AlloyDB Omni Performance Management Guide

Table of contents

Did you find this document helpful? Please send us your feedback.

Introduction

This document outlines the best practices for designing effective performance tests, collecting data, and analyzing performance metrics to tune AlloyDB Omni to efficiently handle your application's requirements.

The goal of performance testing and benchmarking is to assess the current performance of an application or workload and to identify changes to the application, database, or hardware to increase throughput, reduce latency, or both.

- **Throughput**
  Throughput is the amount of work the database can process in a given amount of time. Throughput is usually expressed as transactions per second (tps) or transactions per minute (tpm).
- **Latency**
  Latency, or response time, is the amount of time it takes to process a single request, usually measured in milliseconds, seconds, or sometimes larger time amounts for complex reporting and analytics use cases.

Tuning the performance of database applications requires an understanding of how databases respond to queries. Having a high level picture of how the database system processes a request helps you to understand the different metrics that characterize performance and what actions to take to reach your performance goals.

Database engine architecture

A database engine translates a client's query into an executable plan, finds the data necessary to satisfy the query, performs any necessary filtering, ordering, and aggregation, and returns the results to the client.
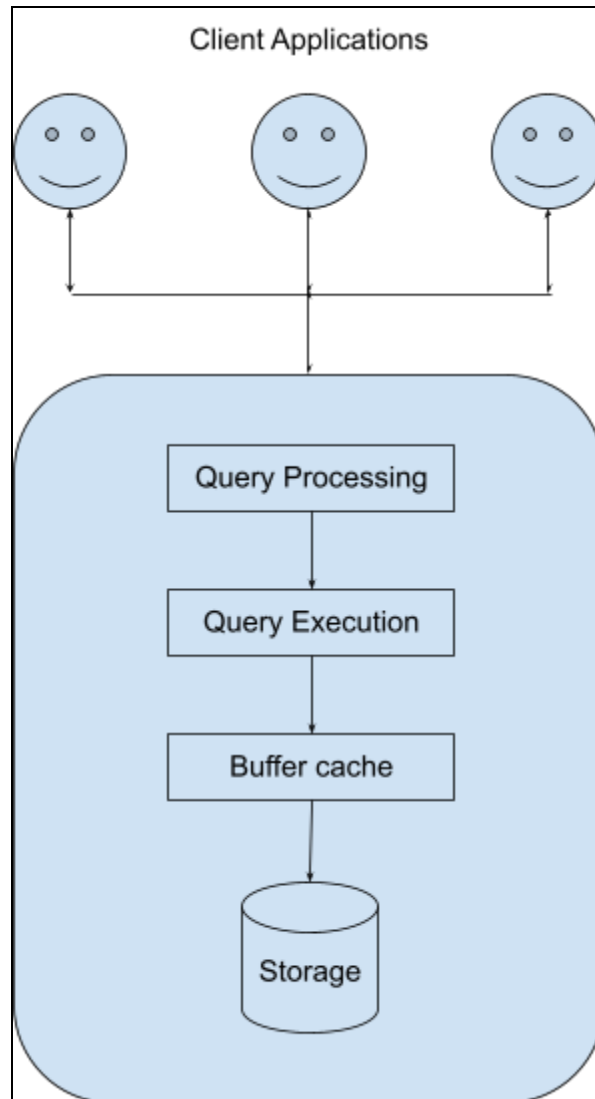
Figure 1 shows the database layers that work together to respond to a client's query.
- First, the query processing layer parses the query and turns it into an execution plan.
- The execution plan is fed to the query execution layer, which performs the operations needed to compute the response to the query.
- During execution, data could be loaded from the buffer cache (if data is present in the cache), or loaded directly from storage (otherwise). If it is loaded from storage, the data is stored in the cache, for future uses.

Resources used when processing the client's query include CPU, memory, I/O, network, and synchronization primitives like database locks. Performance tuning aims to optimize resource utilization during each of the steps in query execution.

The goal of a performant database engine is to respond to a query using the fewest resources required. This goal starts with a good data model and query design. How can queries be answered while looking at the least amount of data? What indexes are needed to reduce the search space and I/O? Sorting data requires CPU and, often, disk access for large data sets, so how can sorting data be avoided?

Data in Postgres is stored in fixed-sized blocks that are stored in the file system. Blocks are also cached in the Postgres buffer pool. When a block is needed, Postgres first checks its buffer pool. If the page is not found, Postgres reads from the file system. Buffer pool reads are memory accesses and significantly faster than reads from storage. Maximizing the buffer pool for the working set of an application is an important factor when choosing the machine specifications for your database server.

AlloyDB Omni introduces dynamic memory management to Postgres. Postgres typically has a fixed buffer pool size. AlloyDB Omni allows the buffer pool to grow and shrink dynamically, within configured bounds, depending on the memory demands of the system. *Therefore, there is no need to tune the buffer pool size.* When diagnosing performance issues, the first metrics to consider are the buffer pool hit rate and the read rate to see if your application is getting the benefit of the buffer pool. If not, that indicates that the application's data set does not fit in the buffer pool, and you could consider resizing to a larger machine with more memory.

Retrieving, filtering, aggregating, sorting, and projecting data all require CPU on the database server. Minimizing the amount of data that needs to be manipulated helps reduce the CPU required to filter, aggregate, and sort the results. You should monitor the CPU utilization on the database server to ensure the steady state utilization is around 70%. This amount leaves sufficient headroom on the server for spikes in utilization or changes in access patterns over time. Running at closer to 100% utilization introduces overhead due to process scheduling and context switching and might create bottlenecks in other parts of the system. High CPU utilization is another key metric to use when making decisions about machine specifications.

I/O is one of the most important factors in database application performance. For all database servers, read performance will be better when data is resident in the buffer pool, which is why efficient use of memory is so important.

