# AlloyDB for PostgreSQL - Analytical (OLAP) Benchmarking Guide

May 2023

## Disclaimer

This AlloyDB for PostgreSQL benchmark guide provides best practices for running an Online Analytical (OLAP) 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 OLAP 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.

The focus of this guide is to provide a step-by-step procedure to evaluate the analytical performance of AlloyDB which is powered by the `Columnar Engine` feature that stores and manages data in the columnar format. The Columnar Engine is designed and optimized for the efficient storage and retrieval of column data for analytical workloads where the emphasis is on efficiently processing large volumes of data compared to row-based data storage and to generate insights, analysis and reporting. The analytical queries execute substantially faster because the Columnar Engine selectively accesses and processes only the columns of data that are pertinent to the query, resulting in significant query performance improvements. Users of AlloyDB have a choice of running only transactional workloads (disable Columnar Engine), run analytical queries along with transactional workloads (enable Columnar Engine and allocate appropriate memory), or run purely analytical workloads on read pools.

Relational database systems typically require database administrators (DBAs) 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 would also vary depending on the size and type of the instance. AlloyDB comes pre-configured with optimal Columnar Engine settings for each machine type and requires very minimal tuning to achieve an optimal OLAP performance.

This document describes a step-by-step procedure to deploy and configure the AlloyDB cluster, a benchmark driving (client) machine, and provides best practices to measure the performance of AlloyDB using a variety of OLAP benchmarks, like HammerDB TPROC-H (derived from TPC-H) with different scale factors and OLAP atomic queries developed internally at Google.

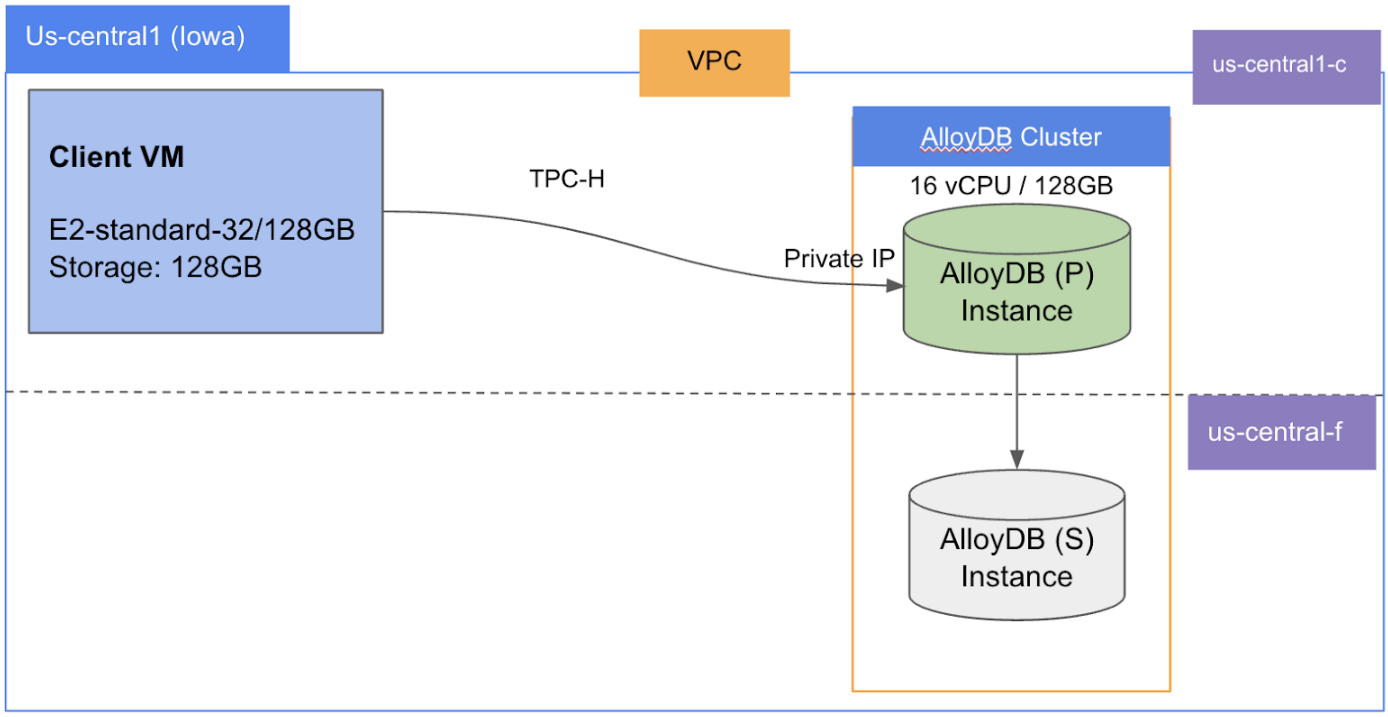
Since HammerDB's TPROC-H implementation is a close variant of the official TPC-H benchmark, we will use the terms TPC-H and TPROC-H interchangeably throughout this user guide.

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.<br>Intel® Xeon® Platinum 8373C Processor (Ice Lake) 3rd Generation* |

| Component | Value |
|---|---|
| Database Version | PostgreSQL 14 compatible (14.4) |
| Region | us-central1 (Iowa) |
| AlloyDB Primary zone | us-central1-c (Auto-selected) |
| AlloyDB Secondary zone | Us-central1-f (Auto-selected) |
| Client VM - Machine Type | E2-standard-32 (Intel-Broadwell) / 128GB / 128 GB persistent disk as boot disk<br><br>NOTE: A large client machine can help you with faster load of TPC-H database. For power run of TPC-H, you don't need a large machine.<br><br>**Operating System**: Debian 5.10.162-1, x86_64 GNU/Linux |
| Zone of Client VM | us-central1-c [same as AlloyDB primary instance] |
| Connectivity | Private IP over VPC |
| Test tools | HammerDB-4.6<br>Psql |
| Workloads | TPC-H benchmark on a 16 vCPU machine with scale-factor of 10, 30 and 100.<br><br>A collection of 11 primitive OLAP queries to measure the decision support capability of the AlloyDB columnar engine. |

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 that region.




# Infrastructure Setup

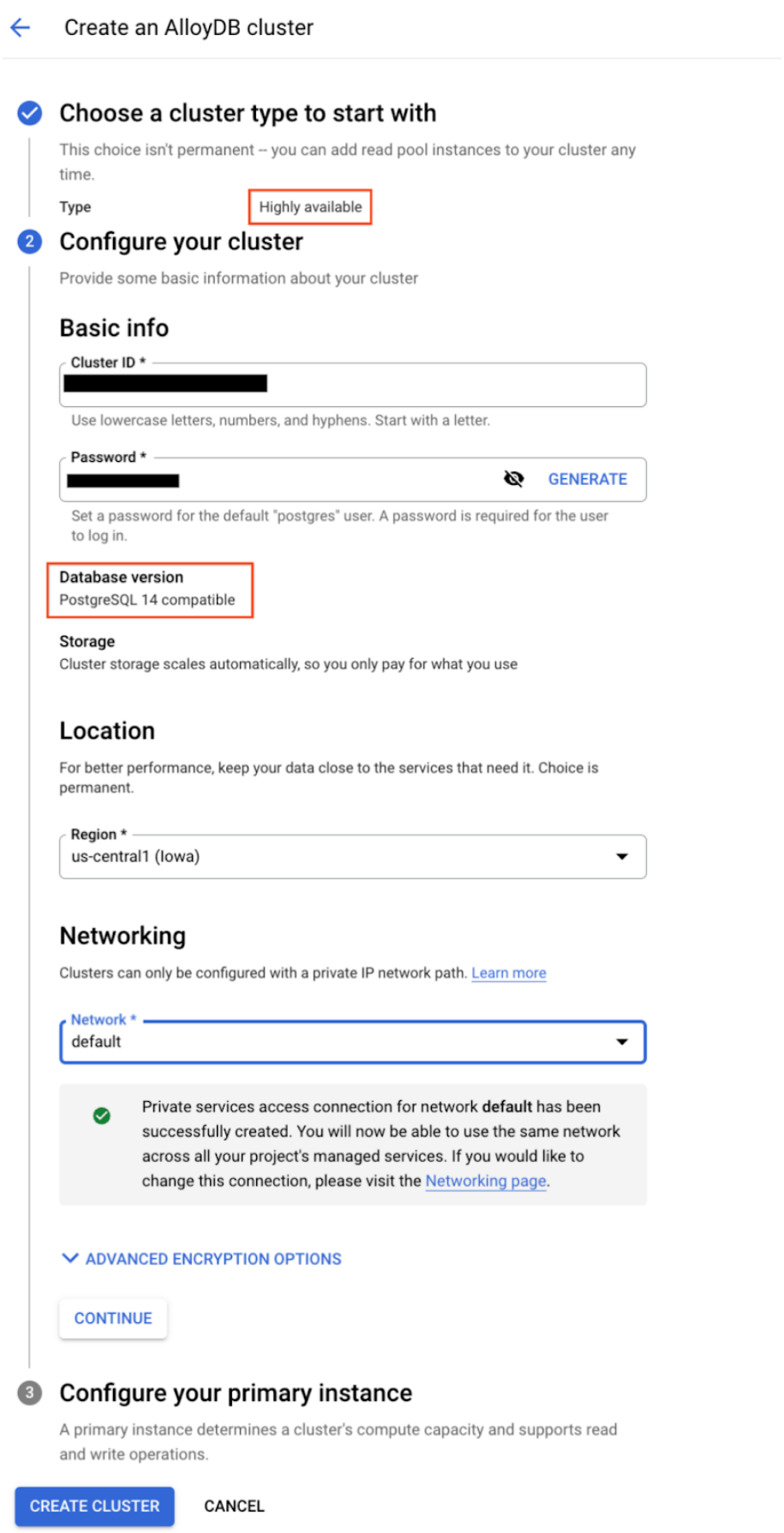
## *Setting-up AlloyDB Cluster and Instance*

