmarking tool.

As mentioned previously, we also tracked the individual hardware metrics during each benchmark, the following subsections detail the results discovered from each run. We've included many of the hardware metrics we've collected through GCP, though we've excluded some of the metrics for the sake of brevity. We will still discuss the results of our memory related metrics later, but the images were excluded to prevent bloat.

Finally, we realized after running the YCSB benchmarks that our local SSD was not included in Google Cloud Observability. Due to this finding, we have removed the disk figures and will present an additional test that demonstrates reads, writes, and caching.

5.1. Insert Workload

We begin our testing by running the YCSB insert workload against both databases, inserting a total of 10,000,000 entries.

Figure 2 shows the CPU utilization on GCP during the insert benchmark for Scylla (top) and PostgreSQL (bottom).

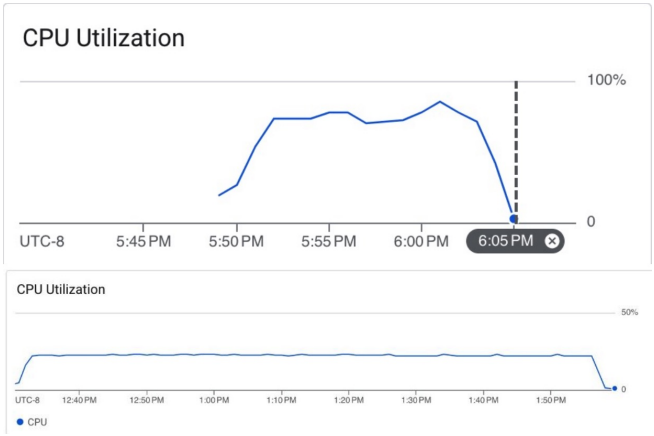

Figure 2: CPU utilization during the insert workload Scylla (top) and PostgreSQL (bottom)

We can see that this single-node Scylla cluster does an excellent job at using its available CPU resources, while Postgres stays steady at very low utilization. This behavior arises due to the way YCSB interacts with the Postgres driver. Postgres typically benefits from connection pooling and parallel client connections; however, in our configuration, YCSB maintained only a single active connection. This effectively serialized request handling, saturating that connection and preventing throughput from exceeding approximately 2,650 operations per second.

5.2. Load Workload

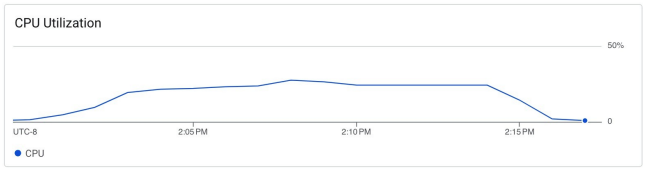

Figure 3: CPU utilization during the load workload (PostgreSQL).

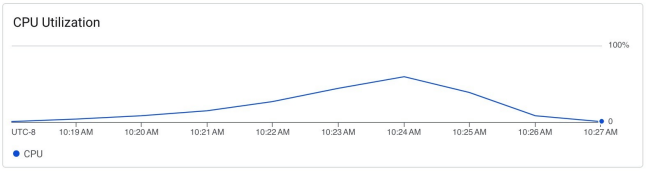

Figure 4: CPU during the load workload (ScyllaDB).

Here we see the benefits of Scylla's parallelization. While Postgres reaches a relatively stable usage state, hovering just over 25%, Scylla quickly ramps up and down without ever reaching it's maximum throughput. While a larger workload may have displayed Scylla reaching a higher or more consistent utilization, the chosen workload more accurately reveals the disparity between Postgres and Scylla, revealing a significantly improved throughput.

5.3. Stress Workload

Figure 5: CPU utilization during the stress workload (PostgreSQL).

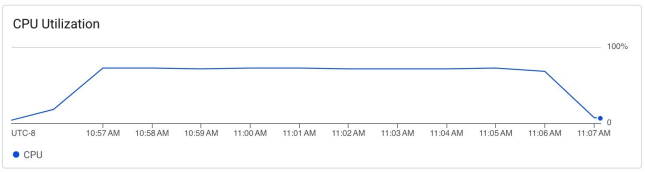

Figure 6: CPU utilization during the stress workload (ScyllaDB).

