General configuration and tuning recommendations for EDB Postgres Advanced Server and PostgreSQL on Linux

Last updated: March 17, 2022

These recommendations for tuning EDB Postgres Advanced Server (EPAS) and PostgreSQL represent a starting point. Benchmarks and other measurements are required for proper tuning. This document is not intended to be an exhaustive list of configuration settings or recommendations, only the most important parameter settings that are not already the default values.

For optimal tuning, we recommend engaging with EDB's Professional Services team or a qualified EDB partner.

Operating system-level recommendations:

Mount Points

We recommend having separate mount points with dedicated IOPs for the EPAS/PG Cluster. For better performance, we recommend using SSDs

1. `/pgdata:` Mount point for data directory of EPAS/PG
   PGDATA directory: `/pgdata/data`

2. `/pgwaldata/:` Mount point for WAL (Write Ahead Log).
   WAL directory for EPAS/PG: `/pgwaldata/wal`

Depending on the usage of indexes and the amount of indexes needed for user-defined tables, as per workload, we recommend having a separate mount point for indexes.

Filesystem

We recommend using the XFS filesystem for PG/EPAS data directory, WAL, and any other mount points hosting PG/EPAS data.

Read-ahead

Increasing disk read ahead improves I/O throughput by reducing the number of requests to disk. This is typically set at 128 kB, but it's generally recommended that this be set to 4096 kB on the disk hosting the database or tablespaces.

You can tune the readahead parameter with the `tuned` daemon, as written below.

I/O scheduler

We recommend using the `deadline` I/O scheduler for CentOS/RHEL 7 and `mq-deadline` for CentOS/RHEL 8 for spindles. For systems equipped with solid state storage or a dedicated IO controller that has its own reordering mechanism, we recommend using `noop` for RHEL 7 and `none` for RHEL 8.

Dirty Memory

Current Linux systems often define dirty memory as a ratio to total RAM size. Once more than this amount of memory is used and modified without being committed, a Linux system will enter sequential access mode and will block on all IO requests not related to flushing the dirty memory to disk. Such events are *extremely* disruptive. Due to database write access patterns, this becomes more likely as volume increases.

In modern hardware and some VMs, even 1% of system RAM may represent more data than a backing storage system can absorb without significant IO waits. This is why modern Linux kernels supply the ability to set dirty memory in bytes instead of a ratio. The recommended setting should be around the cache size of the underlying storage device or controller. In situations where those values are unknown, 1GB is a suitable default.

In addition to considering dirty memory as a ratio of total available RAM, current Linux systems also tend to write dirty memory to disk in the background only after it has crossed a threshold dictated by `vm.dirty_background_ratio`. The default for this setting varies, but is commonly 5-10% of RAM, and can be no lower than 1%.

Similarly to `vm.dirty_bytes`, modern kernels provide the ability to set the background write threshold in bytes instead with `vm.dirty_background_bytes.` Here it's common to use a value 1/4 the size of the dirty bytes allocation. This allows the kernel to trickle data to the disk subsystem long before it reaches the value we specify in `vm.dirty_bytes`, and also prevents excessive background writing caused by using a value that is too small.

Most storage subsystem caches aren't exceedingly large at 1-4GB, so encouraging background writes while also avoiding storage activity overhead is a delicate balancing act. Setting `dirty_background_bytes` to ¼ of `dirty_bytes` is an easy first-approximation that can be adjusted later depending on the environment. When larger caches are available, `dirty_background_bytes` may be a smaller proportion of `dirty_bytes.`

Other Operating System Parameters

Below are EDB's suggestions for tuning a Linux OS for EPAS/PostgreSQL using the above guidelines:

```
mkdir /etc/tuned/edb

export DIRTY_BYTES=$[1024*1024*1024]
export DIRTY_BG=$[${DIRTY_BYTES}/4]
export MEM_TOTAL=$(grep MemTotal /proc/meminfo | awk '{print $2}')

cat<<EOF>/etc/tuned/edb/tuned.conf
[main]
```

```
summary=Tuned profiles for EnterpriseDB Postgres Advanced Server
[cpu]
governor=performance
energy_perf_bias=performance
min_perf_pct=100
[disk]
readahead=4096
[sysctl]
vm.overcommit_memory=2
vm.overcommit_kbytes=${MEM_TOTAL}
vm.swappiness=1
vm.dirty_bytes=${DIRTY_BYTES}
vm.dirty_background_bytes=${DIRTY_BG}
[vm]
transparent_hugepages=never
EOF

systemctl enable --now tuned
tuned-adm profile edb
```