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.

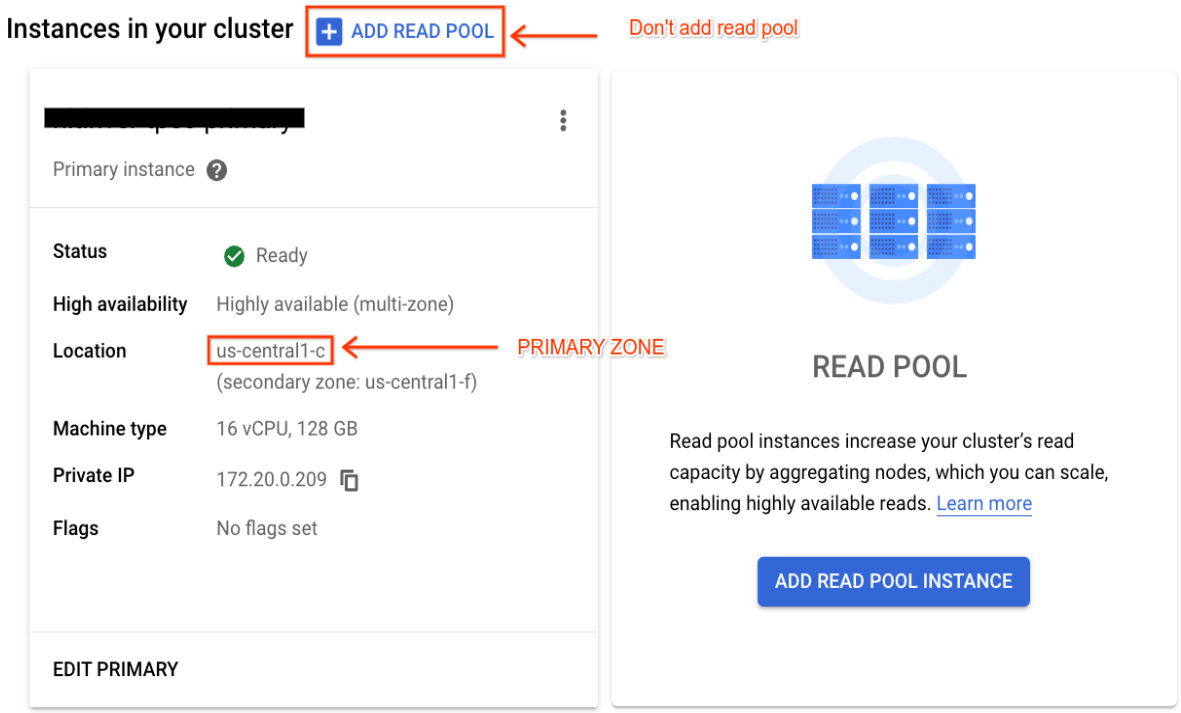
2. Follow these links on the portal: “Products and Solutions” → “All Products” → “Databases” → “AlloyDB for PostgreSQL”.

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.


Create an AlloyDB cluster
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 *
us-central1 (Iowa)
Networking
Clusters can only be configured with a private IP network path. Learn more
Network *
default
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. Unless otherwise specified, in this guide, we used a 16 vCPU with 128 GB RAM as a primary AlloyDB instance deployed in a highly available mode without a readpool. Note the location of the **primary zone** and **private IP**. These will be used when configuring the client machine. Use the illustration below as a guide.




6. Configuring and tuning AlloyDB columnar-engine settings varies depending on the benchmark type and scenario. Those instructions will be covered in a later section.

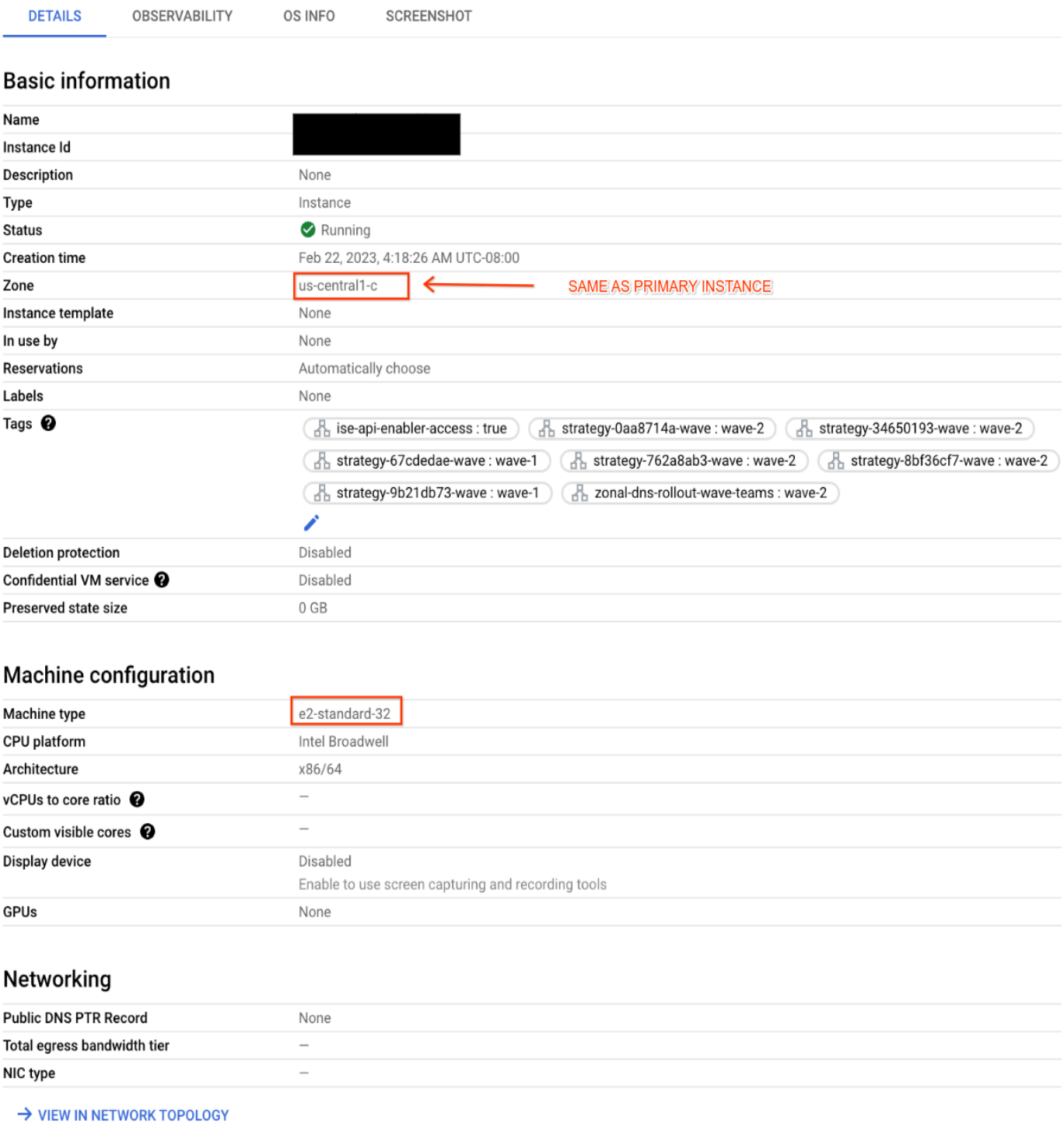
## *Provision Client Machine*

Unless otherwise specified, we used an E2-standard-32 VM with 128 GB disk as a client for the TPC-H benchmarking. The client VM is created in the **same zone** as AlloyDB's primary instance.