The stress workload reveals a similar pattern, Scylla is able to ramp up higher and utilize more of it's allocated CPU resources due to it's parallelization of the workload. Postgres, limited by it's single connection, is never able to reach the same throughput or CPU utilization.

5.4. Soak Workload

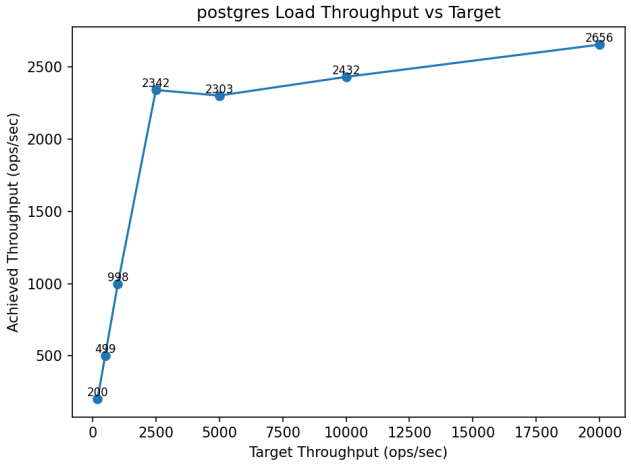

Figure 7: Overall throughput (PostgreSQL).

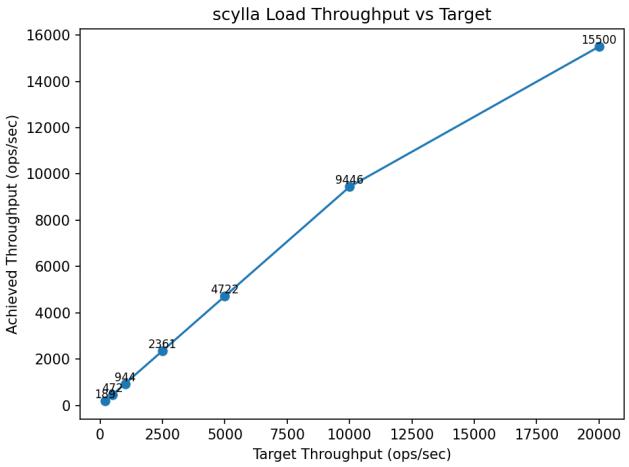

Figure 8: Overall throughput (ScyllaDB).

To illustrate the findings of the previous tests, the soak workload was used to test the databases realistic performance against a target throughput. While Scylla scales linearly and adheres to its target relatively closely, Postgres reaches a bottleneck at a throughput around 2350. While some small gains are made afterwards, this benchmark serves to highlight just how important it is to test Postgres properly.

5.5. Spike

We implemented spike testing without YCSB due to the concurrency issue we experienced with Postgres. This adjustment allows for a much more insightful look into how these databases handle heavy and spiked loads.

Using cassandra-stress, we spike test ScyllaDB starting with 4 client (benchmark query) threads until 404 threads, incrementing by 100. Some cooldown time in between would have been ideal, but this is a limitation of cassandra-stress.

To test Postgres, we implemented our own test using pgbench as a baseline, running 1000 transactions per client, with 8, 75, 8, then 100 clients.

Figure 9: CPU utilization during spike workload (Postgres).

Figure 9 shows that we have finally utilized most of the CPU on the Postgres instance. This is drastically different from YCSB, since we are now opening a new connection for each client, rather than attempting to share the same connection. We hit 100 open connections, which is the default maximum for new Postgres installations. The GCP Observability CPU utilization graph does not appear to spike, which is due to the minute-by-minute reporting that completely smoothed the curve.

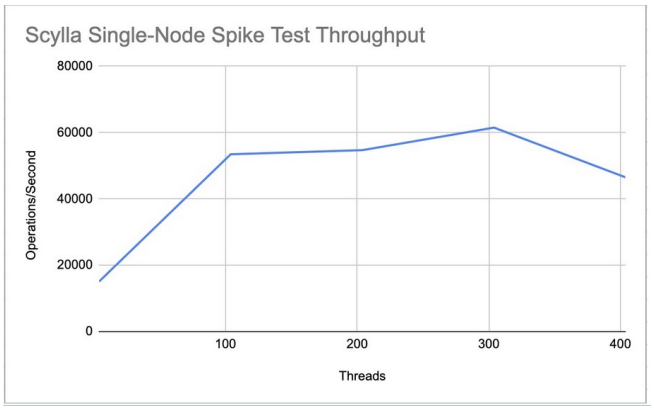

Figure 10: Scylla single-node spike test throughput.

