PostgreSQL 17: Mastering the Art of Monitoring & Managing Long-Running and Blocked Queries

PostgreSQL 17 comes loaded with advanced tools that give database administrators deep visibility into query execution. Whether you're troubleshooting slow queries, resolving blocked sessions, or preventing deadlocks, PostgreSQL provides everything you need — if you know where to look.

In this guide, we'll explore:

✅How to detect slow, long-running queries
✅How to identify blocked queries
✅How to safely terminate problematic sessions
✅How to monitor locks & blocking chains
✅Practical queries every PostgreSQL DBA should memorize

Let's dive into PostgreSQL internals and make you a query monitoring expert. 🔍🔍

🔍 Real-Time Query Monitoring with `pg_stat_activity`

Efficient database management demands insight into what's running at any given moment. PostgreSQL's `pg_stat_activity` system view is the go-to tool for monitoring live connections and queries in real time.

🔍 What Is `pg_stat_activity`?

`pg_stat_activity` is an internal system view that provides a snapshot of all active connections to your PostgreSQL database. It allows DBAs to inspect ongoing queries and connection states—perfect for debugging slow queries, monitoring load, or managing locks.

✅How to View All Current Queries

To explore live activity, simply run:

```
SELECT * FROM pg_stat_activity;
```

This returns one row per database connection with columns such as:

- `pid` – The process ID of the backend worker.
- `usename` – The authenticated username that initiated the connection.
- `datname` – The database to which the connection belongs.
- `state` – The current state: `active`, `idle`, `idle in transaction`, `waiting`, etc.
- `query` – The SQL statement currently being executed (or `<IDLE>` if not executing).
- `query_start` – Timestamp marking when the execution of the current query began.

Understanding key columns helps you interpret workload:

Column	Description
`pid`	Unique process ID—useful for termination or profiling
`usename`	Which user is running the query
`datname`	Which database the activity is happening in
`state`	Indicates activity status (`active`, `idle`, `idle in transaction`, etc.)
`query`	The actual SQL command in progress
`query_start`	How long the query has been running—helps identify long-running statements

⍰ Practical Use Cases

- **Spotting long-running queries**
  Look for queries with high `now() - query_start` intervals.
  Example:

```
SELECT pid, usename, now() - query_start AS duration, query FROM pg_stat_activity WHERE state = 'active' ORDER BY duration DESC LIMIT 5;
```

- This pinpoints the top 5 longest-running queries.
- **Detecting blocked or waiting sessions**
  Filter sessions in the `waiting` state to identify lock or I/O contention.
- **Monitoring idle transactions**
  Sessions in `idle in transaction` state can hold locks; tracking them helps prevent blockages.

□ Tips for Effective Monitoring

- **Restrict output** to avoid clutter with:

```
SELECT pid, usename, datname, state, query, query_start FROM pg_stat_activity WHERE state = 'active';
```

- **Exclude self-session**:

```
WHERE pid <> pg_backend_pid()
```

- **Use dedicated tools** as well — like `pg_activity` or `pgAdmin`—that offer tailing and color-coded views of `pg_stat_activity`.

□ In Summary

`pg_stat_activity` is your real-time diagnostic dashboard for PostgreSQL:

1. **Check who's connected** and what they're doing.
2. **Identify problematic queries** by duration or state.

3. **Uncover idle or blocked connections** to maintain healthy throughput.

Integrating `pg_stat_activity` queries into your monitoring routines empowers you to proactively detect performance issues and keep your PostgreSQL environment running smoothly.

? Count Total Active Connections

To keep an eye on how many clients are currently connected to your database, use:

```
SELECT COUNT(*) AS total_conns
FROM pg_stat_activity;
```

- `total_conns` gives you an immediate view of how many active database sessions are in progress.
- This metric is vital for capacity planning and detecting unexpected workload spikes.

? Bonus: Filter by Client IP

Want to know how many connections come from a specific host? Use:

```
SELECT COUNT(*) AS client_conns
FROM pg_stat_activity
WHERE client_addr = 'your_ip_here';
```

- Replace `'your_ip_here'` with the actual IP you wish to monitor.
- This helps you detect if a specific app server is overloading your database with connections.

