How to Use PostgreSQL Log Files for Troubleshooting Performance

Welcome to this comprehensive guide on leveraging PostgreSQL log files to diagnose and resolve performance issues. Throughout this presentation, we'll explore how proper log configuration and analysis can transform your troubleshooting process from guesswork to precision.

Introduction

My name is Shiv Iyer, Founder and CEO of MinervaDB. We specialize in PostgreSQL optimization and performance tuning for organizations of all sizes. With years of experience diagnosing complex database issues, we've developed systematic approaches to extract actionable insights from PostgreSQL logs.

Presentation Topics

1. Understanding the Importance of PostgreSQL Logging
2. Types of PostgreSQL Log Files and Their Uses
3. Configuring PostgreSQL Logging Parameters for Optimal Performance Analysis
4. Analyzing Query Performance from Logs
5. Detecting Deadlocks and Lock Waits
6. Utilizing External Log Analysis Tools
7. Best Practices for Managing Log File Size and Retention
8. Troubleshooting with Logs: Real-World Examples

Contact Information

Contact: **contact@minervadb.com** | **shiv@minervadb.com**

Why PostgreSQL Logging Matters

Critical for Diagnosis

Log files provide the forensic evidence needed to diagnose performance bottlenecks. They can help pinpoint unexpected errors and lock contention issues that may not be apparent through other monitoring tools. Detailed logs offer insights into the sequence of events leading to an issue, which is invaluable for root cause analysis.

Data-Driven Optimization

Rather than relying on intuition, logs enable precise, evidence-based performance tuning. By revealing exactly which queries, transactions, or connections are causing problems, logs allow database administrators to make informed decisions. This data-driven approach ensures that optimization efforts are targeted and effective.

Compliance and Auditing

Well-configured logs create an audit trail of database activity essential for security compliance, governance requirements, and post-incident investigations. These logs provide a record of who accessed what data and when, facilitating compliance with regulations. This is essential for maintaining data integrity and accountability.

Types of PostgreSQL Log Files

Error Logs

Capture system-level errors, warnings, and fatal issues that indicate problems with the PostgreSQL server itself. These logs are essential for identifying hardware failures, software bugs, or configuration mistakes.

Query Logs

Contain SQL statements executed against the database, along with their execution durations. These logs are invaluable for performance tuning, allowing administrators to identify slow-running queries and optimize them for improved performance.

Connection Logs

Record session starts and ends, as well as authentication events. These logs are crucial for tracking user activity, identifying connection issues, and auditing access to the database.

Security Logs

Security logs monitor access attempts, privilege escalations, and other security-related events. These logs are critical for identifying potential security breaches and ensuring compliance with security policies and regulations.

Each log type serves a specific diagnostic purpose. Error logs help identify system-level issues, while query logs focus on performance optimization opportunities. Connection logs help track user activity patterns and authentication problems. Security logs provide crucial insights into database access and potential security threats.

For a deeper dive into log types and their configurations, visit our detailed guide: PostgreSQL Logging Basics

Where PostgreSQL Logs Are Stored?

Default Location

1

By default, PostgreSQL sends logs to the standard error stream (stderr), which typically appears in the database server console or is captured by the service manager. This is useful for initial debugging or when a full logging setup isn't necessary. To view these logs, you'll usually need to access the server directly or check the system's service management logs.

Enabling File Logging

2

The **logging_collector** parameter must be set to 'on' to capture logs to actual files instead of stderr. This requires a server restart to take effect. After enabling, ensure the PostgreSQL process has write permissions to the log directory. Consider the implications for log rotation and disk space.

Common File Locations

3

Most installations store logs in **/var/log/postgresql/** on Linux systems, **%PROGRAMDATA%\PostgreSQL\log** on Windows, or within the cluster's data directory in a 'pg_log' subdirectory. Check these locations regularly and set up automated scripts to monitor log file sizes. Ensure proper permissions are set to prevent unauthorized access.

