**Top 10 Configuration Parameters to Tune PostgreSQL for Better Performance**

The conventional wisdom is that, if you do not know what you are doing, it’s better off using the default configuration. Databases usually perform well with the default parameter values and premature performance optimisation often results in worse performance. On the other hand, Postgres is a different story. Postgres default configuration is set to work everywhere, including tiny machines like Raspberry Pi. Therefore, the most default values are too conservative for normal usages. It is recommended to tweak with the configuration so that it performs well for your use case. With PostgreSQL, a few small config changes often result in a huge performance improvement.

Having said that, Postgres has more than 200 configuration parameters. Tuning Postgres can be confusing. In this post, I will present top 10 parameters that you should know. By default, these parameters are set way too conservative. Changing these values to be more aggressive is likely to increase the performance.

1. **max\_connection**

If you have a lot of concurrent users and are getting connection rejected, the solution is not so simple as to increase this value. If it is set to high, you will see high latency.

Postgres rejects connections over the value. It is for limiting the number of connection to prevent performance degradation from too many connections. Default is 100. In Postgres, each connection creates a process and consume memory. This means Postgres will suffer from too many connections. Especially, the latency will increase.

If you have an application with many concurrent users, it is sometimes better to deny connections entirely than degrade the performance. If you have too many connections, the solution is to set up [PgBouncer connection pooling](https://www.enterprisedb.com/resources/tutorials/how-set-pgbouncer-connection-pooling-postgres-plus-standard-server). The excessive connections will not be rejected. Check out [this video](https://www.youtube.com/watch?v=IFIXpm73qtk) (about 15 minutes in), explaining max\_connection parameters in details.

1. **shared\_buffer**

The default is too low for most modern hardware. More buffers will be needed for many concurrent queries and many CPUs like web applications or OLAP.

The conventional wisdom is to set it to 1/4th of RAM and effective\_cache\_size 3/4th . However, 1/4th may be too low for small machine and too high for big machines. Start with 1/4th and need to try different values out.

1. **effective\_cache\_size**

Increasing this value will increase the amount of memory available to cache data. The larger value makes database more likely to use index. It should be more than 50% of RAM. shared\_buffer plus the amount of OS cache available is a good place to start.

1. **work\_mem**

This can changed by any user and does not require to reboot the database. The parameter can be allocated per role or task. The value determines the maximum local process memory used for operations such as sort and joins. The optimal value would be vastly different between OLTP, OLAP and DW (small to very large, e.g. like gigabytes for complex queries going for hours. OLTP 10mb, OLAP 100mb, DW 1gb).

Lower it if you have too many connections (e.g. for web applications) and raise it for large querry. If you are swapping Ram, you have too much work\_mem and logging temp files means not enough work\_memory.

1. **maintenance\_work\_mem**

Used for maintenance for vacuuming, index creation, checking foreign keys. Generally speaking, it should be raised. For vacumming, the faster it completes, better.

As it is used for index creation, we should raise it if you are doing a lot of bulk load.

Raise it to 256MB to 1GB for large database.

1. **random\_page\_cost**

The ration of random\_page\_cost and seq\_page\_cost means how much more expensive to read randomly compared to reading sequentially from the disk. The default is random\_page\_cost=4.0 and seq\_page\_cost=1.0. This means random read is 4 times more expensive than sequential read. This is not true for SSD. It is recommended to set the value to 2 or 3 for SSD.

1. **{min, max}\_wal\_size**

When it is too low, it will cause excessive checkpoints. The recommendation is to make max\_wal\_size large enough to rarely reach it. Make min at least 1GB and max several GB up to 50 to 100 GB. Along with changing this value, you also need to increase **checkpoint\_timeout** to get the benefit of increased max\_wal\_size. By raising this value, we can spread checkpoint further apart. To understand checkpoint, I recommend you to read this excellent blog ([Basic of Tuning Checkpoint](https://blog.2ndquadrant.com/basics-of-tuning-checkpoints/)).

1. **autovacuum\_vacuum\_scale\_factor**

Vacuuming is the process to clear out dead tuples created by deletion and insertions. PostgreSQL has a mechanism to automatically vacuum the tables depending of the threshold value specified.

Generally speaking, default values that control auto vacuum are too conservative. You can make them more aggressive. Sometimes I hear people recommending to turn off auto vacuum and do vacuum manually. Especially autovacuum is not so good for batch writes (e.g. data warehouse). It is better to do manual vaccum for bulk load. However, it is better to use auto vacuum if you do not fully understand manual vacuuming process. So, the rule of thumb is not to turn off auto vacuum if you are not confident about vacuuming by yourself.

**Autovacuum\_vacuum\_scale\_factor** determines the percentage of dead tuples in the tables before auto vacuum happens. When it is set to 0.2, auto vacuum happens when 20% of the table contains dead tuples.

20% sounds fine with a small to medium size tables. But, it is too much for a large table (e.g. 200GB for 1TB table). This value can be set per table. If not, vacuuming large tables not often enough is worse than vacuuming small tables too often.

You can use this value in conjunction with **autovacuum\_vacuum\_threashold**, which determines the number of dead tuples for auto vacuuming. For example, autovacuum\_vacuum\_threashold=1000 means auto vacuuming happens when 1000 rows are dead tuples.

To further understand auto vacuuming, you should read [Autovacuum Tuning Basics](https://blog.2ndquadrant.com/autovacuum-tuning-basics/#PostgreSQL%20Performance%20Tuning).

1. **autovacuum\_max\_workers**

Use more **autovacuum\_max\_workers** if you have many cores. Default is 0. It is better to have the same number as the number of cores so that we can do this concurrently. The faster vacuuming finishes, the better database performance will be. In conjunction with this parameter, you should set **multiple\_autovacuum\_workers**. This should not be more than ½ cores.

For other auto vacuum tuning parameters, you can raise **autovacuum\_cost\_limit** significantly and reduce **autovacuum\_vacuum\_cost\_delay** significantly. The default values for these parameters are noticeably conservative.

1. **synchronous\_commit**

With the default configuration, commit happens after the success response returned to the client. Turning this parameter on means commit happens when it is executed in the database server regardless of getting the success message or not. If data integrity is super important than response time, turn it on. The general recommendation is to keep it off for safety.

**REFERENCE**

[PostgreSQL Configuration for Humans](https://www.youtube.com/watch?v=IFIXpm73qtk)

[Tuning Your PostgreSQL Server](https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server)

[Five Steps to PostgreSQL Performance](https://www.slideshare.net/pgconf/five-steps-perform2009)

[Performance Tuning PostgreSQL](https://www.revsys.com/writings/postgresql-performance.html)

[Basic of Tuning Checkpoint](https://blog.2ndquadrant.com/basics-of-tuning-checkpoints/)

[Autovacuum Tuning Basics](https://blog.2ndquadrant.com/autovacuum-tuning-basics/#PostgreSQL%20Performance%20Tuning)