Tuning Your PostgreSQL Server

From PostgreSQL wiki

by Greg Smith, Robert Treat, and Christopher Browne

PostgreSQL ships with a basic configuration tuned for wide compatibility rather than performance. Odds are good the default parameters are very undersized for your system. Rather than get dragged into the details of everything you should eventually know (which you can find if you want it at the GUC Three Hour Tour (http://www.pgcon.org/2008/schedule/events/104.en.html)), here we're going to sprint through a simplified view of the basics, with a look at the most common things people new to PostgreSQL aren't aware of. You should click on the name of the parameter in each section to jump to the relevant documentation in the PostgreSQL manual for more details after reading the quick intro here. There is also additional information available about many of these parameters, as well as a list of parameters you shouldn't adjust, at Server Configuration Tuning (https://www.packtpub.com/article/server-configuration-tuning-postgresql).

Background Information on Configuration Settings

PostgreSQL settings can be manipulated a number of different ways, but generally you will want them changed in your configuration files, either directly or, starting with PostgreSQL 9.4, through `ALTER SYSTEM` (http://www.postgresql.org/docs/current/static/sql-altersystem.html). The specific options available change from release to release, the definitive list is in the source code at src/backend/utils/misc/guc.c for your version of PostgreSQL (but the pg_settings view works well enough for most purposes).

The types of settings

There are several different types of configuration settings, divided up based on the possible inputs they take

- Boolean: true, false, on, off
- Integer: Whole numbers (2112)
- Float: Decimal values (21.12)
- Memory / Disk: Integers (2112) or "computer units" (512MB, 2112GB). Avoid integers--you need to know the underlying unit to figure out what they mean.
- Time: "Time units" aka d,m,s (30s). Sometimes the unit is left out; don't do that
- Strings: Single quoted text ('pg_log')
- ENUMs: Strings, but from a specific list ('WARNING', 'ERROR')
- Lists: A comma separated list of strings ('"$user",public,tsearch2)

When they take effect

PostgreSQL settings have different levels of flexibility for when they can be changed, usually related to internal code restrictions. The complete list of levels is:

- Postmaster: requires restart of server
- Sighup: requires a HUP of the server, either by kill -HUP (usually -1), pg_ctl reload, or `SELECT pg_reload_conf();`
- User: can be set within individual sessions, take effect only within that session
- Internal: set at compile time, can't be changed, mainly for reference

- Backend: settings which must be set before session start
- Superuser: can be set at runtime for the server by superusers

Most of the time you'll only use the first of these, but the second can be useful if you have a server you don't want to take down, while the user session settings can be helpful for some special situations. You can tell which type of parameter a setting is by looking at the "context" field in the pg_settings view.

Important notes about configuration files

- Command line options override postgresql.auto.conf settings override postgresql.conf settings.
- If the same setting is listed multiple times, the last one wins.
- You can figure out the postgresql.conf location with `SHOW config_file`. It will generally be $PGDATA/postgresql.conf (`SHOW data_directory`), but watch out for symbolic links, postmaster.opts (http://www.postgresql.org/docs/current/static/app-pg-ctl.html#AEN93617) and other trickiness
- Lines with # are comments and have no effect. For a new database, this will mean the setting is using the default, but on running systems this may not hold true! Changes to the configuration files do not take effect without a reload/restart, so it's possible for the system to be running something different from what is in the file.

Viewing the current settings

- Look at the configuration files. This is generally not definitive!
- `SHOW ALL`, `SHOW <setting>` will show you the current value of the setting. Watch out for session specific changes
- `SELECT * FROM pg_settings` will label session specific changes as locally modified

Tuning tools

- dbForge Studio for PostgreSQL (https://www.devart.com/dbforge/postgresql/studio/query-profiler.html) helps to identify productivity bottlenecks, and provides PostgreSQL performance tuning.
- The postgresqltuner.pl (https://github.com/jfcoz/postgresqltuner) script can analyze the configuration and make tuning recommendations.
- PgBadger (https://pgbadger.darold.net/) analyse PostgreSQL logs to generate performance reports.
- pgMustard (https://www.pgmustard.com/) provides tuning advice based on EXPLAIN ANALYZE output.

listen_addresses (http://www.postgresql.org/docs/current/static/runtime-config-connection.html#GUC-LISTEN-ADDRESSES)