For this analytical benchmarking guide, we will be primarily using TPC-H and OLAP atomic queries, and we do not need a large client VM to execute the benchmarks (i.e. queries). However, loading a large TPC-H database (especially, scale factor of size 30 or 100) will be faster with a large client machine.

**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-H benchmark on an AlloyDB primary instance with 16 virtual CPUs.




## 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 important tools such as HammerDB and PSQL.

Connect to the client machine using the **"gcloud compute ssh"** command. Refer to this documentation for details "https://cloud.google.com/sdk/gcloud/reference/compute/ssh".

Sample `gcloud` command to connect with the client machine:

```
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.

```
export PGPASSWORD=<password of postgres user set during AlloyDB instance creation>
export PGHOST=<Private IP of your AlloyDB Primary Instance>
psql -U postgres
```

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

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

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

## 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-H Benchmark

HammerDB is a popular benchmarking tool that includes TPC-H (A standard decision support benchmarking tool) implementation for evaluating performance of OLAP support in AlloyDB PostgreSQL. HammerDB TPC-H measures the performance of a database system by executing a set of 22 standard queries. The TPC-H benchmark is a widely accepted industry standard benchmark for decision support systems that involves complex queries and large data sets.

This section provides a comprehensive guide on how users can customize HammerDB to execute the TPC-H 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 Benchmarking Setup*

Connect to the client machine and execute all the following commands from the `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

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

# TPC-H Scale Factor (determines the size of the database that we want to build).
export TPCH_SCALE=10

EOF
```

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

For the purpose of this benchmarking guide, we evaluate the performance using three important scale factor (`TPCH_SCALE`) sizes (i.e. 10, 30 and 100) of the TPC-H benchmark.

In the context of TPC-H benchmark, scale factor refers to the size of the data set used in the benchmarking process. The scale factor is determined by the number of rows in the TPC-H database tables and it represents the volume of data to be processed by TPC-H queries.

The scale factors **10**, **30**, and **100** represent data sets of approximate sizes **20GB**, **60GB** and **200GB**, respectively. The significance of trying these different scale factors is to evaluate the performance of the database system under varying data volumes and workloads.

When a database system is tested with a smaller scale factor, such as 10, it may perform well as the data set size is relatively small. However, as the data set size increases, the performance of the database system may decrease due to increased resource consumption, buffer cache hit misses, and other processing overheads. Testing the database system with larger scale factors, such as 30 or 100, can help identify potential performance bottlenecks and scalability issues in the database system that may arise under heavy workloads and larger data sets.

Furthermore, testing with different scale factors helps to evaluate a database system's ability to scale with increasing data sizes. This information can be useful for organizations that need to handle large amounts of data and require a database system that can scale efficiently to meet their needs.

**NOTE**: The number of users (or clients) is set to 1, since this user guide is only running TPC-H in **power mode** and not the throughput mode.

## *Script to load TPC-H data*

For the TPC-H 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 benchmarking tool inserts data into the `tpch` database tables according to the specified scale factor. 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 in a consistent state with a defined set of initial data, ready to be used for the TPC-H benchmark test.

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

1. Switch to the benchmark home directory.

```
cd hammerdb/HammerDB-4.6
source ./setup.env
```

2. Create **`build-tpch.sh`** file as follows:

```
#!/bin/bash -x
source ./setup.env

./hammerdbcli << EOF

# CONFIGURE PARAMETERS FOR TPC-H BENCHMARK
# -------------------------------------
dbset db pg
dbset bm tpc-h

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

# CONFIGURE TPC-H
# -------------------------------------
diset tpch pg_tpch_superuser postgres
diset tpch pg_tpch_superuserpass $PGPASSWORD
diset tpch pg_tpch_user postgres
diset tpch pg_tpch_pass $PGPASSWORD
diset tpch pg_tpch_dbase tpch
```

```
# ------------------------------------
diset tpch pg_scale_fact $TPCH_SCALE
diset tpch pg_num_tpch_threads 32
diset tpch pg_refresh_on false

diset tpch pg_refresh_verbose false
diset tpch pg_degree_of_parallel 8
vuset vu 1

# logging
vuset logtotemp 1
vuset timestamps 0
vuset unique 0

# load and run benchmarking script
loadscript
buildschema

# terminate when completed

vudestroy
quit
EOF
```

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

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

4. **Validate Load**: The load step is an important aspect of the TPC-H benchmark because it affects the benchmark's accuracy and repeatability. The quality and consistency of the data that is loaded into the database can have a significant impact on the performance measurements, and therefore, it is important to validate that the load step is executed properly.

   Use the following commands to validate the load quickly:

```
$ . ./setup.env
$ psql -h $PGHOST -U postgres
postgres=> \l+ tpch
                                                                    List of
databases
    Name     |      Owner       | Encoding | Collate |  Ctype  |          Access
privileges           |  Size   | Tablespace |                Description
```

```
-------------+-----------------+----------+---------+---------+---------------------
-----------------+---------+------------+-------------------------------------------
 tpch         | postgres              | UTF8     | C.UTF-8 | C.UTF-8 |
                  | --- GB   | pg_default |
```

The scale factors 10, 30, and 100 represent data sets of approximate sizes **20GB**, **60GB** and **200GB**, respectively. Ensure that the size of the `tpch` database matches the scale factor of your choice.

## *Columnar Engine (CE) Flags*

AlloyDB's Columnar engine related parameters (flags) come with optimal settings and no tuning is generally required except that the columnar engine is to be enabled. However, for this user guide, updating them with proper values allows for efficient processing of analytical queries, reduces query response times and improves resource utilization, which are critical factors for organizations that need to handle large volumes of data and require fast and accurate analysis of that data.

### Important Flags to Tune

The following are the database flags that we tune to enhance the efficacy of OLAP workloads:

| Database Flag | Is AlloyDB Unique? | Description | Default Value |
|---|---|---|---|
| work_mem | No | Increasing the work_mem value can improve the performance of queries that perform a lot of temporary work (like sorting, hashing, bitmap, *etc.*). If your AlloyDB instance does not have adequate memory, a very high value of work_mem may cause performance issues. | 16MB |
| default_statistics_target | No | Increasing the default_statistics value can improve the accuracy of the query planner's estimates, which can lead to better performance for queries that access the column. However, significantly high values can also increase the time it takes to analyze the table. | 100 |
| google_columnar_engine.enabled | Yes | This configuration flag in AlloyDB for PostgreSQL specifies whether the Columnar Engine is enabled or not. The Columnar Engine is a new feature in AlloyDB that can significantly improve the performance of analytical queries. | OFF |
| google_columnar_engine.memory_size_in_mb | Yes | This flag in AlloyDB for PostgreSQL specifies the amount of memory that is allocated to the columnar engine. The default value is ~30% of the RAM on the VM, but it can be increased or decreased depending on the needs of your database. | ~30% of the RAM size |

| | | | |
|---|---|---|---|
| google_columnar_engine.relations | Yes | This configuration flag in AlloyDB for PostgreSQL specifies a set of tables and their columns that need to be stored in the columnar format. The columnar format is a more efficient way to store data for analytical queries, so using this flag can improve the performance of those queries. | Empty string. |

## Tuning for Scale Factors 10 and 30

Since the database sizes for scale factors 10 and 30 are significantly smaller than available RAM (128GB) on the 16 vCPU machine type, we can simply allow all the entire `tpch` database (i.e. all the columns of all `tpch` relations) to be populated in the columnar engine.

