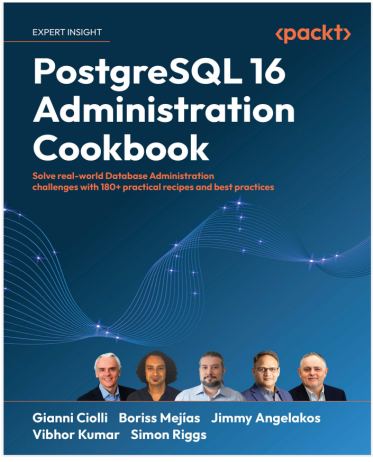
# Database Administration

## Lecture 04: Performance Monitoring and Diagnosis.

Ciolli et al.

11 de agosto de 2025

# Database Administration: Performance Monitoring and Diagnosis.




Content has been extracted from *PostgreSQL 16 Administration Cookbook* by Ciolli, Mejías, Angelakos, Kumar & Riggs, 2023. Visit packtpub.com.

# Plan

# Why check the server log?

- First stop when diagnosing server issues; review it regularly.
- Records server messages with timestamps and PIDs, e.g.:
  - `2023-09-01 19:37:41 GMT [2506-1] LOG: database system is ready to accept connections`

# Where the log can live

- Under the data directory (one or more files).
- In another filesystem directory.
- Redirected to `syslog` (or Windows Event Log).
- There may be *no* server log configured ⇒ configure it!

# Default locations by platform

- Debian/Ubuntu: `/var/log/postgresql`
- Red Hat / RHEL / CentOS / Fedora: `/var/lib/pgsql/data/pg_log`
- TPA deployments: `/var/log/postgres/postgres.log` and `syslog`
- Windows: Windows Event Log

# File naming and rotation

- Current file: `postgresql-`*`MAJOR`*` -`*`SERVER`*`.log` (e.g., `postgresql-14-main.log`).
- Older files: numeric suffix (`.1`, `.2`, ...) — higher number = older file.
- Rotation via OS `logrotate` (Debian/Ubuntu default) or PostgreSQL settings `log_rotation_age`, `log_rotation_size` (when using the logging collector).

# Severity levels (mapping examples)

| PostgreSQL severity | Meaning | Syslog severity | Windows Event Log |
|---|---|---|---|
| DEBUG 1 to DEBUG 5 | This comprises the internal diagnostics. | DEBUG | INFORMATION |
| INFO | This is the command output for the user. | INFO | INFORMATION |
| NOTICE | This is helpful information. | NOTICE | INFORMATION |
| WARNING | This warns of likely problems. | NOTICE | WARNING |
| ERROR | This is the current command that is aborted. | WARNING | ERROR |
| LOG | This is useful for sysadmins. | INFO | INFORMATION |
| FATAL | This is the event that disconnects one session only. | ERR | ERROR |
| PANIC | This is the event that crashes the server. | CRIT | ERROR |

# Control what gets logged

- **log_min_messages**: minimum severity to record (e.g., `WARNING`, `ERROR`).
- **log_error_verbosity**: detail per event (e.g., `TERSE`, `DEFAULT`, `VERBOSE`).
- Other useful knobs:
  - `log_statements`, `log_checkpoints`
  - `log_connections` / `log_disconnections`
  - `log_lock_waits`, `log_verbosity`
  - `log_line_prefix` (prefix fields per line)

# Where logs go & collectors

- **log_destination**: `stderr`, `csvlog`, `syslog`, `eventlog (Windows)`.
- **logging collector**: background process that captures `stderr` output to files; reliable in failure cases.
- Combine with OS tools (`syslog`, `logrotate`) when appropriate.

# Watch-outs & references

- **FATAL** and **PANIC**: unexpected in normal ops (exceptions may occur in replication scenarios).
- For ongoing monitoring, see practices in Chapter 8 (Monitoring and Diagnosis) and log monitoring recipes.

# Plan

# Monitoring and Diagnosis

- Contains recipes for common monitoring and diagnosis tasks within PostgreSQL.
- Designed to answer specific, frequently encountered questions.
- Helps you understand and address issues while using PostgreSQL.

