Bachelor's thesis

Bachelor of Engineering, Embedded Software and IoT

2023

Nicholas Navarro

Comparison of Load Balancers and Replication Methods for Big Data Storage

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Embedded System Software and IOT

2023 | 52

Nicholas Navarro

Comparison of Load Balancers and Replication Methods for Big Data Storage

This thesis will focus on how load-balancing proxies, a new paradigm in cloud computing, can benefit modern database deployments. The theory section will explain the requirements and use cases by describing the features of current database technologies. The thesis will continue to provide test cases that will be carried out to benchmark these systems in both an acute context where data requests can exceed the hardware limits of the database as well as an everyday context where the synthetic load will be comparable to simple data requests. The focus will be on the speed at which the databases can handle the traffic, using Transactions per Second (TPS) as the comparing metric.

Keywords:

Big Data

Database Management Systems

Cloud Computing

Content

Appendices

Appendix 1. Previous Work Placement at HoistGroup Oy

Appendix 2. Table of Results

List of abbreviations (or) symbols

Abbreviation	Explanation of abbreviation (Source)
API	Application Interface
TPS	Transactions Per Second
PG	PostgreSQL
HTTP	Hypertext Transfer Protocol
RDBMS	Relational Database Management System
SSL	Secure Sockets Layer
TLS	Transport Layer Security
TCP	Transmission Control Protocol

1 Introduction

Since the turn of the millennium, society has increasingly relied on the internet to service products and goods necessary to maintain the status quo. As the needs exponentially rise, both in terms of pure computing power and data storage methods, the infrastructure built today must consider tomorrow's requirements.

Furthermore, sharp mobile use increases in both Western and developing nations, particularly in Eastern Asia - with an estimate from Ani Petrosyan[1] of 1.235 Billion Unique Internet Users from Eastern Asia alone provides evidence that modern applications need to be optimised for simultaneous data requests.

Therefore, the services found on the internet today have to keep their infrastructure in mind so that increases in use can be handled without compromising the speed or security of their databases. With this in mind, Load balancing technologies have been developed to help system administrators and DevOps specialists to help them deploy a service that is always available.

This thesis will provide a case for the use of load balancers in database clusters where traffic is dynamic. Its utilisation of resources fluctuates depending on the demands of the database cluster's service.

2 Definition of Load Balancing and Database Management Terminology

Heavy.AI[2] explains load-balancing as a ‘relationship between cooperating programs in an application, composed of clients initiating requests for services and servers providing that function or service.’ (2023). Pre-requisite knowledge of how a server works are required to understand the dilemma and the solution that load-balancing technology provides.

In the thesis’s context, the Server contains the PostgreSQL (PG) Database, where clients are Application Interfaces(API) sending data requests using HTTP protocols.

2.1 HTTP

HTTP (Hypertext Transfer Protocol) is a widely used protocol[3] for communication on the World Wide Web. When clients access a website or any other online resource, the client sends an HTTP request to the server where the resource is hosted. The server then processes the request and sends back an HTTP response to the client.

HTTP traffic refers to the data exchange between clients and the server using HTTP. This traffic can be monitored, analysed, and optimised using specialised tools.

When clients send an HTTP request, it includes information such as the type of request (e.g., GET, POST, PUT, DELETE), the URL of the requested resource, and any additional data needed to complete the request. The server then responds with an HTTP status code indicating whether the bid was successful or not, along with any requested data.

2.2 PostgreSQL

PostgreSQL is an open-source relational database management system[4](RDBMS) that stores and manages large amounts of data. It is a popular choice for many organisations because it is reliable, scalable, and has robust data integrity features.

Data is stored in a structured manner, making it easier to retrieve and manipulate. In PostgreSQL, data is organised into tables consisting of rows and columns. Each table represents a single entity, and the columns represent the attributes or properties of that entity.

PostgreSQL supports various data types, including numeric, character, and date/time. It also supports advanced features such as user-defined functions, triggers, and stored procedures, allowing more complex data management tasks.

PostgreSQL is known for its transactional capabilities[5], which ensure that data is always consistent and reliable. It uses a multi-version concurrency control (MVCC) system to manage concurrent access to data, which allows for high levels of concurrency while maintaining data consistency.

PostgreSQL also has robust security features, including access control and encryption, to protect sensitive data. It supports various authentication methods, including password-based authentication, certificate-based authentication, and external authentication methods such as LDAP or Kerberos.

In summary, PostgreSQL is a robust and reliable RDBMS that many organisations use to manage large amounts of data. It offers advanced features such as user-defined functions, triggers, and stored procedures and is known for its transactional capabilities and robust security features.

2.3 Load Balancers

Load-balancing proxies for database deployments refer to a network infrastructure that helps distribute incoming database requests evenly across multiple servers. This aims to ensure that every database server responds promptly to incoming requests. In other words, Oracle describes: a load balancer is used to distribute client connection requests, provide load balancing and failover across the cluster.’[6]

A load-balancing proxy acts as an intermediary between the client making a request and the database servers. The broker receives incoming requests and then forwards them to one of the available database servers based on pre-determined algorithms.

3 Load Balancing Algorithms

This Thesis focuses on the three most commonly used algorithms used in load balancing – round-robin, least connections and IP hash. Each test case will contain the algorithms labelled to specify the speed differences between each method.

3.1 Round-Robin

The round-robin method of load balancing[7] in database deployments is a technique in which client requests are distributed evenly across a group of database servers. In this method, each request is sent to the next available server in a predefined list of servers, and the process is repeated circularly.

3.2 Least Connections

The least connections method of load balancing[8] in database deployments is a technique that distributes incoming client requests to the database servers with the least number of active connections. In this method, each new request is routed to the server with the fewest active connections, regardless of its current processing capacity.

3.3 IP Hash

The IP hash method of load balancing [9] in database deployments is a technique that distributes incoming client requests to a specific database server based on the client's IP address. In this method, the IP address of the client is hashed, and the resulting value is used to select the server to handle the request.

3.4 Conclusion of Algorithms

In conclusion, of all mainstream algorithms provided by the latest load-balancing technologies, each algorithm has a specific use case depending on the requirements of the service behind it. In the thesis's context, all algorithms are tested in all circumstances - to find the best solution for a database serving data through an API.

4 Issues with Modern Database Deployments

Modern database deployments face various scalability, performance, security, and management challenges[10]. Some of the critical issues with current database deployments include:

4.1 Scalability

As organisations generate and collect ever-increasing amounts of data, their database systems must be able to scale to meet this demand. Scalability can be challenging for traditional relational database systems, which may need help handling large amounts of data or a high volume of concurrent users.

4.2 Performance

As data volumes increase, database performance can become a bottleneck. Slow queries, long response times, and other performance issues can impact user experience and productivity. Modern database deployments must be designed to handle large volumes of data and a high degree of concurrency while also providing fast query response times.

4.3 Security

Databases are a prime target for attackers seeking to steal or compromise sensitive data. Modern database deployments must be designed with security in mind, with features such as encryption, access control, and audit trails to protect against unauthorised access and data breaches.

4.4 Management

As databases become more complex, managing them can become a challenge. Database administrators need to be able to monitor performance, tune queries, and manage backups and recovery processes. They also need to be able to handle large distributed systems with multiple nodes and high availability requirements.

4.5 Interoperability

Ensuring databases can work seamlessly with other systems is critical for ensuring data integrity and consistency across different applications. Modern database deployments must often work with various other applications and data sources. This can be challenging if different systems use different data formats or protocols.

In summary, modern database deployments face various scalability, performance, security, management, and interoperability challenges. Addressing these issues requires careful planning, design, implementation, ongoing monitoring, and leadership to ensure that databases continue to perform effectively over time.

5 Requirements for Load-Balancing Technologies

Load balancers require six critical aspects from a database cluster[11]. The functionality of the proxy depends on the following.

5.1 Multiple database nodes

A database cluster must have various nodes for load balancing. Each node should be identical in configuration and functionality to ensure consistency and optimal performance.

5.2 Network connectivity

All the nodes in the database cluster must have network connectivity with each other, and the load balancer must also have connectivity to all the nodes in the group. This allows the load balancer to distribute incoming requests to the appropriate node.