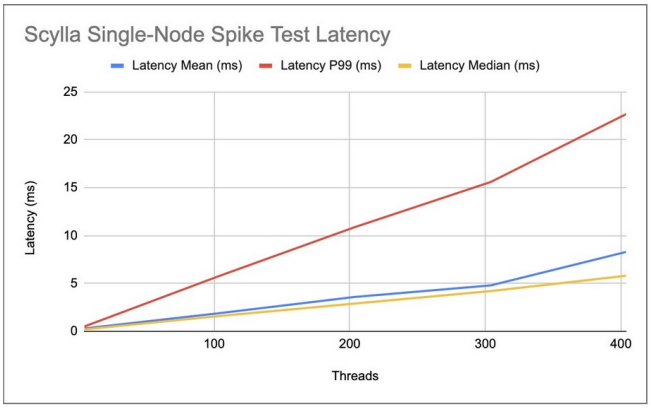

Figure 11: Scylla single-node spike test latency.

For ScyllaDB, we notice consistently strong performance across the entire spike test. We notice the throughput start to fall off once the test gets to 404 threads in Figure 10, and we also observe all latency metrics steadily increasing as the number of client threads grow in Figure 11. Despite these increased metrics, we did not observe a single query fail, and the latency was completely tolerable with the P99 staying below 25ms even with 404 threads.

5.6. Vertical Scaling

We further explore comparing these databases against themselves in a vertical scaling test. For both Postgres and Scylla, we compare throughput and latency for three machine types: n2d-standard-2, n2d-standard-4, and n2d-standard-8. These machines have 2, 4, and 8 vCPUs, as well as 8, 16, and 32GB of RAM respectively. We tested Scylla with 1,000,000 reads against a database with 1,000,000 entries across all machines. We tested Postgres with 8 client connections sending 5000 transactions each.

We observe some expected and some unexpected behavior. We found that increasing the VM size from 2 vCPUs and 8GB RAM to 4 vCPUs and 16GB of RAM showed a negligible gain in both throughput and latency for both Scylla and Postgres. We then see the next jump to 8 vCPUs and 16GB RAM to be quite large, with both databases seeing an almost 50% increase in throughput and a significant reduction in latency.

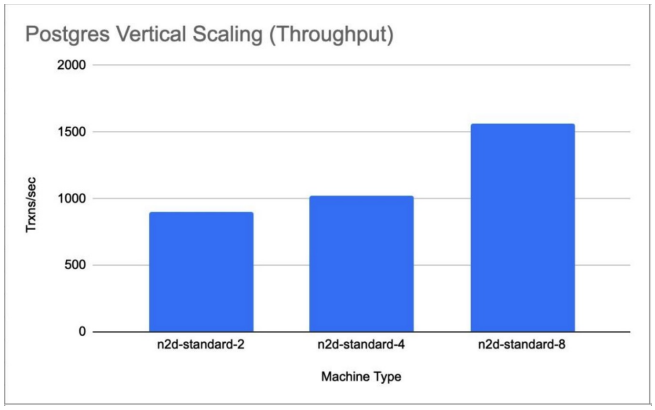

Figure 12: Vertical scaling throughput (Postgres).

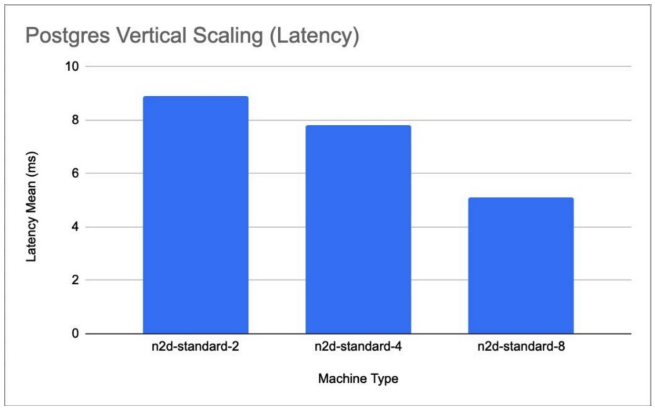

Figure 13: Vertical scaling latency (Postgres).