Performance testing methodology and best practices

When benchmarking performance, it's important to define what you expect to learn from the test before beginning. Some questions you might consider are the following:
- What is the maximum throughput the system can achieve?
- How long does a particular query or workload take?
- How does the performance change as the amount of data increases?
- How does the performance of two different systems compare?
- How much does the Columnar Engine reduce the response time of my query performance?

- How much load can a database handle before I should consider upgrading to a more powerful machine?

Understanding the goals of your performance study informs what benchmark you run, what environment is required, and what metrics you need to collect.

Repeatability

To draw conclusions from performance testing, the results must be repeatable. If there is wide variation in performance from test to test, it becomes difficult to assess the impact of changes you made in the application or the system configuration. Running tests multiple times or for longer periods of time can help lower the amount of variation by providing more data.

Ideally, performance tests should be run on systems that are isolated from others. Running in an environment where other people's actions can affect the performance of your application can lead to drawing incorrect conclusions. Full isolation is often not possible when running in a multi-tenant, cloud environment, so you should expect to see greater variability in the results

Part of repeatability is ensuring that the test workload remains the same between runs. There is usually some randomness in the input to a test, which is acceptable as long as the randomness does not cause significantly different application behavior. For instance, if randomly generated client input changes the mix of reads and writes from run to run, performance will vary significantly.

Database size, caching, and I/O patterns

Ensure the amount of data you are testing with is representative of your application. Running tests with a small amount of data when you have hundreds of gigabytes or terabytes of data will likely not give you a true representation of how your application performs. The size of the data set also influences choices the query optimizer makes. Queries against small test tables may use table scans that give poor performance at larger scales and you won't identify missing indexes in this configuration.

Strive to replicate the I/O patterns of your application. The ratio of reads to writes is important to the performance profile of your application.

Benchmark Duration

In a complex system, there is a lot of state information that is maintained as the system executes: database connections are established, caches are populated, processes and threads are spawned.
At the start of a performance test, the initialization of these components could take up system resources and adversely affect the measured performance if the runtime of the workload is too short.

We recommend running performance tests for at least 20 minutes to minimize the effects of warming up the system. Measure performance during a steady state after startup and long enough to ensure that all aspects of database operations are included. For instance, database checkpoints are a critical feature of database systems and can have a significant impact on performance. Running a short benchmark that completes before the checkpoint interval hides this important factor in your application's behavior.

Methodical testing

When tuning performance, it is important to change only one variable at a time. If you change multiple aspects of a workload between runs, you won't be able to isolate which change improved performance. In fact, it is possible that multiple changes offset each other so you won't see the benefit of an appropriate change. If the database server is overutilized, try switching to a machine with more vCPUs while keeping the load constant. If the database server is underutilized, try increasing the load while keeping the CPU configuration constant.

Network topology and latencies

The network topology of your system can affect the performance test results. Latency between zones differs. When doing performance testing, ensuring that the client and the database cluster are in the same zone minimizes the network latency and yields the best performance–especially for applications with high throughput, short transactions as the network latency can be a big component of the overall transaction response time.

When comparing the performance of two different systems, ensure the network topology is the same for both systems. Note that network latency variability cannot be completely eliminated, even within the same zone there can be differences in latency due to underlying network topologies.

When it comes to deploying your application, you might want to better understand the impact of cross zone latencies by considering a typical high volume web application. The application has a load balancer sending requests to multiple web servers deployed across multiple zones for high availability. The latencies might differ depending on which web server processes a request because of cross-zone latency differences.
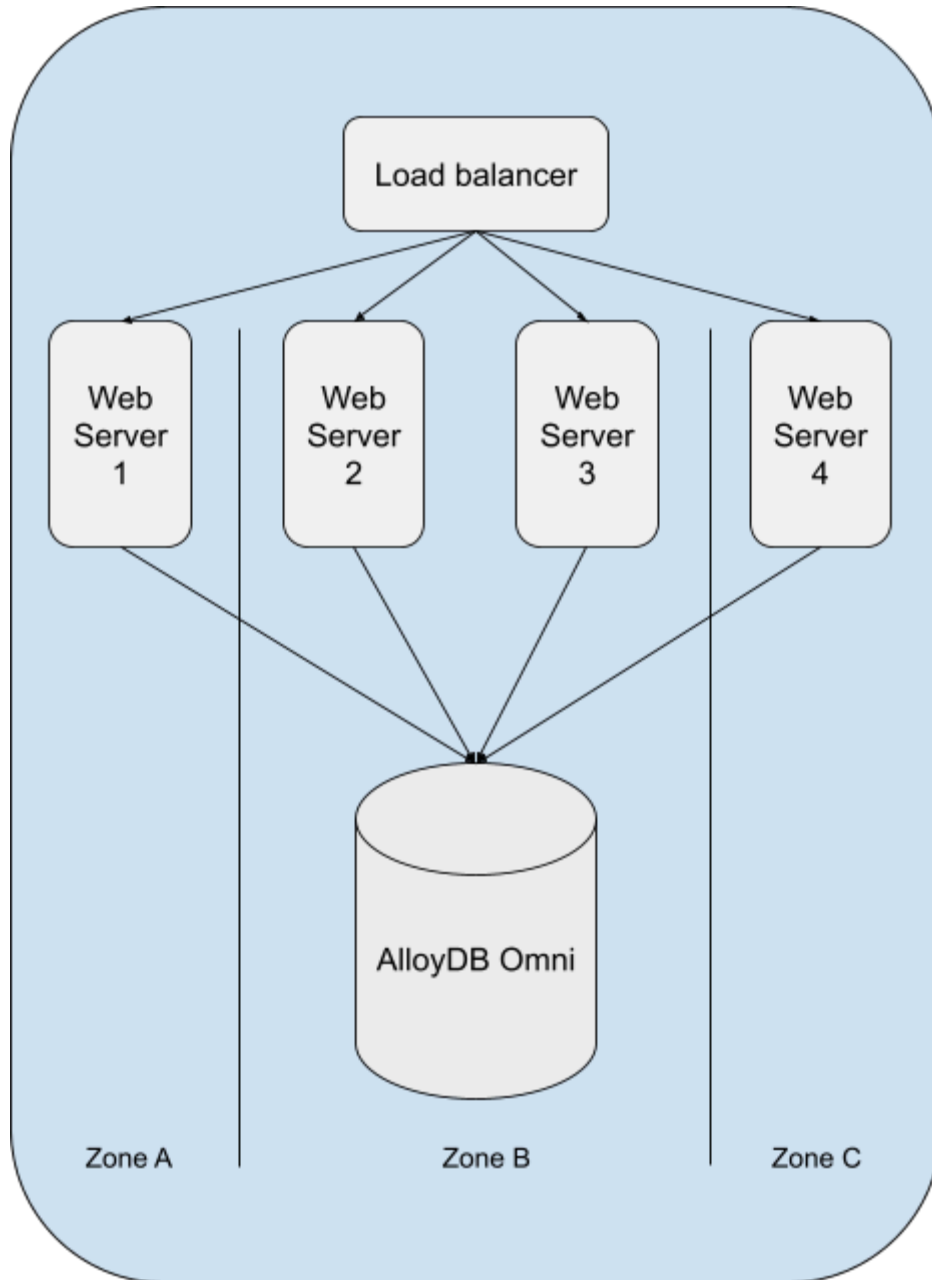
Figure 2 shows the typical architecture of a web application using AlloyDB Omni.
- Client requests are handled by a load balancer, which forwards each request to one web server out of many.
- The web servers are all connected to AlloyDB Omni. Some servers are in a different zone from where AlloyDB Omni is running, and will encounter higher latencies when making database requests.