5.3 Load balancing algorithm

The load balancer must have a load-balancing algorithm that determines how requests are distributed across the nodes in the cluster. Each algorithm has its strengths and weaknesses, and the choice of algorithm will depend on the application's specific requirements.

5.4 Monitoring and failover

The load balancer must monitor the health and status of each node in the cluster. The load balancer must redirect traffic to a healthy node if a node fails or becomes unresponsive. This ensures that the database cluster remains highly available and responsive to incoming requests.

5.5 Security

The load balancer must be configured with appropriate security measures such as SSL/TLS encryption, access control, and authentication to protect against unauthorised access or attacks.

5.6 Scalability

The load balancer should be able to scale horizontally or vertically as the number of nodes in the database cluster increases, or the traffic volume grows. This allows the load balancer to handle increasing traffic without impacting performance or availability.

6 PgBouncer and Replication Methods

6.1 PgBouncer

PgBouncer is an open-source connection pooling and load-balancing software for PostgreSQL databases[12]. It is designed to reduce the overhead of creating and destroying database connections, improving the performance and scalability of PostgreSQL database clusters.

Similarly to PgPool, it supports the three main algorithms tested in this Thesis. Furthermore, PgBouncer boasts additional features in network security, with its documentation describing;

Authentication: PgBouncer can authenticate client connections to the database server, providing an additional layer of security.

SSL encryption: PgBouncer can encrypt client connections using SSL/TLS, providing an additional layer of security.

PgBouncer supports multiple versions of PostgreSQL and is distributed under the GNL open-source agreement.

6.1.1 PgBouncer Settings

The pgbouncer.ini file is the main configuration file for PgBouncer[13]. This file contains various settings that control the behaviour of PgBouncer, such as database connections, user authentication, and logging.

Key sections that are relevant to this thesis's topic, in addition to the test cases that are carried out, are the following:

[databases]: This section defines the connections that PgBouncer will manage. Each link is illustrated with a separate entry that specifies the database name, the host and port, and any additional connection parameters.

[users]: This section defines the user authentication settings for PgBouncer. Each user is illustrated with a separate entry specifying the user name and password and any optional settings like allowed databases or connection limits.

[pgbouncer]: This section contains general settings for PgBouncer itself, such as the log file name, the maximum number of clients, and the server encoding.

[pool]: This section contains settings that control the behaviour of the connection pool. For example, the max_client_conn setting specifies the maximum number of client connections active simultaneously. In contrast, the reserve_pool setting specifies the number of connections to keep in reserve for unexpected spikes in demand. Additionally, this section handles the algorithms used in PgBouncer's load balancing, with the three options described in 'Load-Balancing Algorithms'.

[log]: This section contains settings related to logging, such as the log file name and the log verbosity level.

6.2 Synchronous Replication in PostgreSQL

Synchronous replication is a feature in PostgreSQL that ensures that changes made to the primary database are immediately replicated [14] to the standby database before acknowledging the transaction committed to the client. This provides strong data consistency and durability guarantees. Any failure in the primary database will not result in data loss as long as the standby database is up-to-date.

When a transaction is committed on the primary database, the primary waits for an acknowledgement from the standby database that the changes have been successfully written to the standby's WAL (Write-Ahead Log) before sending an acknowledgement to the client.

If the standby fails to acknowledge the changes within a configured timeout, the primary assumes it has been unable and marks it inactive. The primary then continues to accept transactions but no longer waits for acknowledgements from the standby.

When the standby is restored or becomes available, it requests any missing WAL segments from the primary and catches up to the current state.

Synchronous replication can be configured at the database, table, or transaction level in PostgreSQL, allowing for fine-grained control over data consistency requirements. However, it comes at the cost of increased latency and potential performance overhead, as each transaction must wait for an acknowledgement from the standby before it can be acknowledged to the client.

6.3 Asynchronous Replication in PostgreSQL

Asynchronous replication is a feature in PostgreSQL that allows changes made on the primary database to be asynchronously replicated[15] to one or more standby databases without waiting for confirmation from the standby before acknowledging the transaction committed to the client. This performs better than synchronous replication, as standby has no wait time to recognise the changes.

When a transaction is committed on the primary database, the changes are written to the WAL (Write-Ahead Log) on the primary.

The changes are then asynchronously sent to the standby database(s) using streaming replication. The standby applies the changes to its copy of the database, which can be used for read-only queries or as a backup in case the primary fails.

If the primary fails, one standby can be promoted to become the new primary, and the other standby can start replicating from the new primary.

Asynchronous replication can be configured with different levels of replication delay, depending on the data consistency and availability requirements. For example, a standby can be configured to have a "replication lag" of a certain number of WAL segments or time intervals, which allows it to be slightly behind the primary but still provide a recent copy of the data.

6.4 Master-Slave Replication

Master-Slave Replication is a standard method [16] for replicating PostgreSQL databases. In this method, one database server (the "master") is designated as the primary server that receives read-write requests, while one or more other servers (the "slaves") replicate changes made on the master and can be used for read-only queries or backups.

The master server receives write requests from clients and applies them to its copy of the database.

The changes made on the master are then replicated to one or more slave servers using streaming or file-based replication. The slave servers maintain a copy of the database that is kept up-to-date with changes made on the master.

The slave servers can be used for read-only queries or backups and promoted to the new master if the original master fails.

If a slave fails in replication due to network issues or other problems, it can synchronise with the master using a "streaming base backup" process.

Master-Slave Replication can be configured with various settings and parameters, such as the replication method (streaming or file-based), replication delay, and replication slots. It provides high availability, fault tolerance, and the ability to offload read-only queries to the slave servers for improved performance.

Additionally, this method supports both Synchronous and Asynchronous replication between nodes and, therefore, will be used as the replication method for testing the performance benefits of using it along with a load-balancer.

7 Test Cases

The description of test cases is split between the comparing projects since each contains its unique settings handled by its settings file. All test cases will use the three main algorithms as described.

The standard approach is to test three scenarios where the traffic exceeds the limitations set in the hardware configuration of the PostgreSQL databases. The first scenario is to test the responsiveness of the cluster when the number of clients connected exceeds the hardware limit. The second scenario tests the group's responsiveness when the number of requests per client is excessively high. The third scenario will focus on read-only requests, which signifies the efficiency of how the solutions handle balancing between read-only nodes in the PostgreSQL cluster. The fourth scenario is a standard case where the number of clients connecting matches the hardware set limit, and the transactions per client are low. This signifies an average load of data requests a database expects to handle.

All test cases will be benchmarked against a singular PostgreSQL database with no load-balancing deployed to sanity-check the benefits that load balancers claim to have.

All test cases focus on the high availability of the database cluster as the measuring factor that is focused on during testing is the amount of data transaction carried out per second, as well as measuring SQL query execution between load-balanced data clusters and standalone PostgreSQL databases.

7.1 Testing Enviroment Comparison

The Number of replica nodes are dictated by the test case, with the benchmark readings for no replication or load balancing are done with a singular PostgreSQL Node. This is chosen to accurately test with the performance

improvements with both replication and load balancing proxies in modern database deployments.

7.2 Control Cases

7.2.1 Test Case 1

Edge case where the number of clients exceeds the database hardware set limit. This tets availability when the cluster processes an unexpected spike in traffic on client requests.

PostgreSQL Commands:

‘max_connections=500

Pg_Bench Flags:

Clients: 600

Transactions: 100

7.2.2 Test Case 2

The edge case is where the number of transactions per client spikes to see how the cluster responds to abnormal data requests.

PostgreSQL Commands:

‘max_connections=500

PgBench Flags

Clients: 10

Transactions: 1000

7.2.3 Test Case 3

In this control case, the number of clients connected is lower than the hardware-specified limit, and the number of transactions is comparable to modern data requests from external traffic.

PostgreSQL Commands:

‘max_connections=500’

Pg_Bench Flags:

Clients: 300

Transactions: 100

7.2.4 Test Case 4

Read-Only test to see how the cluster functions with read-only data requests from clients. This test focuses on system utilisation and pure speed between standalone PostgreSQL databases.