# Topics

- Providing PostgreSQL information to monitoring tools
- Monitoring the PostgreSQL message log
- Real-time viewing using pgAdmin
- Checking whether a user is connected
- Checking whether a computer is connected
- Repeatedly executing a query in `psql`

# Topics

- Checking which queries are running
- Monitoring the progress of commands and queries
- Checking which queries are active or blocked
- Knowing who is blocking a query
- Killing a specific session
- Knowing whether anybody is using a specific table

# Topics

- Knowing when a table was last used
- Monitoring I/O statistics
- Usage of disk space by temporary data
- Understanding why queries slow down
- Analyzing the real-time performance of your queries
- Tracking important metrics over time

# Beyond Database Monitoring

- Monitoring only the database is not enough.
- You must also monitor all components involved in database usage:
  - Is the database host available and accepting connections?
  - Network bandwidth usage and possible interruptions or dropped connections.
  - RAM availability for common tasks and remaining free memory.
  - Disk space availability and prediction of when it will run out.

# Beyond Database Monitoring (cont.)

- Disk subsystem performance and capacity for additional load.
- CPU load and available idle cycles.
- Availability of dependent network services (e.g., Kerberos for authentication).
- Number of context switches during database operation.
- Historical trends for these metrics to detect changes over time.
- Identifying when disk usage started to change rapidly.

# Most Relevant Topics

1. **Monitoring the PostgreSQL message log** – primary source for detecting errors, warnings, and server events.
2. **Checking which queries are running** – crucial for identifying real-time performance bottlenecks.
3. **Monitoring the progress of commands and queries** – enables tracking of long-running operations such as `VACUUM`, `ANALYZE`, and complex queries.
4. **Identifying who is blocking a query** – essential for diagnosing lock contention and concurrency issues.
5. **Monitoring I/O statistics** – helps detect disk-related bottlenecks and storage constraints.
6. **Understanding why queries slow down** – supports root cause analysis of performance degradation.
7. **Tracking important metrics over time** – lightweight monitoring of key PostgreSQL metrics and trends.

# Monitoring the PostgreSQL Message Log

- Essential in production for detecting warnings, errors, and performance trends.
- PostgreSQL can produce large volumes of logs daily.
- Use tools like **pgBadger** to:
  - Summarize and analyze logs automatically.
  - Generate reports for specific time periods.
  - Gain insights into server activity without manual log scanning.

# Getting Ready: Installation & Rotation

- Most Linux distribution PostgreSQL packages have log file rotation enabled by default.
- pgBadger can be installed via your package manager:
  - From community repositories.
  - From your distribution's own packages.
- Log rotation ensures logs are manageable and archived automatically.

# Getting Ready: PostgreSQL Settings for pgBadger

- Configure `log_line_prefix` to include at least:
  - `%t` – timestamp (no milliseconds).
  - `%p` – process ID.
- Enable parameters for richer analysis:
  - `log_checkpoints = on`
  - `log_connections = on`
  - `log_disconnections = on`
  - `log_lock_waits = on`
  - `log_temp_files = 0`
  - `log_autovacuum_min_duration = 0`
  - `log_error_verbosity = default`

# Getting Ready: Reducing Log Volume

- Use `log_min_duration_statement` to limit logged queries by runtime.
- Example: `250ms`: only log statements longer than 250 milliseconds.
- Reduces unnecessary log entries while keeping slow queries visible.
- More details:
  - PostgreSQL logging configuration: PostgreSQL Docs
  - pgBadger configuration: pgBadger Docs

# Using pgBadger with Cron

- Schedule pgBadger to run automatically via `cron`.
- Example: run daily at 4:00 AM to create incremental reports:
  - `0 4 * * * /usr/bin/pgbadger -I -q /var/log/postgresql/postgresql.log.1 -O /var/www/pg_reports/`
- Produce:
  - Daily incremental reports.
  - Weekly summary at the end of the week.