Initializing the EPAS/PG Cluster

Begin with this initdb Command to bootstrap the EPAS PGDATA folder:

```
PGSETUP_INITDB_OPTIONS="-X /pgwaldata/12-wal --pgdata=/pgdata/12-data -E
UTF-8 -k --auth=peer --auth-host=scram-sha-256"
/usr/edb/as12/bin/edb-as12-setup initdb
```

Use systemctl to extend the service file as shown below:

```
systemctl edit edb-as-12
```

Then add the following contents in the file in the edit window:

```
[Service]
Environment=PGDATA=/pgdata/12-data
```

Save and exit, then enable and start the edb-as service:

```
systemctl enable --now edb-as-12
```

Configuration & Authentication

max_connections

The optimal maximum number for max_connections is roughly 4 times the number of CPU cores. This formula often gives a very small number which isn't practical in most real world cases, thus we recommend using the formula: `GREATEST(4 x CPU cores, 100)`. Beyond this number, a connection pooler such as pgbouncer should be used.

Note: If you don't need that many connections as given by the above formula, then it is recommended to reduce the value of this parameter.

password_encryption

It is recommended that people should use `scram-sha-256` for this parameter.

Resource Usage

shared_buffers

This parameter has the most variance of all. Some workloads work best with very small values (such as 1GB or 2GB) even with very large database volumes. Other workloads require large values. A reasonable starting point is to use `Total RAM / 4`, with a more nuanced version that accounts for more RAM variance in the following pseudo-code:

```
base = RAM / 4

if RAM < 3 GB:
   base = base * 0.5
else if RAM < 8 GB:
   base = base * 0.75
else if RAM > 64 GB:
   base = greatest(16 GB, RAM / 6)
```

```
shared_buffers = least(base, 64 GB)
```

This accounts for the fact that lower memory systems have less tolerance for high amounts of memory being dedicated to Postgres shared buffers while still handling user sessions. Servers with much higher amounts of RAM can absorb proportionally adjusted shared buffer settings. However, beyond 64GB, there are diminishing returns due to overhead associated with maintaining such a large contiguous memory allocation.

work_mem

The recommended starting point for work_mem is `((Total RAM - shared_buffers)/(16 x CPU cores)).`

maintenance_work_mem

This determines the maximum amount of memory used for maintenance operations like VACUUM, CREATE INDEX, ALTER TABLE ADD FOREIGN KEY, and data-loading operations. These may increase the I/O on the database servers while performing such activities, so allocating more memory to them may lead to these operations finishing more quickly. The calculated value of `15% x (Total RAM - shared_buffers) / autovacuum_max_workers` up to 1GB is a good start.

effective_io_concurrency

This parameter is used for read-ahead during certain operations and should be set to the number of disks used to store the data. However, improvements have been observed by using a multiple of that number. For solid-state disks, it is recommended to set this value to 200.

Write-Ahead Log

wal_compression

When this parameter is on, the PostgreSQL server compresses a full-page image written to WAL when full_page_writes is on or during a base backup. Set this parameter to 'on'

wal_log_hints

This parameter is required in order to use pg_rewind. Set it to 'on'.

Note: In the event the Postgres instance has checksums enabled, this setting is implied as always being activated.

wal_buffers

This controls the amount of memory space available for backends to place WAL data prior to sync. WAL segments are 16MB each by default, so buffering a segment is very inexpensive memory-wise. And larger buffer sizes have been observed to have a potentially very positive effect on performance in testing. Set this parameter to `64MB`.

checkpoint_timeout

Longer timeouts reduce overall WAL volume but make crash recovery take longer. The recommended value is a minimum of `15 minutes` but ultimately, the RPO of business requirements dictates what this should be.

checkpoint_completion_target

This determines the amount of time in which PostgreSQL aims to complete a checkpoint. This means a checkpoint need not result in an I/O spike and instead aims to spread the writes over a certain period of time. The recommended value is `0.9`.

max_wal_size

This parameter can be difficult to set as its optimal value is a function of `checkpoint_timeout` and the maximum WAL throughput of the system. However, if set too small it can dramatically reduce performance so it is well worth taking the time to tune it. The risk of setting it too high is that you run out of disk space, but otherwise performance won't be adversely affected. Note that `max_wal_size` is a **soft** limit.