PostgreSQL Commands:

‘max_connections=500’

Pg_Bench Flags:

Clients: 300

Transactions: 100

Mode: Read-Only

7.3 Synchronous Test Cases

This section covers the test cases on a Master-Slave replication model using PostgreSQL 15 and PgBouncer v1.18 with the containers provided by Bitnami. Synchronous Replication is done by not setting ‘WAL’ settings in the pg-conf file so that the slave node constantly monitors database changes on the master server.

For each test case, PgBouncer will additionally be configured for each load-balancing algorithm previously explained in this thesis.

7.3.1 Test Case 1

PostgreSQL Commands:

‘max_connections=500

Pg_Bench Flags:

Clients: 600

Transactions: 100

7.3.2 Test Case 2

PostgreSQL Commands:

‘max_connections=500’

PgBench Flags:

Clients: 10

Transactions: 1000

-C

7.3.3 Test Case 3

PostgreSQL Commands:

‘max_connections=500’

Pg_Bench Flags:

Clients: 300

Transactions: 100

-C

7.3.4 Test Case 4

PostgreSQL Commands:

‘max_connections=500’

Pg_Bench Flags:

Clients: 300

Transactions: 100

-S ‘Read-Only’

7.4 Asynchronous Test Cases

This section covers the test cases on a Master-Slave replication model using PostgreSQL 15 and PgBouncer v1.18 with the containers provided by Bitnami. The test cases are the same as Synchronous ones with the same flags, steps and pg_bench command-line flags. Asynchronous Replication is done by setting ‘WAL’ settings to five minutes in the pg-conf file so that the slave node checks for updates from the master in five-minute periods. The 5-minute interval was chosen so that the periods were close enough so that the results would show the effect of asynchronous processing without bottlenecking the processer found on the Raspberry Pi.

8 Interpretation of Results

8.1 Test Case 1

Test Case 1 shows that implementing replication and load-balancing has a sharp increase in transactions the cluster can handle per second by roughly 500%. This increase was seen due to the algorithmic approach to client handling, where the load balancer splits read and write requests between stand-by nodes, as well as an increase in CPU and RAM usage. This result was also seen in the previous testing done in collaboration with HoistGroup Oy, where a PostgreSQL server hosted in Google Cloud was tested with and without PgBouncer.

Figure 1 shows comparative results split between each setting and replication method. Asynchronous transactions show the sharpest increases but could not complete all clients, as demonstrated by the second figure. Out of each setting, the Synchronous transaction mode allowed all clients and transactions to be satisfied with the same increase in TPS but slightly behind other methods in raw speed.

Additionally, Figure 3 shows how PgBouncer can justify the sharp increase in performance, no matter the replication method and PgBouncer mode, the system front loads SQL queries in the Begin Statement, with the time graph flat-lining towards the end of the test. In contrast, with no replication or load balancing, the time graphs vary wildly throughout the trial.

From Figure 2, in Statement mode with PgBouncer, no transactions were completed, with the error being: ‘Transaction Blocks not allowed in Statement Mode’. Therefore, no data has been shown regarding performance.

This test case focused on the high availability of a cluster, where a sharp rise in client requests is processed abnormally without the foresight of knowing when this spike could happen. In systems with continuous sharp peaks, Synchronous

replication of PostgreSQL nodes with PgBouncer in Transaction mode provided the most stable response, with a five-fold increase in TPS speed.

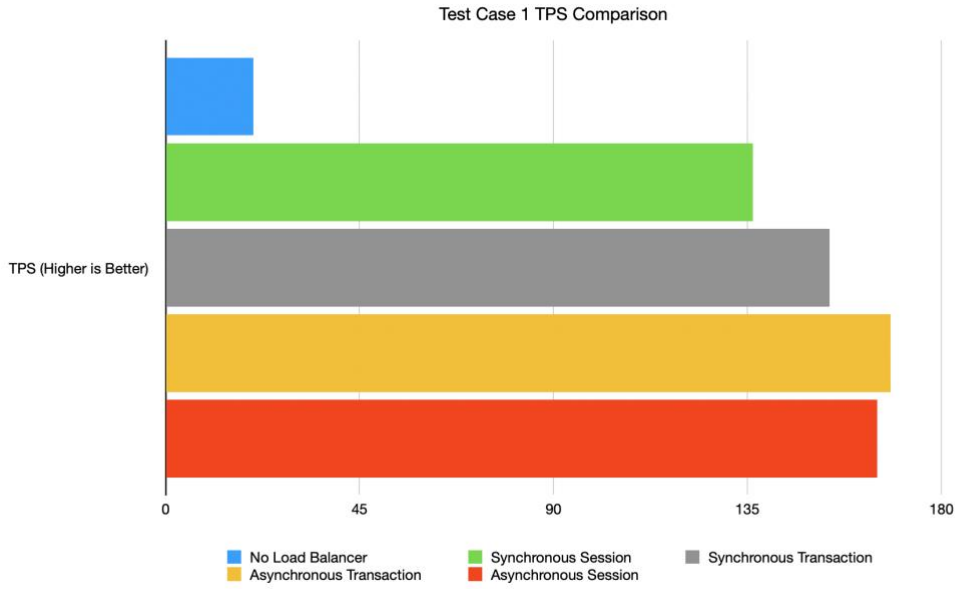


Figure 1. Test Case 1 TPS Comparison

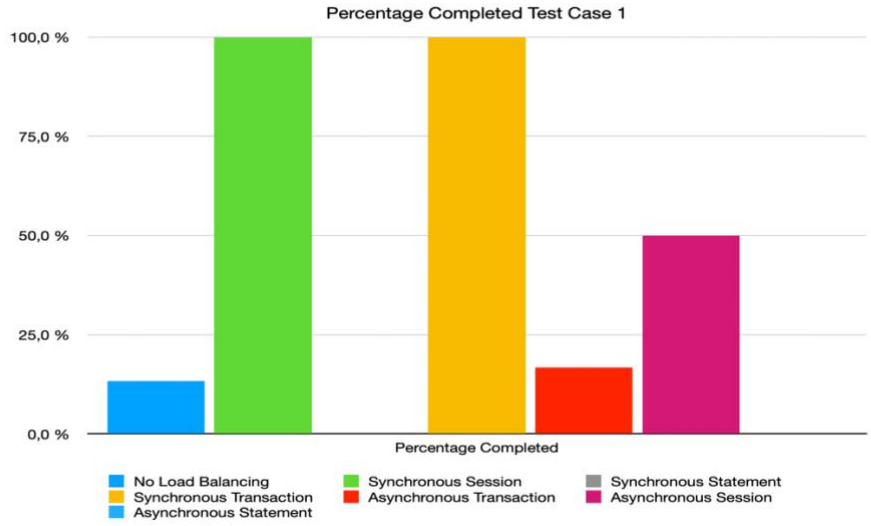


Figure 2. Test Case 1 Percentage Completed

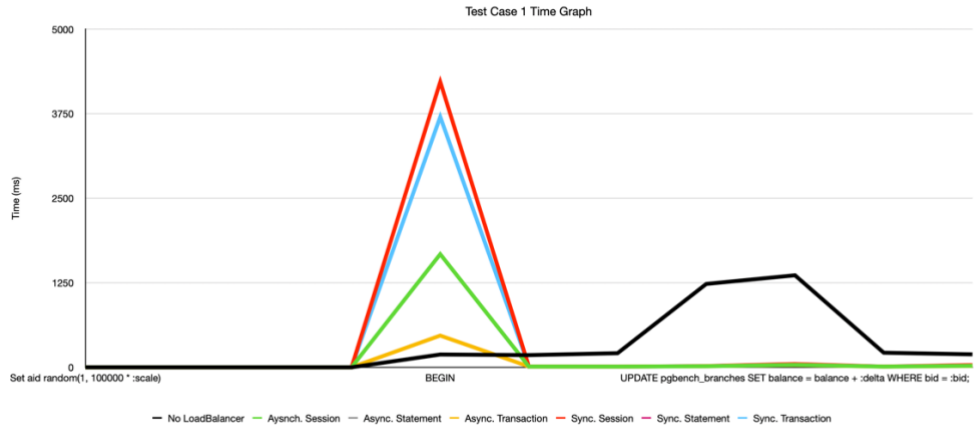


Figure 3. Time-Line Graph

8.2 Test Case 2

Test Case 2 shows similar behaviour to Test Case 1, where a five-fold performance increases when comparing no load balancing to replication and load balances. Results between replications were tighter, and all methods accepted statements provided a 100% completion rate.

