



# AlloyDB for PostgreSQL - Transactional (OLTP) Benchmarking Guide

May 2023

<b>Disclaimer</b>	<b>2</b>
<b>Overview</b>	<b>3</b>
Benchmarking Process	3
<b>Infrastructure Setup</b>	<b>5</b>
Setting-up AlloyDB Cluster and Instance	5
Provision Client Machine	7
<b>Setup of Benchmark Driver Machine (Client)</b>	<b>8</b>
<b>Benchmark Cleanup: An important Prerequisite</b>	<b>10</b>
<b>TPC-C Benchmark</b>	<b>10</b>
Prerequisites	11
Initial Setup on Client Machine	11
Script to load TPC-C data	12
Running the TPC-C benchmark	14
Analyzing TPC-C Results	16
<b>TPC-C Benchmark on 64 vCPU AlloyDB Instance</b>	<b>19</b>
Infrastructure Setup using 64 vCPU Machine Type	19
Running the benchmark	20
Results Observed	21
<b>PGBench OLTP Benchmark</b>	<b>21</b>
PGBench TPC-B Like Benchmark	21
Customized Write Intensive Scenario [Index-Insert Only]	27
Select-Only (Maximum Throughput) Scenario on 64 vCPU Instance	32
<b>Results Summary</b>	<b>35</b>

## Disclaimer

---

This AlloyDB for PostgreSQL benchmark guide provides best practices for running an Online Transactional Processing (OLTP) benchmark. Your results may vary depending on several factors including, but not limited to the type of AlloyDB instance, type of client machine driving the benchmark, region, zone, and network bandwidth at the time of tests. Nothing in this user guide should be construed as a [promise](#) or [guarantee](#) about the results you'll derive from measuring the OLTP performance of AlloyDB.

# Overview

---

AlloyDB for PostgreSQL on Google Cloud is a relational database built to give you enterprise grade reliability, scalability, and performance suitable for critical, enterprise-level workloads. AlloyDB has state-of-the-art log and transaction management, dynamic memory management, artificial intelligence and machine learning integration, a built-in columnar engine, and a multi-tiered cache, and is based on distributed, scalable storage. As a whole, these features enable high performance for your transactional (OLTP) , analytical (OLAP), and hybrid (HTAP) workloads.

Relational database systems typically require a database administrator (DBA) to optimize them for benchmarking, which includes configuring the transaction log settings, establishing the right buffer pool sizes, and tweaking other important database parameters (flags) and characteristics. These settings also vary based on instance size and type. AlloyDB comes pre-configured with optimal settings and does not require extensive database tuning to achieve high OLTP performance.

This document describes step-by-step procedures and best practices to configure an AlloyDB cluster, a client machine, and scripts to setup, load and run benchmarks.

## Benchmarking Process

---

We'll go through the following steps to set up and run various OLTP benchmarks.

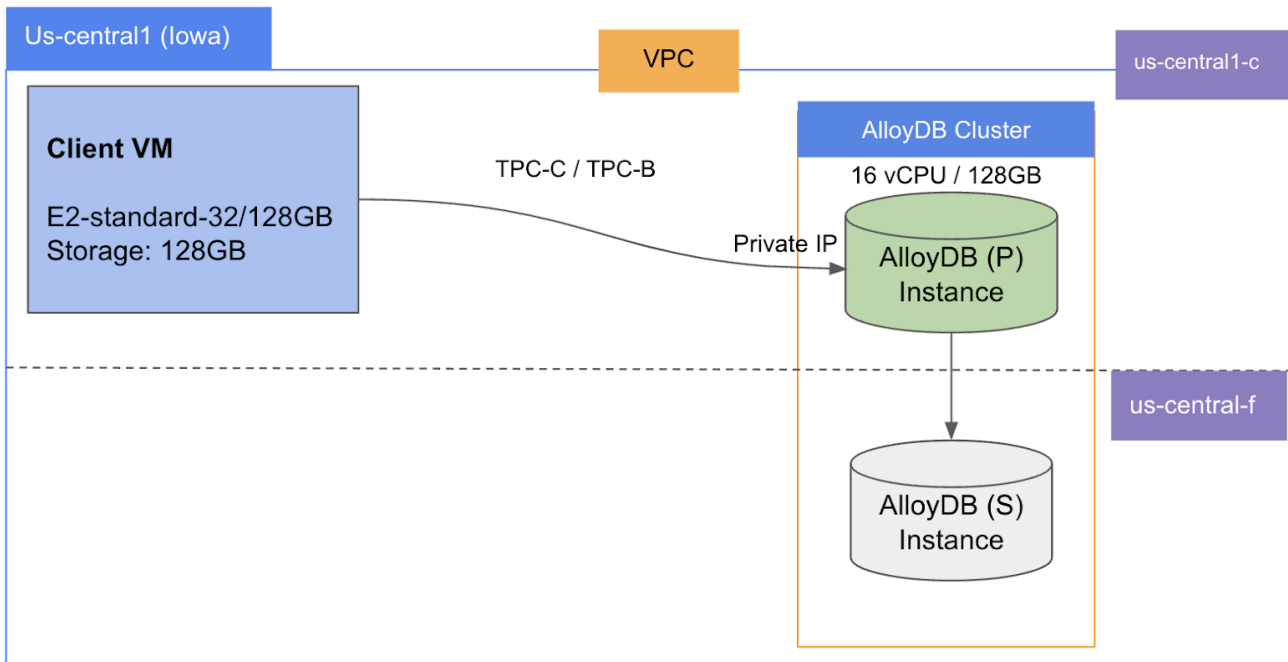
1. Configure an AlloyDB cluster (Server) within a Google Cloud VPC.
2. Setup of benchmark driver client virtual machine running on Google Cloud, where we will install benchmarking tools.
3. Install [HammerDB](#) and [pgbench](#) tools on the client machine.
4. Run TPC-C like benchmark using HammerDB.
5. Run TPC-B like benchmark using pgbench.
6. Run write-intensive workload using pgbench.
7. Run read-intensive workload using pgbench.

Unless otherwise specified, we used following setup for performance benchmarking:

Component	Value
AlloyDB Cluster Type	Highly Available
AlloyDB Machine Type	16vCPU / 128GB / Storage auto-allocated. Intel® Xeon® Platinum 8373C Processor (Ice Lake) 3rd Generation*
Database Version	PostgreSQL 14 compatible (14.4)
Region	us-central1 (Iowa)

Component	Value
AlloyDB Primary zone	us-central1-c (Auto-selected)
AlloyDB Secondary zone	Us-central1-f (Auto-selected)
Client VM - Machine Type	<p><b>For TPCC:</b> E2-standard-32 (Intel-Broadwell) / 128GB / 128 GB persistent disk as boot disk</p> <p><b>For PGBench:</b> E2-standard-16 (intel-Broadwell) / 64GB / 128 GB persistent disk as boot disk</p> <p><b>Operating System:</b> Debian linux</p>
Zone of Client VM	us-central1-c [same as AlloyDB primary instance]
Connectivity	Private IP over VPC
Test tools	<p>HammerDB-4.6</p> <p>PGBench 13.9 (Debian 13.9-0+deb11u1)</p> <p>Psql</p>
Workloads	<p>TPC-C benchmark on 16 and 64 vCPU machines in following modes:</p> <ul style="list-style-type: none"> <li>(1) 30% data on cache</li> <li>(2) 100% data on cache</li> </ul> <p>PGBench:</p> <ul style="list-style-type: none"> <li>(1) TPC-B Like with Partially-cached Database (~650GB)</li> <li>(2) TPC-B Like with fully-cached DB (60GB)</li> <li>(3) Index-Insert-Only</li> <li>(4) Select-Only on 64 vCPU</li> </ul>

\* When you deploy AlloyDB, it will be provisioned on either [Intel Cascade Lake or the newer Intel Ice Lake platform](#) depending on the availability in the region and zone.



## Infrastructure Setup

### Setting-up AlloyDB Cluster and Instance

1. Create or select your GCP project: Go to <https://console.cloud.google.com> and select your project from the drop down menu or create a new one.
2. Follow these links on the portal: “Products and Solutions” → “All Products” → “Databases” → “AlloyDB for PostgreSQL”.
3. Click on the following button to create an AlloyDB cluster.



4. Choose “**Highly Available**” for the cluster type and “**PostgreSQL14**” for the database. For illustration, consider the image below.

✓ Choose a cluster type to start with

This choice isn't permanent – you can add read pool instances to your cluster any time.

