10 Rules for Managing PostgreSQL®

Introduction

PostgreSQL® is one of the most successful open source projects in existence. This is because it provides a robust, thoughtful, well-maintained solution to the common need of data storage and retrieval. It has found adoption at the highest levels of business and academia, and has a foothold in nearly every vertical market.

But, PostgreSQL can also be a bit daunting. It has grown from a small university project managed by a handful of graduate students into a world-class database with all of the advanced features such a project demands.

It becomes harder and harder to just "dive in" to the PostgreSQL ecosystem with each passing year. More features, utilities, enhancements, and concepts enter the fray as fast as a worldwide community can think of them.

It is not difficult to use PostgreSQL at a simple level. Just install it, learn 5 SQL commands, and you're off to the races. But to use it in an enterprise environment is a different story.

In this white paper, we will cover 10 rules that will help you perfect your PostgreSQL installation and get ahead of the curve.

instaclustr
Now part of **Spot by NetApp**

1. Multi-Version Concurrency Control

We'll start right off with the elephant in the refrigerator. PostgreSQL is a highly concurrent database system. As soon as the second user desires to connect to the database, some type of management is required to distinguish users and provide concurrent services.

PostgreSQL has picked a strategy called Multi-Version Concurrency Control, which we will henceforth call MVCC. The basic operation of this method is to keep a copy of every row in the database which has ever existed, thus providing a consistent view of the data to each connection (called an observer in the documentation).

At some point, there will no longer be any connection that needs access to archive rows in the past. These rows are now considered "bloat" or "cruft". The process of getting rid of this dead weight is left to a garbage collection process known as **autovacuum**.

It is critical that you get a basic understanding of why and how the garbage gets created, and how to manage the process of getting rid of it.

Key parameters to understand are:

- max_background_workers
- autovacuum_max_workers
- autovacuum_cost_limit
- autovacuum_vacuum_scale_factor
- autovacuum_vacuum_threshold
- autovacuum_analyze_scale_factor

Also please remember that any operation in PostgreSQL that creates bloat for the table data also creates the same bloat in indexes, blob storage, and tracking tables for the same reason.

Balancing the timing and cleanup operations to your workload is a central task of PostgreSQL maintenance.

2. Hardware Requirements

PostgreSQL scalability can be viewed from two perspectives. Those are vertical and horizontal. Vertical means that the installation of a single hardware system is pushed to its physical limits. Horizontal is the idea of using multiple hardware systems for a single PostgreSQL service provision.

The goal of the PostgreSQL development group for much of the '80s and '90s was to provide as much vertical scalability as possible. This focus has shifted in the last few years to horizontal scalability.

Vertical Scalability

The PostgreSQL project has decided from a very early stage to be generally hardware-agnostic. This means that the project makes no use of hardware features such as GPU computation, ASIC encryption, or hardware-supplied compression.

Vertical scalability is achieved solely through solid mathematical algorithm choices in the query engine and efficient management of the memory and file system blocks. No attempt is made to associate these physical actions with any specific piece of hardware.

This can be seen as both a positive and negative attribute. The positive side is that PostgreSQL can provide services on a very wide array of hardware platforms, from cell phones and embedded devices all the way up to dedicated servers. It also does not have any specific vendor requirements or minimum system hardware.

The negative side of this issue is that it is left as an exercise to the administrator to scale PostgreSQL for any particular hardware. Because a bewildering array of hardware options are available, the question becomes very complex very quickly.

So, how can we make hardware choices if there are virtually no guidelines? To compound the question, how would we do it when we don't even know the access pattern of our users to the data?

We take advantage of hardware agnosticism with virtualism. That is, we use systems that abstract the resources from the operating system. By doing this, we can add resources at will and PostgreSQL will mostly just do the right thing with the newly available capacity.

Or we have a database administrator around with the required experience level to get to an initial system architecture based on the hardware availability at the time of deployment. The settings to pay the most attention to for vertical scalability are:

- work_mem
- maintenance_work_mem
- shared_buffers

Horizontal Scalability

The PostgreSQL project understands that there is only so far that a vertically scaled installation can go. When our architecture reaches that point, there are several options on the table.