By default, PostgreSQL only responds to connections from the local host. If you want your server to be accessible from other systems via standard TCP/IP networking, you need to change listen_addresses from its default. The usual approach is to set it to listen to all addresses like this:

```
listen_addresses = '*'
```

And then control who can and cannot connect via the pg_hba.conf (http://www.postgresql.org/docs/current/static/auth-pg-hba-conf.html) file.

max_connections (http://www.postgresql.org/docs/current/static/runtime-config-connection.html#GUC-MAX-CONNECTIONS)

max_connections sets exactly that: the maximum number of client connections allowed. This is very important to some of the below parameters (particularly work_mem) because there are some memory resources that are or can be allocated on a per-client basis, so the maximum number of clients suggests the maximum possible memory use. Generally, PostgreSQL on good hardware can support a few hundred connections. If you want to have thousands instead, you should consider using connection pooling software to reduce the connection overhead.

shared_buffers (http://www.postgresql.org/docs/current/static/runtime-config-resource.html#GUC-SHARED-BUFFERS)

The shared_buffers configuration parameter determines how much memory is dedicated to PostgreSQL to use for caching data. One reason the defaults are low is because on some platforms (like older Solaris versions and SGI), having large values requires invasive action like recompiling the kernel. Even on a modern Linux system, the stock kernel will likely not allow setting shared_buffers to over 32MB without adjusting kernel settings first. (PostgreSQL 9.4 and later use a different shared memory mechanism, so kernel settings will usually not have to be adjusted there.)

If you have a system with 1GB or more of RAM, a reasonable starting value for shared_buffers is 1/4 of the memory in your system. If you have less RAM you'll have to account more carefully for how much RAM the OS is taking up; closer to 15% is more typical there. There are some workloads where even larger settings for shared_buffers are effective, but given the way PostgreSQL also relies on the operating system cache, it's unlikely you'll find using more than 40% of RAM to work better than a smaller amount.

Be aware that if your system or PostgreSQL build is 32-bit, it might not be practical to set shared_buffers above 2 ~ 2.5GB. See this blog post (http://rhaas.blogspot.jp/2011/05/sharedbuffers-on-32-bit-systems.html) for details.

Note that on Windows, large values for shared_buffers aren't as effective, and you may find better results keeping it relatively low and using the OS cache more instead. On Windows the useful range is 64MB to 512MB.

Changing this setting requires restarting the database. Also, this is a hard allocation of memory; the whole thing gets allocated out of virtual memory when the database starts.

PostgreSQL 9.2 or earlier

If you are running PostgreSQL 9.2 or earlier, it's likely that in order to increase the value of shared_buffers you will have to increase the amount of memory your operating system allows you to allocate to a single shared memory segment. On UNIX-like systems, if you set it above what's supported, you'll get a message like this:

```
IpcMemoryCreate: shmget(key=5432001, size=415776768, 03600) failed: Invalid argument

This error usually means that PostgreSQL's request for a shared memory
segment exceeded your kernel's SHMMAX parameter. You can either
reduce the request size or reconfigure the kernel with larger SHMMAX.
To reduce the request size (currently 415776768 bytes), reduce
PostgreSQL's shared_buffers parameter (currently 50000) and/or
its max_connections parameter (currently 12).
```

See Managing Kernel Resources (http://www.postgresql.org/docs/current/static/kernel-resources.html) for details on how to correct this.

effective_cache_size (http://www.postgresql.org/docs/current/static/runtime-config-query.html#GUC-EFFECTIVE-CACHE-SIZE)

effective_cache_size should be set to an estimate of how much memory is available for disk caching by the operating system and within the database itself, after taking into account what's used by the OS itself and other applications. This is a guideline for how much memory you expect to be available in the OS and PostgreSQL buffer caches, not an allocation! This value is used only by the PostgreSQL query planner to figure out whether plans it's considering would be expected to fit in RAM or not. If it's set too low, indexes may not be used for executing queries the way you'd expect. The setting for shared_buffers is not taken into account here--only the effective_cache_size value is, so it should include memory dedicated to the database too.

