Mukesh Chaurasia – SQL DBA

COURSE: POSTGRESQL DATABASE ADMINISTRATOR

DAY – 8

TOPIC: - Postgres Monitoring, Logs, and Alert Configuration

PostgreSQL monitoring, logging, and alert configuration are essential for ensuring a database's stability, performance, and security. By actively observing the database's health, analyzing logs for issues, and setting up alerts for critical events, administrators can proactively manage their systems.

PostgreSQL Logs 📝

PostgreSQL's logging system is a powerful tool for troubleshooting and performance analysis. Logs record server activity, from startup messages and errors to detailed query information.

Configuration

Log behavior is controlled by parameters in the **postgresql.conf** file. Here's a breakdown of key settings:

- **log_destination**: This parameter specifies where logs are sent. The most common value is stderr, which sends logs to the standard error stream. Using csvlog is also popular as it produces logs in a CSV format, which is easier for automated tools to parse.
- **logging_collector**: Set this to on to enable a background process that captures log messages and writes them to a file. This is the recommended way to handle logging as it ensures logs are organized and rotated.
- **log_directory**: Defines the directory where log files are stored. The default is pg_log within the data directory.
- **log_filename**: Specifies the naming pattern for log files (e.g., postgresql-%Y-%m-%d_%H%M%S.log).
- **log_statement**: Controls which SQL statements are logged.
    - none: No statements are logged.
    - ddl: Logs all Data Definition Language (DDL) statements like CREATE, ALTER, DROP.
    - mod: Logs all DDL and Data Modification Language (DML) statements like INSERT, UPDATE, DELETE.

  - all: Logs all statements, including SELECT. This can be very verbose and is generally used for specific debugging sessions.

- **log_min_duration_statement**: This is a crucial setting for performance tuning. It logs any statement that runs for longer than the specified number of milliseconds. For example, setting it to 1000 will log all queries that take more than one second to execute, helping you identify slow queries.

Example psql script for log configuration:

You can modify these settings directly in the postgresql.conf file or dynamically for the current session using ALTER SYSTEM or SET. The ALTER SYSTEM command modifies the postgresql.conf file and requires a server reload to take effect.

```
-- Set the minimum duration for statements to be logged to 1 second (1000ms)

ALTER SYSTEM SET log_min_duration_statement = '1000';


-- Log all DDL and DML statements

ALTER SYSTEM SET log_statement = 'mod';


-- Enable the logging collector

ALTER SYSTEM SET logging_collector = 'on';


-- Reload the configuration to apply changes

SELECT pg_reload_conf();
```

PostgreSQL Monitoring 📊

Monitoring is the practice of collecting and visualizing key metrics to understand the database's health and performance in real time.

Design Diagram

A typical monitoring setup involves a monitoring agent collecting data from PostgreSQL, a time-series database to store the metrics, and a dashboard tool for visualization and alerting.

1. **PostgreSQL Server**: The source of all metrics.

2. **Monitoring Agent**: A tool like **Prometheus** or an agent from Datadog that connects to PostgreSQL and collects metrics.
3. **Metrics Storage**: A time-series database like **Prometheus** or **InfluxDB** stores the collected data efficiently.
4. **Dashboard/Visualization**: A tool like **Grafana** connects to the metrics storage and displays the data on customizable dashboards.

Key Metrics to Monitor

- **System Metrics**: CPU usage, memory usage, disk I/O, and disk space.
- **Connection Metrics**: Total active connections, idle connections, and connection wait times.
- **Query Performance**: Long-running queries, queries per second (QPS), and transaction throughput.
- **Replication Metrics**: Replication lag (how far behind a replica is from the primary) and replication status.
- **Cache Hit Ratio**: The ratio of data found in the buffer cache versus needing to be read from disk. A low ratio can indicate a memory bottleneck.