? Detecting Long-Running Queries

Queries that run longer than five minutes can hog resources, degrade performance, or indicate application errors. You can identify these problematic queries using:

```
SELECT
 pid,
 usename,
 pg_stat_activity.query_start,
 now() - pg_stat_activity.query_start AS query_time,
 query,
 state,
 wait_event_type,
 wait_event
FROM pg_stat_activity
WHERE (now() - pg_stat_activity.query_start) > interval '5 minutes';
```

⍰ What Each Column Tells You:

- `pid` — The session's process ID; can be used to terminate the query if needed.
- `usename` — The database user who started the query, useful for assigning responsibility.
- `query_start` — When the query began.
- `query_time` — The elapsed time since the query started—ideal for prioritizing the longest ones.
- `query` — The SQL text currently executing.
- `state` — Shows if the query is `active`, `idle`, `waiting`, etc.
- `wait_event_type`, `wait_event` — Provide insight into what the query is waiting on (disk I/O, locks, buffers, etc.).

⍰ Why This Matters

- **Resource hog detection** — Long-running queries can saturate CPU, memory, or I/O.
- **Performance tuning** — Pinpointing slow queries helps you optimize application code and database indexes.

- **Troubleshooting** — If many sessions are waiting or blocked, it may signal locking or replication issues.

? Tip: Target Your Monitoring

To avoid scanning every connection or query, narrow your search:

```
SELECT
  pid,
  usename,
  now() - query_start AS query_duration,
  query
FROM pg_stat_activity
WHERE state = 'active'
  AND (now() - query_start) > interval '5 minutes';
```

This quickly surfaces the top offenders, making it easier to take action — like query cancellation, index tuning, or application optimization.

By routinely checking total connections and snapping up long-running queries, you'll stay ahead of load issues, maintain high performance, and keep your PostgreSQL infrastructure healthy and efficient.

? Analyze Database Load Distribution

To understand where your database is spending its time and which clients are driving most of the traffic, use this query:

```
SELECT
    client_addr,
    usename,
    datname,
    state,
    COUNT(*) AS connection_count
FROM pg_stat_activity
GROUP BY 1, 2, 3, 4
ORDER BY connection_count DESC;
```

? What This Shows

- `client_addr`: IP address of each connected client.
- `usename`: Database user running each connection.
- `datname`: Name of the target database.
- `state`: Connection status (e.g., `active`, `idle`, `idle in transaction`).
- `connection_count`: Number of connections per user/database combination.

❓ Why It Matters

- Highlights which app or user account is generating heavy traffic.
- Reveals whether a particular database is a hotspot.
- Helps identify large numbers of idle or leaked connections.

❓ Investigate Blocking & Locks

Locks are vital to database integrity — but excessive or stale locking can lead to performance bottlenecks. Start by inspecting active locks:

```
SELECT
  relname AS relation_name,
  query,
  pg_locks.*
FROM pg_locks
JOIN pg_class ON pg_locks.relation = pg_class.oid
JOIN pg_stat_activity ON pg_locks.pid = pg_stat_activity.pid;
```

❓ Key Columns

- `relation_name`: The locked table or index.
- `query`: The SQL statement currently holding or waiting for the lock.
- Lock details: columns like `mode`, `granted`, and `locktype` describe the lock's nature.

❓ Outcome

- Identify tables with high lock contention.
- Spot sessions where locks are being held longer than necessary.

🔍 Find Blocking Chains

To track dependencies between queries — who is waiting on whom — you can use:

```
SELECT
  activity.pid AS blocked_pid,
  activity.usename AS blocked_user,
  activity.query AS blocked_query,
  blocking.pid AS blocking_pid,
  blocking.query AS blocking_query
FROM pg_stat_activity AS activity
JOIN pg_stat_activity AS blocking
 ON blocking.pid = ANY(pg_blocking_pids(activity.pid));
```

🧠 What to Look For

- `blocked_pid` / `blocked_user` / `blocked_query`: the session currently blocked.
- `blocking_pid` / `blocking_query`: the session holding the lock that's preventing progress.

🧠 Why It Matters

- Quickly surfaces chains of blocking and blocked sessions.
- Helps you identify which transactions need intervention (e.g., cancellation or optimization).
- Useful for debugging deadlock or long-running dependency issues.

🧠 How to Implement

1. **Schedule periodic checks** (e.g., every 5 minutes) to capture load and blocking patterns.

2. **Include in monitoring dashboards** alongside latency and throughput metrics.