Key Logging Parameters

Parameter	Purpose	Common Values
logging_collector	Enables file-based logging	on, off
log_destination	Output format and target	stderr, csvlog, jsonlog, syslog
log_filename	Naming pattern for log files	postgresql-%Y-%m-%d_%H%M%S.log
log_rotation_age	Time-based rotation interval	1d, 1h
log_rotation_size	Size-based rotation threshold	10MB, 1GB

Properly configuring these parameters is essential for maintaining useful logs without overwhelming storage resources. For a comprehensive parameter reference, see our **Complete Postgres Logging Guide**.

Tuning Log Verbosity

Log Verbosity Modes

PostgreSQL offers different log verbosity modes to control the amount of detail recorded in the logs. Choose the mode that best suits your troubleshooting needs:

1	**TERSE Mode** Records minimal detail, focusing on basic error messages only. Useful for high-level monitoring.
2	**DEFAULT Mode** Provides a standard level of detail, suitable for general monitoring and troubleshooting. This is the recommended setting for most systems.
3	**VERBOSE Mode** Records maximum detail, including context information like SQLSTATE error codes and source code locations. Use this mode for in-depth debugging.

Controlling Message Severity

The **log_min_messages** parameter filters messages based on their severity level. It ranges from DEBUG5 (most verbose) to PANIC (most severe). For performance troubleshooting, setting it to ERROR or WARNING typically provides sufficient detail without generating excessive log noise. Adjust this parameter to focus on critical issues impacting database performance.

Context Included with Each Message

The **log_error_verbosity** parameter determines the amount of context included with each log message. During active debugging, VERBOSE mode provides the most comprehensive context, including SQLSTATE error codes and source code locations, aiding in issue diagnosis. However, DEFAULT mode is often sufficient for routine monitoring, balancing detail and log size. Consider temporarily enabling VERBOSE mode when investigating specific issues, then revert to DEFAULT for normal operations.

Performance Impact

Increasing log verbosity can impact performance due to the increased overhead of writing more data to log files. It's crucial to strike a balance between capturing enough detail for effective troubleshooting and minimizing the performance impact of logging. Regularly monitor log file sizes and system performance when adjusting these parameters to ensure optimal database operation.

Logging Slow Queries

|

Configure Threshold

Set the **log_min_duration_statement** parameter to define the minimum execution time for queries to be logged. This parameter allows you to specify a threshold in milliseconds. Queries that execute longer than this threshold will be logged, allowing you to identify performance bottlenecks.

||

Capture Execution

The system logs all queries that exceed the configured threshold, capturing valuable performance data. Capturing these queries provides insights into which operations are consuming the most resources and impacting overall database performance. This data is crucial for identifying areas that require optimization.

|||

Analyze Patterns

Examine the logs to identify patterns and recurring slow queries that require attention. By analyzing these patterns, you can pinpoint specific queries or database operations that consistently exhibit slow performance. This analysis can lead to targeted optimization efforts and improved overall system efficiency.

||||

Optimize

Improve the performance of identified slow queries through indexing, query rewriting, or other optimization techniques. Optimization techniques include adding indexes to frequently queried columns, rewriting inefficient queries to use more optimal execution plans, and adjusting database configuration parameters to better suit the workload. Effective optimization can significantly reduce query execution times and improve overall database performance.

The **log_min_duration_statement** parameter is crucial for performance troubleshooting. By setting it to a specific millisecond value (e.g., 100, 500, 1000), you instruct PostgreSQL to log all queries whose execution time exceeds that duration, providing a focused list for optimization efforts. A lower value will capture more queries but can lead to larger log files. Conversely, a higher value will reduce log file size but may miss some slow queries. Experimentation and monitoring are key to finding the right balance.

Exercise caution when setting this value too low in high-volume production environments, as excessive logging can introduce a performance bottleneck. Monitor log file sizes and system performance to maintain optimal database operation. Regularly review your logging configuration to ensure it aligns with your performance troubleshooting needs without negatively impacting system performance. Consider implementing log rotation and archiving strategies to manage log file sizes effectively.