Figure 2 shows a typical web application architecture. In this example, we would expect lower latencies when Web Servers 2 and 3 make requests to the database because they are in the same zone as the AlloyDB Omni database.

Resource monitoring

Monitoring the resource utilization of the system you are testing is critical. Monitoring the resource utilization of the client systems you are using to drive the workload is equally important. For example, if you are trying to find the maximum number of clients a system can support before it runs out of CPU, you won't be able to drive the system hard enough if the machines generating load don't have sufficient CPU themselves.

Scalability testing

Scalability testing is another aspect of performance testing. Scalability refers to how performance metrics change as one characteristic of a workload varies. Some examples of scalability studies include:
- How does the increase in the number of concurrent requests change throughput and response times?
- How does the increase in database size change the throughput and response times?

Scalability tests consist of multiple runs or a workload where a single dimension is varied between runs and one or more metrics are collected and plotted. This type of testing informs decisions about what bottlenecks exist in the system, how much load the system can handle given a specific configuration, what the maximum load a system can support, and what the behavior of the system is when the load increases beyond those levels.

Machine size considerations

AlloyDB Omni introduces many new features to Postgres to improve the reliability and availability of the database. The monitoring necessary to do this uses resources on the machine running AlloyDB Omni. On very small machine sizes there are limited memory and CPU resources to begin with, so for benchmarking, we recommend using machine sizes of 4 vCPUs minimally.

Interpret performance results

When you graph throughput over time as another variable is modified, typically you see throughput increase until it reaches a point of resource exhaustion.

Throughput vs. # of Clients
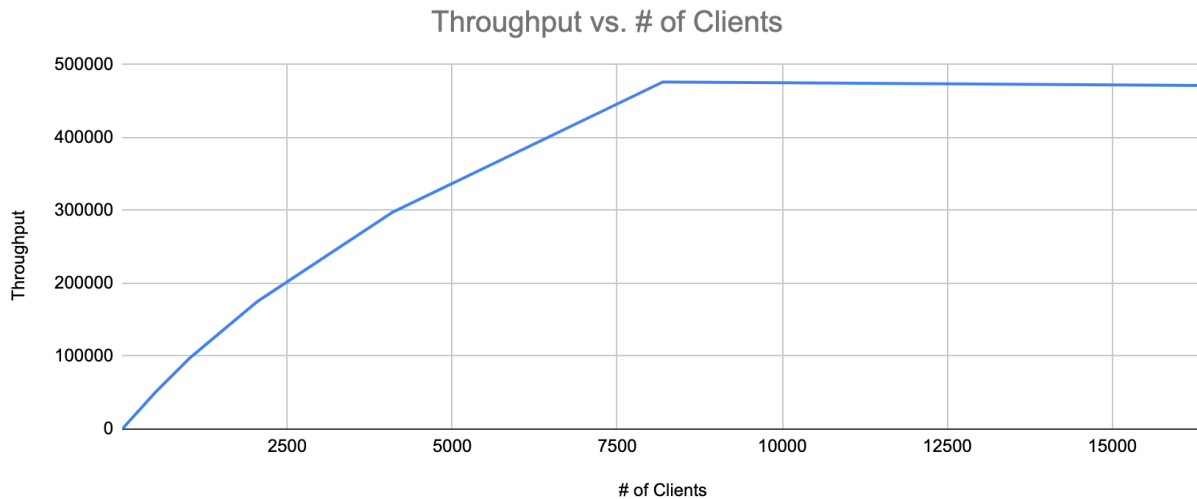
Figure 3 shows a typical throughput scaling graph, with number of clients on the x-axis, and throughput on the y-axis. As the number of clients increases, they generate more workload, and so overall throughput increases. However, at some point, the database is saturated and can handle no more requests, so further increase in the number of clients do not increase throughput any more.

