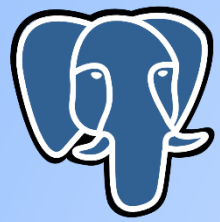
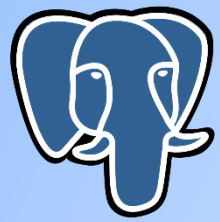


PostgreSQL

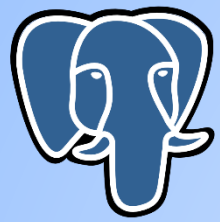
Module 10

Monitoring and Diagnosis



Module Overview

- Checking Whether A User Is Connected
- Checking Which Queries Are Running
- Checking Which Queries Are Active Or Blocked
- Knowing Who Is Blocking A Query
- Finding Slow SQL Statements
- Killing A Specific Session
- Knowing Whether Anybody Is Using A Specific Table
- Analyzing The Real-time Performance Of Your Queries

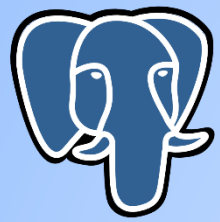


Introduction

- Databases are not isolated entities. They live on computer hardware using CPUs, RAM, and disk subsystems. databases themselves may need network resources to function
- Monitoring only the database is not enough
- Know details like:
  - Is the database host available? Does it accept connections?
  - Is there enough RAM available for the most common tasks? How much of it is left?
  - Is there enough disk space available? When will it run out of disk space?
  - When did the disk usage start changing rapidly?
- Tools – Cacti, Munin, Nagios

PostgreSQL Monitoring Tools

- There are several tools available as front-end to PostgreSQL:
  - **pgAgent**
    - pgAgent is a job scheduler for PostgreSQL
  - **pg_statsinfo**
    - pg_statsinfo in the monitored DB on behalf of the existence of the form, pg_statsinfo regularly collected snaoshot information and stored in the warehouse
  - **pgCluu**
    - pgCluu is a PostgreSQL performances monitoring and auditing tool
  - **phpPgAdmin**
    - It is a web-based administration tool for PostgreSQL written in PHP
  - **pgFouine**
    - It is a log analyzer which creates reports from PostgreSQL log files

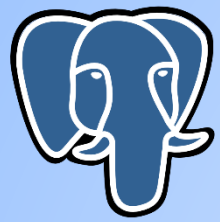


Checking Whether A User Is Connected

- Whether a certain database user is currently connected to the database.
  - Issue the following query to see whether the bob user is connected:
    - **SELECT datname FROM pg_stat_activity WHERE usename = 'bob';**
- **pg_stat_activity** system view keeps track of all running PostgreSQL backends.
- Several different processes may connect as the same database user. In that case, you may actually want to know whether there is a connection from a specific computer.
  - **SELECT datname, usename, client_addr, client_port, application_name FROM pg_stat_activity;**
- The client_addr and client_port parameters help you look up the exact computer and even the process on that computer that has connected to the specific database.

Checking Which Queries Are Running

- This can be done either in the postgresql.conf file or by the superuser, using the following SQL statement:
  - **SET track_activities = on**
  - When track_activities = on is set, PostgreSQL collects data about all running queries
  - Command to see the active records:
    - **SELECT datname, usename, state, query FROM pg_stat_activity**
  - To get active queries only, limit your selection to only those records that have state set to active:



Checking Which Queries Are Active Or Blocked

- Show you how to know whether a query is actually running or it is waiting for another query.

```
SELECT datname , username , wait_event_type , wait_event , query FROM pg_stat_activity WHERE wait_event IS NOT NULL;
```

- The pg_stat_activity system view has a Boolean field named waiting. This field indicates that a certain backend is waiting on a system lock.
- As the waiting column is already Boolean, you can safely omit the = true part from the query and simply write the following:
  - **SELECT datname, username, query FROM pg_stat_activity WHERE waiting;**

Knowing Who Is Blocking A Query

- Once you have found out that a query is blocked, you need to know who or what is blocking them

```
SELECT datname , usename , wait_event_type , wait_event ,
pg_blocking_pids(pid) AS blocked_by , query FROM pg_stat_activity
WHERE wait_event IS NOT NULL;
```

- This returns the process ID, user, current query about both blocked and blocking backends, and the fully qualified name of the table that causes the blocking.