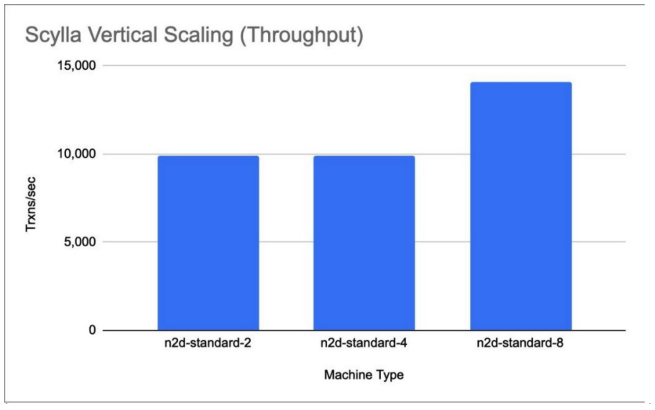

Figure 14: Vertical scaling throughput (Scylla).

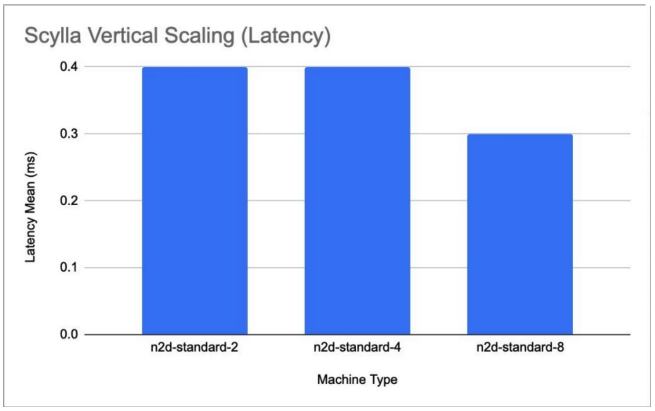

Figure 15: Vertical scaling latency (Scylla).

5.7. Horizontal Scaling

We also tested ScyllaDB horizontal scaling, as it is unusual to run Scylla with a single node due to its distributed design and all of the benefits that come with it (fault tolerance, horizontally scaling storage, etc.).

For this test, we run cassandra-stress with 1,000,000 writes and reads on 1, 2, and 3-node Scylla clusters. All nodes were provisioned using Terraform, and run on Scylla's proprietary OS image.

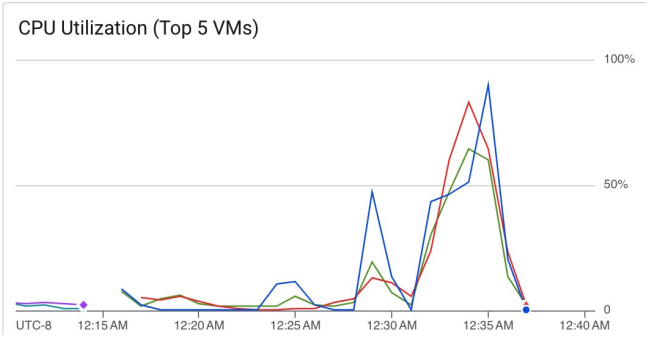

Figure 16: Scylla 2-nodes CPU usage.

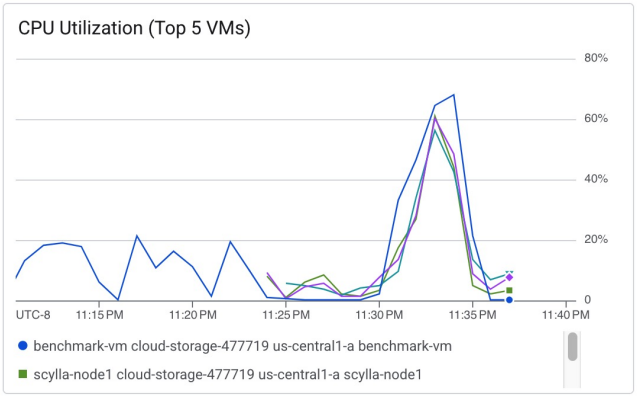

Figure 17: Scylla 3-nodes CPU usage.