Example: Enabling Slow Query Logging

```
# postgresql.conf example for production slow query logging:

# Log queries running longer than 2 seconds
log_min_duration_statement = 2000

# Enable logging of statement duration
log_duration = on

# Recommended log format for easier parsing
log_line_prefix = '%t [%p]: user=%u,db=%d '

# Enable auto_explain extension for automatic query plan logging
shared_preload_libraries = 'auto_explain'

# Auto explain configuration
auto_explain.log_min_duration = 2000 # aligned with log_min_duration_statement
auto_explain.log_analyze = true
auto_explain.log_verbose = true
auto_explain.log_format = json

# Apply changes with:
SELECT pg_reload_conf();  # or restart for shared_preload_libraries changes
```

Once configured, slow queries and their execution plans will appear in logs. Example:

> 2023-11-01 15:32:47.123 UTC [12345]: user=app_user,db=production LOG: duration: 2345.678 ms statement: SELECT * FROM orders o JOIN customers c ON o.customer_id = c.id WHERE c.signup_date < '2023-01-01' AND o.total_amount > 100;
>
> 2023-11-01 15:32:47.124 UTC [12345]: user=app_user,db=production LOG: QUERY PLAN: [JSON formatted query plan from auto_explain]

Analyzing Query Performance from Logs

What to Look For

- Recurring slow queries from the same source IP or application
- Patterns in execution time variations correlated with specific SQL functions or operators
- Queries that consistently exhibit slow performance versus intermittent slowdowns due to resource contention
- Correlation with peak time of day or specific background workload patterns (e.g., backups)
- Sequential scans on large tables lacking appropriate indexes; identify missing index candidates

Analysis Techniques

- Aggregate similar queries (normalized by removing literal values) to identify query hotspots and common performance bottlenecks
- Correlate slow queries with system resource metrics (CPU, memory, I/O) using tools like pg_stats and system monitoring dashboards
- Track queries across related transactions using transaction IDs to identify multi-query performance issues
- Compare query execution plans before and after optimization attempts (e.g., index creation, query rewriting) to validate improvements
- Use specialized log analysis tools (e.g., pgbadger, pgMonitor) for visualization and automated anomaly detection

For detailed query performance analysis strategies, see our guide: **PostgreSQL Query Performance Analysis**

Detecting Deadlocks and Lock Waits

1

Enable Lock Logging

Set **log_lock_waits = on** in **postgresql.conf**. This setting ensures that PostgreSQL logs details when a session waits longer than **deadlock_timeout** (default: 1s) to acquire a lock, aiding in identifying lock contention issues.

2

Examine Log Output

PostgreSQL logs include details about the blocking and waiting processes, such as process IDs (PIDs), queries involved, lock types (e.g., ACCESS SHARE, ROW EXCLUSIVE), and the resources being contended.

3

Identify Contention Patterns

Analyze logs for recurring lock contention patterns by aggregating related events. Focus on problematic tables or application behaviors that frequently cause blocking. Use transaction IDs (**xid**) to trace lock waits across multiple queries within a transaction.

4

Implement Mitigation Strategies

Restructure transactions to minimize lock duration, add targeted indexes to reduce contention, and refactor application logic to avoid long-held locks. Consider using advisory locks for application-level concurrency control.

Using External Log Analysis Tools

Leverage specialized tools for in-depth PostgreSQL log analysis, providing insights into performance bottlenecks and system behavior:

pgBadger

Generates detailed HTML reports with query performance metrics, error distributions, and connection statistics. Efficiently parses large log files and supports all PostgreSQL log formats. It offers features like slow query analysis, autovacuum monitoring, and identification of the most resource-intensive operations. pgBadger is a powerful tool for understanding your PostgreSQL server's performance and identifying areas for optimization.

ELK Stack (Elasticsearch, Logstash, Kibana)

Provides centralized log aggregation, indexing, and visualization for enterprise environments. Enables correlation of logs across multiple database instances for comprehensive analysis. Elasticsearch provides scalable search and analytics, Logstash handles log ingestion and parsing, and Kibana offers interactive dashboards for visualizing log data. The ELK stack is ideal for large-scale deployments requiring real-time monitoring and alerting.