Partitioning: Dividing the data into multiple tables inside the individual system level, and treating that table set as if it were an individual table.

Sharding: Dividing the data into multiple tables across a cluster of machines so that CPU power is added for every shard.

Geo-location: Putting the data that matters on the servers physically closest to the user.

Replication: Putting copies of the data everywhere we need them.

3. Replication and Redundancy

This brings us to our next topic, replication. The minimum system for online transaction processes (OLTP) system is four machines.

PostgreSQL may only take data modifying statements on a single host, at least when we're speaking about vertical scalability. This ability to make modifications makes this host the primary.

The first replica (fancy name for copy) of that primary host is a failover target in case the primary is incapacitated. The second replica is to have another system available when the failure situation occurs. It is advisable that at least one of these replicants resides in a geographically different region.

The fourth machine is where we put all of the tools and utilities. This includes products like patroni, repmgr, backup tools, and backups.

There are many forms of replication in PostgreSQL that have been developed over time for different reasons.

The first and simplest is **physical streaming replication**, which is a fancy name for copying the file system continuously to another host. This system is simple to set up, easy to understand, and fairly fast. It does not provide any selection criteria. That is, the entire system is blindly copied to a destination without any regard to contents.

The next replication type is **logical streaming replication**. This system uses a publisher and subscriber model. The primary host makes a publication with the administrator's choice of tables. Any number of replicants may subscribe to that publication. This system is much more flexible, very intelligent, and slow. It also requires a lot of CPU power based on that intelligence.

There are also external replication systems such as slony, bucardo, londiste, pgsync, etc. These replication systems are all based on tracking changes in the database, and then repeating those changes elsewhere using an external agent that performs the actual replication. They are mostly unnecessary in modern systems, except possibly in live streaming upgrade scenarios.

4. Security

PostgreSQL relies primarily on the host-based authentication model (HBA) for authentication. That is, it only provides a very rudimentary security based on MD5 or SCRAM within the product. Otherwise, it delegates the authentication to some other service. This list of services may include the operating system, GSSAPI, kerberos, RADIUS, LDAP, AD, and many others.

Security at rest is provided on a per-column basis using pgcrypto library, which is a wrapper for libcrypt, which in turn uses libssh and libssl. For installations that require complete encryption of the entire data structure, the exercise is left to the operating system and external data storage provider.

5. Locking and Blocking

In PostgreSQL, there is a fundamental difference between locking and blocking. There is a table in the system catalog called pg_locks where the act of cooperation for the aforementioned concurrency occurs. It is the not-so-humble opinion of this author that the table should be named pg_concurrency or pg_semaphore, as the name pg_locks is a bit misleading.

The vast majority of the entries in the pg_locks table do not in fact lock anything at all. They are rows for indicating that data is still being used, so that the background processes (especially the autovacuum process) do not destroy the data while it is being accessed. It also shows some precision of what type of process is accessing the data.

Because of this, it is very easy to look at the large number of rows in this table, and assume that operations are blocked. Actually, quite the opposite is true. There is a lot of communication across processes going on there to ensure that nothing is blocked.

The rows that are actually blocking something will have Exclusive somewhere in the lock type name. Even then, it may not be critical. PostgreSQL has a lock type AccessSharedExclusive for interruptible system services, which would take a few milliseconds to shut down and return control to the user processes. This would make for a slight queueing delay, but does not actually block user interaction.

The only locks to look out for are AccessExclusiveLock, which are actually blocking, but are also used as sparingly as possible for that very reason. Much work in the PostgreSQL community goes into reducing the dwell time of such locks to the point of having vanishingly small lifetimes.

6. Extensions

PostgreSQL has a unique feature among databases. This feature allows you to extend the database functionality itself using functions. This feature justifies its own article, but for now we'll just talk about the basics.

The administrator can create operators, data types, conversions, and just about anything else that exists in the database. Many features of the database started out as an extension or part of an extension in another project. But then they went on to become a core part of the PostgreSQL project.