Figures 16 and 17 very clearly show the difference in CPU utilization between 2 and 3-node clusters. In both figures, the blue line (highest CPU utilization peak) represents the benchmark VM, where the others are Scylla nodes. We can see that in a 2-node cluster (Figure 16), one of the nodes shows roughly 80% CPU utilization, where the other peaks around 60%. The 3-node cluster (Figure 17) shows all Scylla nodes perfectly load balanced, peaking at roughly 60% CPU. This could have to do with the data partitioning in the workload, where a disproportionate amount of the data ended up on one node in the 2-node cluster.

5.8. ScyllaDB Cloud

To supplement our monitoring mistakes from Google Cloud Observability, we ran one more test on a 3-node Scylla cluster, deployed on the same GCP datacenter as our other tests, this time using ScyllaDB Cloud. This test is run on a free-trial cluster of three n2-highmem-2 machines with 2 vCPUs and 16GB RAM each. We ran a cassandra-stress test of 10M reads against a database with 10M records. cassandra-stress runs the read test with an increasing number of client threads, from 4 to 404 in increments of 100. We ran this test specifically to observe Scylla caching patterns.

Figures 18 and 19 show the reads and cache hits/misses respectively. We observed that Scylla records a cache hit for nearly 80% of read operations across each spike. This is impressive, and it is due to the fact that Scylla attempts to keep mostly hot rows [12] in its cache, which likely coincides nicely with the benchmark workload. We also noticed not a single cache miss despite the roughly 20% difference in cache hits vs actual reads. This is quite strange, and our only theory is that Scylla knows when a row is cold enough to not even check the cache at all.

Figure 18: Scylla 3-node cluster reads with 4, 104, 204, 304, and 404 threads.

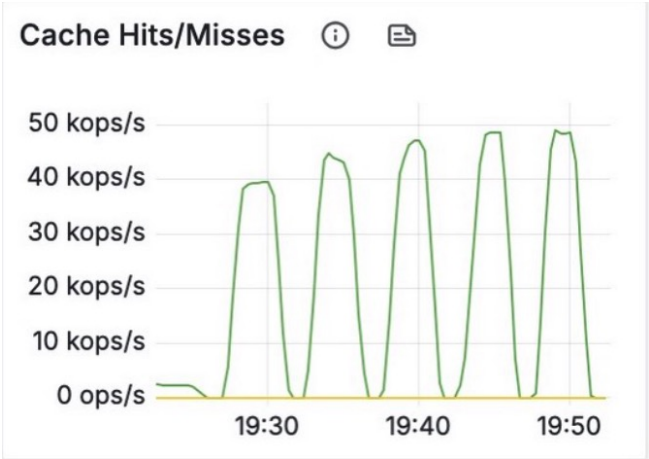

Figure 19: Scylla 3-node cluster cache hits/misses with 4, 104, 204, 304, and 404 threads reading.

6. Resources Needed

6.1. Hardware

When deploying on a cloud provider like Google Cloud Platform (GCP), all hardware resources are provisioned on demand and no physical infrastructure is required from the user. For local execution, however, it is important that the benchmarking machine is sufficiently provisioned so that the client, rather than the database, does not become the performance bottleneck.

The table below lists the recommended hardware for PostgreSQL and ScyllaDB, as well as some more minimal viable configurations.

TABLE 1: Recommended Hardware Guidelines for PostgreSQL and ScyllaDB

Resource	PostgreSQL	ScyllaDB
CPU	2–8 cores for OLTP; 8–32 cores for analytics	Shard-per-core; 8+ dedicated cores recommended
Memory	8–32 GB; relies heavily on OS page cache	16–64 GB; memory-critical for caching SSTables
Disk	NVMe SSD strongly recommended; low latency for WAL writes	High-IOPS SSD/NVMe; disk latency limits shard throughput
Network	1 Gbps minimum; 10 Gbps for replication-heavy systems	10 Gbps preferred in distributed clusters
Minimal Config	1 vCPU, 1–2 GB RAM (non-production)	4+ cores, 8+ GB RAM (minimum stable)

As for our own hardware, we utilized VM sizes of between 2 and 8 vCPUs, between 8 and 32GB of RAM, and a 375GB NVME local SSD for each database instance. We wanted to start small and test the system as hardware increased, rather than the other way around.