Figure 4 shows comparative results using TPS as the deciding factor of performance. Asynchronous replication with PgBouncer set to transaction provided the most significant increase in TPS speeds.

Figure 5 shows that with load balancing and replication, the curve closely follows what no load-balancing produces but transforms by a factor of five down the x-axis, meaning that the speed of the test was five-fold, which relates to the five-fold increase in TPS.

This test case focused on instances where a large number of requests can be carried out by a singular client, which closely replicates situations where many updates are required on the cluster. This focuses on the high availability aspect of load-balancing. As evident in Figure 6.

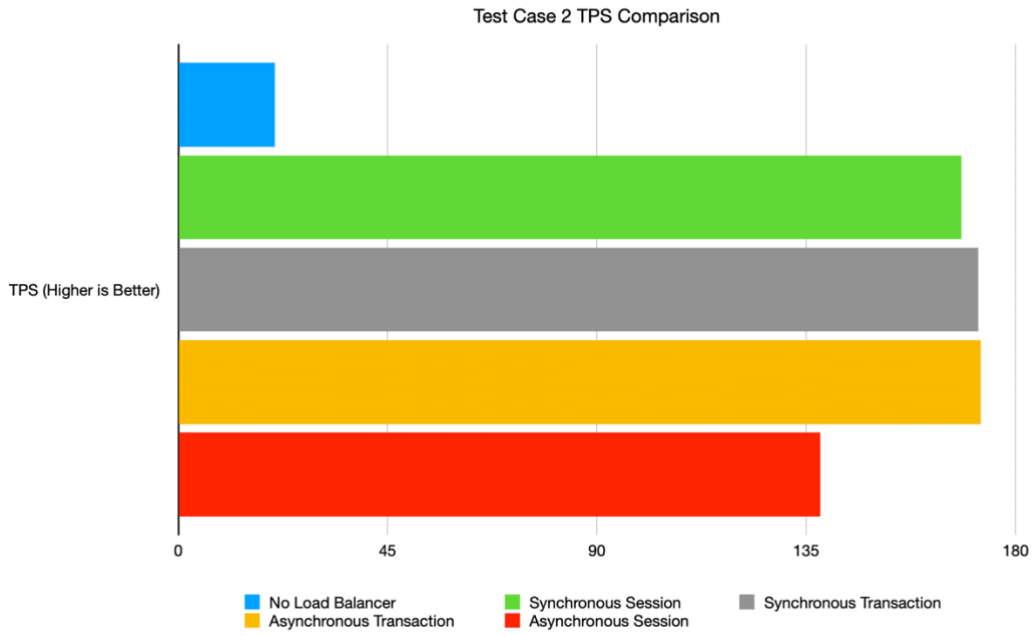


Figure 4 Test Case 2 TPS

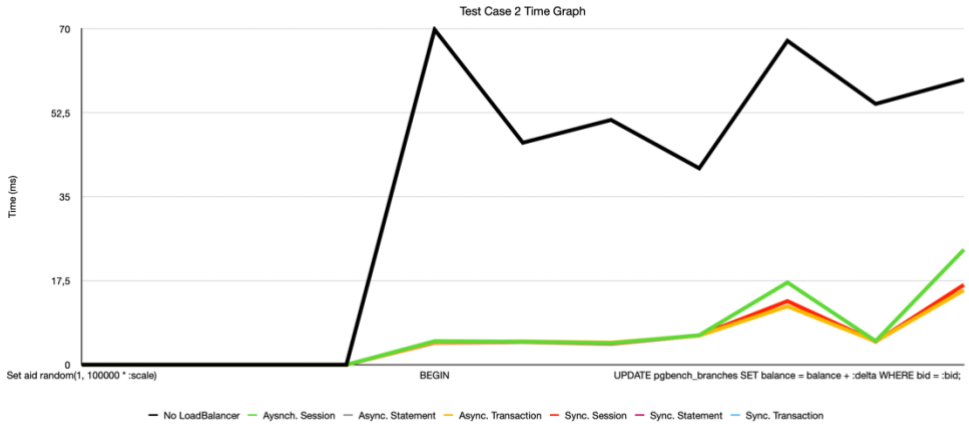


Figure 5. Test Case 2 Time Graph

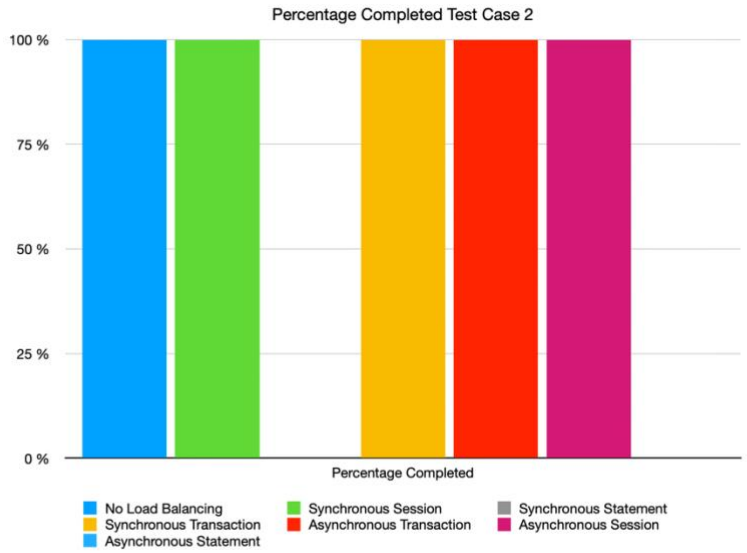


Figure 6. Test Case 2 Percentage Completed

8.3 Test Case 3

Test Case Three again produces similar results to Test Cases One and Two, where a significant increase in TPS was seen. The PostgreSQL server couldn't complete all transactions without replication or load-balancing, producing only 33% completion. Furthermore, Statement Mode had no result with the same 'Transaction Blocks not allowed in Statement Mode' error being made.

Figure 7 shows that Asynchronous replication with session pooling mode produced the most significant increase in TPS with a sixfold increase. Furthermore, all replication modes and pooling methods made a 100% completion rate besides the session pooling mode.

Figure 8 shows similar results to Test Case One, where the load balancer front loads from the beginning statement with a downward straight trend to the end. In comparison, the time graph with no replication has two peaks, with the first

peak appearing at the beginning statements of the test and the second peak reaching towards the end of the SQL statements.

This test case focused on day-to-day data requests with equal client transactions. It operated within the hardware set limit found on the PostgreSQL server so that no anomaly data points would be produced and was operating to the status quo. The results suggest a considerable gain in performance handling and hardware utilisation when replicating and load-balancing, even when the database isn't being pushed to the limit. As evident in Figure 9.