Setting effective_cache_size to 1/2 of total memory would be a normal conservative setting, and 3/4 of memory is a more aggressive but still reasonable amount. You might find a better estimate by looking at your operating system's statistics. On UNIX-like systems, add the free+cached numbers from free or top to get an estimate. On Windows see the "System Cache" size in the Windows Task Manager's Performance tab. Changing this setting does not require restarting the database (HUP is enough).

checkpoint_segments checkpoint_completion_target (http://www.postgresql.org/docs/current/static/runtime-config-wal.html#RUNTIME-CONFIG-WAL-CHECKPOINTS)

Note: This applies to PostgreSQL 9.4 and below. PostgreSQL 9.5 introduced min_wal_size and max_wal_size (https://www.postgresql.org/docs/9.5/release-9-5.html) configuration parameters and removed checkpoint_segments. Please review the release notes and the documentation on min_wal_size (https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-MIN-WAL-SIZE), max_wal_size (https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-MAX-WAL-SIZE) and WAL configuration (https://www.postgresql.org/docs/current/wal-configuration.html).

PostgreSQL writes new transactions to the database in files called WAL segments that are 16MB in size. Every time checkpoint_segments worth of these files have been written, by default 3, a checkpoint occurs. Checkpoints can be resource intensive, and on a modern system doing one every 48MB will be a serious performance bottleneck. Setting checkpoint_segments to a much larger value improves that. Unless you're running on a very small configuration, you'll almost certainly be better setting this to at least 10, which also allows usefully increasing the completion target.

For more write-heavy systems, values from 32 (checkpoint every 512MB) to 256 (every 4GB) are popular nowadays. Very large settings use a lot more disk and will cause your database to take longer to recover, so make sure you're comfortable with both those things before large increases. Normally the large settings (>64/1GB) are only used for bulk loading. Note that whatever you choose for the segments, you'll still get a checkpoint at least every 5 minutes unless you also increase checkpoint_timeout (which isn't necessary on most systems).

Checkpoint writes are spread out a bit while the system starts working toward the next checkpoint. You can spread those writes out further, lowering the average write overhead, by increasing the checkpoint_completion_target parameter to its useful maximum of 0.9 (aim to finish by the time 90% of the next checkpoint is here) rather than the default of 0.5 (aim to finish when the next one is 50% done). A setting of 0 gives something similar to the behavior of obsolete versions. The main reason the default isn't just 0.9 is that you need a larger checkpoint_segments value than the default for broader spreading to work well. For lots more information on checkpoint tuning, see Checkpoints and the Background Writer (http://www.westnet.com/~gsmith/content/postgresql/chkp-bgw-83.htm) (where you'll also learn why tuning the background writer parameters is challenging to do usefully).

autovacuum (http://www.postgresql.org/docs/current/static/routine-vacuuming.html#AUTOVACUUM)

The autovacuum process takes care of several maintenance chores inside your database that you really need. Generally, if you think you need to turn regular vacuuming off because it's taking too much time or resources, that means you're doing it wrong. The answer to almost all vacuuming problems is to vacuum more often, not less, so that each individual vacuum operation has less to clean up.

However, it's acceptable to disable autovacuum for short periods of time, for instance when bulk loading large amounts of data.

Logging (http://www.postgresql.org/docs/current/static/runtime-config-logging.html)

There are many things you can log that may or may not be important to you. You should investigate the documentation on all of the options, but here are some tips & tricks to get you started:

- log_destination & log_directory (& log_filename): What you set these options to is not as important as knowing they can give you hints to determine where your database server is logging to. Best practice would be to try and make this as similar as possible across your servers. Note that in some cases, the init script starting your database may be customizing the log destination in the command line used to start the database, overriding what's in the configuration files (and making it so you'll get different behavior if you run pg_ctl manually instead of using the init script).
- log_min_error_statement: You should probably make sure this is at least on error, so that you will see any SQL commands which cause an error. should be the default on recent versions.
- log_min_duration_statement: Not necessary for everyday use, but this can generate logs of "slow queries" on your system.
- log_line_prefix: Appends information to the start of each line. A good generic recommendation is '%t:%r:%u@%d:[%p]: ' : %t=timestamp, %u=db user name, %r=host connecting from, %d=database connecting to, %p=PID of connection. It may not be obvious what the PID is useful at first, but it can be vital for trying to troubleshoot problems in the future so better to put in the logs from the start.
- log_statement: Choices of none, ddl, mod, all. Using all or mod in production would introduce overhead of the logging. The performance penalties for "all" would be significant if the workload is select intensive and less significant for write-intensive workloads. Turning the synchronous_commit to off would cause a more severe performance regression. DDL can sometime be helpful to discover rogue changes made outside of your recommend processes, by "cowboy DBAs" for example.

