Logging Basics for PostgreSQL

In PostgreSQL, managing logs serves as a vital tool for identifying and resolving issues within your application and database. However, navigating through logs can be overwhelming due to the volume of information they contain. To address this, it's essential to implement a well-defined logs management strategy.

Customizing PostgreSQL logs involves adjusting various parameters to suit your specific needs. Each organization may have unique requirements for logging, depending on factors such as the type of data stored and compliance standards.

In this article, we will explain parameters used to customize logs in PostgreSQL. Furthermore, we describe how to record queries in PostgreSQL and finally recommend a tool for managing PostgreSQL logs at granular level.

How to enable logs in PostgreSQL?

To enable logging in PostgreSQL, you'll need to adjust the configuration settings in the postgresql.conf file. Here's how you can do it:

Log Destination

PostgreSQL supports various methods for logging server messages, including stderr, csvlog, jsonlog, and syslog. You can specify multiple destinations separated by commas. For example:

```
log_destination = 'stderr,csvlog'Copy to Clipboard
```

Logging Collector

Enabling the logging collector background process captures log messages and redirects them into log files. This parameter can only be set at server start.

```
logging_collector = onCopy to Clipboard
```

Log Directory

When the logging collector is enabled, this parameter determines the directory where log files will be created. It can be an absolute path or relative to the cluster data directory.

```
log_directory = '/mnt/data/logs/pg_log'Copy to Clipboard
```

Log Filename

Sets the file names of the created log files when the logging collector is enabled. By default, it follows the pattern:

```
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'Copy to Clipboard
```

For daily logging, you can change it to:

```
log_filename = 'postgresql-%Y-%m-%d.log'Copy to Clipboard
```

How to set log rotation?

log_rotation_age

The log_rotation_age parameter controls automatic log file rotation based on time, specifying how often log files should rotate. For instance, setting it to 1d means log rotation will occur daily. If set to 0, automatic rotation based on time is disabled.

Log_rotation_size

The log_rotation_size parameter determines automatic log file rotation based on file size, specifying the size threshold at which rotation occurs. For example, setting it to 10MB means rotation happens when the log file reaches 10 megabytes. Setting it to 0 disables automatic rotation based on size.

Log_truncate_on_rotation

The log_truncate_on_rotation parameter, when enabled alongside logging_collector, instructs PostgreSQL to truncate (overwrite) any existing log file with the same name when rotation occurs. For instance, with a log_filename pattern like postgresql-%H.log, this setting would generate twenty-four hourly log files, and then overwrite them instead of appending.

What logging levels are available in PostgreSQL?

In PostgreSQL, logging levels determine the severity of messages that are recorded in the log files. Here's a description of each logging level available:

1. **Debug1…Debug5**: The most detailed level, providing verbose information for troubleshooting specific issues. Mostly used by developers.
2. **Info**: Offers general informational messages about PostgreSQL's operations, such as startup, shutdown, or task completion. For example: output from VACUUM VERBOSE. **INFO** level messages are always sent to the client.
3. **Notice**: Indicates non-error conditions that don't require immediate attention, providing status updates about ongoing operations.
4. **Warning**: Highlights potential issues or conditions that may lead to problems if not addressed, serving as warnings to administrators or developers.
5. **Error**: Indicates errors within PostgreSQL that caused the current command to abort.
6. **Fatal**: Indicates errors causing PostgreSQL current session to terminate abruptly.
7. **Panic**: The most severe level, indicating catastrophic errors that force PostgreSQL to close all database sessions immediately followed with a shut down. PostgreSQL will restart in recovery mode.

By adjusting the **log_min_messages** and **log_min_error_statement** parameters in the PostgreSQL configuration file, you can control which messages are recorded in the log files based on their severity.

How to log slow queries?

The **log_min_duration_statement** parameter logs SQL statements that run for at least a specified time. For example, if you set it to 100ms, then all SQL statements that run 100ms or longer will be logged. Enabling this parameter can be helpful in tracking down unoptimized queries in your applications.