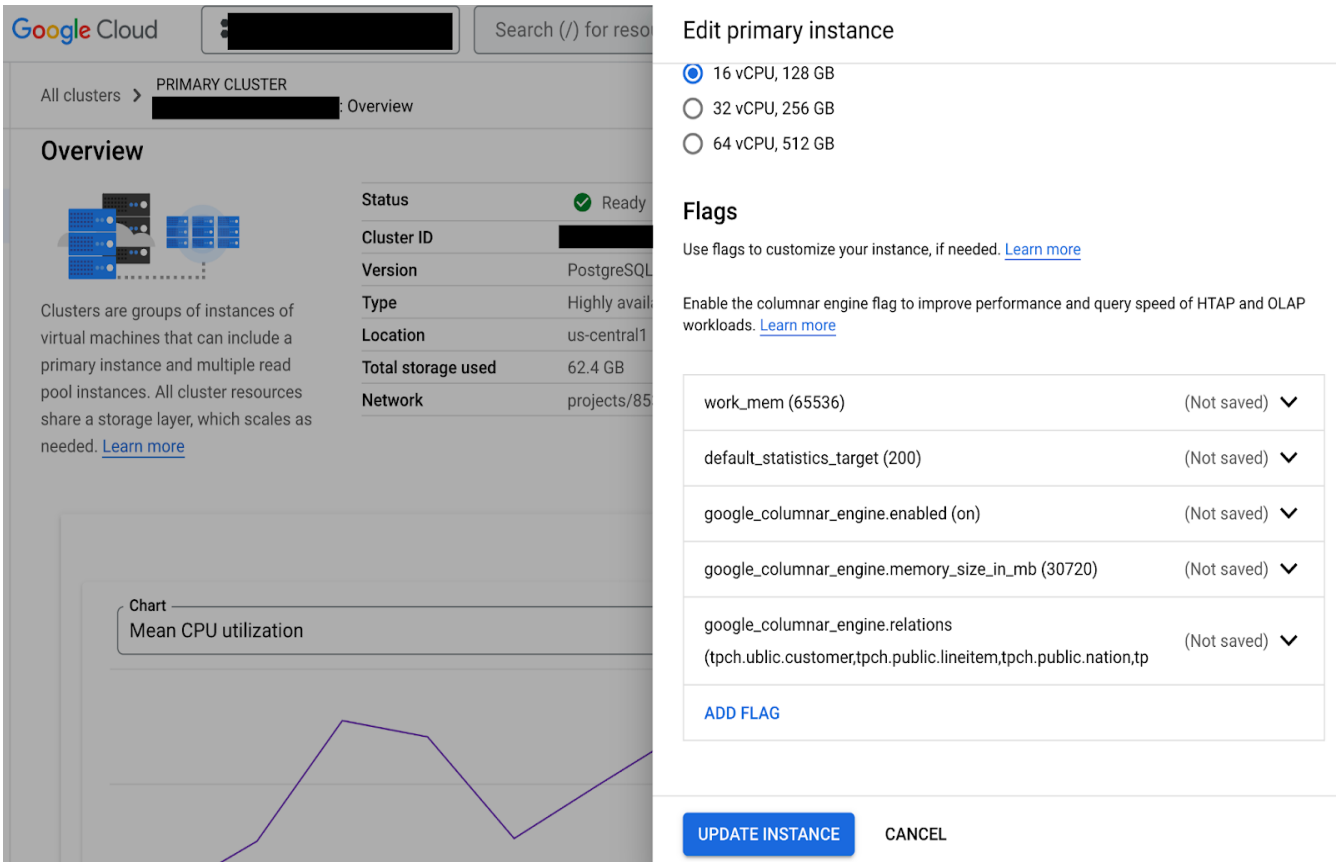
Below are the simple tuning steps:

1. Open https://console.cloud.google.com and go to the AlloyDB Primary Cluster → AlloyDB Primary Instance page.

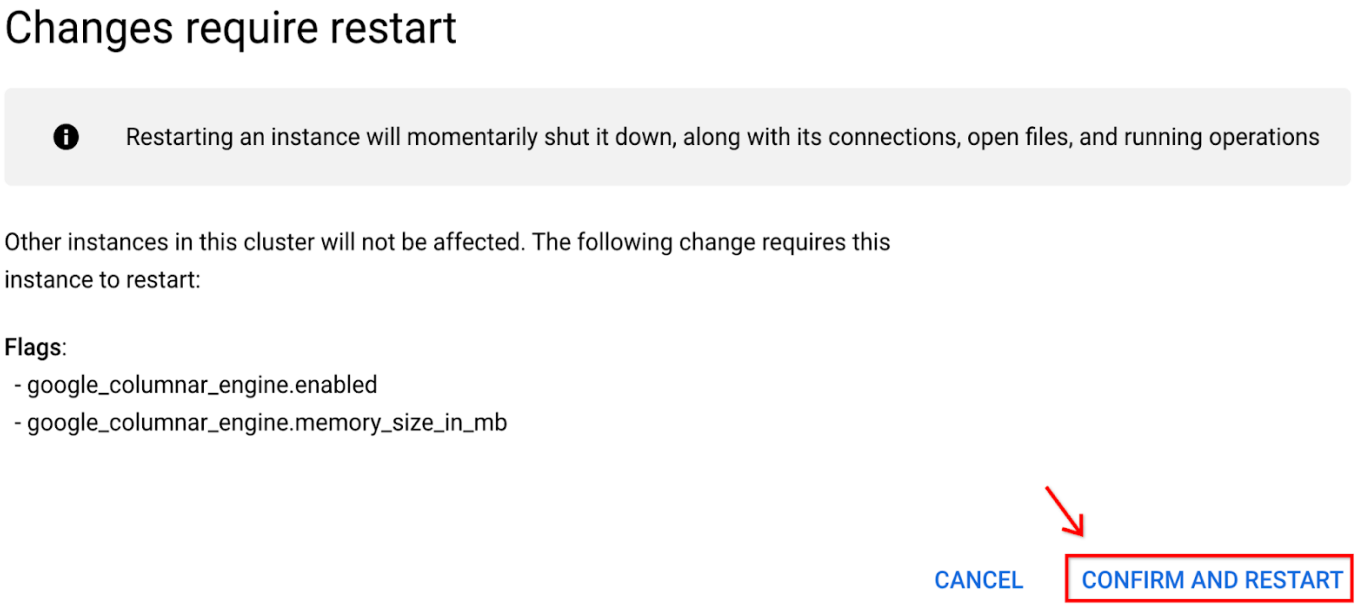
2. Edit the AlloyDB primary instance and add or update the following `Flags from the UI`:

    - work_mem = 65536
    - default_statistics_target = 200
    - google_columnar_engine.enabled = ON
    - google_columnar_engine.memory_size_in_mb = 30720
    - google_columnar_engine.relations = tpch.public.customer,tpch.public.lineitem,tpch.public.nation,tpch.public.orders,tpch.public.part,tpch.public.partsupp,tpch.public.region,tpch.public.supplier

    Below is a screenshot for your reference:




3. Click on the `UPDATE INSTANCE` button and then you should see the following screen. Since a few settings would require the instance to restart, you must allow it to restart by clicking the `CONFIRM AND RESTART` button (as shown in the image below).



4. Wait for the restart operation to finish. It will take a few minutes to complete since the AlloyDB instance needs to be restarted.

5. Monitor the population of columnar-engine as follows:

   a. Confirm that `google_columnar_engine.enabled` is set to `on`. Use the command `psql -h $PGHOST -U postgres -c "SHOW google_columnar_engine.enabled"` for this purpose.

   b. Check the status of columnar engine population within the `tpch` database by using the following command.

```
psql -U postgres -d tpch
...
tpch=> select * from g_columnar_relations; \watch 10
```

   Note `\watch 10` at the end of the SQL command which executes the command every 10 seconds. You should observe the output of this command until it no longer changes. Once the output stops changing, specifically, check block_count_in_cc=total_block_count for every relation, go to the next step for validation of the output. Also as a general rule of thumb, ensure that the `total_block_count` matches `block_count_in_cc` for all the relations.

   c. Validate the status of columnar engine population as follows:

   **Validation State for Scale Factor = 10**

   Below is the final state of the columnar engine after population was done for scale factor 10:

```
psql -U postgres -d tpch

tpch=> select * from g_columnar_relations

 database_name | schema_name | relation_name | status |    size    |
uncompressed_size | columnar_unit_count | invalid_block_count | block_count_in_cc |
total_block_count | auto_refresh_trigger_count | auto_refresh_failure_count |
auto_refresh_recent_status
---------------+-------------+---------------+--------+------------+---------------
----+---------------------+---------------------+-------------------+--------------
-----+----------------------------+----------------------------+------------------
---------
 tpch          | public      | supplier      | Usable |   17275132 |
17275132 |                   1 |                   0 |              2268 |
2268 |                     0 |                         0 | NONE YET
 tpch          | public      | part          | Usable |  140325324 |
140325324 |                  11 |                   0 |             41942 |
```

```
41942 |                          0 |                          0 | NONE YET
 tpch          | public       | region        | Usable |       3642 |
3642 |                    1 |                    0 |                    1 |
1 |                          0 |                          0 | NONE YET
 tpch          | public       | nation        | Usable |       6048 |
6048 |                    1 |                    0 |                    1 |
1 |                          0 |                          0 | NONE YET
 tpch          | public       | orders        | Usable | 1509117771 |
1509117771 |                   69 |                    0 |             278710 |
278710 |                          0 |                          0 | NONE YET
 tpch          | public       | lineitem      | Usable | 4831946995 |
4831946995 |                  325 |                    0 |            1330899 |
1330899 |                          0 |                          0 | NONE YET
 tpch          | public       | customer      | Usable |  276859377 |
276859377 |                    9 |                    0 |              36658 |
36658 |                          0 |                          0 | NONE YET
 tpch          | public       | partsupp      | Usable | 1258041408 |
1258041408 |                   45 |                    0 |             183648 |
183648 |                          0 |                          0 | NONE YET
(8 rows)

Alternatively, you can use

tpch =>select relation_name, block_count_in_cc, total_block_count,
block_count_in_cc=total_block_count from g_columnar_relations order by 1;
```