Figure 3 shows a typical throughput graph. In an ideal situation, as you double the load on the system, throughput should double. In practice there will be contention on resources that leads to smaller throughput increases. At some point resource exhaustion or contention will cause throughput to flatten out or even decrease. If you are optimizing for throughput, this is a key point to identify as it drives your efforts into where to tune the application or database system to improve throughput.

Typical reasons for throughput to level off or drop include:
- CPU exhaustion on the database server
- CPU exhaustion on the client so the database server is not being sent more work
- Database lock contention
- I/O wait time when data exceeds the size of the Postgres buffer pool
- I/O wait time due to storage engine utilization
- Network bandwidth bottlenecks returning data to the client

Typically, latency and throughput are inversely proportional. As latency increases, throughput decreases. Intuitively this makes sense. As a bottleneck begins to materialize, operations start to take longer and the system performs fewer operations per second.

Figure 4: Typical latency scaling graph

Figure 4 shows the typical shape of how latency changes as the load placed on a system increases. Latency stays relatively constant until friction occurs due to resource contention. The inflection point of this curve generally corresponds to the flattening of the throughput curve in Figure 3.

Another useful way to evaluate latency is as a histogram. In this representation, we group latencies into buckets and count how many requests fall into each bucket.
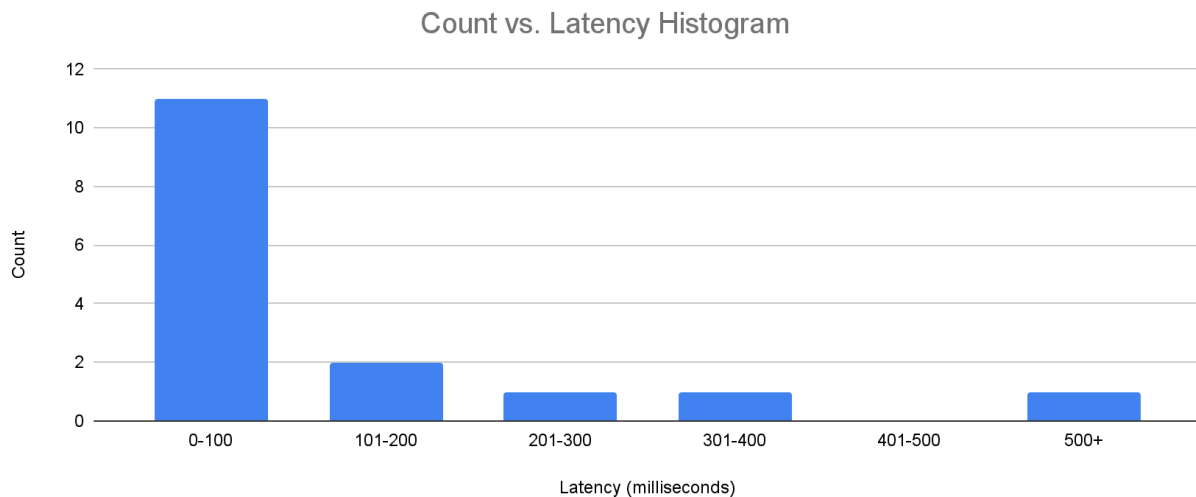
Figure 5: Typical latency histogram

Figure 5 illustrates a typical histogram. Most requests take under 100 milliseconds. There's a tail of requests with longer latencies. Understanding the cause of the tail can help explain variation seen in application performance. The causes of the tail correspond to the increased latencies

---

seen in the typical latency scaling graph (Figure 4) and the flattening of the throughput graph (Figure 3).

Where the latency histogram is most useful is when there are multiple modalities in an application. A modality is a normal set of operating conditions. For instance, most of the time the application is accessing pages that are in buffer cache. Most of the time, the application is updating existing rows, however, there might be multiple modes. Some of the time, the application is retrieving pages from storage, inserting new rows, or is experiencing lock contention.

When an application encounters these different modes of operation over time, the latency histogram shows these multiple modalities.
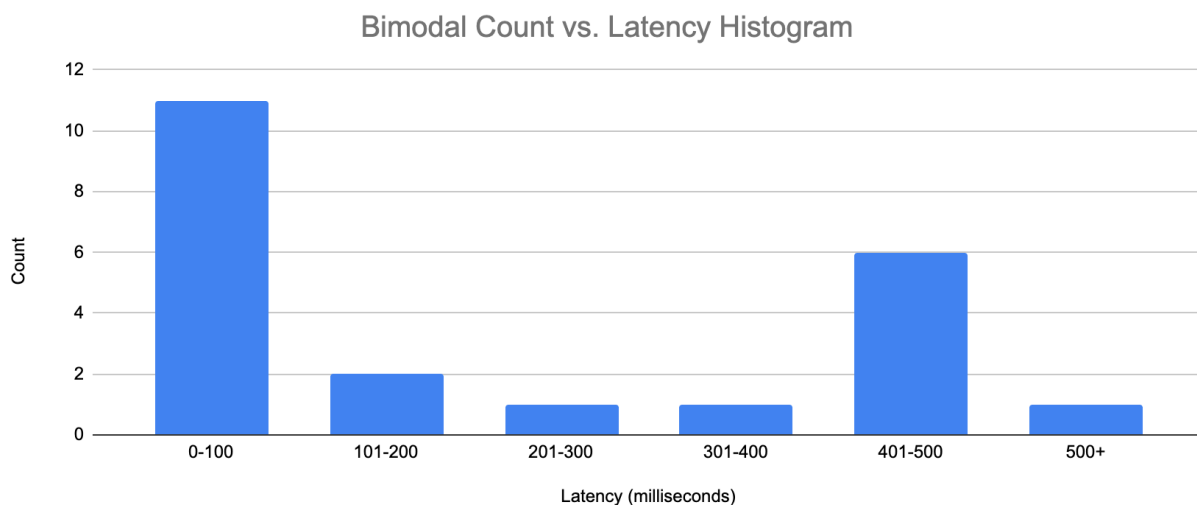
Figure 6: Bimodal latency histogram

Figure 6 shows what a bimodal histogram looks like. Most of the requests are serviced in under 100 milliseconds, but there's another cluster of requests that take 401-500 milliseconds. Understanding the cause of this second modality can help improve the performance of your application. There can be more than two modalities as well.