Grafana + Loki

Offers a modern log aggregation and visualization platform with tight integration with system metrics. Facilitates real-time monitoring and alerting based on log patterns and anomalies. Loki is a horizontally scalable, highly available, multi-tenant log aggregation system inspired by Prometheus. Grafana provides rich dashboards and alerting capabilities, allowing you to correlate logs with other system metrics for comprehensive troubleshooting.

Learn more about automating log analysis in our guide: Automating Postgres Log Analysis

Managing Log File Size and Retention

1 Size-Based Rotation

Set **log_rotation_size** to automatically create new log files when the current file reaches a specified size (e.g., 100MB). This prevents individual log files from growing unmanageably large. The server checks the log file size after each new log message, and performs a rotation if the size exceeds the configured value. This helps in managing disk space and makes it easier to analyze specific time frames.

2 Time-Based Rotation

Use **log_rotation_age** to rotate logs based on elapsed time (e.g., 1d for daily rotation). This creates predictable log file boundaries that align with operational time periods. It is useful for organizing logs by day, week, or month, which can simplify auditing and troubleshooting efforts. When this parameter is set, the PostgreSQL server checks the log file's age each time a new log message is written.

3 File Management

Enable **log_truncate_on_rotation** to automatically overwrite old log files in a cyclical pattern, maintaining a fixed number of historical logs without manual cleanup. When this setting is enabled, the server truncates the existing log file instead of appending to it. This is commonly used in conjunction with time-based or size-based rotation to manage log file sizes and prevent disk space exhaustion.

4 Retention Policies

Implement external retention scripts to archive or purge logs beyond your required retention period, balancing troubleshooting needs with storage constraints. These scripts can automatically move older logs to cheaper storage or completely remove them. It is crucial to define a retention policy that meets both compliance requirements and operational needs.

Best Practices for Log Storage

Storage Performance

Store PostgreSQL logs on disks separate from your database files to prevent log writes from competing with database I/O. This separation ensures that logging activities do not degrade the performance of critical database operations. SSDs are highly recommended for high-volume logging environments to minimize the impact on overall system responsiveness and reduce latency.

Security Considerations

Restrict log file permissions to database administrators only (typically 0600 or 0640) as logs may contain sensitive query information, including values from INSERT statements if full query logging is enabled. Regularly audit access to these logs to ensure compliance with security policies. Consider encrypting log files at rest to further protect sensitive data.

Archiving Strategy

Implement automated processes to compress and archive older logs to cold storage, such as cloud-based object storage or network-attached storage (NAS), while maintaining recent logs on faster storage for immediate analysis. Employ log shipping to a centralized logging infrastructure to facilitate easier querying, monitoring, and alerting across multiple PostgreSQL instances. Ensure that your archiving strategy complies with data retention requirements and regulations.

Common Logging Pitfalls

1 Excessive Verbosity

Overly detailed logging, especially with debug levels enabled, can significantly degrade performance and create overwhelming data volumes, obscuring critical insights. Use DEBUG levels judiciously for targeted troubleshooting.

Ensure your logging configuration aligns with monitoring goals, avoiding redundant data capture. Regularly review and adjust logging levels for optimal performance.

2 Insufficient Detail

Conversely, inadequate logging deprives troubleshooters of the necessary context for diagnosing complex issues. Balance is crucial: capture enough data to be useful without overwhelming the system.

Implement logging for key events, transactions, and user actions to create a comprehensive audit trail. Aim for detail that enables root cause analysis without excessive data volume.

3 Neglecting Log Rotation

Failure to implement proper log rotation can exhaust disk space, potentially causing database outages when the disk fills. Always configure size and time-based rotation with appropriate retention policies.

Automate log rotation frequently to prevent disk space issues, and set up alerts for when disk usage nears critical levels. Use compression to minimize storage while preserving data integrity.