# pgBadger Reports

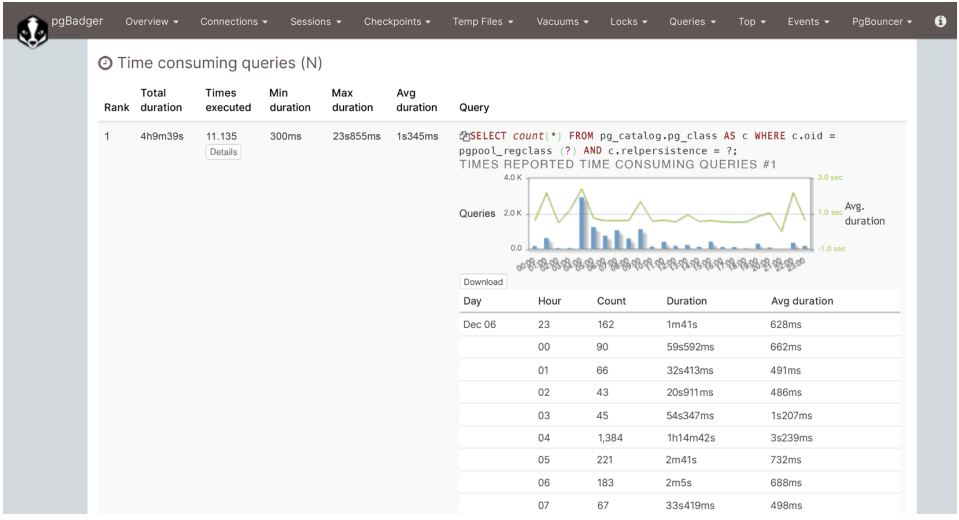
- Generates HTML reports with interactive JavaScript charts.
- Viewable directly in your browser.
- Can highlight:
  - Top $N$ most time-consuming queries.
  - Performance trends and anomalies.

# pgBadger Report Example

- Example of a **Time Consuming Queries** report.
- Shows:
  - Total and average execution time.
  - Minimum and maximum duration.
  - Execution count.
  - Hourly breakdown with durations.

# pgBadger Report Example (Cont.)

- Interactive charts help visualize query performance trends.



# Checking Which Queries Are Running

- Purpose: Identify currently running queries in PostgreSQL.
- Requirements:
  - Logged in as a superuser **or** as the target database user.
  - Ensure `track_activities = on` (default setting).
- If `track_activities` is disabled:
  - Refer to the "Updating the parameter file" recipe in Chapter 3 (Server Configuration).

# Listing All Running Queries

- To view queries currently being executed by connected users:

```
SELECT
    datname, usename, state, backend_type, query
FROM
    pg_stat_activity;
```

- `backend_type = 'client backend'` filters out PostgreSQL worker processes.
- Many sessions may show `state = 'idle'`:
  - No query running — waiting for new commands.
  - `query` field stores the last executed statement.

# Listing Active Queries Only

- To list only active client queries:

```
SELECT
    datname, usename, state, query
FROM
    pg_stat_activity
WHERE
    state = 'active' AND backend_type = 'client backend';
```

- Focuses on currently executing statements.
- Helps identify ongoing workloads and potential slow queries.

# How It Works: Query Activity Tracking

- With `track_activities = on`, PostgreSQL records details for all running queries:
  - Start time, state, query text, executing user, and more.
- Data is accessible to users with sufficient rights via the `pg_stat_activity` system view.
- This view relies on the function:
  `pg_stat_get_activity(procpid int)`
- Pass a specific `process ID` to monitor a single backend.
- Pass `NULL` to retrieve information for all backends.

# Watching the Longest Queries

- Identify long-running queries in PostgreSQL:

```
SELECT
    current_timestamp - query_start AS runtime,
    datname, usename, query
FROM pg_stat_activity
WHERE state = 'active'
ORDER BY 1 DESC;
```

- Shows running queries ordered by execution time (longest first).
- Optionally add `LIMIT 10` to display only the top queries.

# Filtering by Minimum Runtime

- To return only queries running for more than 1 minute:

```
SELECT
    current_timestamp - query_start AS runtime,
    datname, usename, query
FROM pg_stat_activity
WHERE state = 'active' AND current_timestamp - query_start > '1 min'
ORDER BY 1 DESC;
```

- Focuses on queries with significant execution times.
- Useful on busy systems to highlight potential performance issues.

# Knowing Who is Blocking a Query

- Purpose: Identify the session or process causing a query to be blocked.
- Once a blocked query is detected, determine the blocker to resolve the issue.
- Requirements:
  - Logged in as a **superuser** for full access to monitoring data.
- Enables faster diagnosis and resolution of lock contention problems.