These extensions include:

- GiST and GiN indexes
- Bloom indexes
- JSON
- XML
- …and some others.

There are no programming limits to what is possible with extensions. Any part of the query planner, file system, maintenance, and user functions can be enhanced using this system.

There are hundreds of extensions available on the pgxn website. PostgreSQL uses a tool called pgxnclient. This tool works in much the same way as gem for Ruby, cpan for Perl, or pip for Python. It is available in most major system package managers such as yum or apt.

7. Languages

Okay, this is really a continuation of the extensions topic above for a specific type of extension, but it's so amazing that it is worth noting separately.

PostgreSQL has the ability (and again another unique one) to install the language of your choice for internal functions.

This may be any one of:

- Java
- Javascript
- Perl
- Python
- Ruby
- C
- Bash
- PHP
- R
- Scheme
- Tcl
- C#
- …and about a hundred others.

If you are worried about having the personnel resources for PostgreSQL, just ask around which skills are already on deck. Plug those into PostgreSQL, and you have just trained your staff to use PostgreSQL.

8. Backup and Point-In-Time Recovery

The PostgreSQL concurrency model allows for hot backups. For the uninitiated, **hot backups** are when a copy is made of the database while still running and processing queries.

PostgreSQL also provides a log of every transaction as it occurs. This log can be applied to a new installation after a restoration process, such that PostgreSQL can catch up to the current transaction position of the primary.

These two techniques taken together are called **point-in-time recovery**. This allows PostgreSQL to make regular backups without service interruption, while also maintaining the ability to recover up to the point of failure in a disaster scenario.

Over time, this system has been appropriated to provide a simple replication model. The reproduction of a primary is called **physical streaming replication**. It allows the user to create a secondary system that is concurrent with the primary system. The secondary system may also provide query results, thus making PostgreSQL extremely attractive for distributed systems with high query demands.

9. Internationalism

PostgreSQL started life as a university project with a wide appeal to academia. Many of these students were not in the United States. Indeed, most of them came from Europe and Asia. PostgreSQL was built from the ground up with this geographic distribution as a fact of life.

Unicode and UTF-8 were not afterthoughts of implementation in PostgreSQL. They have been fully supported in every release since the beginning. Using the operating system for internationalization and localization have been at the core of PostgreSQL for decades. Sorting, indexing, encoding, and translation are all just part of the core.

10. Monitoring

The PostgreSQL project does not provide any monitoring tools for PostgreSQL directly. These are supplied by outside vendors, and there are many to choose from.

Some of the more popular ones are Nagios (iCinga), Graphite, Data Dog, Zabbix, cacti, munin, SolarWinds, AppOptics, Paessler, OmniDB, and a whole host of others that are loosely based on the check_postgres.pl perl script from Nagios.

Monitoring does not necessarily include alerting in the open source world, so in some instances you may need additional integration from products such as PagerDuty, Prometheus, Bosun, Grafana, or some other such tool.

Conclusion

PostgreSQL is a powerful and extensible technology that can solve a lot of problems in all different kinds of environments. Use cases for PostgreSQL are growing exponentially over time. That is, the number of applications that PostgreSQL can target is becoming greater as time goes by, and (more importantly) the number of applications that it can't target is going down.

But PostgreSQL is also a complex technology to configure, deploy, and manage. Observing best practices—like those outlined in this white paper—will help you have the greatest success on your own PostgreSQL journey.

About Instaclustr

Instaclustr helps organizations deliver applications at scale through its managed platform for open source technologies such as **Apache Cassandra®**, **Apache Kafka®**, **Apache Spark™**, **Redis™**, **OpenSearch®**, **PostgreSQL®**, and **Cadence®**.

Instaclustr combines a complete data infrastructure environment with hands-on technology expertise to ensure ongoing performance and optimization. By removing the infrastructure complexity, we enable companies to focus internal development and operational resources on building cutting edge customer-facing applications at lower cost. Instaclustr customers include some of the largest and most innovative Fortune 500 companies.