4 Missing Correlation Context

Without application-specific identifiers in queries (e.g., comment tags), linking database activity to application events becomes challenging. Add context via query comments for easier troubleshooting.

Embed unique transaction IDs, user IDs, or session identifiers in query comments to connect database operations with originating application logic. Ensure consistent identifier application across all relevant database interactions.

Performance Impact: Logging Overhead

5-15%

I/O Overhead

Detailed logging can increase disk I/O by 5-15% on active systems. CSV logging has the greatest impact due to its structured format.

<1ms

Per-Query Cost

Each logged statement adds under 1ms of overhead. This cost accumulates with high query volumes.

2-3x

Log Size Growth

Full statement logging can increase log size by 2-3x, compared to logging only errors, affecting storage.

24h

Retention Balance

Most performance issues can be diagnosed using 24 hours of detailed logs, allowing for efficient rotation policies.

Real-World Example: Troubleshooting with Logs

Problem Identification

1

A financial application experienced intermittent slowdowns each afternoon between 2:00 PM and 3:00 PM. Users reported transaction delays exceeding 5 seconds for balance inquiries and transaction postings. System monitoring indicated no resource saturation; CPU, memory, and I/O metrics remained within normal ranges. The preliminary investigation suggested a bottleneck related to end-of-day reporting.

Logging Configuration

2

To isolate the root cause, we enabled **log_min_duration_statement = 1000** in PostgreSQL to capture queries exceeding 1 second. This helped identify slow-running SQL statements. We also enabled **log_lock_waits = on** to detect potential lock contention contributing to performance degradation. **log_statement='all'** was briefly enabled but quickly disabled due to excessive log generation.

Log Analysis

3

Using pgBadger, a PostgreSQL log analyzer, on 24 hours of logs, we discovered a specific reporting query executing daily at 2:00 PM. This query, designed to generate daily transaction summaries, triggered full table scans on large transactional tables, which blocked OLTP operations. These table scans locked significant database portions, causing the observed slowdowns.

Solution Implementation

4

To eliminate the bottleneck, we created a materialized view to pre-compute data for the reporting query. This view was refreshed during off-peak hours (midnight) to minimize the impact on production. The reporting query was modified to utilize the materialized view, avoiding scans of live transactional tables. This change resolved the slowdown during peak hours without altering report output or functionality, improving application performance and user experience.

Integrating Logs with Monitoring

Integrating PostgreSQL logs with system monitoring enhances observability. Combining log events with system metrics (CPU, memory, I/O) and application telemetry offers comprehensive context for troubleshooting. Each integration type varies in complexity and value.

Integration Type	Implementation Complexity (1-10)	Troubleshooting Value (1-10)
Basic File Collection	2	4
Centralized Logging	5	7
Real-time Analysis	7	8
Alerts Integration	6	9
AI-Powered Analysis	8	10

1

Basic File Collection

Basic File Collection: The simplest log integration, periodically collecting log files. Implementation is easy with scripts, but real-time analysis is limited, requiring manual data sifting.

2

Centralized Logging

Centralized Logging: Uses a dedicated logging server (e.g., rsyslog, Fluentd) for PostgreSQL logs, improving searchability and aggregation, but lacking real-time insights. Implementation involves configuring PostgreSQL to forward logs. It enhances log management across servers but lacks immediate alerting.

3

Real-time Analysis

Real-time Analysis: Processes logs in real-time with tools like Elasticsearch or Splunk, enabling immediate detection of anomalies. It requires a pipeline to ingest and index log data, offering proactive issue detection and faster root cause analysis, but demands specialized expertise.

4

Alerts Integration

Alerts Integration: Extends real-time analysis by setting up alerts for specific log patterns, notifying operations teams for immediate action, minimizing downtime. Setting up alerts involves defining rules and notification channels, ensuring timely responses to critical issues.

5

AI-Powered Analysis

AI-Powered Analysis: Employs machine learning to automatically identify anomalies and predict issues, detecting subtle patterns for advanced insights. It requires integrating machine learning models with the logging pipeline, significantly improving troubleshooting efficiency but demanding data science expertise.