Type Highly available

2 Configure your cluster

Provide some basic information about your cluster

Basic info

Cluster ID \*

Use lowercase letters, numbers, and hyphens. Start with a letter.

Password \*  [GENERATE](#)

Set a password for the default "postgres" user. A password is required for the user to log in.

Database version  
PostgreSQL 14 compatible

Storage  
Cluster storage scales automatically, so you only pay for what you use

Location

For better performance, keep your data close to the services that need it. Choice is permanent.

Region \*

Networking

Clusters can only be configured with a private IP network path. [Learn more](#)

Network \*

✓ Private services access connection for network **default** has been successfully created. You will now be able to use the same network across all your project's managed services. If you would like to change this connection, please visit the [Networking page](#).

▼ ADVANCED ENCRYPTION OPTIONS

[CONTINUE](#)

3 Configure your primary instance


A primary instance determines a cluster's compute capacity and supports read and write operations.

[CREATE CLUSTER](#) [CANCEL](#)

- 5. The primary instance is configured using 16 vCPU with 128 GB RAM. Note the location of the **primary zone** and **private IP**. This benchmarking exercise is conducted without a readpool. The instance should be a multi-zone instance (i.e. highly available). Use the illustration below as a guide.

Instances in your cluster + ADD READ POOL ← Don't add read pool

Primary instance ?	
Status	Ready
High availability	Highly available (multi-zone)
Location	us-central1-c ← PRIMARY ZONE (secondary zone: us-central1-f)
Machine type	16 vCPU, 128 GB
Private IP	172.20.0.209
Flags	No flags set
EDIT PRIMARY	



### READ POOL

Read pool instances increase your cluster's read capacity by aggregating nodes, which you can scale, enabling highly available reads. [Learn more](#)

ADD READ POOL INSTANCE

## Provision Client Machine

To run the OLTP benchmarks, you will require a client machine with enough processing power. The benchmark drivers like HammerDB and PGBench runs in a highly parallel fashion and consumes a significant amount of CPU. The client machines' configurations are chosen in a way that they should not be a bottleneck for this experiment.

**For HammerDB TPC-C benchmark:** An **E2-standard-32** machine with 128 GB RAM and 128 GB disk as a client for driving TPC-C benchmark. The client machine is created in the same zone as AlloyDB's primary instance.

**For the PGBench benchmark:** An **E2-standard-16** machine with 64 GB RAM and 128 GB disk is used. The client machine is created in the same zone as AlloyDB's primary instance.

**Important:** For this exercise, the **Debian linux** client must be provisioned in the same region, zone, and VPC as AlloyDB's primary instance. Benchmarking tools directly access the AlloyDB instance over private IP.

Below is a sample client machine we provisioned to execute the TPC-C benchmark on an AlloyDB primary instance with 16 virtual CPUs.

## Basic information

Name	[REDACTED]
Instance Id	[REDACTED]
Description	None
Type	Instance
Status	✓ Running
Creation time	Feb 22, 2023, 4:18:26 AM UTC-08:00
Zone	us-central1-c ← SAME AS PRIMARY INSTANCE
Instance template	None
In use by	None
Reservations	Automatically choose
Labels	None
Tags <sup>?</sup>	<a href="#">ise-api-enabler-access : true</a> <a href="#">strategy-0aa8714a-wave : wave-2</a> <a href="#">strategy-34650193-wave : wave-2</a> <a href="#">strategy-67cdeade-wave : wave-1</a> <a href="#">strategy-762a8ab3-wave : wave-2</a> <a href="#">strategy-8bf36cf7-wave : wave-2</a> <a href="#">strategy-9b21db73-wave : wave-1</a> <a href="#">zonal-dns-rollout-wave-teams : wave-2</a>
Deletion protection	Disabled
Confidential VM service <sup>?</sup>	Disabled
Preserved state size	0 GB

## Machine configuration

Machine type	e2-standard-32
CPU platform	Intel Broadwell
Architecture	x86/64
vCPUs to core ratio <sup>?</sup>	—
Custom visible cores <sup>?</sup>	—
Display device	Disabled Enable to use screen capturing and recording tools
GPUs	None

## Networking

Public DNS PTR Record	None
Total egress bandwidth tier	—
NIC type	—

→ [VIEW IN NETWORK TOPOLOGY](#)

# Setup of Benchmark Driver Machine (Client)

This section will guide you through the steps of configuring the client machine running on Google Cloud, where we will install benchmarking tools such as HammerDB and PGBench.



Connect to the client machine using the “`gcloud compute ssh`” command. Refer this documentation for details “<https://cloud.google.com/sdk/gcloud/reference/compute/ssh>”.

Sample command:

```
gcloud compute ssh --zone "<primary zone>" "<client machine name>" --project "<google-project>"
```

## Install PostgreSQL client

You will need a `psql` client application to connect to AlloyDB PostgreSQL. Use the following command to install a `postgresql` client that includes a `psql` application and then ensure you are able to connect.

```
sudo apt-get update
sudo apt install postgresql-client
```

Now ensure that it works and you are able to connect to the AlloyDB PostgreSQL. Use the “Private IP” address of your primary AlloyDB instance.

```
psql -h <Private IP> -U postgres
```

## Install HammerDB-4.6 Driver for TPC-C benchmark

For this benchmarking guide, we utilized HammerDB-4.6 driver. Execute the following commands to install the HammerDB driver.

```
mkdir hammerdb
pushd hammerdb
curl -OL
https://github.com/TPC-Council/HammerDB/releases/download/v4.6/HammerDB-4.6-Linux.tar.gz
tar zxvf HammerDB-4.6-Linux.tar.gz
```

## Install pgbench Driver for TPCB-Like and other OLTP benchmarking

We utilized the open-source [pgbench utility](#) to assess the performance of `TPCB-like` apps. `TPC-B` mode in `pgbench` provides a standardized and customizable way to measure the performance of a PostgreSQL database system in a transactional workload that resembles a banking scenario. We will also cover a few customized read and write-intensive OLTP situations, such as `Index-Insert-Only` and `Select-Only`. The user may choose the scenario that best fits their unique business requirements.

Use following commands to install the PGBench utility:

```
sudo apt-get update
```

```
sudo apt-get install postgresql-contrib
pgbench --version
```

The version that we got is the following: `pgbench (PostgreSQL) 13.9 (Debian 13.9-0+deb11u1)`.

## Benchmark Cleanup: An important Prerequisite

---

This step is important if you are planning to execute multiple benchmarks in succession. Performing a proper cleanup between each benchmark is a critical prerequisite for accurate and reliable benchmarking results. This includes deleting previous benchmark data (i.e. benchmark database), and rebooting the AlloyDB instance (that clears caches at database and operating systems level) before running another benchmark. A proper benchmark cleanup ensures that residual effects from previous benchmarks do not affect the performance measurements of the new benchmark. It also helps to ensure consistency and repeatability of the benchmark results, which is essential for making meaningful comparisons between different systems or identifying areas for optimization in hardware, software, or configuration.

Follow the URL <https://cloud.google.com/alloydb/docs/instance-restart> to learn more about how to reboot an AlloyDB instance.

To drop the previous benchmark database, you can use the following psql command from the client machine.

```
psql -h <Private IP> -U postgres -c "DROP DATABASE [IF EXISTS] <database_name>;"
```

## TPC-C Benchmark

---

HammerDB is a benchmarking tool that includes a [TPC-C](#) benchmark implementation for evaluating the performance of OLTP systems. HammerDB's TPC-C implementation allows you to simulate a workload similar to the TPC-C benchmark, including a mix of transactions that mimic the behavior of a wholesale supplier environment. HammerDB measures the system's performance in terms of transactions per minute (TPM) and generates reports that include detailed statistics and performance metrics. Additionally, HammerDB supports customization of the benchmark parameters, allowing users to adjust the database size, the number of warehouses, and other workload characteristics to simulate different scenarios.

This section provides a comprehensive guide on how to execute the HammerDB TPC-C benchmark to gauge the performance of the AlloyDB PostgreSQL database system.

## Prerequisites

---

- A. You need to run the following steps from a client (driver) machine. Ensure that you have completed the setup steps listed in the [“Setup of Benchmark Driver Machine \(Client\)”](#) section (especially installation of the HammerDB utility).
- B. **Cleanup:** If you are running multiple benchmarks in succession, ensure you follow the [“Cleanup: An important Prerequisite”](#) section before doing your subsequent run.