3. **Automate alerts**, for example:

- If `connection_count` by a specific IP spikes above a threshold.
- If any blocked session is waiting longer than 1–2 minutes.

4. **Enable observation across environments** — especially in development, staging, and production to anticipate where locking or concurrency issues might emerge.

By using these techniques, PostgreSQL DBAs can visualize workload distribution, detect high-impact sessions, and preemptively resolve blocking issues — helping maintain a performant, reliable database environment.

⚠️ Terminate Problem Queries Safely

Occasionally, you'll encounter runaway or blocked queries that need intervention. PostgreSQL offers two methods to cancel these, depending on severity and urgency:

1⃣ Soft Cancel (Recommended)

Use this to **gently ask a query to stop**, allowing it to clean up resources safely:

```
SELECT pg_cancel_backend(pid);
```

- Replaces `pid` with the actual process ID from `pg_stat_activity`.
- Ideal for queries that are simply too long or consuming excessive resources.
- The backend gets a cancellation signal and typically halts soon afterward — without rolling back the entire transaction.

2️⃣ Hard Kill (Use Cautiously)

If the backend ignores the soft cancel or is stuck, escalate to:

```
SELECT pg_terminate_backend(pid);
```

- This **forcefully terminates** the session associated with `pid`.
- Any open transaction is rolled back.
- Note: This may cause temporary spikes in load or leave locks behind — use sparingly and monitor afterward.

📝 Quick Reference: `pg_stat_activity` Columns

Below is a handy reference to some key columns you'll often use when identifying and managing problematic sessions:

Column	Description
datname	The database the session is connected to
pid	The backend process ID
usename	The user who initiated the session
client_addr	The client's IP address
backend_start	When the session began
query_start	When the current or last query began
xact_start	When the current transaction started
state	Session status (`active` , `idle` , etc.)
query	The SQL being executed or last executed

🧠 How This Fits Together

- Use **`pg_stat_activity`** to find long-running or locked queries.
- **Soft cancel (`pg_cancel_backend`)**: your first line of defense—minimal disruption.
- **Hard kill (`pg_terminate_backend`)**: a last resort when sessions won't end cleanly.

- Always review relevant logs and monitor for side-effects, such as locked rows or rollbacks that took time to complete.

Knowing when to cancel or terminate a session — and how to do it cleanly — is crucial for maintaining database health and keeping your PostgreSQL environment running smoothly.

==========
==========

- **SELECT pg_cancel_backend(pid);**

Action: Attempts to cancel the currently executing query of the backend with the given PID.

Scope: Only cancels the current query, not the entire session.

Effect: The session remains connected and can continue executing other queries.

Use case: When you want to stop a long-running query without disconnecting the user.

Safe: Yes — it's the least disruptive option.

✅Example use case:

SELECT pg_cancel_backend(12345);

This will try to cancel the active query running on PID 12345.

- ⛔**SELECT pg_terminate_backend(pid);**

Action: Forcefully terminates the entire backend process for the given PID.

Scope: Ends the whole session and all its current activities.

Effect: The user will be disconnected, and any open transactions will be rolled back.

Use case: When a session is stuck, idle in transaction, or not responding.

Safe?: More disruptive — should be used with caution, especially on production systems.

✅Example use case:

SELECT pg_terminate_backend(12345);

This will kill the PostgreSQL session on PID 12345.

==========
==========

🧠 Essential DBA Queries for PostgreSQL

Here's a refined suite of must-have `pg_stat_activity` queries for real-time insight into database usage, performance issues, and connection patterns:

1️⃣ Total Active Queries

Get a quick count of how many sessions are currently connected:

```
SELECT COUNT(*) AS total_active_queries
FROM pg_stat_activity;
```

– Great for spotting spikes in connection load or sudden increases in database usage.

2️⃣ Identify Waiting Queries

Spot sessions that are blocked or waiting on disk, locks, or other resources:

```
SELECT *
FROM pg_stat_activity
WHERE wait_event IS NOT NULL
 AND backend_type = 'client backend';
```

– `wait_event` reveals what the session is waiting for (e.g., `Lock`, `IO`, etc.).
– Helps catch bottlenecks caused by long waits or contention.

3️⃣ Group Connections by Client

See where the load is coming from — group sessions by client IP, user, and database:

```
SELECT
  client_addr,
  usename,
  datname,
  state,
  COUNT(*) AS session_count
FROM pg_stat_activity
GROUP BY 1, 2, 3, 4
ORDER BY session_count DESC;
```

– Useful for trending which apps or users are most active, and how many concurrent sessions each generates.
— Helps uncover idle sessions that may be hogging resources unnecessarily.