What information can we log?

Based on the mentioned parameters, the following information can be logged:

- Logging Server connections and disconnections(**log_connections** and **log_disconnections**)
- Performance Logging which includes:
    1. Logging autovacuum activity (**log_autovacuum_min_duration**)

2. Checkpoints (**log_checkpoints**)
3. Logging the duration of queries (**log_duration**)
4. Monitoring lock waits(**log_lock_waits**)
5. Standby recovery conflict waits(**log_recovery_conflict_waits**)

- Specifying the information included in log lines, such as application name, user name, database name, timestamp, and session ID. (**log_line_prefix**)
- Configuring which types of SQL statements to log(**log_statement**), The log_statement parameter in PostgreSQL configuration specifies which SQL statements are logged. Only superusers can change it and It has the following values:
-

1. **none** – no logging(default)
2. **ddl** – logging all data definition operations like CREATE, DROP, and ALTER
3. **mod** – same as ddl, and also logs operations that modify data, like INSERT, UPDATE and DELETE
4. **all** – logs all SQL queries.

- Logging of temporary files for disk based operations, specifying the minimum size for logging.(**log_temp_files**)
  For moderately busy PostgreSQL instances, setting this parameter to all is suitable but make sure to check the log file size. Certain monitoring/reporting applications such as pgBadger also expect this parameter to be set to **all**.

What is the best setting for log_line_prefix?

To enhance the readability and parsing of logs, you can prefix each variable with a key as follows:

- Time of the event (without milliseconds): **%t**
- Remote client name or IP address: **%h**
- User name: **%u**
- Database: **%d**
- Application name: **%a**
- Process ID: **%p**

```
log_line_prefix = 'time=%t, pid=%p, user=%u, db=%d, clientIP=%h, appname=%a'Copy to Clipboard
```

Log Output

```
time=2024-04-25 01:01:45.613 UTC pid=6978 user=postgres db=postgres clientIP=10.0.0.5
appname=psql LOG:  statement: SELECT count(*) from test;Copy to Clipboard
```

What are the benefits of pgaudit extension?

- pgAudit provides more advanced and customizable auditing features compared to built-in PostgreSQL logging. It allows for detailed tracking of database activities, including **SELECT, INSERT, UPDATE, DELETE** and **DDL** commands.
- With pgAudit, you can define custom audit policies to specify exactly which database actions should be logged and under what conditions. This level of granularity enables organizations to tailor auditing to meet specific compliance requirements or security needs.
- pgAudit logs the details of each operation, with its full internals, using a structured output that is suitable for audit searches.
  For example, here is a DO statement, and the log generated by **log_statement=all** compared to the log generated by pgAudit. This code was shared in the pgAudit documentation:

```
DO $$

BEGIN

EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';

END $$;
```

```
Standard logging will give you this:

LOG:  statement: DO $$

BEGIN

EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';

END $$;

For the same input, it will produce this output in the log with pgaudit:

AUDIT: SESSION,33,1,FUNCTION,DO,,,"DO $$

BEGIN

EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';

END $$;"

AUDIT: SESSION,33,2,DDL,CREATE TABLE,TABLE,public.important_table,CREATE TABLE important_table
(id INT);Copy to Clipboard
```

Not only is the DO block logged, but statement 2 contains the full text of the CREATE TABLE with the statement type, object type, and fully-qualified name to make searches easy.

- pgAudit offers **Object audit logging**, which specifically logs statements impacting a particular object or relation. Supported commands include **SELECT, INSERT, UPDATE, and DELETE**, contributing to the reduction of log volume generated. Object-level audit logging is implemented through the **roles** system, where the role must have necessary grants on the object to capture logs effectively. This feature ensures targeted monitoring of object-related activities, providing valuable insights while efficiently managing log data.
- pgAudit provides PostgreSQL users with capability to produce audit logs often required to comply with government, financial, or ISO certifications.