# Identifying Blocking Sessions (Step 1)

- To find which session is blocking a query, run:

```
SELECT datname, usename, wait_event_type, wait_event, pid,
       pg_blocking_pids(pid) AS blocked_by,
       backend_type, query
FROM pg_stat_activity
WHERE wait_event_type IS NOT NULL
  AND wait_event_type NOT IN ('Activity', 'Client');
```

- Adds the `blocked_by` column to show blocking session PIDs.

# Identifying Blocking Sessions (Step 2)

- Example output:

```
-[ RECORD 1 ]---+-----------------
datname         | postgres
usename         | gianni
wait_event_type| Lock
wait_event      | relation
pid             | 19502
blocked_by      | {18142}
backend_type    | client backend
query           | select * from t;
```

- `blocked_by` lists one or more PIDs causing the block.
- PIDs are unique OS-level session identifiers.

# Interpreting Blocking Session Data

- `pg_blocking_pids(pid)` returns a list of blocking session PIDs.
- This query builds on the previous recipe for active or blocked queries.
- The `pid` column comes from the operating system and identifies each session.
- For more details on PIDs, see Chapter 4 (*Server Control*).

# How It Works: Blocking Session Detection

- Uses the `pg_blocking_pids()` function:
  - Returns an array of PIDs for all sessions blocking the given PID.
- Parallel queries lock through their leader process.
- This behavior does not add complexity when monitoring locks.

# Monitoring I/O Statistics

- Purpose: Assess system performance in relation to hardware resource usage.
- Focus: Examine I/O rates for the database and its contained objects.
- Requirements:
  - Membership in the `pg_monitor` role for full access to I/O statistics.
- Helps identify storage bottlenecks and optimize resource usage.

# Monitoring I/O Statistics with pg_stat_io

- Since PostgreSQL 16, pg_stat_io provides I/O statistics for the entire server.
- Includes all databases and backend types.
- To get running totals in bytes for each backend type:

```
SELECT
    backend_type,
    reads * op_bytes AS bytes_read,
    writes * op_bytes AS bytes_written
FROM
    pg_stat_io;
```

# PostgreSQL Maintenance Backends

- PostgreSQL uses a **process-based architecture**:
  - The main server process (`postmaster` or `postgres`) spawns multiple backend processes for connections and internal tasks.
- **Maintenance backends:**
  - **autovacuum launcher** – Oversees autovacuum activity and schedules cleanup tasks.
  - **autovacuum worker** – Executes vacuum and analyze operations.
  - **background writer** – Flushes dirty buffers to disk to reduce checkpoint spikes.
  - **checkpointer** – Writes all dirty buffers at checkpoints and updates control files.
  - **startup** – Runs during database startup and recovery.

# Sample Output: pg_stat_io

- Example output (truncated for brevity):

```
backend_type         | bytes_read | bytes_written
---------------------+------------+---------------
autovacuum launcher |       0     |       0
client backend       |  7086080    |       0
background writer    |       0     |       0
checkpointer         |       0     | 152690688
...
(30 rows)
```

- Shows read and write byte totals by backend type.

# Per-Table I/O Statistics

- Use `pg_statio_user_tables` for table-level I/O stats.
- Example query:

```
WITH b AS (
    SELECT current_setting('block_size')::int AS blcksz
)
SELECT
    heap_blks_read * blcksz AS heap_read,
    heap_blks_hit  * blcksz AS heap_hit,
    idx_blks_read  * blcksz AS idx_read,
    idx_blks_hit   * blcksz AS idx_hit
FROM
    pg_statio_user_tables, b
WHERE
    relname = <table name>;
```

# Sample Output: Per-Table I/O

- Example output for table `job_details`:

```
heap_read  | heap_hit    | idx_read | idx_hit
-----------+-------------+----------+------------
8192       | 841138176   | 8192     | 1637752832
(1 row)
```