6.2. Software

The benchmarking framework relies on a minimal set of software dependencies. The following tools and versions were used in our experiments:

- **Terraform** for infrastructure provisioning.
- **Google Cloud CLI** (gcloud) for authentication and VM management.
- **Docker** and **Docker Compose** for local containerized deployment.
- **JDK8** a dependency for the Scylla YCSB binding.
- **YCSB** (0.17.0) installed into our Docker image for workload execution.
- **PostgreSQL** (9.8)
- **ScyllaDB** (2025.2)

These dependencies allow the benchmarking pipeline to operate consistently across cloud and local environments. Version alignment is strongly encouraged, components like YCSB are prone to failures when the recommended versions are not strictly adhered to.

7. Timeline

Nov 11: Initial Terraform infrastructure created.
Nov 18: First benchmark iteration completed.
Nov 22: First Docker-Compose file complete, terraform initializes databases properly.
Nov 30: Specific insert, load, stress, and soak benchmarks created.
Dec 2: Data processing prototype script complete.
Dec 7: Horizontal and Vertical scaling benchmarks complete, data processing script finalized.

8. Discussion

The results across the 4 YCSB workloads show a clear and expected performance difference between Postgres and Scylla, which stem from their architectural differences. The aggregate runtime and throughput measurements (Figures **??**–1) demonstrate that ScyllaDB consistently delivers higher sustained throughput and lower end-to-end runtime under write heavy and highly concurrent workloads. This is largely due to the age of the Postgres version utilized.

8.1. Insights

The flaws of the benchmarking process serve to understate the necessity of fair benchmarks between equivalent databases. Without fair benchmarks, accurate decisions cannot be made and trust in the tool is eroded, making the tool unreliable. Despite these flaws, we successfully demonstrated that the system designed functioned as intended. While a dependency of the project forced concessions to be made, the underlying system developed still has value. If YCSB were to be modernized, by either the maintainers or the users of the tool, the system would still function as intended, providing users with the most accurate comparisons possible.

With that in mind, we perform analysis on the following workloads as if we were comparing equivalent systems, to demonstrate an example of the conclusions the user would draw if YCSB was implemented optimally.

8.1.1. Insert, Load, and Stress Workloads. These workloads reveal that Scylla is able to better utilize it's allocated resources, reaching a higher and more consistent CPU utilization. This indicates that the database is efficiently converting the available compute into useful work. The system is more optimally saturating the CPU, revealing its effective parallelization of the workload.

8.1.2. Soak Workload. In the soak test, a target throughput was specified and compared against the empirically measured throughput to evaluate the gap between theoretical and realized performance. The difference is immediately apparent: Scylla consistently maintains throughput closer to it's target (as seen in Figure 8), whereas Postgres falls noticeably behind (Figure 7). This behavior arises largely from the way YCSB interacts with the Postgres driver. Postgres typically benefits from connection pooling and parallel client connections; however, in our configuration, YCSB maintained only a single active connection. This effectively serialized request handling, saturating that connection and preventing throughput from exceeding approximately 2,650 operations per second.

8.1.3. Resource utilization patterns.

- CPU: Scylla generally maintained a higher and more consistent CPU utilization.
- Memory: Provided a sufficiently large workload, Scylla will utilize all available memory due to it's aggressive caching strategy. This leads to optimal latencies during high usage periods.
- Disk: Unfortunately, it was discovered that GCP Monitoring could not accurately track the disk usage of either database. If deployed on Kubernetes, it is strongly recommended to utilize a tool like Jaeger for monitoring, though the setup of such a tool was outside of the scope of this work, as GCP was the primary platform considered. We did notice that, even in a database of 10M entries, Scylla hit its cache nearly 80% of the time. Only 20% of reads ever reached the disk, and the NVME local storage was unlikely to be a bottleneck.
- Networking: Networking bottlenecks were never reached and network usage remained relatively consistent.