**Validation state for Scale Factor = 30**

Execute the command `psql -U postgres -d tpch -c "select * from g_columnar_relations"` and verify that the output is columnar-engine population is close to the following numbers:

```
tpch=> database_name | schema_name | relation_name | status |    size     |
uncompressed_size | columnar_unit_count | invalid_block_count | block_count_in_cc
| total_block_count | auto_refresh_trigger_count | auto_refresh_failure_count |
auto_refresh_recent_status
---------------+-------------+---------------+--------+-------------+-------------
------+--------------------+--------------------+------------------+-----------
--------+----------------------------+----------------------------+--------------
-------------
 tpch          | public       | orders        | Usable |  4721850499 |
4721850499 |                  206 |                    0 |             840965 |
840965 |                          0 |                          0 | NONE YET
 tpch          | public       | partsupp      | Usable |  3802494436 |
3802494436 |                  135 |                    0 |             551724 |
551724 |                          0 |                          0 | NONE YET
 tpch          | public       | customer      | Usable |   830579812 |
```

```
830579812 |                  27 |                   0 |            109974 |
109974 |                       0 |                         0 | NONE YET
 tpch          | public      | part          | Usable |   423694950 |
423694950 |                  31 |                   0 |            125855 |
125855 |                       0 |                         0 | NONE YET
 tpch          | public      | supplier      | Usable |    52037086 |
52037086 |                   2 |                     0 |              6807 |
6807 |                          0 |                         0 | NONE YET
 tpch          | public      | nation        | Usable |        6062 |
6062 |                       1 |                     0 |                 1 |
1 |                            0 |                         0 | NONE YET
 tpch          | public      | region        | Usable |        3551 |
3551 |                       1 |                     0 |                 1 |
1 |                            0 |                         0 | NONE YET
 tpch          | public      | lineitem      | Usable | 15045734637 |
15045734637 |                 983 |                     0 |           4025408 |
4025408 |                       0 |                         0 | NONE YET
(8 rows)
```

## Tuning for Scale Factor 100

The size of the `tpch` database that we load with `TPCH_SCALE=100` is approximately 205GB. This database size is substantially larger than the size of available RAM on the machine (128 GB) of type 16 virtual CPUs. We cannot therefore populate the columnar engine for the entire database. This is where AlloyDB Columnar Engine's `auto columnarization` comes into action. Now that we must let CE observe the workload first, the tuning steps here differ slightly. After we enable the `Columnar Engine`, we need to execute the entire set of TPC-H queries once. That enables the recommendation engine to make suggestions on the optimal values to set for `google_columnar_engine.relations` and `google_columnar_engine.memory_size_in_mb` database flags.

Below are the simple tuning steps:

1. Open https://console.cloud.google.com and go to the AlloyDB Primary Cluster -> AlloyDB Primary Instance page.

2. Edit the AlloyDB primary instance and add or update the following `Flags` (refer Important Flags to Tune to learn more about these flags):

    - work_mem = 65536
    - default_statistics_target = 200
    - google_columnar_engine.enabled = ON
    - google_columnar_engine.memory_size_in_mb = 40960

3. Click on the `UPDATE INSTANCE` button and allow the AlloyDB instance to restart by clicking the `CONFIRM AND RESTART` button.

4. Wait for the AlloyDB instance to finish the update and restart operation. It will take a few minutes to complete since the AlloyDB instance needs to be restarted.

5. Confirm that `google_columnar_engine.enabled` is set to `on`. Use following command to confirm this:

```
psql -h $PGHOST -U postgres
postgres=> SHOW google_columnar_engine.enabled;
```

6. Reset the columnar engine recommendation by using the following command:

```
psql -h $PGHOST -U postgres -d tpch -c "SELECT
google_columnar_engine_reset_recommendation('true')";
```

7. **`Observe workload`**: In this step, you simply execute all of the 22 TPC-H queries (just once) that will let Columnar Engine observe the workload to make optimal tuning suggestions. You can create and execute the following script to train the engine (execute it from `hammerdb/HammerDB-4.6` directory):

   Create and execute train-recommendation-engine.sh script

```
#!/bin/bash -x

source ./setup.env

./hammerdbcli << EOF

# CONFIGURE PARAMETERS FOR TPC-H BENCHMARK
# -------------------------------------
dbset db pg
dbset bm tpc-h

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

# CONFIGURE TPC-H
# -------------------------------------
diset tpch pg_tpch_superuser postgres
diset tpch pg_tpch_superuserpass $PGPASSWORD
diset tpch pg_tpch_user postgres
diset tpch pg_tpch_pass $PGPASSWORD
diset tpch pg_tpch_dbase tpch
```

```
diset tpch pg_scale_fact $TPCH_SCALE
diset tpch pg_num_tpch_threads 1

diset tpch pg_degree_of_parallel 8
vuset vu 1

# logging
vuset logtotemp 1
vuset timestamps 0
vuset unique 0

# load tpc-h script and run benchmark
loadscript
vurun

# terminate when completed
waittocomplete
vudestroy
quit

EOF
```

8. **Optimal tuning suggestion**: Once all the queries from previous step finish to execute, run the following command to find the optimal columnar engine tuning for `tpch` database:

```
psql -h $PGHOST -U postgres -d tpch  -c "SELECT
google_columnar_engine_recommend('RECOMMEND_SIZE')"
```

   a. This command uses the recommendation engine to recommend the performance optimal memory size and recommended column.

   b. **Output** looks like following:

```
(39454,"tpch.public.customer(c_acctbal,c_address,c_comment,c_custkey,c_mktsegment,c
_name,c_nationkey,c_phone),tpch.public.lineitem(l_commitdate,l_discount,l_extendedp
rice,l_linestatus,l_orderkey,l_partkey,l_quantity,l_receiptdate,l_returnflag,l_ship
date,l_shipinstruct,l_shipmode,l_suppkey,l_tax),tpch.public.orders(o_custkey,o_orde
rdate,o_orderkey,o_orderpriority),tpch.public.part(p_brand,p_container,p_name,p_par
tkey,p_size,p_type),tpch.public.partsupp(ps_partkey,ps_suppkey,ps_supplycost),tpch.
public.supplier(s_address,s_comment,s_name,s_nationkey,s_suppkey)")
```

c. Note the 2 parts of the above output:
   i. **First Part**: It is an integer (in this case, `39454`). This is the recommended value for the `google_columnar_engine.memory_size_in_mb` parameter. However, we can safely disregard this parameter since it requires a restart of AlloyDB and the difference between the new suggested value and the original value we specified (`40960`) is not significant.

   ii. **Second Part**: A string containing a list of recommended relations and their important columns `"tpch.public.customer(c_acctbal,c_address,c_comment,c_custkey,c_mktsegment,c_name,c_nationkey,c_phone),tpch.public.lineitem(l_commitdate,l_discount,l_extendedprice,l_linestatus,l_orderkey,l_partkey,l_quantity,l_receiptdate,l_returnflag,l_shipdate,l_shipinstruct,l_shipmode,l_suppkey,l_tax),tpch.public.orders(o_custkey,o_orderdate,o_orderkey,o_orderpriority),tpch.public.part(p_brand,p_container,p_name,p_partkey,p_size,p_type),tpch.public.partsupp(ps_partkey,ps_suppkey,ps_supplycost),tpch.public.supplier(s_address,s_comment,s_name,s_nationkey,s_suppkey)"`.