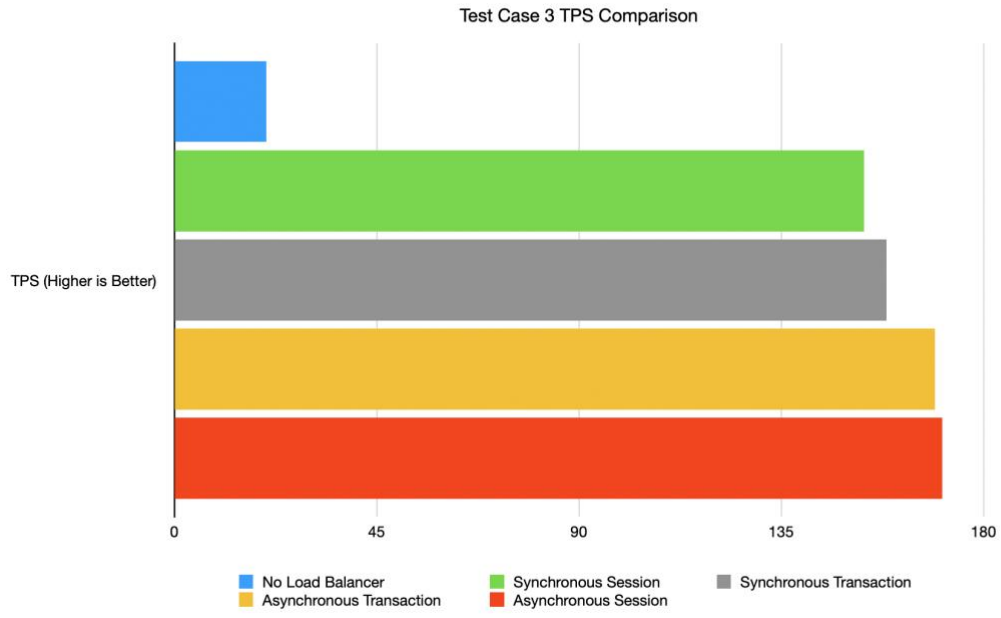


Figure 7. Test Case 3 TPS

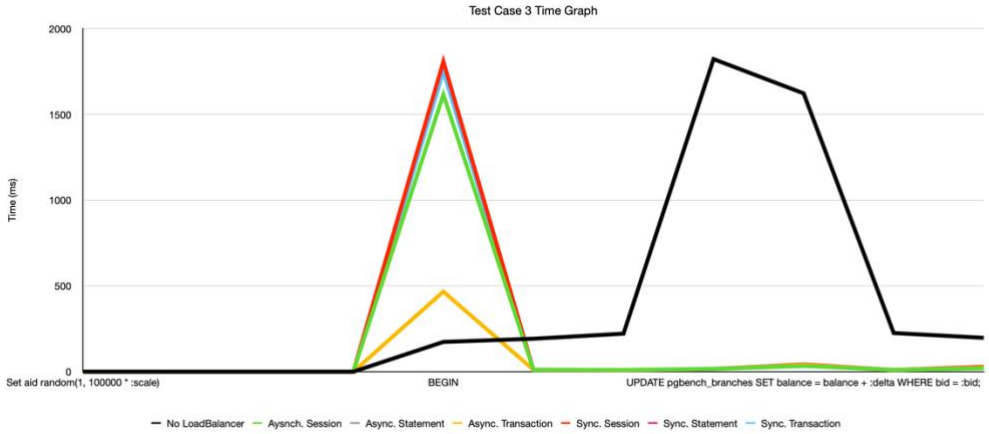


Figure 8. Test Case 3 Time Graph

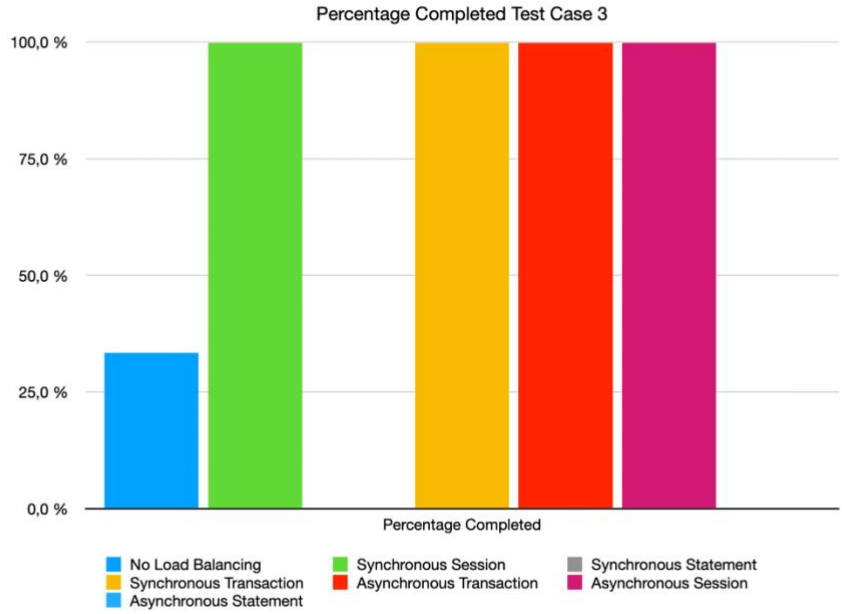


Figure 9. Test Case 3 Percentage Completed

8.4 Test Case 4

Test case four focused on read-only situations where the replication method would significantly affect performance metrics. Choosing the correct replication method heavily depends on the server's experience in day-to-day operations serving its host. The test case clients and transactions were the same as in Test Case Three.

Statement Transactions were compatible in this test case and provided the most significant increase seen with Asynchronous Statements. TPS reported around 350 compared to no load-balancing or replication, provided only 20 transactions per second, as shown in Figure 10.

Additionally, all replication and load balancing methods could complete 100% of transactions compared to no load balancing, only providing a 33% completion result, as shown in Figure 11.

Timeline curvature where same across the test cases as only two SQL statements were carried out during this test – although all replication and pooling methods provided a smaller gradient.