## Initial Setup on Client Machine

---

Execute all commands from `hammerdb/HammerDB-4.6` directory.

```
cd hammerdb/HammerDB-4.6
```

Then create `setup.env` file as follows:

```
cat << EOF > setup.env

# Private IP of the AlloyDB primary instance
export PGHOST=111.222.333.444

# Postgres default port address. You do not need to change it unless you use non-default port
address.
export PGPORT=5432 # default port to connect with postgres

# Number of TPC-C warehouses to load. This determines the overall database size.
export NUM_WAREHOUSE=576

# Set the password that you used during AlloyDB instance creation.
export PGPASSWORD='<postgres_user_password>'

# Number of users for running the benchmark.
export NUM_USERS=256
EOF
```

Edit the generated `setup.env` file and change all the **highlighted** parameter values to those that are suitable to your environment setup.

For the purpose of this benchmarking guide, we evaluate the performance in the following two crucial scenarios:

1. **Partially (~30%) cached mode:** In this mode, we generate a large TPC-C database which can only partially fit in the buffer cache. The transactions in this mode will not be always served from

memory and will incur IO to the underlying storage subsystems. This scenario is more realistic to the OLTP needs of the majority of customers with large data set.

To test this scenario, change `NUM_WAREHOUSE` as `3200` in the `setup.env` file.

2. **Fully (100%) cached mode**, where the TPC-C database fully fits in the buffer cache. AlloyDB instance utilizes approximately 90% of the available 128 GB RAM including buffer cache. Since TPC-C transactions perform minimal IO's (as reads are mostly served from buffer cache) in this mode, higher TPM is expected compared to partially-cached runs.

To test this scenario, change `NUM_WAREHOUSE` as `576` in the `setup.env` file.

**NOTE:** The number of users (or clients) is set to 256 for this test. This number of users has been tuned to provide the best throughput with acceptable latency on both of these configurations.

### *Script to load TPC-C data*

---

In the context of the TPC-C benchmark, a "load step" refers to the process of populating the benchmark database with initial data before running the actual performance test.

During this step, the database is populated with a specified number of warehouses, customers, and other entities according to the TPC-C specifications. The purpose of the load step is to create a realistic workload for the performance test, and to ensure that the test results are comparable across different systems.

After the load step is completed, the database is pre-populated with a defined set of initial data, and ready to be used for the TPC-C benchmark test.

Follow the steps below to load the TPC-C database:

1. Switch to the benchmark home directory.

```
cd hammerdb/HammerDB-4.6
```

2. Create `build-tpcc.sh` file as follows:

```
#!/bin/bash -x

source ./setup.env

# create role tpcc with superuser login as 'postgres' and password as 'AlloyDB#123';
# -----

./hammerdbcli << EOF
```

```

# CONFIGURE PARAMETERS FOR TPCC BENCHMARK
# -----
dbset db pg
dbset bm tpc-c

# CONFIGURE POSTGRES HOST AND PORT
# -----
diset connection pg_host $PGHOST
diset connection pg_port $PGPORT

# CONFIGURE TPCC
# -----
diset tpcc pg_superuser postgres
diset tpcc pg_superuserpass $PGPASSWORD
diset tpcc pg_user tpcc
diset tpcc pg_pass $PGPASSWORD
diset tpcc pg_dbase tpcc

# SET NUMBER OF WAREHOUSES AND USERS TO MANAGE EACH WAREHOUSE
# THIS IMPORTANT METRIC ESTABLISHES THE DATABASE SCALE/SIZE
# -----
diset tpcc pg_count_ware $NUM_WAREHOUSE
diset tpcc pg_num_vu 10

# LOG OUTPUT AND CONFIGURATION DETAILS
# -----
vuset logtotemp 1
print dict

# CREATE AND POPULATE DATABASE SCHEMA
# -----
buildschema

waittocomplete
vudestroy
quit

EOF

```

3. Execute the load command as shown below and wait for the command to finish.

```

chmod +x ./build-tpcc.sh
mkdir results
sudo nohup ./build-tpcc.sh > results/build-tpcc.out 2>&1

```

4. **Validate Load:** After the aforementioned script completes, it is recommended to confirm that the database load was successful. The database's size can be quickly verified by doing as follows:

```
$ psql -h $PGHOST -p 5432 -U postgres
postgres=> \l+ tpcc
```

List of databases					
Name	Owner	Encoding	Collate	Ctype	Access privileges
	Size	Tablespace		Description	
tpcc	tpcc	UTF8	C.UTF-8	C.UTF-8	
	--- GB	pg_default			

In 30% cached TPC-C configuration (with 3200 warehouses), expect the size of the TPC-C database to be around **300 GB**.

In 100% cached TPC-C configuration (with 576 warehouses), expect the size of the TPC-C database to be around **55 GB**.

### Running the TPC-C benchmark

In this step, we will initiate the actual TPC-C performance test. The TPC-C benchmark will be executed using the populated database from the load step. The benchmark generates a series of transactions that simulate a typical business environment, including order entry, payment processing, and inventory management. The workload is measured in "transactions per minute" (TPM), which represents the number of complete business transactions that the system can handle in one minute.

The run step is designed to stress the database system under realistic conditions and provide a standard way of measuring performance that can be compared across different database systems. Vendors and customers widely use the results of the TPC-C benchmark to evaluate the performance of different database systems and hardware configurations.

The following script will run the TPC-C benchmark for about 1 hour after approximately 10 minutes of warm up.

1. Switch to benchmark home directory:

```
cd hammerdb/HammerDB-4.6
```

2. Create `run-tpcc.sh` script as follows:

```
#!/bin/bash -x
```

```

source ./setup.env

./hammerdbcli << EOF
dbset db pg
dbset bm tpc-c

# CONFIGURE PG HOST and PORT
# -----
diset connection pg_host $PGHOST
diset connection pg_port $PGPORT

# CONFIGURE TPCC DB
# -----
diset tpcc pg_superuser postgres
diset tpcc pg_superuserpass $PGPASSWORD
diset tpcc pg_user postgres
diset tpcc pg_pass $PGPASSWORD
diset tpcc pg_dbase tpcc

# BENCHMARKING PARAMETERS
# -----
diset tpcc pg_driver timed
diset tpcc pg_rampup 10
diset tpcc pg_duration 60
diset tpcc pg_vacuum false
diset tpcc pg_partition false
diset tpcc pg_allwarehouse true
diset tpcc pg_timeprofile true
diset tpcc pg_connect_pool false
diset tpcc pg_dritasnap false
diset tpcc pg_count_ware $NUM_WAREHOUSE
diset tpcc pg_num_vu 1

loadscript
print dict
vuset logtotemp 1
vuset vu $NUM_USERS
vucreate
vurun
waittocomplete
quit
EOF

```

### 3. Run the script as follows:

```

chmod +x run-tpcc.sh

```

```
mkdir results
sudo nohup ./run-tpcc.sh > results/run-tpcc.out 2>&1
```

Now wait for the `run-tpcc.sh` script to finish. The script will take approximately 1 hour and 10 minutes to complete.

## Analyzing TPC-C Results

---

In the context of the TPC-C benchmark, **NOPM** and **TPM** are performance metrics used to measure the performance of a database system. **NOPM** stands for "New Orders Per Minute" and measures the number of new order transactions that the system can handle in one minute. The New Order transaction is one of the most important transactions in the TPC-C benchmark and involves creating a new order for a customer.

**TPM** stands for "Transactions Per Minute" and measures the total number of completed business transactions that the system can handle in one minute. This includes not only **New Order** transactions but also **Payment**, **Delivery**, **Order Status**, and other types of transactions defined in the TPC-C benchmark.

In general, TPM is considered to be the primary performance metric for the TPC-C benchmark, as it provides an overall measure of the system's ability to handle a realistic workload. However, NOPM can also be a useful metric for systems that are heavily focused on processing new orders, such as e-commerce or retail systems.

## Measured Results With AlloyDB

With 30% cached TPC-C database on 16 vCPU machine (i.e. `NUM_WAREHOUSE=3200` and `NUM_USERS=256`), we observed **252,970** tpm-C (New Order Per Minute) from a cumulative **582,385** AlloyDB TPM. These performance numbers can be extracted using following command:

```
$ grep NOPM results/run-tpcc.out
Vuser 1:TEST RESULT : System achieved 252970 NOPM from 582385 PostgreSQL TPM
```

On a 100% cached TPC-C database on 16 vCPU machine (i.e. `NUM_WAREHOUSE=576` and `NUM_USERS=256`), we got **428,316** tpm-C (New Order Per Minute) from a cumulative **974,264** AlloyDB TPM :

```
$ grep NOPM results/tpcc-run.out
Vuser 1:TEST RESULT : System achieved 428316 NOPM from 974264 PostgreSQL TPM
```

Summary of performance results on 16 vCPU.