There are also external tools such pgbadger (https://pgbadger.darold.net/) that can analyze Postgres logs, see Monitoring for a comprehensive list.

default_statistics_target (http://www.postgresql.org/docs/current/static/runtime-config-query.html#GUC-DEFAULT-STATISTICS-TARGET)

The database software collects statistics about each of the tables in your database to decide how to execute queries against it. If you're not getting good execution query plans particularly on larger (or more varied) tables you should increase default_statistics_target then ANALYZE the database again (or wait for autovacuum to do it for you).

Increasing the default_statistics_target may be useful but the default value shipped with PostgreSQL is a reasonable starting point.

PostgreSQL 8.3 and earlier

In PostgreSQL 8.3 and earlier increasing the supplied default_statistics_target would often greatly improve query plans. The starting default_statistics_target value was raised from 10 to 100 in PostgreSQL 8.4. The maximum value for the parameter was also increased from 1000 to 10,000 in 8.4.

work_mem (http://www.postgresql.org/docs/current/static/runtime-config-resource.html#GUC-WORK-MEM)

If you do a lot of complex sorts, and have a lot of memory, then increasing the `work_mem` parameter allows PostgreSQL to do larger in-memory sorts which, unsurprisingly, will be faster than disk-based equivalents.

This size is applied to each and every sort done by each user, and complex queries can use multiple working memory sort buffers. Set it to 50MB, and have 30 users submitting queries, and you are soon using 1.5GB of real memory. Furthermore, if a query involves doing merge sorts of 8 tables, that requires 8 times work_mem. You need to consider what you set max_connections to in order to size this parameter correctly. This is a setting where data warehouse systems, where users are submitting very large queries, can readily make use of many gigabytes of memory.

log_temp_files (http://www.postgresql.org/docs/9.3/static/runtime-config-logging.html#GUC-LOG-TEMP-FILES) can be used to log sorts, hashes, and temp files which can be useful in figuring out if sorts are spilling to disk instead of fitting in memory. You can see sorts spilling to disk using `EXPLAIN ANALYZE` plans as well. For example, if you see a line like `Sort Method: external merge Disk: 7526kB` in the output of EXPLAIN ANALYZE, a `work_mem` of at least 8MB would keep the intermediate data in memory and likely improve the query response time (although it may take substantially more than 8MB to do the sort entirely in memory, as data on disk is stored in a more compact format).

maintenance_work_mem (http://www.postgresql.org/docs/current/static/runtime-config-resource.html#GUC-MAINTENANCE-WORK-MEM)

Specifies the maximum amount of memory to be used by maintenance operations, such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY. It defaults to 64 megabytes (64MB) since version 9.4. Since only one of these operations can be executed at a time by a database session, and an installation normally doesn't have many of them running concurrently, it's safe to set this value significantly larger than work_mem. Larger settings might improve performance for vacuuming and for restoring database dumps.

wal_sync_method wal_buffers (http://www.postgresql.org/docs/current/static/runtime-config-wal.html#GUC-WAL-SYNC-METHOD)

After every transaction, PostgreSQL forces a commit to disk out to its write-ahead log. This can be done a couple of ways, and on some platforms other options than the shipped wal_sync_method are considerably faster than the conservative default. open_sync is the most common non-default setting switched to, on platforms that support it but default to one of the fsync methods. See Tuning PostgreSQL WAL Synchronization (http://www.westnet.com/~gsmith/content/postgresql/TuningPGWAL.htm) for a lot of background on this topic. Note that open_sync writing is buggy on some platforms (such as Linux (http://lwn.net/Articles/350219/)), and you should (as always) do plenty of tests under a heavy write load to make sure that you haven't made your system less stable with this change. Reliable Writes contains more information on this topic.

wal_buffers defaults to 1/32 of the size of shared_buffers, with an upper limit of 16MB (reached when shared_buffers=512MB). Adjustments to the default are required much less often than in earlier PostgreSQL releases.