Internal Resources & Advanced Techniques

For deeper insights into advanced PostgreSQL troubleshooting techniques, **explore our comprehensive guide: Comprehensive Postgres Troubleshooting**

Advanced users should also consider exploring custom log processing with pg_stat_statements, integrating with time-series databases for long-term analysis, and implementing automated anomaly detection to proactively identify performance issues before they impact users.

Custom Log Processing with pg_stat_statements:

This extension tracks SQL execution statistics. By enabling it and querying its views, you can identify frequently executed and slow queries. This enables targeted optimization efforts, reducing overall database load.

Integrating with Time-Series Databases:

Time-series databases like TimescaleDB or InfluxDB are optimized for storing and analyzing data points indexed in time order. Integrating PostgreSQL logs with these databases facilitates long-term trend analysis, capacity planning, and identifying performance regressions over time.

Automated Anomaly Detection:

Implementing automated anomaly detection involves setting up algorithms to monitor log patterns and detect deviations from normal behavior. Tools like Anodot or custom scripts can be used to analyze log data in real-time, triggering alerts when anomalies are detected. This helps in proactively addressing performance issues and preventing potential outages.

Contact MinervaDB for PostgreSQL Expertise

Our Services

- **PostgreSQL Health Checks & Performance Audits:** Comprehensive assessments to identify bottlenecks and areas for improvement in your PostgreSQL environment.
- **Database Architecture Design & Optimization:** Crafting efficient and scalable database architectures tailored to your specific business needs, ensuring optimal performance and resource utilization.
- **24/7 Production Support & Incident Response:** Round-the-clock support to address critical issues and minimize downtime, ensuring business continuity.
- **Customized Monitoring Solutions:** Tailored monitoring setups to proactively detect and alert on performance anomalies, providing real-time insights into your database health.
- **Migration Planning & Implementation:** Seamlessly migrate your databases with minimal disruption, leveraging our expertise to ensure a smooth transition.
- **Team Training & Knowledge Transfer:** Empower your team with the knowledge and skills to effectively manage and optimize your PostgreSQL databases.

Our Expertise

MinervaDB boasts a team of seasoned PostgreSQL experts with decades of experience in database administration, performance tuning, and troubleshooting. Our deep understanding of PostgreSQL internals enables us to deliver unparalleled solutions and support.

Contact Information

General Inquiries:
contact@minervadb.com

Shiv Iyer
Founder & CEO
shiv@minervadb.com

Website:
minervadb.xyz/blog/

Commitment to Excellence

At MinervaDB, we are committed to providing comprehensive PostgreSQL solutions that meet the evolving needs of our clients. Our focus on innovation and continuous improvement ensures that you receive the highest quality services and support.

Conclusion & Next Steps

Assess Your Logging Configuration

Evaluate your current PostgreSQL logging setup against the recommended practices. Pinpoint any areas where your existing strategy might be less effective for troubleshooting. Check whether your logging levels are appropriately set to capture necessary information without overwhelming the system with excessive data. Review your log rotation policies to ensure logs are managed efficiently.

Automate Log Analysis

Implement pgBadger or a similar tool to generate daily reports from your database logs. Build a dashboard to visualize slow query trends over time for proactive monitoring. Automate the process to quickly identify performance bottlenecks and optimize query performance. Set up alerts for unexpected spikes in slow queries or error rates.

Integrate Logs with Monitoring

Link your PostgreSQL logs to your current monitoring system. This allows you to correlate database events with broader system metrics and application performance data for comprehensive insights. This integration provides a holistic view of system health, allowing you to correlate database events with application performance. Use tools like Prometheus or Grafana to build comprehensive dashboards.

Engage PostgreSQL Experts

For complicated setups or ongoing problems, reach out to MinervaDB for specialized guidance on PostgreSQL performance optimization. Delve into our comprehensive resources at the **MinervaDB Blog**. Benefit from our team of seasoned PostgreSQL experts with decades of experience in database administration and performance tuning. Contact us for customized solutions tailored to your business needs.