9. Now go back to the AlloyDB Primary Instance page URL on https://console.cloud.google.com, edit the instance and add the following Flag:

   a. Set `google_columnar_engine.relations = "<Second Part>"`. For example, with the above output, you would set `google_columnar_engine.relations = "tpch.public.customer(c_acctbal,c_address,c_comment,c_custkey,c_mktsegment,c_name,c_nationkey,c_phone),tpch.public.lineitem(l_commitdate,l_discount,l_extendedprice,l_linestatus,l_orderkey,l_partkey,l_quantity,l_receiptdate,l_returnflag,l_shipdate,l_shipinstruct,l_shipmode,l_suppkey,l_tax),tpch.public.orders(o_custkey,o_orderdate,o_orderkey,o_orderpriority),tpch.public.part(p_brand,p_container,p_name,p_partkey,p_size,p_type),tpch.public.partsupp(ps_partkey,ps_suppkey,ps_supplycost),tpch.public.supplier(s_address,s_comment,s_name,s_nationkey,s_suppkey)"`

10. Click on the `UPDATE INSTANCE` button and allow the AlloyDB instance to restart (if needed) by clicking the `CONFIRM AND RESTART` button. The AlloyDB database will restart to pick up the new flag settings.

11. Run the query `SELECT * from g_columnar_relations` regularly, and wait until values do not change any further. Use the following SQL command with `\watch 10` switch to allow the query to execute in every 10 seconds.

```
psql -h 172.20.0.209 -U postgres -d tpch
...
tpch=> SELECT * FROM g_columnar_relations; \watch 10
```

Observe the output of this command until it no longer changes. Once the output stops changing, ensure that the `total_block_count` matches `block_count_in_cc` for all the relations. The final state of `g_columnar_relations` should be close to the following:

```
 database_name | schema_name | relation_name | status |    size     | uncompressed_size |
columnar_unit_count | invalid_block_count | block_count_in_cc | total_block_count |
auto_refresh_trigger_count | auto_refresh_failure_count | auto_refresh_recent_status
---------------+-------------+---------------+--------+-------------+-------------------+
--------------------+---------------------+-------------------+-------------------+-----
----------------------+----------------------------+---------------------------
 tpch          | public      | orders        | Usable |  4064416519 |        4064416519 |
688 |                   0 |           2816904 |           2816904 |
0 |                         0 | NONE YET
 tpch          | public      | part          | Usable |  1140509489 |        1140509489 |
103 |                   0 |            419463 |            419463 |
0 |                         0 | NONE YET
 tpch          | public      | partsupp      | Usable |  1651209445 |        1651209445 |
450 |                   0 |           1839485 |           1839485 |
0 |                         0 | NONE YET
 tpch          | public      | supplier      | Usable |   144812152 |         144812152 |
6 |                    0 |             22687 |             22687 |
0 |                         0 | NONE YET
 tpch          | public      | customer      | Usable |  2753610591 |        2753610591 |
90 |                   0 |            366510 |            366510 |
0 |                         0 | NONE YET
 tpch          | public      | lineitem      | Usable | 31066817949 |       31066817949 |
3296 |                  0 |          13496579 |          13496579 |
0 |                         0 | NONE YET
(6 rows)
```

## *Running the TPC-H benchmark*

In this stage, we perform the TPC-H benchmark's "**Power Test**" with one client running 22 TPC-H queries, monitoring each query's response time, and calculating a final "**Geometric mean of query times returning rows.**"

### What is Power Test in TPROC-H?

The TPROC-H **"Power Test"** in HammerDB is a performance test that measures the ability of a database system to handle large-scale data warehousing workloads. HammerDB utilizes a modified version of TPC-H "power test" that does not have refresh functions. In this test, a single client generates a series of 22 queries that simulate typical data warehousing operations, such as generating reports, analyzing data, and performing complex joins. The test is based on the TPC-H benchmark, which is a standard benchmark used

to evaluate the performance of database systems for data warehousing applications. The goal of the TPROC-H Power test is to measure the minimum query latency (or response time) that can be achieved by a single client, which provides an indication of the overall performance and scalability of the database system under test.

## What is “Geometric Mean” Metric?

The geometric mean is a measure of central tendency that is used in the TPROC-H benchmark. It is calculated by taking the product of all of the query times and then taking the n-th root of the product, where *n* is the number of queries. The geometric mean is used in the TPROC-H benchmark because it is less sensitive to outliers than the arithmetic mean. The arithmetic mean is the average of all of the query times. However, if there is one query that takes a very long time, the arithmetic mean will be skewed by that query. The geometric mean, on the other hand, is not as sensitive to such outliers. Even if one query takes a very long time, the product of all of the query times will not be as affected by that query. Refer to TPC-H official documentation to learn more about this metric.

A lower geometric mean of query times returning rows is desirable, as it indicates that the database system can process queries more quickly and efficiently and can handle larger data volumes more effectively.

Use the following script to execute the "**Power Test**" benchmark for TPROC-H. This script repeats the series of 22 queries. The first set of the query executions is intended to warm up the database caches, while the second set is used for actual performance measurement.

1. Switch to benchmark home directory:

```
cd hammerdb/HammerDB-4.6

source ./setup.env
```

2. Create **run-tpch.sh** script as follows:

```
#!/bin/bash -x

source ./setup.env

./hammerdbcli << EOF

# CONFIGURE PARAMETERS FOR TPC-H BENCHMARK
# ---------------------------------------
dbset db pg
dbset bm tpc-h

# CONFIGURE POSTGRES HOST AND PORT
# ---------------------------------------
```

```
diset connection pg_host $PGHOST
diset connection pg_port $PGPORT

# CONFIGURE TPC-H
# ---------------------------------------
diset tpch pg_tpch_superuser postgres
diset tpch pg_tpch_superuserpass $PGPASSWORD
diset tpch pg_tpch_user postgres
diset tpch pg_tpch_pass $PGPASSWORD
diset tpch pg_tpch_dbase tpch
diset tpch pg_scale_fact $TPCH_SCALE
diset tpch pg_num_tpch_threads 1
diset tpch pg_refresh_on false
diset tpch pg_refresh_verbose false
diset tpch pg_degree_of_parallel 8
diset tpch pg_trickle_refresh 1000
diset tpch pg_tpch_tspace pg_default
diset tpch pg_tpch_gpcompat false
diset tpch pg_tpch_gpcompress false
diset tpch pg_cloud_query false
diset tpch pg_rs_compat false
diset tpch pg_update_sets 1
diset tpch pg_total_querysets 1

vuset vu 1

# logging
vuset logtotemp 1
vuset timestamps 0
vuset unique 0

# load tpc-h script and run benchmark
loadscript
# Warmup run
vurun

# Measurement run
vurun

# terminate when completed
waittocomplete
vudestroy
quit

EOF
```

3. Run the script as follows:

```
chmod +x run-tpch.sh
mkdir -p results
sudo nohup ./run-tpch.sh > results/run-tpch.out 2>&1
```

4. Below is a sample output of run-tpch.sh script obtained for scenario where Columnar-Engine (CE) is enabled and TPC-H `scale factor` is set to `30`:

```
TPROC-H Driver Script
Script loaded, Type "print script" to view


Vuser 1 created - WAIT IDLE
Failed to create virtual users: Could not open tempfile /tmp/hammerdb.log
Vuser 1:RUNNING
Vuser 1:Executing Query 14 (1 of 22)
Vuser 1:query 14 completed in 9.569 seconds
Vuser 1:Executing Query 2 (2 of 22)
Vuser 1:query 2 completed in 18.363 seconds
Vuser 1:Executing Query 9 (3 of 22)
...
...
Vuser 1:query 12 completed in 4.194 seconds
Vuser 1:Completed 1 query set(s) in 314 seconds
Vuser 1:Geometric mean of query times returning rows (22) is 7.37605
Vuser 1:FINISHED SUCCESS
ALL VIRTUAL USERS COMPLETE
TPROC-H Driver Script
jobid=642777185F8303E203936333

Vuser 1:RUNNING
Vuser 1:Executing Query 14 (1 of 22)
Vuser 1:query 14 completed in 3.086 seconds
Vuser 1:Executing Query 2 (2 of 22)
Vuser 1:query 2 completed in 14.241 seconds
Vuser 1:Executing Query 9 (3 of 22)
...
...
Vuser 1:query 12 completed in 4.271 seconds
Vuser 1:Completed 1 query set(s) in 264 seconds
Vuser 1:Geometric mean of query times returning rows (22) is 6.05468
Vuser 1:FINISHED SUCCESS
ALL VIRTUAL USERS COMPLETE
TPROC-H Driver Script
jobid=642778545F8303E273233383
```

**NOTE:** As stated previously, we only evaluate the second round of query execution when measuring performance. The initial round serves as a warm-up.