PostgreSQL 9.0 and earlier

Linux kernels starting with version 2.6.33 cause PostgreSQL versions before 9.0.2 to default to wal_sync_method=open_datasync; before kernel 2.6.32 the default picked was always fdatasync. This can cause a significant performance decrease when combined with small writes and/or small values for wal_buffers. PostgreSQL versions starting with 9.0.2 again default wal_sync_method to fdatasync when running on Linux.

On PostgreSQL 9.0 and earlier, increasing wal_buffers from its tiny default of a small number of kilobytes is helpful for write-heavy systems. Benchmarking generally suggests that just increasing to 1MB is enough for some large systems, and given the amount of RAM in modern servers allocating a full WAL segment (16MB, the useful upper-limit here) is reasonable.

Changing wal_buffers requires a database restart.

constraint_exclusion (http://www.postgresql.org/docs/current/static/runtime-config-query.html#GUC-CONSTRAINT-EXCLUSION)

`constraint_exclusion` now defaults to a new choice: `partition`. This will only enable constraint exclusion for partitioned tables which is the right thing to do in nearly all cases.

max_prepared_transactions (http://www.postgresql.org/docs/current/static/runtime-config-resource.html#GUC-MAX-PREPARED-TRANSACTIONS)

This setting is used for managing 2 phase commit. If you do not use two phase commit (and if you don't know what it is, you don't use it), then you can set this value to 0. That will save a little bit of shared memory. For database systems with a large number (at least hundreds) of concurrent connections, be aware that this setting also affects the number of available lock-slots in pg_locks, so you may want to leave it at the default setting. There is a formula for how much memory gets allocated in the docs (http://www.postgresql.org/docs/current/static/kernel-resources.html#SHARED-MEMORY-PARAMETERS) and in the default postgresql.conf.

Changing max_prepared_transactions requires a server restart.

synchronous_commit (http://www.postgresql.org/docs/current/static/runtime-config-wal.html#GUC-SYNCHRONOUS-COMMIT)

PostgreSQL can only safely use a write cache if it has a battery backup. See WAL reliability (http://www.postgresql.org/docs/current/static/wal-reliability.html) for an essential introduction to this topic. No, really; go read that right now, it's vital to understand that if you want your database to work right.

You may be limited to approximately 100 transaction commits per second per client in situations where you don't have such a durable write cache (and perhaps only 500/second even with lots of clients).

For situations where a small amount of data loss is acceptable in return for a large boost in how many updates you can do to the database per second, consider switching synchronous commit off. This is particularly useful in the situation where you do not have a battery-backed write cache on your disk controller, because you could potentially get thousands of commits per second instead of just a few hundred.

For obsolete versions of PostgreSQL, you may find people recommending that you set *fsync=off* to speed up writes on busy systems. This is dangerous--a power loss could result in your database getting corrupted and not able to start again. Synchronous commit doesn't introduce the risk of *corruption*, which is really bad, just some risk of data *loss*.

random_page_cost (http://www.postgresql.org/docs/current/static/runtime-config-query.html#GUC-RANDOM-PAGE-COST)

This setting suggests to the optimizer how long it will take your disks to seek to a random disk page, as a multiple of how long a sequential read (with a cost of 1.0) takes. If you have particularly fast disks, as commonly found with RAID arrays of SCSI disks, it may be appropriate to lower random_page_cost, which will encourage the query optimizer to use random access index scans. Some feel that 4.0 is always too large on current hardware; it's not unusual for administrators to standardize on always setting this between 2.0 and 3.0 instead. In some cases that behavior is a holdover from earlier PostgreSQL versions where having random_page_cost too high was more likely to screw up plan optimization than it is now (and setting at or below 2.0 was regularly necessary). Since these cost estimates are just that--estimates--it shouldn't hurt to try lower values.

But this not where you should start to search for plan problems. Note that random_page_cost is pretty far down this list (at the end in fact). If you are getting bad plans, this shouldn't be the first thing you look at, even though lowering this value may be effective. Instead, you should start by making sure autovacuum is working properly, that you are collecting enough statistics, and that you have correctly sized the memory parameters for your server--all the things gone over above. After you've done all those much more important things, if you're still getting bad plans *then* you should see if lowering random_page_cost is still useful.

Retrieved from "https://wiki.postgresql.org/index.php?title=Tuning_Your_PostgreSQL_Server&oldid=35462"

- This page was last edited on 14 October 2020, at 11:57.