TPC-C Scenario	NUM_WAREHOUSE	NUM_USERS	New Order Per Minute (NOPM)	Cumulative TPM
----------------	---------------	-----------	-----------------------------	----------------



30% cached	3200	256	252,970	582,385
100% cached	576	256	428,316	974,264

## Observability

To further understand the behavior of the database system, AlloyDB users can monitor important system metrics, such as CPU usage, memory usage, transactions per second, etc. from the AlloyDB instance overview page and/or navigate to the **Monitoring** page on <https://console.cloud.google.com>.

For instance, the mean CPU utilization we got for 100% cached TPC-C run is almost 90% as shown in the picture below.

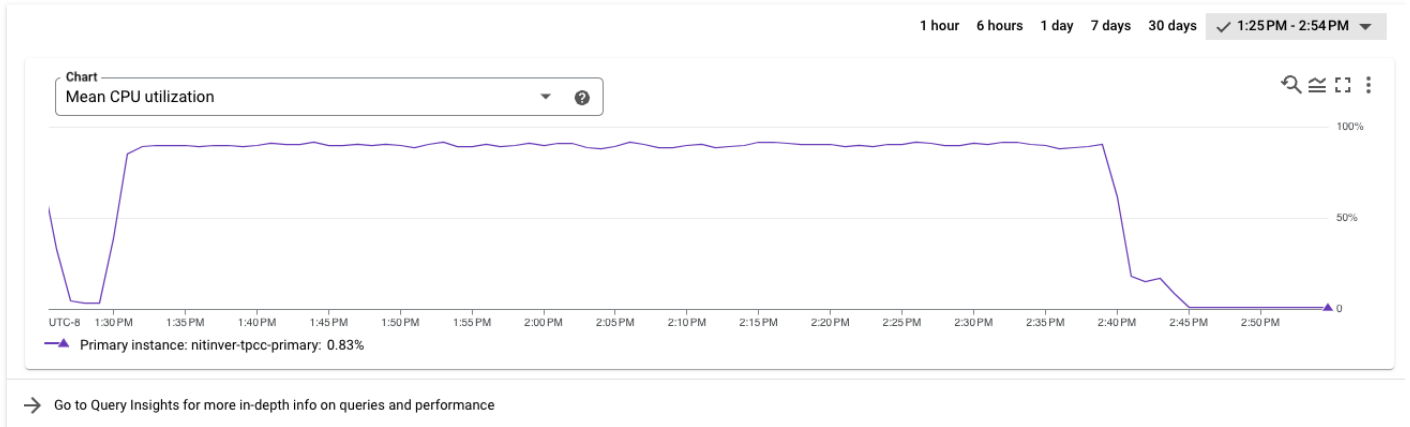
## Overview



Clusters are groups of instances of virtual machines that can include a primary instance and multiple read pool instances. All cluster resources share a storage layer, which scales as needed. [Learn more](#)

Status	Ready
Cluster ID	[REDACTED]
Version	PostgreSQL 14 compatible
Type	Highly available
Location	us-central1 (Iowa)  Low CO2
Total storage used	313.33 GB
Network	[REDACTED]/global/networks/default

TPCC 100% CACHED CPU UTILIZATION



## Instances in your cluster [+ ADD READ POOL](#)

[REDACTED]

Primary instance

Status	Ready
High availability	Highly available (multi-zone)
Location	us-central1-c (secondary zone: us-central1-f)
Machine type	16 vCPU, 128 GB
Private IP	[REDACTED]
Flags	No flags set

[EDIT PRIMARY](#)

### READ POOL

Read pool instances increase your cluster's read capacity by aggregating nodes, which you can scale, enabling highly available reads. [Learn more](#)

[ADD READ POOL INSTANCE](#)

AlloyDB provides high transaction concurrency at low latency, allowing the database system to fully consume the available CPU bandwidth. The speed of transaction commits enables efficient CPU use as opposed to stalling caused by bottlenecks and suboptimal log processing in other database systems.

# TPC-C Benchmark on 64 vCPU AlloyDB Instance

## Infrastructure Setup using 64 vCPU Machine Type

### AlloyDB Setup

The overall instructions for the setup of AlloyDB Postgres with 64 vCPU machine type are similar to the steps outlined in section “[Setting-up AlloyDB Cluster and Instance](#)”. The **Machine Type** is the only parameter that is different from those instructions. The user needs to pick the **Machine Type** as **64 vCPU, 512 GB**. Below is the snapshot of a 64 vCPU AlloyDB instance that we created for this benchmarking guide. That concludes the database server setup!

Instances in your cluster [+ ADD READ POOL](#)

████████████████████


Primary instance ?

---

Status	<span>✓</span> Ready
High availability	Highly available (multi-zone)
Location	us-west1-a (secondary zone: us-west1-b)
Machine type	64 vCPU, 512 GB
Private IP	██████████
Flags	No flags set

---

EDIT PRIMARY



### READ POOL



Read pool instances increase your cluster's read capacity by aggregating nodes, which you can scale, enabling highly available reads. [Learn more](#)

[ADD READ POOL INSTANCE](#)

### Client Machine Setup

To setup a client machine, you need to follow the steps outlined in “[Provision Client Machine](#)” except the **Machine Type** parameter that changes to **n2-standard-64** machine. Ensure that the client machine is located in the zone of AlloyDB primary instance. Below is the screenshot of our client machine configuration.

## Machine configuration

Machine type	n2-standard-64
CPU platform	Intel Cascade Lake
Architecture	x86/64
vCPUs to core ratio 	—
Custom visible cores 	—
Display device	Disabled Enable to use screen capturing and recording tools
GPUs	None

## Networking

Public DNS PTR Record	None
Total egress bandwidth tier	—
NIC type	—

Then follow the instructions outlined in the section “[Setup of Benchmark Driver Machine \(Client\)](#)”.

## *Running the benchmark*

---

Follow the steps outlined below to run the benchmark:

1. Follow the “[Prerequisites](#)” section.
2. Then follow “[Initial Setup on Client Machine](#)” and use following parameter values:
  - Set `PGHOST` to the “Private IP” of your new 64 vCPU AlloyDB instance.
  - For 30% Cached TPC-C scenario, set `NUM_WAREHOUSE=128000` and `NUM_USERS=1024`.
  - For 100% Cached TPC-C scenario, set `NUM_WAREHOUSE=2304` and `NUM_USERS=1024`.
3. To setup and load a TPC-C database, follow the “[Load TPC-C script](#)” section.  
**NOTE:** In order to speed-up the load, change the value of `pg_num_vu` to 64 in `build-tpcc.sh` as `diset tpcc pg_num_vu 64`.
4. Then follow the exact steps in “[Running the TPC-C benchmark](#)”.

## Results Observed

Benchmark Mode	NUM_WAREHOUSE	NUM_USERS	New Order Per Minute (NOPM)	Cumulative TPM
30% cached	128000	1024	589,598	1,371,160
100% cached	2304	1024	716,138	1,665,438

## PGBench OLTP Benchmark

PGBench is a benchmarking tool that comes bundled with PostgreSQL. It allows you to simulate transaction workloads such as inserting, updating, selecting data and measuring the database system's performance in Transactions Per Second (TPS). With PGBench, you can customize the database size, number of clients and transaction mix to emulate your production workload and obtain insights into the system's behavior under different scenarios.

### General Prerequisites:

- A. You need to run the following steps from a client (driver) machine. Ensure that you have completed the setup steps listed in the [“Setup of Benchmark Driver Machine \(Client\)”](#) section (especially installation of the PGBench utility).
- B. **Cleanup:** If you are running multiple benchmarks in succession, ensure you follow the [“Cleanup: An important Prerequisite”](#) section before your subsequent run.

## PGBench TPC-B Like Benchmark

[TPC-B](#) (Transaction Processing Performance Council Benchmark B) is one of the benchmark modes available in PGBench, a benchmarking tool for PostgreSQL. TPC-B simulates a banking scenario where multiple tellers execute transactions on customer accounts. The workload consists of three types of transactions: *deposits*, *withdrawals*, and *balance* inquiries. The benchmark measures the performance of the database system by simulating a mix of these transactions and measuring the number of transactions per second that the system can handle.