## *Expected TPC-H Results*

The table below summarizes the execution time (in seconds) for each of the 22 TPC-H queries. As stated previously, three distinct scenarios with scale factors of 10, 30, and 100 have been explored. In each scenario, the query execution durations and geometric mean for all queries with Columnar-Engine (CE) population are presented. You should anticipate TPC-H (power test) performance results similar to the following:

| Query Id | Query Execution Time (in seconds) | | |
|---|---|---|---|
| | TPCH_SCALE **= 10** | TPCH_SCALE **= 30** | TPCH_SCALE **= 100** |
| 1 | 4.00 | 12.36 | 55.68 |
| 2 | 4.78 | 12.84 | 68.01 |
| 3 | 1.91 | 5.15 | 92.17 |
| 4 | 0.44 | 1.24 | 47.88 |
| 5 | 0.86 | 3.68 | 8.11 |
| 6 | 0.05 | 0.12 | 0.44 |
| 7 | 0.87 | 2.75 | 30.45 |
| 8 | 0.59 | 1.92 | 237.98 |
| 9 | 4.45 | 15.67 | 733.11 |
| 10 | 1.81 | 5.41 | 22.21 |
| 11 | 0.83 | 2.46 | 75.19 |
| 12 | 0.55 | 1.64 | 6.09 |
| 13 | 4.59 | 12.75 | 78.18 |
| 14 | 0.38 | 0.98 | 3.19 |
| 15 | 2.71 | 5.52 | 26.14 |
| 16 | 1.41 | 3.63 | 15.36 |
| 17 | 5.96 | 21.42 | 140.02 |
| 18 | 18.84 | 56.27 | 389.47 |
| 19 | 0.08 | 0.23 | 0.80 |
| 20 | 3.76 | 12.66 | 3,273.24 |
| 21 | 1.64 | 6.25 | 494.35 |
| 22 | 0.24 | 0.31 | 1.08 |

| | | | |
|---|---|---|---|
| **Geometric mean (seconds)** | **1.23** | **3.56** | **38.00** |

# OLAP Atomics Benchmarking

To evaluate and improve the OLAP capabilities of AlloyDB's Columnar-Engine, the engineering at Google has developed a custom benchmark known as **OLAP atomics**, which consists of a collection of 11 primitive queries executed over a large volume of data and covering the fundamental operations of an OLAP system. This set of primitive OLAP queries can perform the fundamental data manipulation and analysis of any typical OLAP system, including selection, slice-and-dice, joins, roll-up (also known as aggregation or consolidation), drill-down, etc.

Measuring OLAP atomics on a database system is important because it can reveal performance bottlenecks at the primitive level. These primitive queries are a valuable tool for ensuring that the OLAP system fulfills the requirements for your large-scale data analysis and decision support.

For the purposes of this user guide, a TPC-H database with a **scale factor of 30** was utilized.

## *Setup, Configuration and Tuning*

Before you can execute the OLAP atomic queries, you must perform the database configuration and tuning described in this section.

### 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

Connect to the client machine and execute the following commands:

```
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

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

# TPC-H Scale Factor (determines the size of the database that we want to build).
export TPCH_SCALE=30

EOF
```

Edit the above file and all the settings (excluding `TPCH_SCALE`, that should remain as 30) to suit your environment.

Now, to load the TPC-H database, follow the exact steps outlined in the "Script to load TPC-H data" section. The load steps are identical to the TPC-H benchmarking.

## Altering the TPC-H schema

For the purpose of OLAP atomics, we only need the `lineitem` and `supplier` tables from `tpch` database (i.e. without any constraints or indices). In this section, we provide minimal instructions to prepare the database for query execution.

1. Connect to the client machine.
2. Connect to the `tpch` database by using `psql -h $PGHOST -U postgres -d tpch` command.
3. Now run the following commands to drop all the `constraints` and `indices` from `lineitem` and `supplier` tables:

```
--- Drop constraints from lineitem table:
ALTER TABLE lineitem DROP CONSTRAINT IF EXISTS lineitem_pk CASCADE;
ALTER TABLE lineitem DROP CONSTRAINT IF EXISTS lineitem_partsupp_fk CASCADE;
ALTER TABLE lineitem DROP CONSTRAINT IF EXISTS lineitem_order_fk CASCADE;

--- Drop all indexes of lineitem table:
DROP INDEX IF EXISTS lineitem_part_supp_fkidx CASCADE;
```

```
DROP INDEX IF EXISTS idx_lineitem_orderkey_fkidx CASCADE;
DROP INDEX IF EXISTS lineitem_pk CASCADE;

--- Drop constraints from supplier table:

ALTER TABLE supplier DROP CONSTRAINT IF EXISTS supplier_pk CASCADE;
ALTER TABLE supplier DROP CONSTRAINT IF EXISTS supplier_nation_fk CASCADE;
ALTER TABLE supplier DROP CONSTRAINT IF EXISTS "2200_127555_1_not_null" CASCADE;

--- Drop all indexes of supplier table:
DROP INDEX IF EXISTS supplier_nation_fkidx CASCADE;

--- Drop all the tables that are not needed:
DROP TABLE customer CASCADE;
DROP TABLE nation CASCADE;
DROP TABLE orders CASCADE;
DROP TABLE part CASCADE;
DROP TABLE partsupp CASCADE;
DROP TABLE region CASCADE;
```

4. Verify that you only see the following objects in the `tpch` database after executing the preceding commands:

```
tpch=> \dti+
                                 List of relations
 Schema |   Name   | Type  |  Owner   | Table | Persistence | Size  | Description
--------+----------+-------+----------+-------+-------------+-------+-------------
 public | lineitem | table | postgres |       | permanent   | 31 GB |
 public | supplier | table | postgres |       | permanent   | 53 MB |
(2 rows)
```

## Tuning Columnar Engine

Here are the recommended procedures for tuning the AlloyDB columnar engine:

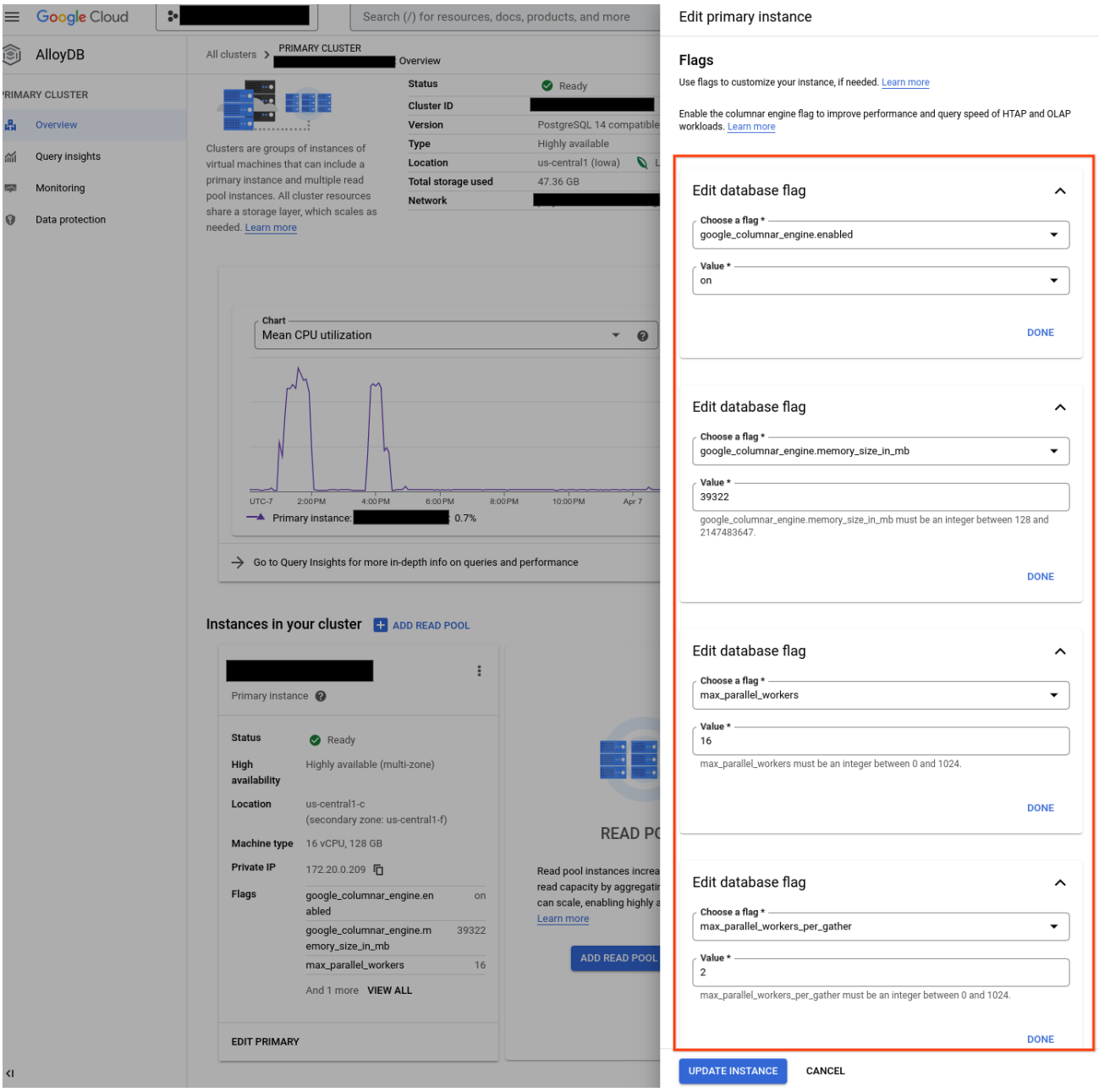
1. Open https://console.cloud.google.com and go to the AlloyDB Primary Cluster -> AlloyDB Primary Instance page.