The second modality might be due to normal database operations, heterogeneous infrastructure and topology or application behavior. Some examples to consider are the following:
- Most data accesses are from the PostgreSQL buffer pool, but some come from storage
- Differences in network latencies for some clients to the database server
- Application logic that performs different operations depending on input or time of day
- Sporadic lock contention
- Spikes in client activity

Tools for analyzing performance

PostgreSQL cumulative statistics system

AlloyDB Omni exposes PostgreSQL's cumulative statistics system, which collects and reports information about server activity. This information is exposed as system tables, which can be queried through SQL.

For example, the query reports the number of disk blocks found in the buffer cache and number of disk blocks read from disk:

```
Unset
SELECT blks_hit, blks_read FROM pg_stat_database;
```

By observing the values of these counters over time, you can infer whether the database is making good use of the buffer cache.

The PostgreSQL documentation has more information about its [cumulative statistics system](#).

Perfsnap

Perfsnap is a performance analysis tool, similar to Oracle's AWR, that captures snapshots of crucial system metrics like CPU usage, memory usage, disk I/O, and wait events. By comparing these metrics to a performance baseline, the tool provides insights and visibility into vital performance metrics for database workloads. It is a convenient tool for gathering a number of metrics from PostgreSQL's cumulative statistics system, as well as additional metrics in AlloyDB Omni, into a concise and readable format.

To use Perfsnap, you capture two system snapshots while running your database workload.
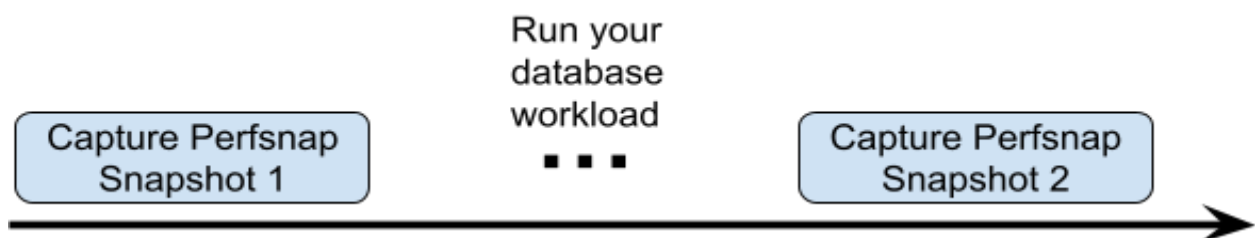
Then, from the two snapshots, you can generate a Perfsnap report, which summarizes changes in metrics values in between the two snapshots.

Install Perfsnap

Perfsnap is the schema name that contains SQL functions that allow users to capture snapshots or generate reports. Currently it's a part of the `g_stats` extension, which was developed for AlloyDB specifically. Perfsnap has to be installed by a superuser role. To use the Perfsnap APIs, connect to any database where users want to install the extension, and create the `g_stats` extension:

```
Unset

CREATE EXTENSION g_stats;
```

You don't need to install Perfsnap in multiple databases since the Perfsnap functions collect system-level metrics, not limited to specific databases. It doesn't matter which database users install Perfsnap in, the snapshot content doesn't rely on it.

Usage

The first step is connecting to the database where users install Perfsnap.

Create snapshots

```
Unset

SELECT perfsnap.snap();
postgres=# select perfsnap.snap();
 snap
------
    1
(1 row)
```

View a list of snapshots

```
Unset

SELECT * FROM perfsnap.g$snapshots;
postgres=# select * from perfsnap.g$snapshots;
 snap_id |          snap_time          | instance_id | node_id | snap_description | snap_type | is_baseline
---------+-----------------------------+-------------+---------+------------------+-----------+-------------
       1 | 2023-11-13 22:13:43.159237+00 | sr-primary  |         | Manual snapshot  | Manual    | f
(1 row)
```

Generate a perfsnap report

Generate a diff report between snapshots with `snap_id` 1 and 2.

```Unset
SELECT perfsnap.report(1,2)
```

Perfsnap report

The performance report consists of several sections:
- Performance report metadata
- Host Information
- Load Profile
- Response Time and Wait Class Breakdown
- Vacuum Information
- Database Information
- Database Conflict information
- Database Sizing Information
- Backend Wait Event Histogram Information
- Background Wait Event Histogram Information
- Parameter Section

Performance report metadata

Each performance report starts with metadata about the report about which system the report created for, when the snapshot collection started, and when the snapshot collection ended. The performance report also shows the PostgreSQL release when the snapshot was taken, making it easy to compare the performance to before and after upgrades.

In addition the performance report lists key memory sizing parameters for the Postgres cluster. This information allows users to quality the values reported in some of the later sections of the performance report.

```
PGSNAP DB Report for:

 Snapshot details
 ---------------------------------------
 Host                i841-sr-primary-c105d1fb-wkb3
 Release             14.7
 Startup Time        2023-11-05 00:45:10+00

            Snap Id    Snap Time
 ------------ ----------- ------------------------
 Begin Snap:         1 13.11.2023 22:13:43 (UTC) Manual snapshot
   End Snap:         2 13.11.2023 22:43:54 (UTC) Manual snapshot
    Elapsed:                 00:30:10.886373


 Database Cache sizes
 ~~~~~~~~~~~~~
         Shared Buffers:        31 GB      Block Size:          8192
    Effective Cache Size:       25 GB      WAL Buffers:        16384
```

Host information

The section shows how many CPU resources the machine had available when the report was created. The displayed percentage values are the average over the measurement interval. The host memory shows key metrics for the host memory: total, available, and free memory.