The "tpcb-like" mode in PGBench generates a synthetic database and simulates a mix of transactions that resembles the TPC-B workload but it is not officially certified by the TPC organization. Therefore, while the "tpcb-like" mode in PGBench provides a useful approximation of TPC-B performance, it should not be used to claim compliance with TPC-B standards.

In this section, we provide a step by step guide to measure TPCB-Like performance in the following two critical modes. The only parameter that is different in these two modes is the value of `SCALE_FACTOR` parameter.

## Partially-cached Database Scenario

In this scenario, we setup and initialize a large database (approximately 650GB in size by using `--scale = 50000`). Having a large database that does not fit in memory and causes significant disk I/O provides a more realistic representation of many production workloads. A large database that causes significant disk I/O can underscore the importance of database design and query optimization. It can expose performance issues related to disk I/O, such as slow disk access or inefficient queries, that may not be apparent in a small or entirely memory-resident database.

## Fully-cached Database Scenario

In this scenario, we setup and initialize a database of approximately 60GB in size by using `--scale=4000` so that it resides in the buffer pool. Benchmarking a memory-resident database is important because it allows you to assess the maximum performance of the database system in a controlled environment. A memory-resident database stores all data in the Postgres buffer pool, eliminating the I/O bottleneck that can occur when accessing data from disk. This mode can help identify performance bottlenecks that are not related to I/O, such as CPU usage or locking issues, that may not be apparent when benchmarking a database that relies on disk I/O.

## Infrastructure Setup

**Database Server:** AlloyDB PostgreSQL with machine type as 16 vCPU and 128 GB RAM

**Client Machine:** E2-standard-16 (minimum) as indicated in the section [“Provision Client Machine”](#).

## Steps to run the TPCB-Like benchmark

Follow these steps to run TPC-B like benchmark:

1. Connect to the client machine using gcloud command as follows:

```
$ gcloud compute ssh --zone "<primary zone>" "<client machine name>" --project
"<google-project>"
```

2. Create the `pgbench-setup.env` file as follows:

```
$ cat << EOF > pgbench-setup.env

# Private IP of the AlloyDB primary instance
export PGHOST=<private_ip>
```

```
# In pgbench, the scale factor represents the size of the test database.
# and is defined as the number of 1 MB-sized data pages to be generated per client.
export SCALE_FACTOR=<scale_factor>

# Set the password that you used during AlloyDB instance creation.
export PGPASSWORD='<postgres_user_password>'

EOF
```

Edit the generated `setup.env` file and change the following parameter values to those that are suitable to your environment setup.

`<private_ip>`: The private IP of your AlloyDB instance.

`<scale_factor>`: You need to pick the scale factor according to your scenario and stick to it for all the benchmarking steps in this section.

- For a partially-cached database scenario pick value as **50000** (i.e. `--scale=50000`).
- For the Fully-cached database scenario pick value as **4000** (i.e. `--scale=4000`).

`<postgres_user_password>`: Set the password that you provided during AlloyDB instance creation.

3. Create a `pgbench` database as follows after editing the parameters in the environment file.

```
$ source ./pgbench-setup.env
$ psql -h $PGHOST -p 5432 -U postgres

postgres=> create database pgbench;
CREATE DATABASE
```

4. **Initialize and load PGBench database:** This step ensures that the benchmarking dataset is created and populated with realistic data, allowing you to accurately simulate TPC-B like workload on the `pgbench` database. You just need to run the following command:

```
$ source ./pgbench-setup.env

$ sudo nohup pgbench -i --host=$PGHOST --user=postgres --scale=$SCALE_FACTOR pgbench >
/tmp/pgbench-tpcb-partially-cached-db-init.out 2>&1
```

#### Expected Load Time:

- The partially-cached database takes approximately 6 hours to load.
- The fully-cached database takes approximately 45 minutes to load.

#### Load Accuracy Checks (Optional):

Ensure that the contents of the `/tmp/pgbench-tpcb-partially-cached-db-init.out` file are similar to the following:

```
creating tables...
generating data (client-side)...
100000 of 400000000 tuples (0%) done (elapsed 0.02 s, remaining 99.82 s)
. . . . .
. . . . .
399800000 of 400000000 tuples (99%) done (elapsed 534.60 s, remaining 0.27 s)
399900000 of 400000000 tuples (99%) done (elapsed 534.72 s, remaining 0.13 s)
400000000 of 400000000 tuples (100%) done (elapsed 534.85 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 1481.92 s (drop tables 0.01 s, create tables 0.04 s, client-side
generate 540.93 s, vacuum 615.11 s, primary keys 325.84 s).
```

Optionally, if you want to further validate the accuracy of your load, you can run following PostgreSQL command measuring size of all pgbench tables:

Connect to the `pgbench` database:

```
$ source ./pgbench-setup.env
$ psql -h $PGHOST -p 5432 -U postgres -d pgbench
```

Then run the following SQL command:

```
pgbench=> SELECT nspname AS schema_name, relname AS table_name,
pg_size_pretty(pg_total_relation_size(C.oid)) AS size FROM pg_class C
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
WHERE nspname NOT LIKE 'pg_%' AND nspname != 'information_schema'
ORDER BY pg_total_relation_size(C.oid) DESC;
```

Compare the output of the above command with the output that we got for the partially-cached database run (`SCALE_FACTOR=50000`).

schema_name	table_name	size
public	pgbench_accounts	731 GB
public	pgbench_accounts_pkey	105 GB
public	pgbench_tellers	32 MB
public	pgbench_tellers_pkey	11 MB
public	pgbench_branches	2952 kB
public	pgbench_branches_pkey	1112 kB
...		
public	pgbench_history	0 bytes



```
.. .. ..  
(29 rows)
```

**NOTE:** The size of tables and indices will be much smaller for `SCALE_FACTOR=4000`.

5. Now, we are ready to execute the final TPCB-like run step that simulates a financial accounting system workload by executing a series of transactions involving deposits, transfers and payments, to measure the database's performance under a heavy workload.

```
$ source ./pgbench-setup.env  
$ mkdir -p ~/results/alloydb/pgbench  
$ sudo nohup pgbench --host=$PGHOST --user=postgres --builtin=tpcb-like --time=3900  
--jobs=256 --client=256 --scale=$SCALE_FACTOR --protocol=simple --progress=1 pgbench  
> ~/results/alloydb/pgbench/pgbench.run.out 2>&1
```

## Results Observed

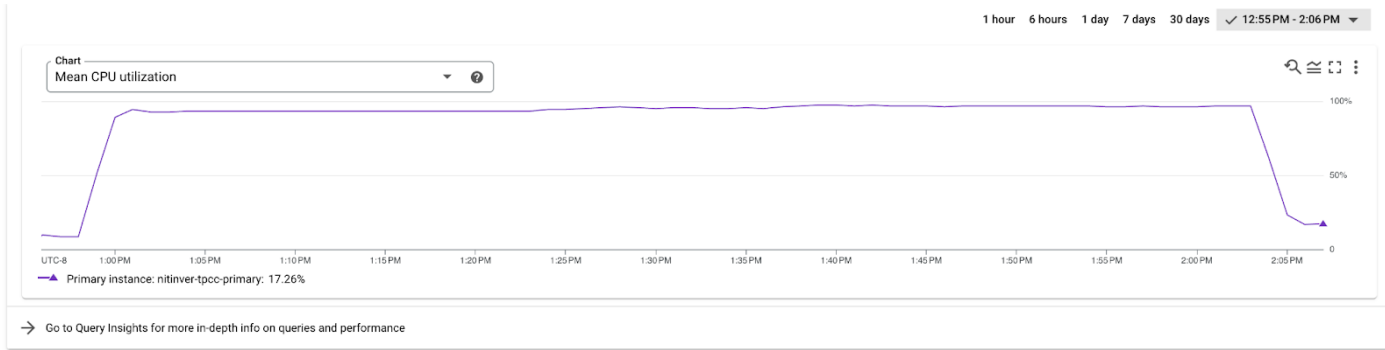
Check the output of the last command in `~/results/alloydb/pgbench/pgbench.run.out` file. The TPS (Transactions Per Second) number that you see in the report should be close to the numbers that we see below.