Example psql script for monitoring:

PostgreSQL provides several built-in views to monitor its state. pg_stat_activity is one of the most useful for real-time monitoring.

```
-- View all active queries and their state

SELECT pid, datname, usename, client_addr, application_name, state, query_start,
now() - query_start AS duration, query

FROM pg_stat_activity

WHERE state != 'idle'

ORDER BY duration DESC;


-- Check the cache hit ratio for a specific table

SELECT relname,

    blks_read AS disk_reads,

    blks_hit AS buffer_hits,
    blks_hit * 100 / (blks_hit + blks_read) AS cache_hit_ratio
FROM pg_statio_all_tables
WHERE relname = 'your_table_name';
```

Alert Configuration 🚨

Alerts are the proactive component of monitoring. They notify administrators when a predefined condition is met, preventing potential issues from escalating into major problems.

Alert Conditions

Alerts are typically configured in your monitoring tool (e.g., Grafana, Datadog) based on the metrics you're collecting. Common alert conditions include:

- **High CPU Usage**: The average CPU usage is above 85% for 5 consecutive minutes.
- **Low Disk Space**: The available disk space on the volume containing the data directory is less than 10 GB.
- **High Replication Lag**: The replication lag on a replica server is greater than 100 MB or 60 seconds.
- **Long-Running Queries**: The number of queries running for longer than a specified duration (e.g., 5 minutes) exceeds a certain threshold.
- **Error Logs**: A specific critical error message appears in the logs (e.g., "PANIC: could not write to log file").

Notification Channels

Alerts are useless if they don't reach the right people. Most monitoring tools support various notification channels:

- **Email**: A direct and universal way to send notifications.
- **Slack/Microsoft Teams**: Integrations with these platforms can send alerts to a dedicated channel, making it easy for a team to see and respond.
- **SMS**: For critical alerts that require immediate attention.
- **PagerDuty/VictorOps**: Integration with incident management tools to handle on-call schedules and escalations.

Severity Levels

Assigning severity levels (e.g., Critical, Warning, Information) to alerts helps prioritize responses. A Critical alert for disk space running out would trigger an immediate response, while a Warning for a slightly high number of idle connections might only require a routine check later.

Understanding PostgreSQL Log Destinations

PostgreSQL provides flexible options for where to send server messages and activity logs. The log_destination parameter, found in the **postgresql.conf** file, controls this behavior. By default, logs are sent to stderr, which is the standard error stream. However, you can configure PostgreSQL to send logs to multiple destinations at once.

Available Log Destinations

PostgreSQL offers several built-in log destinations, each serving a different purpose:

- **stderr**: This is the default destination. Logs are written to a file, typically located in the pg_log subdirectory of your PostgreSQL data directory. This is the most common option for file-based logging.
- **csvlog**: When you include csvlog in your log_destination, PostgreSQL outputs log entries in a comma-separated value (CSV) format. This is particularly useful for automated log analysis because the structured data can be easily parsed by scripts or loaded into other programs. For csvlog to work, the logging_collector parameter must be enabled.
- **jsonlog**: Similar to csvlog, this option outputs log entries in JSON format. JSON is a modern, widely-used data format that makes it easy for applications to consume and process log data. Like csvlog, it requires the logging_collector to be enabled.
- **syslog**: This option sends log messages to the system's syslog daemon. This is ideal for centralizing logs from multiple applications on a single server or a log management system. However, on most Unix-like systems, you'll need to configure your syslog daemon (e.g., by adding a line like local0.* /var/log/postgresql to its configuration file) to ensure PostgreSQL's messages aren't discarded.
- **eventlog**: Available on Windows, this option sends logs to the Windows Event Log. This integrates PostgreSQL logs with the native Windows logging system, making them viewable in the Event Viewer. You'll need to register an event source with the operating system to ensure messages are displayed correctly.

The current_logfiles File

When you use stderr, csvlog, or jsonlog as a log destination, PostgreSQL creates a special file called **current_logfiles**. This file acts as a pointer to the log files currently in use by the logging collector. This is a convenient way to find the latest logs without needing to know the exact file name, which changes during log rotation.