2. Edit the AlloyDB primary instance and add the following flags (remove any other flags if you see them):

    - google_columnar_engine.enabled = ON
    - google_columnar_engine.memory_size_in_mb = 39322

- max_parallel_workers_per_gather = 2
- max_parallel_workers = 16

Below is a screenshot for your reference:




3. Click on the `UPDATE INSTANCE` button and allow the AlloyDB instance to restart by clicking the `CONFIRM AND RESTART` button.

4. Wait for the AlloyDB instance to finish the update and restart operation. It will take a few minutes to complete since the AlloyDB instance needs to be restarted.

5. Connect to the client machine and confirm that `google_columnar_engine.enabled` is set to `on`. Use following command to confirm this:

```
psql -h $PGHOST -U postgres
postgres=> SHOW google_columnar_engine.enabled;
```

6. Connect to the `tpch` database by using `psql -h $PGHOST -U postgres -d tpch` command and then run the following commands to add `lineitem` and `supplier` tables to the columnar-engine.

```
SELECT google_columnar_engine_add('lineitem');
SELECT google_columnar_engine_add('supplier');
```

7. Validation of columnar-engine population: Use the command `psql -h $PGHOST -U postgres -d tpch -c "select * from g_columnar_relations"` and ensure that the output is similar to following:

```
 database_name | schema_name | relation_name | status |    size     | uncompressed_size |
columnar_unit_count | invalid_block_count | block_count_in_cc | total_block_count |
auto_refresh_trigger_count | auto_refresh_failure_count | auto_refresh_recent_status
---------------+-------------+---------------+--------+-------------+-------------------+-
--------------------+---------------------+-------------------+-------------------+-------
---------------------+----------------------------+---------------------------
 tpch          | public      | supplier      | Usable |    52051365 |          52051365 |
2 |                   0 |              6807 |              6807 |
0 |                     0 | NONE YET
 tpch          | public      | lineitem      | Usable | 15042480452 |       15042480452 |
983 |                   0 |           4025186 |           4025186 |
0 |                     0 | NONE YET
(2 rows)
```

Now we are ready to execute the OLAP atomics benchmark.

## *Queries in OLAP Atomics*

The following table summarizes the customized OLAP queries that are executed on the `tpch` database that we just loaded. The engineering team at Google AlloyDB develops these queries.

| Scenario Description | Query | Query Id |
|---|---|---|

| | | |
|---|---|---|
| Aggregation (count operation) with a filter covering approximately 10% of the large lineitem table. | `select count(l_orderkey) from lineitem where l_discount = 0;` | Q1 |
| Aggregation (SUM) on an integer column with a filter covering approximately 10% of the lineitem table. | `select sum(l_linenumber) from lineitem where l_discount = 0;` | Q2 |
| Aggregation (SUM) on numeric column with a filter covering approximately 10% of the lineitem table. | `select sum(l_quantity) from lineitem where l_discount = 0;` | Q3 |
| Summarization using GROUP BY and AGGREGATION on the entire lineitem table. | `select count(l_shipmode), l_shipmode from lineitem group by l_shipmode;` | Q4 |
| Full table scan without any filters | `select count(l_comment) from lineitem;` | Q5 |
| Full table scan with equality predicate (filter) | `select count(*) from lineitem where l_quantity=25.99;` | Q6 |
| Sorting of the entire table and presenting the top values | `select l_orderkey, l_commitdate, l_shipmode from lineitem order by 1,2,3 limit 10;` | Q7 |
| Full table scan with LIKE predicate | `select count(*) from lineitem where l_shipinstruct like '%DE%';` | Q8 |
| LIST based selection on the entire table | `select count(*) from lineitem where l_tax in (0.01, 0.02, 0.05);` | Q9 |
| MIX and MAX aggregation on the entire table | `select min(l_quantity), max(l_discount) from lineitem;` | Q10 |
| Join with a predicate | `select count(*) from supplier, lineitem where s_acctbal = l_extendedprice;` | Q11 |

## *Execute OLAP Atomics*

The execution of OLAP atomic queries is as simple as connecting to the `tpch` database and executing the queries introduced in section Queries in OLAP Atomics.

It is **recommended** to execute the queries using **"EXPLAIN ANALYZE** <query> **.."** prefix clause, which will display the query plan and execution time.

Below is an example of executing Q1 from `tpch` database:

```
tpch=> explain analyze select count(l_orderkey) from lineitem where l_discount = 0;

QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=137113.77..137113.78 rows=1 width=8) (actual time=151.708..153.953 rows=1 loops=1)
   ->  Gather  (cost=137113.56..137113.77 rows=2 width=8) (actual time=151.693..153.944 rows=3 loops=1)
```

```
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=136113.56..136113.57 rows=1 width=8) (actual
time=145.574..145.576 rows=1 loops=3)
               ->  Parallel Append  (cost=20.00..119102.90 rows=6804263 width=7) (actual
time=0.063..145.569 rows=5454623 loops=3)
                     ->  Parallel Custom Scan (columnar scan) on lineitem
(cost=20.00..119098.89 rows=6804262 width=7) (actual time=0.062..145.565 rows=5454623 loops=3)
                           Filter: (l_discount = '0'::numeric)
                           Rows Removed by Columnar Filter: 54546385
                           Rows Aggregated by Columnar Scan: 1904168
                           Columnar cache search mode: native
                     ->  Parallel Seq Scan on lineitem  (cost=0.00..4.01 rows=1 width=7) (never
executed)
                           Filter: (l_discount = '0'::numeric)

 Planning Time: 2.283 ms
 Execution Time: 154.008 ms
(15 rows)
```

You should note the `Execution Time` in the above output, which is significantly faster for `AlloyDB` columnar-engine.

## *Expected Results*

The following table gives a summary of the queries to execute and their expected execution and planning time.

| Query Id | Query To Execute | Execution Time (milliseconds) | Planning Time (ms) |
|---|---|---|---|
| Q1 | `EXPLAIN ANALYZE SELECT COUNT(l_orderkey) FROM lineitem WHERE l_discount = 0;` | 154.00 | 2.28 |
| Q2 | `EXPLAIN ANALYZE SELECT SUM(l_linenumber) FROM lineitem WHERE l_discount = 0;` | 278.00 | 2.25 |
| Q3 | `EXPLAIN ANALYZE SELECT SUM(l_quantity) FROM lineitem WHERE l_discount = 0;` | 278.00 | 2.29 |
| Q4 | `EXPLAIN ANALYZE SELECT COUNT(l_shipmode), l_shipmode FROM lineitem GROUP BY l_shipmode;` | 997.00 | 2.20 |
| Q5 | `EXPLAIN ANALYZE SELECT COUNT(l_comment) FROM lineitem;` | 206.00 | 1.90 |
| Q6 | `EXPLAIN ANALYZE SELECT COUNT(*) FROM lineitem WHERE l_quantity=25.99;` | 1.53 | 2.24 |
| Q7 | `EXPLAIN ANALYZE SELECT l_orderkey, l_commitdate, l_shipmode` | 1,909.00 | 1.80 |
| Q8 | `EXPLAIN ANALYZE SELECT COUNT(*) FROM lineitem WHERE l_shipinstruct like '%DE%';` | 276.00 | 2.33 |

| | | | |
|---|---|---|---|
| | `FROM lineitem ORDER BY 1,2,3 LIMIT 10;` | | |
| Q9 | `EXPLAIN ANALYZE SELECT COUNT(*) FROM lineitem WHERE l_tax in (0.01, 0.02, 0.05);` | 352.00 | 2.30 |
| Q10 | `EXPLAIN ANALYZE SELECT MIN(l_quantity), MAX(l_disCOUNT) FROM lineitem;` | 364.00 | 4.40 |
| Q11 | `EXPLAIN ANALYZE SELECT COUNT(*) FROM supplier, lineitem WHERE s_acctbal = l_extendedprice;` | 5,734.00 | 2.09 |

**Authors**

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