4️⃣ Find Queries Running Longer than 1 Second

Detect all sessions where the current transaction or query has been running for over one second:

```
SELECT
  client_addr,
  usename,
  datname,
  now() - xact_start AS xact_age,
  now() - query_start AS query_age,
  state,
  query
FROM pg_stat_activity
WHERE (
    (now() - xact_start) > '00:00:01'::interval
 OR (now() - query_start) > '00:00:01'::interval
)
AND state <> 'idle'
ORDER BY COALESCE(xact_start, query_start) DESC;
```

- `xact_age`: how long the transaction has been open.
- `query_age`: shows how long the individual query has been running.
- Filtering out `idle` sessions ensures you're looking at active workload only.

❓ Why These Queries Matter

- **Session count** helps you manage connection limits and capacity planning.
- **Waiting query detection** isolates load issues and potential performance killers.
- **Grouping by client** reveals patterns, overloaded hosts, or connection leaks.
- **Monitoring sub-second query durations** helps catch slow queries early — before they grow into larger problems.

✔️Pro Tips for Use

1. **Schedule these queries regularly** (every minute if needed) for near-real-time monitoring.
2. **Visualize results** with a monitoring tool like Grafana or pgAdmin for correlations with system metrics.
3. **Combine with alerting**: e.g., send a warning if more than 100 queries run longer than 1 second.

4. **Tailer insight**: pair with PostgreSQL's `log_min_duration_statement` to log slow queries for investigation.

By integrating these core queries into your DBA routine, you gain clarity over database loads, responsiveness, and growth — empowering you to keep your PostgreSQL environment reliable and performant.

🔒 In-Depth Lock Management via `pg_locks`

Locking is fundamental to PostgreSQL's consistency mechanisms, but locks can also introduce contention that impacts performance. Exploring locks in detail helps you understand who holds them, what they're waiting for, and whether any sessions are blocking others.

🔍 1️⃣ Examine All Active Locks

The `pg_locks` view provides a snapshot of every lock currently held or awaited by active sessions:

```
SELECT * FROM pg_locks;
```

This raw output includes columns such as:

- `pid` – Process ID of the backend involved
- `locktype` – Lock category (e.g., `relation`, `transactionid`)
- `mode` – Type of lock (e.g., `RowExclusiveLock`, `AccessShareLock`)
- `granted` – Indicates if the lock is held or still pending

Use this to get a big-picture sense of lock activity, though interpretation often requires context about the associated queries.

🔗 2️⃣ Map Locks to Sessions and Queries

To understand which queries are associated with which locks, join `pg_locks` with `pg_class` (for relation names) and `pg_stat_activity` (for query details):

```
SELECT
  relname AS relation_name,
  pg_stat_activity.query,
  pg_locks.*
FROM pg_locks
JOIN pg_class ON pg_locks.relation = pg_class.oid
JOIN pg_stat_activity ON pg_locks.pid = pg_stat_activity.pid;
```

Key insights you'll gain:

- `relation_name` reveals the table or index involved
- `query` shows the exact SQL holding or requesting the lock
- All `pg_locks.*` fields provide lock type, mode, and grant status

This combined view helps you pinpoint sessions causing lock contention and enables you to take targeted action, such as query optimization or cancellation.

⍰ Pro Tip: Use pgAdmin for Visual Query Monitoring

For visually inspecting and managing active sessions and locks, pgAdmin is a valuable tool:

- **Query Tool**: Run any of the SQL snippets above directly with a graphical interface.
- **Dashboard → Sessions**: Use live filtering, sorting, and real-time stats to monitor connections at a glance.
- **Context menu actions**: With one click, you can cancel queries (`pg_cancel_backend`) or terminate sessions (`pg_terminate_backend`), making lock resolution rapid and intuitive.

✔Summary

- `pg_locks` shows every lock—held or waiting—in your system.
- Joining with `pg_class` and `pg_stat_activity` gives you context: *who* is holding *what* lock and *why*.
- pgAdmin complements CLI tools with powerful visual monitoring and one-click management.

By mastering lock inspection and management, you'll better diagnose performance issues, reduce contention, and maintain a smoother PostgreSQL experience.