- Fully-cached Database (`--scale=4000`)

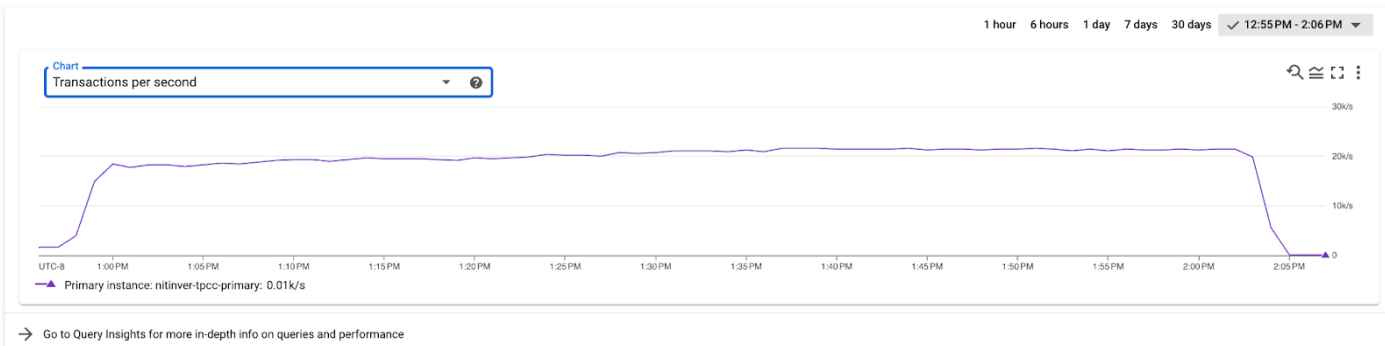
```
transaction type: <builtin: TPC-B (sort of)>  
scaling factor: 4000  
query mode: simple  
number of clients: 256  
number of threads: 256  
duration: 3900 s  
number of transactions actually processed: 79392806  
latency average = 12.573 ms  
latency stddev = 13.625 ms  
tps = 20356.543420 (including connections establishing)  
tps = 20359.357116 (excluding connections establishing)
```

To further understand the behavior of the database system, AlloyDB users can monitor important system metrics, such as CPU usage, memory usage, transactions per second, etc. from the AlloyDB instance overview page on <https://console.cloud.google.com>.

CPU Utilization: ~96%



## TPS chart:



NOTE: It is important to run the benchmark for a longer duration as the throughput is lower in the first few seconds and takes time to reach steady state.

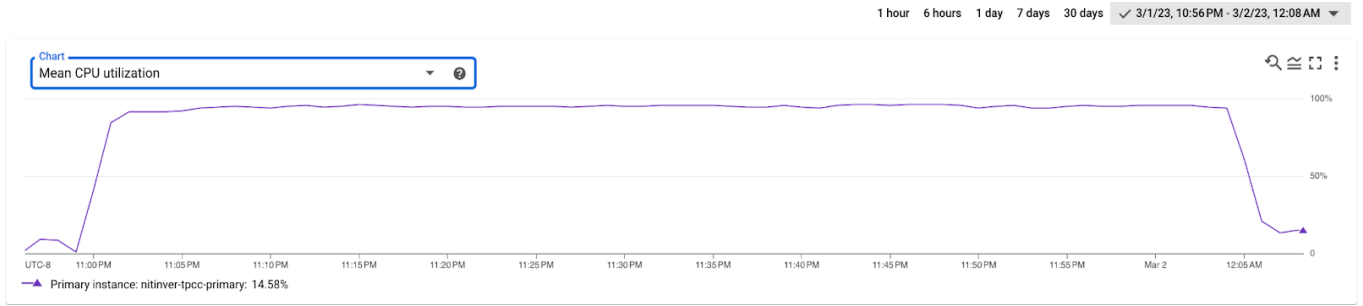
- **Partially-cached Database (--scale=50000):**

```

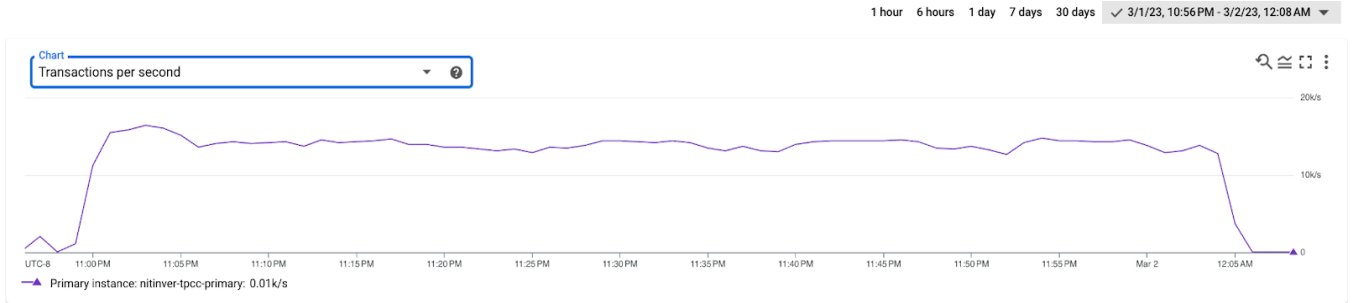
pgbench: warning: scale option ignored, using count from pgbench_branches table
(50000)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 50000
query mode: simple
number of clients: 256
number of threads: 256
duration: 3900 s
number of transactions actually processed: 54828812
latency average = 18.210 ms
tps = 14058.224129 (including connections establishing)
tps = 14060.221209 (excluding connections establishing)

```

**CPU utilization: 94%**



**TPS chart:**



**Summary:**

TPC-B Scenario	SCALE_FACTOR	TPS	CPU Utilization (%)
Partially cached	50000	14,060	96%
Fully Cached	4000	20,359	94%

**Customized Write Intensive Scenario [Index-Insert Only]**

PGBench benchmarking tool can be customizable to measure results based on specific use cases and simulate real-world scenarios. We want to measure performance for a write intensive workload.

**Why is this scenario important?**

When a relational database like PostgreSQL is compared against NoSQL databases, such as MongoDB or Cassandra, the write performance of PostgreSQL is not typically high due to overheads including strict ACID requirements.

One of AlloyDB's design objectives is to enhance PostgreSQL's write performance, which in turn improves OLTP throughput. To boost the performance of write-intensive OLTP situations, AlloyDB PostgreSQL has made a number of architectural innovations including tiered cache layer to help with reads and a distributed and write-scaling using highly scalable storage engine technology to which write processing is offloaded.

The “**Index Insert Only**” benchmark is a highly concurrent write intensive scenario that has been customized to showcase the performance benefits of AlloyDB for the majority of OLTP applications. In this scenario, we create multiple indices on the `pgbench_history` table and then repeatedly perform INSERT operations on the `pgbench_history` table from multiple client connections.

In this section, we provide a step by step guide to measure the performance of “Index-Insert Only” workload.

## Infrastructure Setup

**Database Server:** AlloyDB PostgreSQL with machine type as 16 vCPU and 128 GB RAM

**Client Machine:** E2-standard-16 (minimum) as indicated in the section “[Provision Client Machine](#)”.

## Steps to run “Index-insert Only” benchmark

1. Connect to the client machine. The following command is an example:

```
$ gcloud compute ssh --zone "<primary zone>" "<client machine name>" --project
"<google-project>"
```

2. **Setup the environment:** It is advisable to run all the following commands from the same client terminal. Then all you need to do is export the PGHOST environment variable once and assign it to the private IP of your AlloyDB instance. If you connect multiple terminals to the client machine, then export the following environment variable on all the terminals.

```
$ export PGHOST=<private_ip>
```

3. Create a “pgbench” database following the example below (*if the database already exists, then drop the database and recreate it. Alternatively, you can create a database with another name*):

```
$ psql -h $PGHOST -p 5432 -U postgres
psql (13.9 (Debian 13.9-0+deb11u1), server 14.4)
...

postgres=> create database pgbench;
CREATE DATABASE
```

4. **Initialize and load PGBench database:** This step ensures that the benchmarking dataset is created and populated with realistic data. You just need to run the following command after editing highlighted parameters:

```
$ sudo nohup pgbench -i --host=$PGHOST --user=postgres --scale=25000 pgbench > /tmp/pgbench-index-insert-only-init.out 2>&1
```