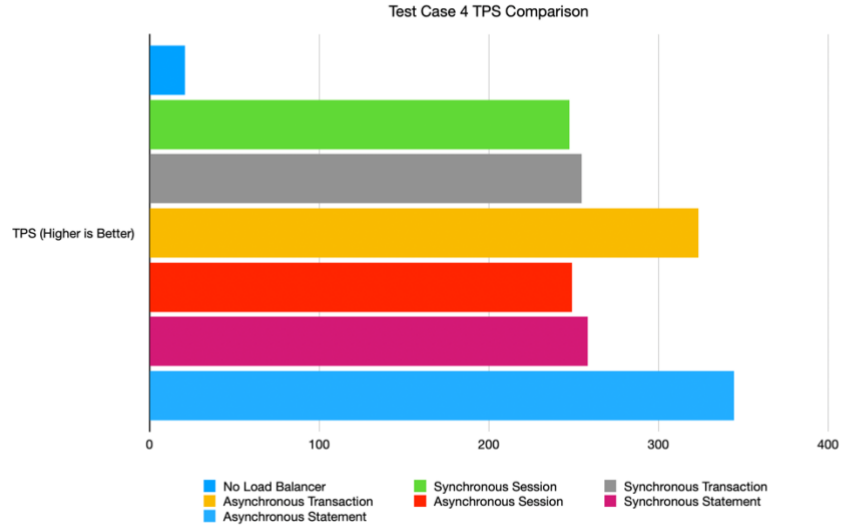


Figure 10. Test Case 4 TPS Comparison

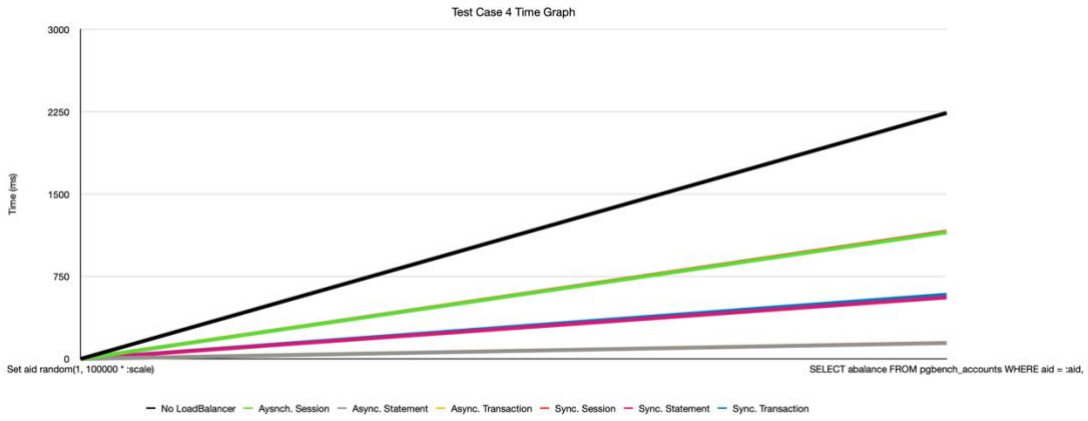


Figure 11. Test Case 4 Time Graph

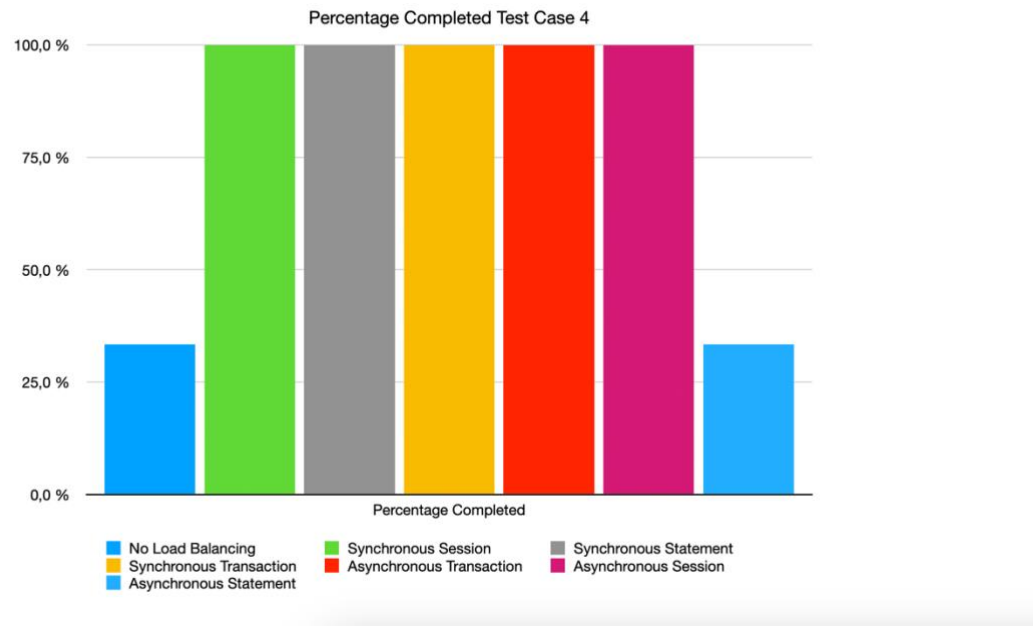


Figure 12. Test Case 4 Percentage Completed.

9 Tools Used

9.1 Raspberry Pi 4 8GB

The Raspberry Pi 4 Model B [17] is a small, single-board computer developed by the Raspberry Pi Foundation. It is the fourth generation of the Raspberry Pi series and was released in June 2019. The Raspberry Pi 4 Model B has a range of RAM options, including 2GB, 4GB, and 8 GB.

The Raspberry Pi 4 Model B with 8GB of RAM is the most powerful yet.

Key specifications of the Raspberry Pi 4 Model B with 8GB of RAM include:

- Broadcom BCM2711 quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- 8GB LPDDR4-3200 SDRAM (depending on model)
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE Gigabit Ethernet
- 2 USB 3.0 ports; 2 USB 2.0 ports.
- 2 × micro-HDMI ports (up to 4kp60 supported)
- H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode)
- OpenGL ES 3.0 graphics
- 5V DC via USB-C connector (minimum 3A*)
- Operating temperature: 0 – 50 degrees C ambient

The Raspberry Pi hosts the PostgreSQL server on a local network, using Docker containers to manage the start-up process, networking, replication methods, and pooling modes.

9.2 PostgreSQL 14

PostgreSQL version 14 is the version used as the operating software for the database server clusters being run on the Raspberry Pi. Bitnami’s PostgreSQL images were used for testing.

9.3 PG_Bench

Pgbench is a benchmarking tool for PostgreSQL used to test a PostgreSQL [18] database's performance by simulating different workloads. Pgbench can be used to test the performance of a PostgreSQL server in various scenarios, such as testing the maximum number of connections that can be handled or testing the version of a specific database schema or query.

Pgbench creates a database with a specified schema and populates it with a data set. It then simulates client connections and performs a selected workload on the database, measuring the time taken for the workload to complete.

PgBench was deployed using my created Docker Image based on Bitnami’s PostgreSQL docker image.

Version 5.2 was used for testing.

9.4 Excel

Microsoft Excel 2023 was used to format data points, plot the TPS bar graph comparisons and time-line graphs, and display data in a format compliant with the Turku University of Applied Science guidelines.

10 Conclusion

In conclusion, database replication and load balancing benefit modern businesses that rely on their databases for critical applications and services. Database replication enables organisations to create a redundant copy of their databases in real-time, ensuring high availability and disaster recovery capabilities. Load balancing, on the other hand, distributes incoming traffic across multiple database servers, improving scalability, performance, and availability.

By leveraging these technologies, organisations can achieve excellent resiliency, scalability, and performance for their databases, ensuring their critical applications and services remain available and responsive. With the ability to distribute traffic across multiple servers and maintain real-time backups of their databases, businesses can ensure that their databases are always accessible and can easily handle traffic spikes and unexpected failures.

Implementing database replication and load balancing is essential for businesses that rely on their databases to support their operations. By investing in these technologies, organisations can ensure the reliability and availability of their databases and improve the performance and scalability of their critical applications and services. Through testing, we saw performance improvements of a factor of five, with each session pooling mode and replication method having its strengths and benefits in context with what the database is meant to serve. A combination of thorough architecture planning and considering the needs of the database cluster – Load Balancing and Replication provides a more reliable, safe and readily accessible database to its cause.

References

1. Petrosyan A (2023) Statista, Internet users by global regions 2019, 2023. Available from: https://www.statista.com/statistics/249562/number-of-worldwide-internet-users-by-region/.

2. What is Client-Server? Definition and FAQs | OmniSci www.heavy.ai. Available from: https://www.heavy.ai/technical-glossary/client-server.

3. CLOUDFLARE What is HTTP? | Cloudflare UK. *Cloudflare*.

4. PostgreSQL (2019) Postgresql.org, PostgreSQL: About, 2019. Available from: https://www.postgresql.org/about/.

5. Thomas SM (2017) PostgreSQL High Availability Cookbook , Packt.

6. NGINX (2018) NGINX, What Is Load Balancing? How Load Balancers Work, 2018. Available from: https://www.nginx.com/resources/glossary/load-balancing/.

7. What is Round Robin Load Balancing? Definition & FAQs Avi Networks. Available from: https://avinetworks.com/glossary/round-robin-load-balancing/#:~:text=Round%20robin%20load%20balancing%20is.

8. Valverde R Research Gate, Least Connections Method. Available from: https://www.researchgate.net/figure/Least-connection-algorithm_fig2_347808307.

9. VMware Knowledge Base (2020) Vmware.com, 2020. Available from: https://kb.vmware.com/s/article/2006129.

10. Ltd RGS, Fritchey G (2020) Redgate, Challenges to Database DevOps: Which Do you Deploy First, the Database or the Application?, 2020. Available from: https://www.red-gate.com/blog/challenges-to-database-devops-which-do-you-deploy-first-the-database-or-the-application.

11. Common Requirements for Load Balancers help.sap.com. Available from: https://help.sap.com/doc/a29eaeb9079948e2ac594421b23f7e38/3.0.12/en-US/30c42f5506704fc8900b23892b72979d.html.

12. PgBouncer features www.pgbouncer.org. Available from: https://www.pgbouncer.org/features.html.

13. PgBouncer config www.pgbouncer.org. Available from: https://www.pgbouncer.org/config.html.

14. Youatt D (2020) Crunchy Data, Synchronous Replication in PostgreSQL, 2020. Available from: https://www.crunchydata.com/blog/synchronous-replication-in-postgresql.