```
Host CPU
~~~~~~~~~~
    %User   %Nice %System   %Idle    %WIO    %IRQ   %SIRQ  %Steal  %Guest
   ------- ------- ------- ------- ------- ------- ------- ------- -------
     0.84    0.33    0.63   98.03    0.02    0.00    0.15    0.00    0.00

Host Memory
~~~~~~~~~~~~
          Total Memory:        63 GB
      Available Memory:        17 GB
           Free Memory:       701 MB
        Buffers Memory:      1015 MB
```

Load profile

The load profile makes it easier to qualify a workload. The per transaction counters make it easy to detect changes in the workload. If there are changes in the per transaction values, the workload has most likely changed.

```
Load profile (in bytes)
~~~~~~~~~~~~~~~~~~~~~~~~~          Per Second       Per Transaction
                                 -------------      ----------------
            Redo size:             55588.02              6583.19
         Logical reads:              409.51                48.50
        Physical reads:                 .02                  .00
       Physical writes:                 .04                  .00
       Tuples inserted:                 .01                  .00
        Tuples updated:                 .38                  .04
        Tuples deleted:                 .00                  .00
           Connections:                2.36                  .28
          Transactions:                8.44
```

Response time and wait class breakdown

The response time profile categorizes the response time of the Postgres cluster: how much time is spent servicing requests and how much time is spent in waiting. This information gives a clear indication on how to focus for performance tuning. If most of the time is consumed by waiting, the wait class breakdown reports which wait class is responsible for most of the waits. Further performance improvement efforts can then focus on this area.

```
Response Time Profile (in s)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CPU time:              120 (   0.47%)
```

```
Wait time:           25511 (  99.53%)
Total time:          25631


Backend Processes Wait Class Breakdown (in s)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
IO                 .815 (  99.99%)
Client             .000 (   0.01%)
LWLock             .000 (   0.00%)
IPC                .000 (   0.00%)
Lock               .000 (   0.00%)
```

Background process wait information

The background process wait information shows the corresponding wait event breakdown for background processes. This information also shows how much CPU time was consumed by all the backend processes, like the number of waits, the cumulative wait time, and the average time per wait. In the following example image it is clear that most of the time was spent on the WALWrite wait event. Why WalWrite? The server has idle wait events that are issued to wait for new work to arrive and time out after a predefined time. *LogicalLauncherMain* and *AutoVacuumMain* are idle wait events.

```
Background Processes Wait Information
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Event                              Class        Waits      Time (us)      Avg (us)
---------------------------------- -----------  --------   ------------   ----------
CPU                                                          99674090
Extension                          Extension     179981     7403180526        41133
WalSenderMain                      Activity       11208     3620954848       323068
WalFilePreallocatorMain            Activity           6     1972690671    328781778
AutoVacuumMain                     Activity         545     1808113412      3317639
WalWriterMain                      Activity     1696974     1807660541         1065
LogicalLauncherMain                Activity          10     1800798762    180079876
ArchiverMain                       Activity          33     1799933392     54543436
CheckpointerMain                   Activity           9     1791543040    199060337
BgWriterHibernate                  Activity         710     1776280219      2501803
LuxBgWriterMain                    Timeout            4     1680281741    420070435
BgWriterMain                       Activity         715       35804858        50076
BgWorkerShutdown                   IPC                4        9328456      2332114
CheckpointWriteDelay               Timeout           19        1902284       100120
WALSync                            IO              1628        1619015          994
WalFlushWhileSwitchingWALSegment   IO                 6         248719        41453
WALWrite                           IO              1628          96661           59
CommitWaitFlush                    IO                30          71784         2392
WALRead                            IO              4763          62888           13
ReplicationSlotSync                IO                12          22634         1886
DataFileSync                       IO                15          19122         1274
StorageAdvanceFlushPosition        IO              1628          13037            8
StorageAwaitMayWrite               IO              1628          12574            7
WalFlushByReplicationSlot          IO                12           9850          820
WalFlushDuringCheckPoint           IO                 6           4371          728
WalBatchWrite                      IO                 1           1071         1071
StorageAdvancePreFlushPosition     IO              1628            599            0
DataFileWrite                      IO                19            440           23
DataFileFlush                      IO                15            260           17
ReplicationSlotWrite               IO                12            195           16
SLRUWrite                          IO                 1             26           26
```

Database information

The database section shows key metrics for each database: rollbacks, commit, hit ratio, and information about temporary tables and sort operations.

```
Per Database Information
~~~~~~~~~~~~~~~~~~~~~~~~~
```

Name	Commits	Rollbacks	BlkRds	Blkhits	TempFiles	TempBytes
alloydbadmin	12973	0	0	255707	0	0 bytes
bench	700	0	0	140938	0	0 bytes
postgres	828	0	31	170528	0	0 bytes
template0	60	0	0	22560	0	0 bytes
template1	60	0	0	22440	0	0 bytes
vj1	670	0	0	129380	0	0 bytes

```
Per Database DML & DQL Information
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Name	Tuples returned	Tuples fetched	Tuples inserted	Tuples updated	Tuples deleted	Index splits	Index Only heap fetches	HOT updates
alloydbadmin	245247	155134	0	684	0	0	60	654
bench	321988	90150	0	0	0	0	0	0
postgres	429015	103276	12	0	0	0	0	0
template0	51210	12000	0	0	0	0	0	0
template1	52560	12000	0	0	0	0	0	0
vj1	178812	83990	0	0	0	0	0	0

```
Per Database Conflict Information
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Name	Lock Timeout	Old Snapshot	Buffer Pins	Deadlock
alloydbadmin	0	0	0	0
bench	0	0	0	0
postgres	0	0	0	0
template0	0	0	0	0
template1	0	0	0	0
vj1	0	0	0	0

Database sizing information

The database sizing section shows how much the database has grown in the snapshot interval

---

and also reports the database size itself. In addition it displays the collation and if the database has established connection limits.