Validate that the output of above command is similar to the following:

```
dropping old tables...
creating tables...
generating data (client-side)...
100000 of 2500000000 tuples (0%) done (elapsed 0.03 s, remaining 636.43 s)
200000 of 2500000000 tuples (0%) done (elapsed 0.05 s, remaining 649.12 s)
. . . . .
. . . . .
2499900000 of 2500000000 tuples (99%) done (elapsed 3425.42 s, remaining 0.14 s)
2500000000 of 2500000000 tuples (100%) done (elapsed 3425.57 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 12851.19 s (drop tables 998.62 s, create tables 0.02 s, client-side generate
3460.33 s, vacuum 5299.93 s, primary keys 3092.29 s).
```

5. Create `index-init.sql` script as follows:

```
$ cat > index-init.sql << EOF
CREATE INDEX tid ON pgbench_history(tid);
CREATE INDEX bid ON pgbench_history(bid);
CREATE INDEX aid ON pgbench_history(aid);
CREATE INDEX delta ON pgbench_history(delta);
CREATE INDEX mtime ON pgbench_history(mtime);
EOF
```

6. Now execute the `index-init.sql` script as follows:

```
$ psql -h $PGHOST -U postgres -d pgbench -f ./index-init.sql
Password for user postgres:
CREATE INDEX
```

7. Optional steps to validate the database schema and initial load:

```
$ psql -h $PGHOST -U postgres -d pgbench

pgbench=> \dt
          List of relations
Schema |          Name          | Type  | Owner
-----+-----+-----+-----
public | pgbench_accounts      | table | postgres
```

```

public | pgbench_branches | table | postgres
public | pgbench_history   | table | postgres
public | pgbench_tellers    | table | postgres
(4 rows)

```

```
pgbench=> \di
```

```

                                List of relations
 Schema |          Name          | Type  | Owner  | Table
-----+-----+-----+-----+-----
public | aid                    | index | postgres | pgbench_history
public | bid                    | index | postgres | pgbench_history
public | delta                  | index | postgres | pgbench_history
public | mtime                  | index | postgres | pgbench_history
public | pgbench_accounts_pkey | index | postgres | pgbench_accounts
public | pgbench_branches_pkey | index | postgres | pgbench_branches
public | pgbench_tellers_pkey  | index | postgres | pgbench_tellers
public | tid                    | index | postgres | pgbench_history
(8 rows)

```

Database size is expected to be around **365GB** after the load.

```
pgbench=> \l+ pgbench
```

```

databases
 Name          | Owner  | Encoding | Collate | Ctype | Access
-----+-----+-----+-----+-----+-----
privileges    | Size  | Tablespace |          | Description
-----+-----+-----+-----+-----+-----
...
pgbench       | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
| 365 GB | pg_default |
...

```

8. Create the `index-inserts-only.sql` script as shown below:

```

$ cat > index-inserts-only.sql << EOF
\set aid random(1, 100000000)
\set bid random(1, 100000000)
\set tid random(1, 100000000)
\set delta random(-50000000, 50000000)
BEGIN;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid,
:delta, CURRENT_TIMESTAMP);
END;
EOF

```

9. Now run the PGBench benchmark using following command:

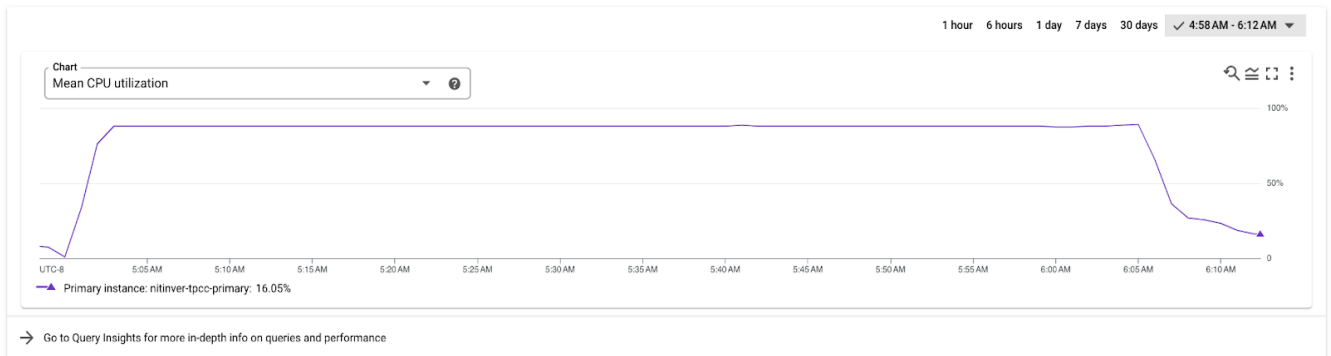
```
$ sudo nohup pgbench --host=$PGHOST --user=postgres --time=3900 --client=256  
--jobs=256 --scale=25000 --progress=1 --file=./index-inserts-only.sql pgbench >  
/tmp/pgbench-index-insert-only-run.out 2>&1
```

## Results Observed

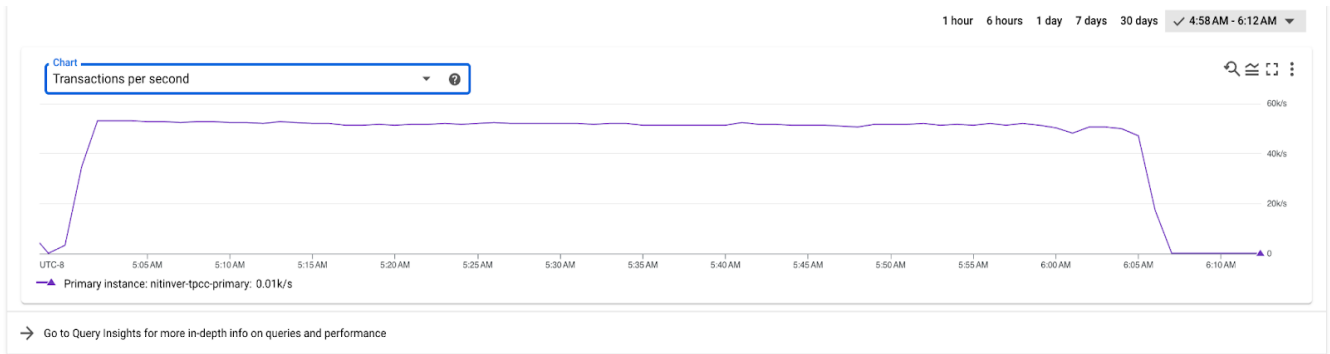
Check the output of the last command in the `/tmp/pgbench-index-insert-only-run.out` file. We observed approximately 52K transactions per sec during this benchmark test (as shown below).

```
scaling factor: 25000  
query mode: simple  
number of clients: 256  
number of threads: 256  
duration: 3900 s  
number of transactions actually processed: 201785196  
latency average = 4.947 ms  
latency stddev = 3.488 ms  
tps = 51738.604613 (including connections establishing)  
tps = 51742.459757 (excluding connections establishing)
```

CPU Utilization: ~88%



TPS Chart:



## Select-Only (Maximum Throughput) Scenario on 64 vCPU Instance

PGBench supports a built-in `select-only` scenario that repeatedly executes SELECT queries from multiple client connections against a specified database. It is used to measure the random read performance of the database, without introducing the overhead of data modification operations like INSERT, UPDATE, or DELETE. These SELECT queries are essentially point lookup queries that are the fastest and most efficient type of select queries as they involve accessing only a single row of data directly from the index structures

### Why is this scenario important?

- **Achieving Maximum Throughput:** Since point lookups on an index are the most efficient form of queries in a database system, we can measure maximum possible throughput that AlloyDB PostgreSQL can achieve.
- **Scalability:** This scenario is also ideal for testing the scalability of AlloyDB from 2 vCPU to the maximum vCPU configuration offered by AlloyDB PostgreSQL.

### Infrastructure Setup

Follow the precise instructions of section “[Infrastructure Setup using 64 vCPU Machine Type](#)”.