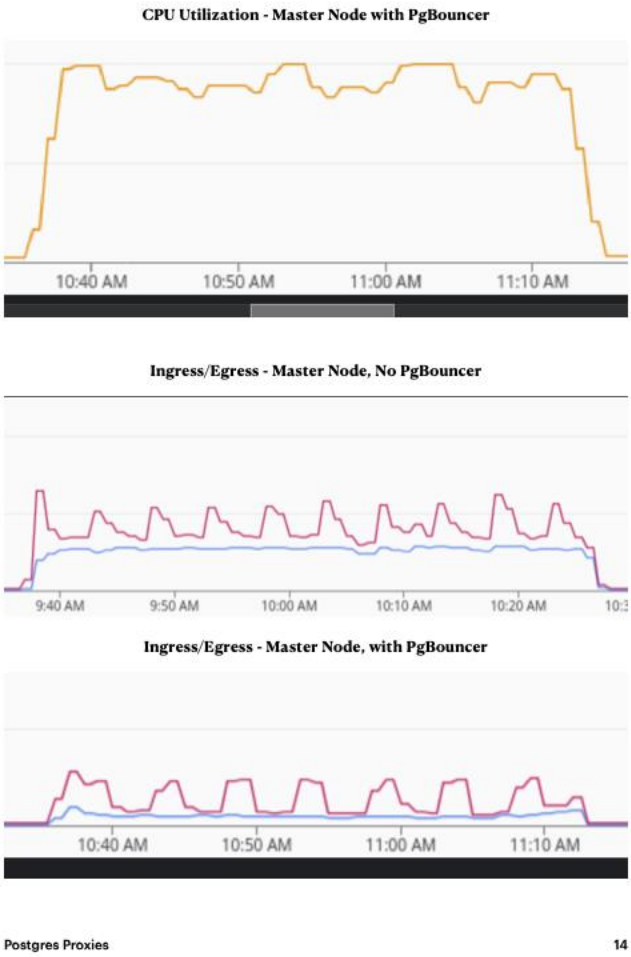
15. Synchronous and asynchronous replication in PostgreSQL CYBERTEC. Available from: https://www.cybertec-postgresql.com/en/services/postgresql-replication/synchronous-asynchronous-replication/.

16. Achieving PostgreSQL Master Slave Replication: 7 Easy Steps (2020) HevoData, 2020. Available from: https://hevodata.com/learn/postgresql-master-slave-replication/.

17. Ltd RP (Trading) Raspberry Pi, Raspberry Pi 4 Model B specifications. Available from: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

18. pgbench (2023) PostgreSQL Documentation, 2023. Available from: https://www.postgresql.org/docs/current/pgbench.html.

Previous Work Placement Report



Page 1 - CPU Utilization in x86 Duo Core Intel

```
FROM alpine:latest as builder

RUN apk add --no-cache postgresql==14

FROM alpine:latest

RUN apk add --no-cache libpq==11.5-r0
COPY --from=builder /usr/bin/pgbench /usr/bin/pgbench

ENTRYPOINT ["pgbench"]
```

Page 2 - Docker Build File for PgBench

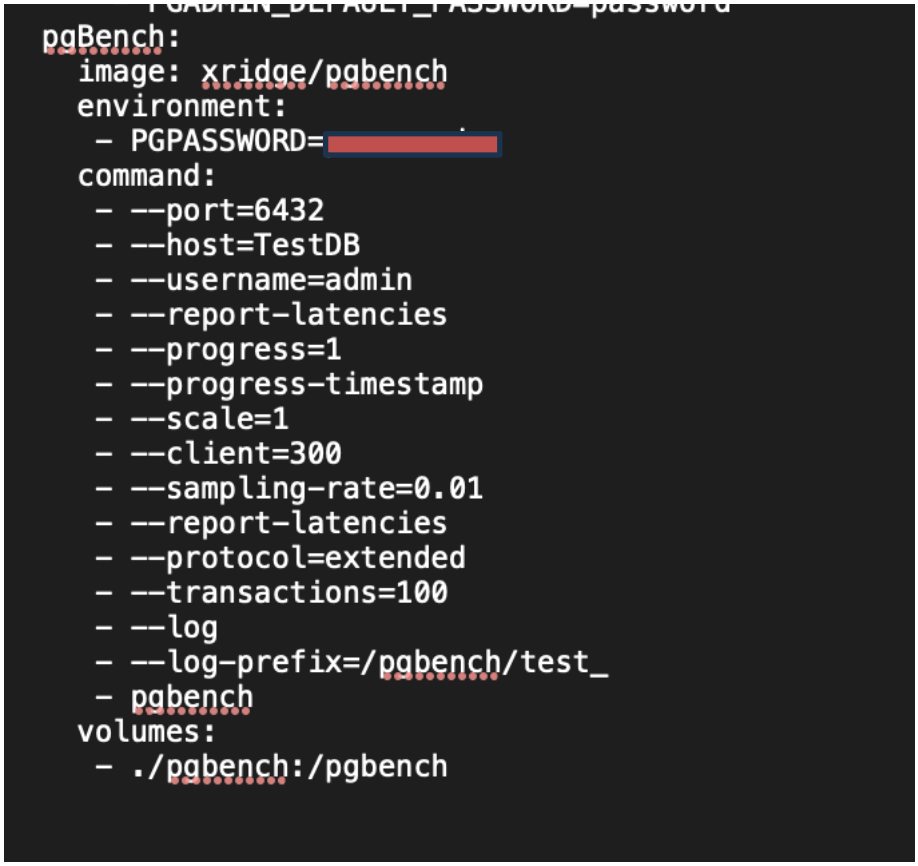
```
version: '3'
networks:
  testNetwork:
    driver: bridge

services:
  MasterDB:
    image: bitnami/postgresql:latest
    ports:
      - '5432:5432'
    environment:
      - POSTGRESQL_PGAUDIT_LOG=READ,WRITE
      - POSTGRESQL_LOG_HOSTNAME=true
      - POSTGRESQL_REPLICATION_MODE=master
      - POSTGRESQL_REPLICATION_USER=
      - POSTGRESQL_REPLICATION_PASSWORD=
      - POSTGRESQL_SYNCHRONOUS_COMMIT_MODE=on
      - POSTGRES_NUM_SYNCHRONOUS_REPLICAS=1
      - POSTGRESQL_USERNAME=
      - POSTGRESQL_DATABASE=
      - POSTGRESQL_PASSWORD=
      - ALLOW_EMPTY_PASSWORD=yes

  SlaveDB:
    image: bitnami/postgresql:latest
    ports:
      - '5433:5432'
    environment:
      - POSTGRESQL_PGAUDIT_LOG=READ,WRITE
      - POSTGRESQL_LOG_HOSTNAME=true
      - POSTGRESQL_MASTER_HOST=TestDB
      - POSTGRESQL_REPLICATION_MODE=slave
      - POSTGRESQL_REPLICATION_USER=repl_user
      - POSTGRESQL_REPLICATION_PASSWORD=
      - POSTGRESQL_SYNCHRONOUS_COMMIT_MODE=on
      - POSTGRESQL_USERNAME=
      - POSTGRESQL_MASTER_PORT_NUMBER=
      - ALLOW_EMPTY_PASSWORD=yes

  pgadmin:
    image: dpage/pgadmin4
    depends_on:
      - TestDB
    restart: always
    ports:
      - '5555:5555'
    environment:
      - PGADMIN_DEFAULT_EMAIL=
      - PGADMIN_DEFAULT_PASSWORD=
```

Page 3 - Docker Compose for PostgreSQL



Page 4 - Docker Compose for PgBench

Table of Results

Table 1. Test Case 1 Results split by SQL Query

Test Case 1

Statement Latencies in MS	No LoadBalancer	Aysnch. Session	Async. Statement	Async. Transaction	Sync. Session	Sync. Statement	Sync. Transaction
Set aid random(1, 100000 * :scale)	0,01	0,01	N/A	0,011	0,011	N/A	0,011
set bid random (1, 1 * :scale)	0,002	0,003	N/A	0,002	0,003	N/A	0,003
set tid random (1, 10 * : scale)	0,002	0,002	N/A	0,002	0,002	N/A	0,002
set delta random (-5000, 5000)	0,001	0,002	N/A	0,001	0,001	N/A	0,002
BEGIN	189,465	1672,994	N/A	469,136	4220,211	N/A	3703,125
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;	180,465	10,755	N/A	11,830	10,755	N/A	12,767
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;	208,833	10,059	N/A	10,690	9,684	N/A	10,111
UPDATE pgbench_tellers SET balance = tbalance + :delta WHERE tid = :tid;	1233,019	15,866	N/A	16,429	18,551	N/A	17,488
UPDATE pgbench_branches SET balance = balance + :delta WHERE bid = :bid;	1360,581	35,959	N/A	33,594	48,441	N/A	43,633
INSERT INTO pbench_history (tid, bid, aid, delta, mtime) VALUES (tid, bid, aid, :delta, CURRENT-TIMESTAMP);	215,772	10,607	N/A	9,990	11,277	N/A	11,442
END	191,74	22,403	N/A	19,827	33,196	N/A	27,585