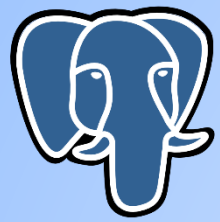
```
SELECT
    w.query AS waiting_query,
    w.pid AS waiting_pid,
    w.usename AS waiting_user,
    l.query AS locking_query,
    l.pid AS locking_pid,
    l.usename AS locking_user,
    t.schemaname || '.' || t.relname AS tablename
 FROM pg_stat_activity w
       JOIN pg_locks l1 ON w.pid = l1.pid AND NOT l1.granted
       JOIN pg_locks l2 ON l1.relation = l2.relation AND l2.granted
       JOIN pg_stat_activity l ON l2.pid = l.pid
       JOIN pg_stat_user_tables t ON l1.relation = t.relid
 WHERE w.waiting;
```

Killing A Specific Session

- Once you have figured out the backend you need to kill, use the function named pg_terminate_backend(pid) to kill it.
- Trying to cancel the query first:
  - Try **pg_cancel_backend(pid)**, a milder version of **pg_terminate_backend(pid).**
  - The difference between these two is that pg_cancel_backend() just cancels the current query, whereas pg_terminate_backend() really kills the backend.
- If pg_terminate_backend(pid) fails to kill the backend, another option— sending SIGKILL to the offending backend.
  - **kill -9 <backend_pid>**
- Using statement_timeout to clean up queries that take too long to run

```
SET statement_timeout TO '3 s';

Select sleep(10);
```



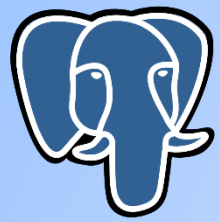
Finding Slow SQL Statements

- Several ways to find the statements that are either slow or cause the database as a whole to slow down.
- Set up logging queries over 10 seconds by defining the following in postgresql.conf:
  - **log_min_duration_statement = 10000;**
- Spot long queries is to look them up in the pg_stat_activity system view by repeatedly running this query:
  - **SELECT now() - query_start AS running_for, query FROM pg_stat_activity WHERE state = 'active' ORDER BY 1 DESC LIMIT 5;**

Knowing Whether Anybody Is Using A Specific Table

- Sometimes are in doubt whether some obscure table is used any more or it is left over from old times and just takes up space
- To see whether a table is currently in active use ,run the following query on the database you plan to inspect:
  - **SELECT * FROM pg_stat_user_tables;**
  - The pg_stat_user_tables view shows the current statistics for table usage

```
select * from pg_stat_user_tables n
join tmp_stat_user_tables t
on n.relid=t.relid
and (n.seq_scan,n.idx_scan,n.n_tup_ins,n.n_tup_upd,n.n_tup_del) <>
(t.seq_scan,t.idx_scan,t.n_tup_ins,t.n_tup_upd,t.n_tup_del);
```

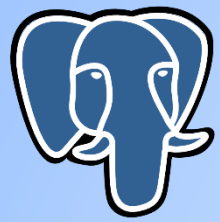


PostgreSQL

Producing a daily summary of log file errors

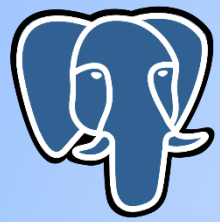
- PostgreSQL has a hierarchy of log entries that ranges from DEBUG messages to PANIC

- To the administrator, the following three error levels are of great importance:
  - ERROR - ERROR is used for problems such as syntax errors, permission-related problems
  - FATAL - FATAL is more scary than ERROR, messages such as could not allocate memory for shared memory name or unexpected walreceiver state
  - PANIC - that something is really, really wrong. Like lock table corrupted



Producing a daily summary of log file errors

- It makes sense to inspect the log to see what is going on
- You can have a logging configuration of your PostgreSQL server
  - log_destination = syslog
  - log_statement = ddl
  - log_min_duration_statement = 1000
  - log_min_messages = info
  - log_checkpoints = on
  - log_lock_waits = on



Analyzing The Real-time Performance Of Your Queries

- The pg_stat_statements extension adds the capability to track execution statistics of queries that are run in a database, including the number of calls, total execution time, total number of returned rows, as well as internal information on memory and I/O access.
- The pg_stat_statements module is available as a contrib module of PostgreSQL. The extension must be installed as a superuser in the desired databases.
- library in the postgresql.conf file, as follows:
  - shared_preload_libraries = 'pg_stat_statements'
- You can start by retrieving the list of the most frequent queries:
  - **SELECT query FROM pg_stat_statements ORDER BY calls DESC;**