### Steps to run “Select-Only” benchmark

1. Connect to the client machine. The following command is an example:

```
$ gcloud compute ssh --zone "<primary zone>" "<client machine name>" --project
"<google-project>"
```

2. **Setup environment:** It is advisable to run all the following commands from the same terminal on the client machine. Then all you need to do is export the PGHOST environment variable once and assign it to the private IP of your AlloyDB instance.



```
$ export PGHOST=<private_ip>
```

3. Create the `pgbench` database following the example below. (If the database already exists, then you may want to drop it and recreate it. Alternatively, you can create a database with another name):

```
$ psql -h $PGHOST -p 5432 -U postgres  
  
postgres=> create database pgbench;  
CREATE DATABASE
```

4. **Initialize PGBench database:** This step will initialize `pgbench` database with approximately 220 GB of realistic data. We use `--scale=15000` for the fully cached `Select-Only` benchmark. You just need to execute the following command:

```
$ sudo nohup pgbench -i --host=$PGHOST --user=postgres --scale=15000 pgbench >  
/tmp/pgbench-select-only-init.out 2>&1
```

Validate that the output of above command is similar to the following:

```
$ cat /tmp/pgbench-select-only-init.out  
nohup: ignoring input  
dropping old tables...  
creating tables...  
generating data (client-side)...  
100000 of 1500000000 tuples (0%) done (elapsed 0.01 s, remaining 161.60 s)  
200000 of 1500000000 tuples (0%) done (elapsed 0.03 s, remaining 224.35 s)  
300000 of 1500000000 tuples (0%) done (elapsed 0.09 s, remaining 448.97 s)  
.. .. ..  
.. .. ..  
1499900000 of 1500000000 tuples (99%) done (elapsed 1251.03 s, remaining 0.08 s)  
1500000000 of 1500000000 tuples (100%) done (elapsed 1251.10 s, remaining 0.00 s)  
vacuuming...  
creating primary keys...  
done in 2204.62 s (drop tables 2.29 s, create tables 0.01 s, client-side generate  
1271.82 s, vacuum 427.83 s, primary keys 502.66 s).
```

5. **Run PGBench:** Now run the last benchmarking step as follows. This step will take over one hour to complete.

```
$ sudo nohup pgbench --host=$PGHOST --user=postgres --builtin=select-only --time=3900  
--jobs=256 --client=256 --scale=15000 --protocol=simple --progress=1 pgbench >  
/tmp/pgbench-select-only-run.out 2>&1
```

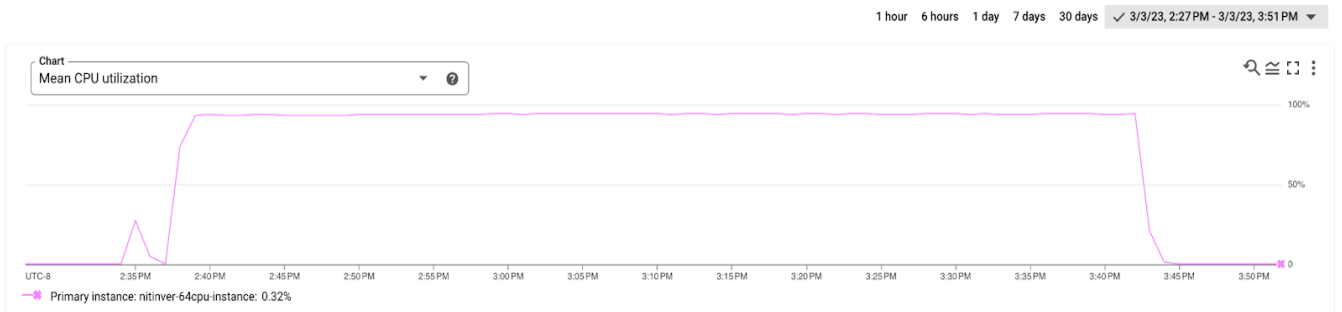
Check the `/tmp/pgbench-select-only-run.out` file for the final results after the above benchmark run completes.

## Results Observed

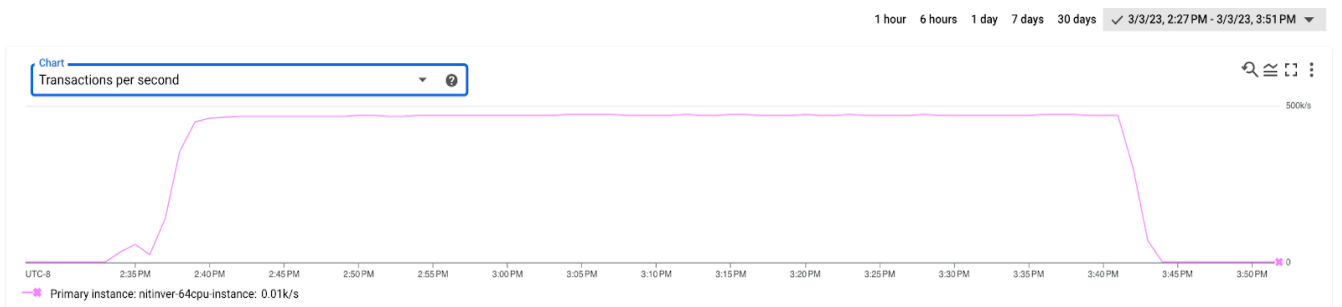
We observed approximately 467k transactions per sec during this benchmark test (as shown below).

```
$ cat /tmp/pgbench-select-only-run.out
transaction type: <builtin: select only>
scaling factor: 15000
query mode: simple
number of clients: 256
number of threads: 256
duration: 3900 s
number of transactions actually processed: 1823506174
latency average = 0.547 ms
latency stddev = 0.267 ms
tps = 467563.144333 (including connections establishing)
tps = 467583.398400 (excluding connections establishing)
```

**CPU Utilization:** ~95%



**TPS Chart:**



# Results Summary

This section is intended to provide a summary of our observations based on the benchmarks explained in this document.

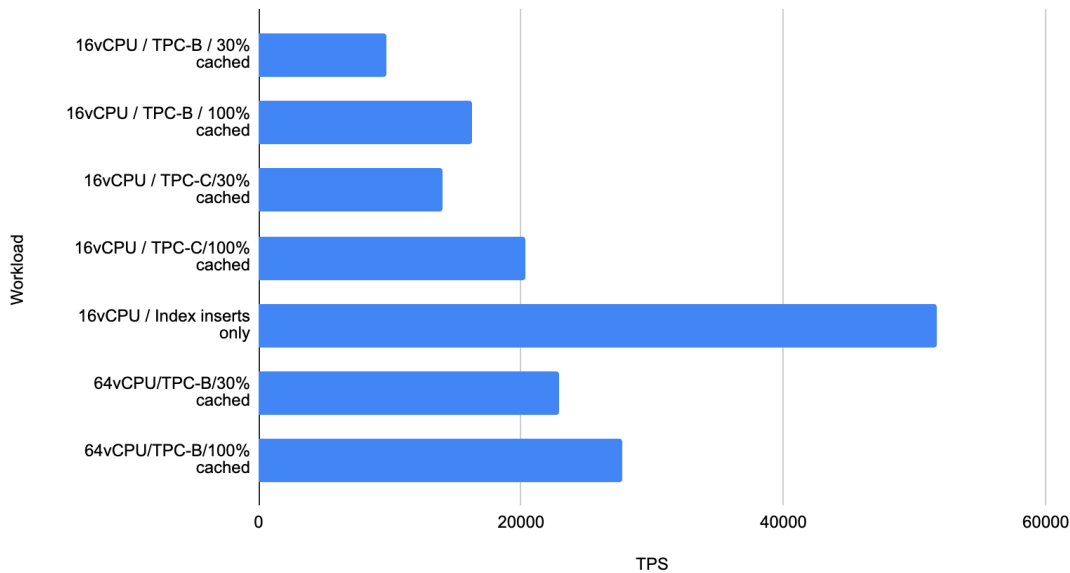
## HammerDB TPC-C Performance Summary

AlloyDB Machine Type	TPC-C Workload Scenario	NUM_WAREHOUSE	NUM_USERS	New Orders Per Minute (NOPM)	Cumulative TPM	Converted to TPS
16vCPU	30% cached	3200	256	252,970	582,385	9,706
16vCPU	100% cached	576	256	428,316	974,264	16,238
64vCPU	30% cached	12800	1024	589,598	1,371,160	22,853
64vCPU	100% cached	2304	1024	716,138	1,665,438	27,757

## PGBench Performance Summary

AllotyDB Machine Type	PgBench Workload Scenario	Scale Factor	TPS	CPU %
16vCPU	TPC-B Like, Fully Cached	4000	20,359	96%
16vCPU	TPC-B Like, Partially Cached	50000	14,060	94%
16vCPU	Index inserts only	25000	51,742	88%
64vCPU	Max. Throughput (Select Only)	15000	467,583	95%

TPS vs. Workload



## **Authors**

Nitin Verma, Software Engineer, AlloyDB, Google Cloud  
Sridhar Ranganathan, Product Manager, AlloyDB, Google Cloud