Table 2. Test Case 2 Results split by SQL Query

Test Case 2

Statement Latencies in MS	No LoadBalancer	Aysnch. Session	Async. Statement	Async. Transaction	Sync. Session	Sync. Statement	Sync. Transaction
Set aid random(1, 100000 * :scale)	0,009	0,01	N/A	0,01	0,01	N/A	0,010
set bid random (1, 1 * :scale)	0,002	0,002	N/A	0,002	0,002	N/A	0,002
set tid random (1, 10 * : scale)	0,002	0,002	N/A	0,002	0,002	N/A	0,002
set delta random (-5000, 5000)	0,001	0,001	N/A	0,001	0,001	N/A	0,001
BEGIN	69,829	4,919	N/A	4,718	4,610	N/A	4,545
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;	46,265	4,817	N/A	4,873	4,750	N/A	4,588
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;	51,015	4,478	N/A	4,615	4,362	N/A	4,280
UPDATE pgbench_tellers SET balance = tbalance + :delta WHERE tid = :tid;	40,932	6,159	N/A	6,068	6,113	N/A	6,176
UPDATE pgbench_branches SET balance = balance + :delta WHERE bid = :bid;	67,503	17,146	N/A	12,187	13,261	N/A	12,276
INSERT INTO pbench_history (tid, bid, aid, delta, mtime) VALUES (tid, bid, aid, :delta, CURRENT-TIMESTAMP);	54,352	4,939	N/A	4,880	4,804	N/A	4,684
END	59,397	23,989	N/A	15,533	16,655	N/A	16,808

Table 3. Test Case 3 Results split by SQL Query

Test Case 3

Statement Latencies in MS	No LoadBalancer	Aysnch. Session	Async. Statement	Async. Transaction	Sync. Session	Sync. Statement	Sync. Transaction
Set aid random(1, 100000 * :scale)	0,01	0,011	N/A	0,011	0,011	N/A	0,010
set bid random (1, 1 * :scale)	0,002	0,003	N/A	0,002	0,003	N/A	0,002
set tid random (1, 10 * : scale)	0,002	0,002	N/A	0,002	0,002	N/A	0,002
set delta random (-5000, 5000)	0,001	0,002	N/A	0,001	0,001	N/A	0,001
BEGIN	173,743	1615,517	N/A	467,143	1808,677	N/A	1751,775
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;	192,308	10,946	N/A	11,703	9,916	N/A	12,086
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;	221,886	10,338	N/A	9,960	8,567	N/A	9,620
UPDATE pgbench_tellers SET balance = tbalance + :delta WHERE tid = :tid;	1822,522	16,005	N/A	16,378	16,330	N/A	17,468
UPDATE pgbench_branches SET balance = balance + :delta WHERE bid = :bid;	1622,71	33,532	N/A	39,226	42,506	N/A	42,349
INSERT INTO pbench_history (tid, bid, aid, delta, mtime) VALUES (tid, bid, aid, :delta, CURRENT-TIMESTAMP);	225,494	10,664	N/A	9,896	10,350	N/A	10,756
END	197,588	20,015	N/A	23,754	28,889	N/A	27,727

Table 4. Test Case 4 Results split by SQL Query

Test Case 4 READ ONLY

Statement Latencies in MS	No LoadBalancer	Aysnch. Session	Async. Statement	Async. Transaction	Sync. Session	Sync. Statement	Sync. Transaction
Set aid random(1, 100000 * :scale)	0,01	0,008	0,009	0,009	0,009	0,009	0,009
SELECT abalance FROM pgbench_accounts WHERE aid = :aid,	2239,806	1152,532	144,132	146,462	1159,313	559,479	584,848

Table 5. Asynchronous Session Results Split by SQL Query

PgBouncer Session

	Test Case 1	Test Case 2	Test Case 3	Test Case 4
TPS (Excluding Initial Connection)	165,030707	137,974060	170,823056	248,943860
Transactions Completed (x / total)	30000/60000	10000/10000	30000/30000	30000/30000
Latency Average (ms)	1778,660	66,461	1717,543	1152,540
Latency Std Dev. (ms)	443,408	112,036	343,556	1084,893
Average Connection Time (ms)	3,408	3,118	3,408	3,842
Percentage Completed	50 %	100 %	100 %	100 %

Table 6. Asynchronous Statement Results Split by SQL Query

PgBouncer Statement

	Test Case 1	Test Case 2	Test Case 3	Test Case 4
TPS (Excluding Initial Connection)	N/A	N/A	N/A	344,607014
Transactions Completed (x / total)	N/A	N/A	N/A	10000/30000
Latency Average (ms)	N/A	N/A	N/A	144,141
Latency Std Dev. (ms)	N/A	N/A	N/A	87,581
Average Connection Time (ms)	N/A	N/A	N/A	2,724
Percentage Completed	0 %	0 %	0 %	33,3 %

Table 7. Asynchronous Transaction Results Split by SQL Query

PgBouncer Transaction

	Test Case 1	Test Case 2	Test Case 3	Test Case 4
TPS (Excluding Initial Connection)	168,163193	172,490914	169,246210	323,487286
Transactions Completed (x / total)	10000/60000	10000/10000	30000/30000	10000/10000
Latency Average (ms)	571,515	52,889	577,797	146,472
Latency Std Dev. (ms)	407,579	51,454	244,523	91,999
Average Connection Time (ms)	3,529	2,999	3,303	2,968
Percentage Completed	16,7 %	100 %	100 %	100 %

Table 8. Synchronous Statement Results Split by SQL Query

PgBouncer Session

	Test Case 1	Test Case 2	Test Case 3	Test Case 4
TPS (Excluding Initial Connection)	136,185628	168,249895	153,400339	247,462444
Transactions Completed (x / total)	60000/60000	10000/10000	30000/30000	30000/30000
Latency Average (ms)	4352,132	54,570	1925,251	1159,321
Latency Std Dev. (ms)	975,567	83,705	471,085	1111,746
Average Connection Time (ms)	3,378	2,958	3,363	3,869
Percentage Completed	100 %	100 %	100 %	100 %

Table 9. Synchronous Transaction Results Split by SQL Query

PgBouncer Transaction

	Test Case 1	Test Case 2	Test Case 3	Test Case 4
TPS (Excluding Initial Connection)	154,073458	171,951865	158,449087	254,819685
Transactions Completed (x / total)	60000/60000	10000/10000	30000/30000	30000/30000
Latency Average (ms)	3830,169	53,372	1871,798	584,858
Latency Std Dev. (ms)	665,186	69,677	499,645	748,236
Average Connection Time (ms)	3,319	2,917	3,330	3,799
Percentage Completed	100 %	100 %	100 %	100 %

Table 10. Synchronous Statement Results Split by SQL Query

PgBouncer Statement

	Test Case 1	Test Case 2	Test Case 3	Test Case 4
TPS (Excluding Initial Connection)	N/A	N/A	N/A	258,257379
Transactions Completed (x / total)	N/A	N/A	N/A	30000/30000
Latency Average (ms)	N/A	N/A	N/A	589,489
Latency Std Dev. (ms)	N/A	N/A	N/A	709,744
Average Connection Time (ms)	N/A	N/A	N/A	3,748
Percentage Completed	0 %	0 %	0 %	100 %

Table 11. No Load Balancing Test Results split by SQL Query

No Load Balancing

	Test Case 1	Test Case 2	Test Case 3	Test Case 4
TPS (Excluding Initial Connection)	20,305216	20,761583	20,492837	21,740884
Transactions Completed (x / total)	8000/60000	10000/10000	10000/30000	10000/30000
Latency Average (ms)	3580,043	389,307	4456,266	2239,015
Latency Std Dev. (ms)	3253,457	138,397	4497,239	1275,022
Average Connection Time (ms)	46,419	44,892	46,685	45,674
Percentage Completed	13,3 %	100 %	33,3 %	33,3 %