```
Per Database Sizing Information
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                              Conn.
Name                Collation Limit  Tablespace           DB Size    Growth
------------------- --------- -----  -------------------- ---------- ----------
alloydbadmin        C.UTF-8      -1  pg_default              25 MB   0 bytes
bench               C.UTF-8      -1  pg_default              77 GB   0 bytes
postgres            C.UTF-8      -1  pg_default             133 MB    272 kB
template0           C.UTF-8      -1  pg_default              19 MB   0 bytes
template1           C.UTF-8      -1  pg_default              20 MB   0 bytes
vj1                 C.UTF-8      -1  pg_default              89 MB   0 bytes
```

Vacuum information

The vacuum information reports cumulative information since the beginning snapshot. It shows how many analysis and vacuum operations have been performed and how much I/O was necessary to fulfill these requests.

```
Vacuum Information
~~~~~~~~~~~~~~~~~~~~
              Num Analyze operations:        10
               Num Vacuum operations:        13
  Num Vacuum large table operations:         0
          Num Mini Vacuum operations:         1
                    Total time spent:     18125
    Total time spent in vacuum index:      1236
         Time spent on large tables:         0
  Time spent on large table indexes:         0
     Total Time Spent in Mini Vacuum:        54
 Wait Time Spent In Truncating Heap:         0
                        Reads in MB:         0
                       Writes in MB:         0
           Num canceled auto vacuums:         0
             Num failed auto vacuums:         0

                      Scanned pages:       305
                   Pin skipped pages:         0
                Frozen skipped pages:         0
   Operations which need more memory:         0

           Percent towards wraparound:       10%
  Percent towards emergency autovac:        97%
          Inflight Vacuum operations:         0

  Reasons for delayed vacuum:
                Oldest running XACT:    no XACT
               Oldest prepared XACT:         0
                 Oldest replica slot:         9
                 Oldest replica XACT:         0
```

Backend wait event histogram

If the wait histogram has been enabled, the report also contains the wait histogram information. The postgres cluster maintains wait histograms for the top 10 wait events. The list of wait events is not dynamically calculated; these wait events are predefined. The below report is shortened and has been reformatted to better suit this document. In reality the wait event histogram contains 32 buckets: from 1us to more than 16s. The list contains the wait event histogram information for the backend processes.

```
Backend Wait Event Histogram
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Event	Class	Waits	<= 1us	<= 2us	<= 4us	<= 8us	<= 16us	<= 32us	<= 64us	<= 128us	<= 256us	<= 512us
ChillCacheAcquireBucketLock	Lock	42	42	0	0	0	0	0	0	0	0	0
ChillCacheAcquireLRULock	Lock	19	19	0	0	0	0	0	0	0	0	0
CommitWaitFlush	IO	667	0	0	0	0	0	0	0	0	0	176
LockManager	LWLock	2	0	0	1	1	0	0	0	0	0	0
StorageDataFileReadUncached	IO	4	0	0	0	0	0	0	0	0	0	1

Event	Class	Waits	<= 1ms	<= 2ms	<= 4ms	<= 8ms	<= 16ms	<= 32ms	<= 64ms	<= 128ms	<= 256ms	<= 512ms
ChillCacheAcquireBucketLock	Lock	42	0	0	0	0	0	0	0	0	0	0
ChillCacheAcquireLRULock	Lock	19	0	0	0	0	0	0	0	0	0	0
CommitWaitFlush	IO	667	483	5	3	0	0	0	0	0	0	0
LockManager	LWLock	2	0	0	0	0	0	0	0	0	0	0
StorageDataFileReadUncached	IO	4	3	0	0	0	0	0	0	0	0	0

Event	Class	Waits	<= 1s	<= 2s	<= 4s	<= 8s	<= 16s	> 16s
ChillCacheAcquireBucketLock	Lock	42	0	0	0	0	0	0
ChillCacheAcquireLRULock	Lock	19	0	0	0	0	0	0
CommitWaitFlush	IO	667	0	0	0	0	0	0
LockManager	LWLock	2	0	0	0	0	0	0
StorageDataFileReadUncached	IO	4	0	0	0	0	0	0

Parameter settings

The parameter setting section contains key postgres configuration parameters and all other postgres configuration parameters that are different from the default value.

```
Parameter                       Value
--------------------------------------------------------------------------------