8.1.4. Cost Comparison. The hardware configuration used as a baseline in our tests (n2d-standard-2) incurs a monthly cost of roughly $61.63. For every double in vCPUs and memory (e.g., n2d-standard-2 to n2d-standard-4), the monthly cost also doubles.

When vertically scaling Postgres, we see a throughput speedup of 1.14x when going from 2 to 4 vCPUs and 8 to 16GB of RAM, and a cost increase of 2x. From 2 to 8 vCPUs and 8 to 32GB of RAM, we see a speedup of 1.74x and a cost increase of 4x. Vertical scaling is common practice for Postgres instances, but the cost does not scale linearly with throughput from our testing.

When horizontally scaling Scylla nodes, we noticed that 1 to 2 nodes showed minimal improvement while incurring twice the monthly cost. On the other end, scaling Scylla from 2 to 3 nodes yielded 1.42x throughput speedup for 1.5x the monthly cost. This was the most cost effective upgrade from our tests.

In terms of raw throughput and latency, Scylla performed significantly better for less resources than Postgres.

8.2. Limitations

It's imperative to note that while benchmarks are valuable, they are not flawless indicators of real-world performance. A real-world system will face many scenarios that are difficult or even impossible to predict, and the loads encountered by deployed systems are often dependent on the individual use case. The fact that the workload of YCSB is variable is certainly beneficial, users can tune the benchmarks to match their expected workloads, but the most accurate benchmarking suite would most certainly require additional benchmarks straining every component of the system, both individually and in conjunction with each other.

As mentioned previously, the last published version of YCSB was released in 2019. While some databases, such as Scylla, maintain strong backward and forward compatibility, we discovered that the bindings for Postgres and its dependencies were incompatible with a contemporary version of the database. Although it's technically possible to manually update these bindings, doing so introduces significant complexity and undermines one of the primary objectives of this project: ease of use and reproducibility. Incorporating a custom-modified version of YCSB directly into the codebase would substantially increase the size and maintenance burden of the project. Furthermore, it would require manual integration of any future upstream YCSB updates. Consequently, to preserve simplicity, usability, and long-term maintainability, the decision was made to evaluate Postgres using an older, YCSB-compatible version.

We had multiple issues with GCP Observability. The metrics are refreshed once per minute, which completely misses quick spikes in performance. Additionally, there are no metrics available for local SSDs. Grafana would have been an excellent choice for this project, but Container-Optimized OS does not have a straightforward way to install this. A different OS image with Grafana installed would have been a better approach, but was not possible due to time constraints.

8.3. Peer group testimonial

We got positive feedback from the group conducting the peer review. They were able to follow all of our local deployment steps to get both databases running and initialized with Docker Compose, then run short benchmarks against each one. The whole process took less than 10 minutes. When running the test on MacOS, Ramon was not able to immediately pull the Docker image, since it is not built for ARM, and Docker Hub does not pull from different architectures by default. This was resolved by explicitly pulling the x86 variant of the image first, which we have added to our README.

9. Conclusions and further works

All source code, graphs, and results are available at https://github.com/char26/cse239-cloud-storage

9.1. Key Takeaways

- Scylla consistently outperforms Postgres, demonstrating superior throughput and runtime characteristics.
- Postgres maintains competitive latency and predictable behavior, but shows scalability limitations.
- Resource utilization patterns reveal that Scylla achieves higher but less consistent utilization, whereas Postgres maintains higher disk write speeds.
- Scylla shows strong stability during long duration workloads, with consistent throughput and minimal performance degradation over time.
- The benchmarking framework successfully provides reproducible, automated comparisons and can be extended to additional databases and workloads with minimal effort.

9.2. Possible Extensions

As mentioned previously in the paper, this system is intended to enable a DevOps team to compare a multitude of databases with a trivial amount of effort, so a natural extension of this work would entail the addition of the other databases included in the YCSB suite.