If you have lots of disk space available for WAL (hundreds of gigabytes or more), then set it to a high value. On a very high performance system, 200GB or even higher may not be unreasonable.

If your disk space is constrained, set `max_wal_size` to the highest value you can, that will avoid the risk of running out of space, leaving a little headroom. If your WAL is on a dedicated disk partition (which is always recommended), this may be 50-75% of the partition size" to be safe(r).

In order to more precisely tune `max_wal_size`, it is recommended that you monitor the `checkpoints_timed` and `checkpoints_req` values in the `pg_stat_bgwriter` view. If `max_wal_size` is too small, the ratio of requested checkpoints to timed checkpoints will rise, indicating that checkpoints are occurring because there is not enough space to hold the WAL segments required to reach the next timed checkpoint.

It's perfectly fine to have some requested checkpoints as a result of occasional activity spikes, however this should not be the norm. Increase `max_wal_size` as needed to minimize the number of requested checkpoints, without exhausting the available disk space.

archive_mode

Because changing this requires a restart, it should be set to ‘`on`’.

archive_command

A valid archive_command is required if archive_mode is on. Until archiving is ready to be configured, a default of '`: to be configured`' on a POSIX system.

Query Tuning

random_page_cost

If using SSD disks, the recommended value is `1.1`.

effective_cache_size

This should be the sum of `shared_buffers` and the Linux buffer cache (as seen in the buff/cache column of the `free` command).

cpu_tuple_cost

Specifies the relative cost of processing each row during a query. It is currently set to `0.01`, but this is likely to be lower than optimal and should be increased to `0.03` for a more realistic costing.

Reporting and Logging

logging_collector

This parameter should be `on` if log_destination includes `stderr` or `csvlog`.

log_directory

If the logging_collector is on, this should be set to someplace outside of the data directory. This way, the logs are not part of base backups.

log_checkpoints

This should be set to on.

log_min_duration_statement

It's important to identify badly performing queries early, so we suggest setting this parameter to `1s` to log any queries that run for one second or longer. This may be too verbose in some instances, but serves as a good initial default.

log_line_prefix

The prefix should at least contain the time, the process id, the line number, the user and database, and the application name.

Suggested value: `'%m [%p]: u=[%u] db=[%d] app=[%a] c=[%h] s=[%c:%l] tx=[%v:%x]'`

Note: Don't forget the space at the end

log_lock_waits

Set to `on`. This parameter is essential in diagnosing slow queries.

log_statement

Set to '`ddl`'. In addition to leaving a basic audit trail, this will help determine at what time a catastrophic human error occurred, such as dropping the wrong table.

log_temp_files

Set to `0`. This will log all temporary files created, suggesting that work_mem is incorrectly tuned.

timed_statistics (EPAS)

Controls the collection of timing data for the Dynamic Runtime Instrumentation Tools Architecture (DRITA) feature. When set to on, timing data is collected. Set this parameter to on.

Autovacuum

log_autovacuum_min_duration

Monitoring autovacuum activity will help in tuning it. Suggested value: `0`.

autovacuum_max_workers

This is the number of workers that autovacuum has. Default value is 3 and requires a DB restart to be updated. Please note that each table can have only one worker working on it. So increasing workers only helps in parallel and more frequent vacuuming across tables. The default value is low, therefore it is recommended to increase this value to `5`.

autovacuum_vacuum_cost_limit

To prevent excessive load on the DB due to the autovacuum, there is an I/O quota imposed by Postgres. So every read/write causes depletion of this quota and once it is exhausted the autovacuum sleeps for a fixed time. This configuration increases the quota limit, therefore increasing the amount of I/O that the vacuum can do. The default value is low, we recommend increasing this value to `5000`.

Client Connection Defaults

idle_in_transaction_session_timeout

Sessions that remain idle in a transaction can hold locks and prevent a vacuum. Suggested value: `10 minutes`.

lc_messages

Log analyzers only understand untranslated messages. Set this to '`C`'.

shared_preload_libraries

Adding `pg_stat_statements` is low overhead and high value. This is recommended but optional.