DateStyle                       ISO, MDY
TimeZone                        UTC
alloydb.enable_anon             on
alloydb.logical_decoding        on
application_name                psql
archive_command                 /bin/true
archive_mode                    on
archive_timeout                 300
autovacuum                      on
…
```

Linux performance tools

There are also many Linux tools and utilities that provide visibility into CPU and memory usage, independent of the database. For example, `top` is a common tool for observing CPU utilization, while `iostat` and `iotop` can give deep insights into disk performance.

A full write up of Linux tools and how to use them is beyond the scope of this document. Users may refer to the book "Systems Performance (2nd edition)" by Brendan Gregg for a more detailed treatment.

Resource considerations that affect performance

Instance sizing

Choosing the correct machine size for AlloyDB Omni is critical to getting the best performance for your application. Bringing together the concepts in this document, the inputs to making a machine size decision are about ensuring the system has enough resources to service your application, such as the correct amount of CPU, RAM, and storage.

CPU

The instance should have enough CPU power so that steady state operations can be serviced at or under 70% utilization. Having enough CPU power leaves headroom for the instance to handle spikes in utilization and to keep operating if your application utilization grows over time. Having more CPU power also ensures that there are resources available for periodic maintenance operations like vacuum. Running at, or near, 100% utilization can lead to poor performance due to process or thread context switching or queuing effects in other parts of the system as they contend for scarce CPU cycles.

If CPU utilization is consistently above 70% or has frequent, sustained spikes over 95%, consider moving to a larger instance size. Similarly, if steady state utilization is low, with peaks that are well below 100%, consider downsizing to a smaller instance to realize some cost savings.

Memory

The buffer pool hit rate is an important factor for application performance. AlloyDB Omni performs dynamic memory management, which adapts to changing memory needs of the database. The maximum size of the buffer pool, without the columnar engine enabled, is 80% of the memory available on the machine type used. As the database runs, AlloyDB Omni grows and shrinks the buffer pool to accommodate queries that need additional memory, for example, analytical queries.

Postgres buffer pool utilization is a key factor in system performance. You can look at the buffer pool hit rate to get an idea of how much data the application is accessing from the buffer pool while it is running. If the miss rate is high, consider increasing the amount of memory to make more RAM available to the Postgres buffer pool.

A database might have a large amount of data, but a smaller subset, called the working set, is actually used by the application. It's important to size your instance to the working set. If the working set is small enough to fit entirely in the buffer pool you will get the best performance.

AlloyDB Omni tuning parameters

Suggested parameters

No two applications have the same performance characteristics or requirements. We've selected a set of workloads that exhibit different read/write patterns, caching behaviors, and index access patterns. Running these workloads, we've identified the generally optimal default values for some parameters.

Most default Postgres parameter values remain unchanged, as we limited the default changes to parameters that will benefit the majority of applications. The suggested values of these parameters should generally improve performance on modern hardware.

Database parameter	AlloyDB Omni default	Suggested setting
random_page_cost	4.0	1.1

temp_buffers	8 MB	128 MB
max_wal_size	1 GB	20 GB
min_wal_size	80 MB	10 GB
default_toast_compression	pglz	lz4
maintenance_work_mem	64 MB	1 GB for smaller machines 2 GB for larger machines
max_worker_processes	64	# vCPUs
max_parallel_workers	8	max(8, # vCPUs)
max_parallel_workers_per_gather	2	max(2, # vCPUs / 2)
work_mem	4 MB	128 to 512 MB, depending on workload and the amount of memory available
effective_cache_size	40% * DRAM	80% * DRAM
alloydb.enable_auto_explain	off	on: avoids restart to use auto_explain
auto_explain.log_min_duration	N/A	-1: turns the auto explain off, but will allow a user to set in a session should auto explain need to be used
auto_explain.log_buffers	N/A	on
auto_explain.log_nested_statements	N/A	on
auto_explain.log_settings	N/A	on
auto_explain.log_triggers	N/A	on
auto_explain.log_verbose	N/A	on
auto_explain.log_wal	N/A	on
alloydb.enable_pg_hint_plan	off	on: avoids a restart to use auto_explain

Columnar engine tuning

Setting the following flag enables columnar engine on primary or read replica instances. The default is `OFF`. The database needs to be restarted after setting this flag. Once enabled, the columnar engine allocates 1024 MB to the columnar engine. You can configure the amount of memory with the `google_columnar_engine.memory_size_in_mb` flag:

```
Unset
google_columnar_engine.enabled = ON
```

Once enabled, the ML/AI based recommendation and auto-columnarization feature begins to monitor the workload and automatically populate tables into the columnar memory every hour.

The following flag can be used to change the schedule of auto-columnarization based upon anticipated scan-heavy workloads. The flag does not require a database restart. The lowest value that can be set is 1 `HOUR`.

```
Unset
google_columnar_engine.auto_columnarization_schedule = 'EVERY 6 HOURS'
```

To ensure that the default memory is sufficient for populating the recommended columns, run the following command after running the workload.

```
Unset
SELECT * FROM google_columnar_engine_recommend('RECOMMEND_SIZE')
```

Compare the required memory size to the configured memory size. Configured memory size can be obtained by running the following command:

```
Unset
SHOW google_columnar_engine.memory_size_in_mb
```

If the configured size is less than recommended size, increase the configured size to a higher value using the following command. The database needs to be restarted after setting this flag. After the database restarts, run the workload again for recommendation to populate the necessary tables.

```
Unset
google_columnar_engine.memory_size_in_mb = <size in MB>
```

Did you find this document helpful? [Please send us your feedback](#).

If manual population is a preferred choice, the recommendation and auto-columnarization features can be turned off using the following flag. The default setting is ON. This change does not require a database restart.

```
Unset
google_columnar_engine.enable_auto_columnarization = OFF
```

Columnar engine uses up to 20% of CPU cores for background maintenance jobs, such as population, refresh, and recommendation. The following flag can be used to adjust the CPU resources allowed for maintenance tasks. Valid values are 0% to 100%, with 20% as the default. The lowest CPU utilization for background maintenance tasks is 1 vCPU.

```
Unset
google_job_scheduler.maintenance_cpu_percentage = 50
```

Columnar engine uses up to 2 worker processes for population and refresh background jobs. The following flag can be used to speed those operations by increasing the number of worker processes for background jobs. The default setting is 2 worker processes. These processes run on limited CPU resources as specified in the `google_job_scheduler.maintenance_cpu_percentage` flag. Setting this flag does not require the database to be restarted.

```
Unset
google_job_scheduler.max_parallel_workers_per_job = 4
```

Performance benchmarking guides

The following documents describe step-by-step procedures to setup, load and run benchmarks on AlloyDB Omni. They describe running 2 popular benchmarks, TPC-C, which simulates a workload for transactional processing, and TPC-H, which simulates a workload for analytical queries.

1. 🗐 User Guide: OLTP Performance benchmarking on AlloyDB OneOmni
2. 🗐 User Guide: OLAP Performance benchmarking on AlloyDB OneOmni

---

Did you find this document helpful? [Please send us your feedback](#).