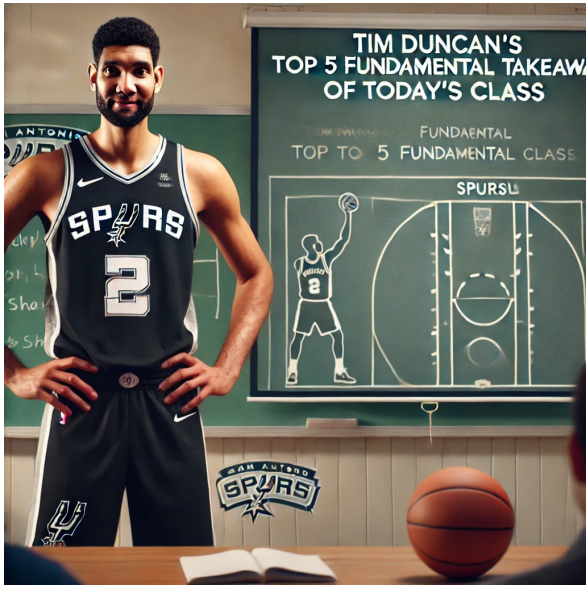
- Values are in bytes, calculated from block counts × block size.

# Interpreting PostgreSQL I/O Metrics

- `pg_stat_io`: server-wide I/O metrics by backend type.
- `pg_statio_user_tables`: per-table heap and index I/O.
- Useful for:
  - Detecting heavy I/O consumers.
  - Balancing workloads across storage resources.
  - Identifying caching efficiency (`hit` vs `read` ratios).

# End of Lecture 4.

# TDT5FTOTC


TIM DUNCAN'S
TOP 5 FUNDAMENTAL TAKEAW
OF TODAY'S CLASS
FUNDAENTAL
TOP TO 5 FUNDAMENTAL CLASS
SPURS
SPURS
SPURS

# TDT5FTOTC

5 **Log Analysis:** PostgreSQL message logs are essential for detecting issues and should be regularly analyzed with tools like `pgBadger`.

# TDT5FTOTC

5 **Log Analysis:** PostgreSQL message logs are essential for detecting issues and should be regularly analyzed with tools like `pgBadger`.

4 **Query Monitoring:** Monitoring running queries in real time via `pg_stat_activity` helps identify and address performance bottlenecks.

# TDT5FTOTC

5 **Log Analysis:** PostgreSQL message logs are essential for detecting issues and should be regularly analyzed with tools like `pgBadger`.

4 **Query Monitoring:** Monitoring running queries in real time via `pg_stat_activity` helps identify and address performance bottlenecks.

3 **Lock Diagnosis:** Lock contention can be diagnosed by finding blocking sessions using `pg_blocking_pids()`.

# TDT5FTOTC

5 **Log Analysis:** PostgreSQL message logs are essential for detecting issues and should be regularly analyzed with tools like `pgBadger`.

4 **Query Monitoring:** Monitoring running queries in real time via `pg_stat_activity` helps identify and address performance bottlenecks.

3 **Lock Diagnosis:** Lock contention can be diagnosed by finding blocking sessions using `pg_blocking_pids()`.

2 **I/O Tracking:** I/O statistics from `pg_stat_io` and `pg_statio_user_tables` reveal storage performance and caching efficiency.

# TDT5FTOTC

5 **Log Analysis:** PostgreSQL message logs are essential for detecting issues and should be regularly analyzed with tools like `pgBadger`.

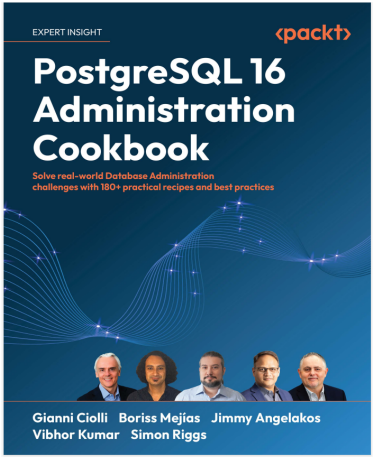
4 **Query Monitoring:** Monitoring running queries in real time via `pg_stat_activity` helps identify and address performance bottlenecks.

3 **Lock Diagnosis:** Lock contention can be diagnosed by finding blocking sessions using `pg_blocking_pids()`.

2 **I/O Tracking:** I/O statistics from `pg_stat_io` and `pg_statio_user_tables` reveal storage performance and caching efficiency.

1 **System Health:** Full performance monitoring includes tracking CPU, RAM, disk, network, and dependent service health alongside database metrics.

# Database Administration: Performance Monitoring and Diagnosis.




Content has been extracted from *PostgreSQL 16 Administration Cookbook* by Ciolli, Mejías, Angelakos, Kumar & Riggs, 2023. Visit packtpub.com.