Additionally, it's important to note some of the flaws of the YCSB suite as a whole. The workload utilized by YCSB was quite simple, it prioritized universality as opposed to depth, so the insights derived from the benchmark may not be indicative of real-world performance. A second natural expansion would be in relation to the benchmarking suite, to create a set of workloads more similar to real-world interactions, enabling the best possible comparisons. In a similar vein, the flaws mentioned in the limitations section would also need to be addressed.

This would would also benefit from an extension in terms of depth as opposed to breadth. A deeper exploration of Scylla's horizontal scalability through large-scale, multi-node deployments would be beneficial. The architecture of Scylla is intended to scale linearly in throughput with the addition of nodes; however, we encountered practical challenges in achieving consistent scaling behavior. These limitations may stem from configuration complexity or deployment assumptions, the precise cause remained elusive during our testing. Future work should include systematic evaluation across a greater number of nodes and a wider range of replication and sharding configurations. Such an investigation would help to determine whether the scaling

limitations were due to configuration or architectural constraints. In theory, Scylla's architecture should enable near infinite scaling, and validating this property under rigorous experimental conditions is an important direction for continued study.

A final extension would come in the form of a quick comparison guide, detailing the feature sets of each benchmark. While data and metrics are great first step, understanding the functionality of a benchmark is imperative when choosing a database. Therefore, providing an overview of each benchmark, the pros and cons per se, could assist a user in narrowing down the databases they wish to benchmark, making this tool the only resource required when selecting a database for a product or service.

Acknowledgments

Despite our qualms with YCSB, the technical implementation of a benchmarking tool that accurately compares the performance between SQL and NoSQL databases is a feat that cannot be understated. To create the benchmarks we utilized would have been a tremendous undertaking, so it's important to recognize the fact that YCSB was integral to the development of this work. In a similar vein, pgbench and cassandra-stress enabled us to push the VMs running the containers to their limits, revealing their overall efficiency and when scaling might be required.

We also want to acknowledge our peers who assisted in this project, specifically Ramon Torres who provided feedback regarding the design and implementation of our system.

References

[1] B. Ingram. (2023, Mar. 6) How discord stores trillions of messages. Accessed: 2025-12-01. [Online]. Available: https://discord.com/blog/how-discord-stores-trillions-of-messages

[2] B. F. Cooper *et al.*, "Ycsb: Yahoo! cloud serving benchmark," 2010, accessed: 2025-11-26. [Online]. Available: https://github.com/brianfrankcooper/YCSB

[3] HashiCorp, "Terraform — infrastructure as code," 2025, accessed: 2025-11-20. [Online]. Available: https://www.terraform.io/

[4] G. Cloud. (2025) Container-optimized os overview. Accessed: 2025-12-08. [Online]. Available: https://docs.cloud.google.com/container-optimized-os/docs/concepts/features-and-benefits

[5] S. Inc., "Scylladb - monstrously fast + scalable nosql database," 2025, accessed: 2025-11-24. [Online]. Available: https://www.scylladb.com/

[6] P. G. D. Group, "Postgresql documentation," 2025, accessed: 2025-11-24. [Online]. Available: https://www.postgresql.org/

[7] D. Hub, "scylladb/scylla: Official docker image," 2025, accessed: 2025-11-24. [Online]. Available: https://hub.docker.com/r/scylladb/scylla

[8] D. Inc., "Docker — empowering app development for developers," 2025, accessed: 2025-11-22. [Online]. Available: https://www.docker.com/

[9] S. Inc. (2025) Shard-per-core architecture. Accessed: 2025-12-04. [Online]. Available: https://www.scylladb.com/product/technology/shard-per-core-architecture/

[10] L. Albe. (2018) How the postgresql query optimizer works. Accessed: 2025-12-04. [Online]. Available: https://www.cybertec-postgresql.com/en/how-the-postgresql-query-optimizer-works/

[11] M. Makovetsky. (2024) Postgresql b-tree internals. Accessed: 2025-12-04. [Online]. Available: https://makotons.substack.com/p/postgresql-btree-internals

[12] T. Grabiec, "How scylla data cache works," *ScyllaDB Blog*, Jul. 2018, accessed: 2025-12-08. [Online]. Available: https://www.scylladb.com/2018/07/26/how-scylla-data-cache-works/