The current_logfiles file is automatically updated whenever a new log file is created (due to rotation) or when log_destination is reloaded. It is removed if you disable the logging collector or if none of the file-based destinations (stderr, csvlog, jsonlog) are being used.

For example, if you have log_destination = 'stderr,csvlog', the current_logfiles file would contain:

```
stderr log/postgresql.log

csvlog log/postgresql.csv
```

Note

On most Unix systems, you will need to alter the configuration of your system's syslog daemon in order to make use of the syslog option for log_destination. PostgreSQL can log to syslog facilities LOCAL0 through LOCAL7 but the default syslog configuration on most platforms will discard all such messages. You will need to add something like:

```
local0.*    /var/log/postgresql
```

to the syslog daemon's configuration file to make it work.

On Windows, when you use the eventlog option for log_destination, you should register an event source and its library with the operating system so that the Windows Event Viewer can display event log messages cleanly.

Detailed Guide to PostgreSQL Logging Configuration

PostgreSQL's logging system is highly configurable, allowing you to control where, what, and how logs are generated. The core of this system is the **logging collector**, a background process that manages log output.

---

1. The Logging Collector: Purpose and Architecture

The **logging_collector** is a crucial background process that captures log messages from the PostgreSQL server. Instead of the server writing logs directly, which can be inefficient and less flexible, the collector intercepts messages sent to stderr and manages them. This approach offers several advantages over sending logs directly to syslog:

- **Reliability**: The collector can capture all messages, including those from the dynamic linker or failed scripts, which might not be picked up by syslog.
- **Centralized File Management**: It provides fine-grained control over log file naming, rotation, and storage location.
- **Format Flexibility**: The collector supports various log formats, including stderr, csvlog, and jsonlog.

Logging Architecture Diagram

1. **PostgreSQL Server Processes**: The main database processes generate log messages.
2. **stderr Stream**: All log messages are first sent to the standard error stream.
3. **Logging Collector**: When enabled, this background process intercepts the stderr stream.
4. **Log Files**: The collector writes the captured messages to log files based on the configured settings (location, format, and rotation rules).
5. **External Tools**: Monitoring and analysis tools can then read and process these log files.

---

2. Configuration Parameters: Methods and Scripts

All of these logging parameters are configured in the **postgresql.conf** file. Most require a server restart to take effect, while some can be reloaded.

logging_collector (boolean)

- **Purpose**: To enable or disable the background process that manages log files.
- **Method**: Set to on to enable.
- **Script**: This is a server start-time parameter and cannot be changed dynamically. You must edit postgresql.conf and restart the server.

```
# postgresql.conf
```

```
logging_collector = on
```

log_directory (string)

- **Purpose**: To specify where the log files will be stored.
- **Method**: Provide an absolute path or a path relative to the data directory. The default is log.
- **Script**:

```
# postgresql.conf
log_directory = 'pg_log'
```

log_filename (string)

- **Purpose**: To define the naming convention for log files.
- **Method**: Uses strftime patterns to create time-varying file names.
  - %Y: Year
  - %m: Month
  - %d: Day
  - %H: Hour
  - %M: Minute
  - %S: Second
- **Script**:

```
# This will create log files like `postgresql-2025-08-10_154044.log`
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'


# Example for daily logs named `server_log.Mon`, `server_log.Tue`
log_filename = 'server_log.%a'
```

log_file_mode (integer)

- **Purpose**: To set the file permissions for log files on Unix systems.
- **Method**: Use an octal number (e.g., 0600) to define permissions.
- **Script**:

```
# This sets permissions to r/w for the owner only.
```

log_file_mode = 0600

# This allows the server owner's group to read the logs.

log_file_mode = 0640

Note: To use 0640, the log directory should be outside the data directory for security.

log_rotation_age (integer)

- **Purpose**: To define the maximum time a single log file is used before a new one is created.
- **Method**: Set the value in minutes. 24h is the default. Setting to 0 disables time-based rotation.
- **Script**:

# Rotate logs every 24 hours.

log_rotation_age = 24h

# Rotate logs every 60 minutes.

log_rotation_age = 60

log_rotation_size (integer)

- **Purpose**: To define the maximum size a log file can reach before a new one is created.
- **Method**: Set the value in kilobytes. 10MB is the default. Setting to 0 disables size-based rotation.
- **Script**:

# Rotate logs when they reach 100 megabytes.

log_rotation_size = 100MB

log_truncate_on_rotation (boolean)

- **Purpose**: To control whether new log files overwrite old ones of the same name.
- **Method**: Set to on to truncate. This is useful for creating a cyclical log history. Truncation only occurs during time-based rotation.
- **Script**:

# Rotate hourly, overwriting the log file from the same hour last day.

```
log_filename = 'server_log.%H'

log_truncate_on_rotation = on

log_rotation_age = 60
```

---

3. Syslog and Event Log Parameters

These parameters are relevant only when you've included syslog or eventlog in your log_destination.

syslog_facility (enum)

- **Purpose**: To specify the syslog "facility" to use for logging.
- **Method**: Choose from LOCAL0 to LOCAL7. The default is LOCAL0.
- **Script**:

```
# postgresql.conf

syslog_facility = 'LOCAL5'
```

syslog_ident (string)

- **Purpose**: To set the program name that identifies PostgreSQL messages in syslog.
- **Method**: The default is postgres.
- **Script**:

```
# postgresql.conf

syslog_ident = 'my_db_instance'
```

syslog_sequence_numbers (boolean)

- **Purpose**: To prefix log messages with a sequence number, preventing syslog from suppressing repeated messages.
- **Method**: Defaults to on. You can set to off if you want to suppress repeated messages.

syslog_split_messages (boolean)

- **Purpose**: To control how long log messages are handled by syslog.
- **Method**: Defaults to on, which splits long messages into lines of 1024 bytes to fit typical syslog limits.

event_source (string)

- **Purpose**: To set the program name for PostgreSQL messages in the Windows Event Log.
- **Method**: The default is PostgreSQL.
- **Script**:

```
# postgresql.conf
event_source = 'MyPostgreSQLServer'
```

---

1. Controlling Log Verbosity with Message Levels 📢

The primary way to control the types of messages written to the log is through message levels. The log_min_messages parameter defines the minimum severity level a message must have to be logged.

- **log_min_messages (enum)**
  - **Purpose**: To filter messages based on severity.
  - **Method**: You can choose from levels ranging from DEBUG5 (most verbose) to PANIC (least verbose). Each level includes all the levels that are more severe. For instance, setting log_min_messages to WARNING will log WARNING, ERROR, LOG, FATAL, and PANIC messages, but will ignore INFO and NOTICE.
  - **Default**: WARNING. This provides a good balance, capturing important issues without logging routine informational messages.

---

2. Logging SQL Statements Based on Errors and Duration ⏳

To diagnose problems, it's often essential to log the SQL statements that cause them. PostgreSQL offers several parameters to control this behavior.

- **log_min_error_statement (enum)**
  - **Purpose**: To log the SQL statement that caused an error of a certain severity or higher.
  - **Method**: The value is an enum of severity levels, similar to log_min_messages. For example, setting this to ERROR (the default) will

log the statement that caused an ERROR, LOG, FATAL, or PANIC. This is invaluable for identifying the exact query that led to a problem.

- **Default**: ERROR.

- **log_min_duration_statement (integer)**
  - **Purpose**: To log the duration of queries that run for a minimum amount of time. This is a crucial tool for performance tuning.
  - **Method**: Set a value in milliseconds. Any query that runs for this duration or longer will have its duration logged. Setting it to 0 logs all query durations, and -1 (the default) disables this feature.
  - **Note**: This parameter logs all qualifying queries, regardless of traffic. If your database has high traffic and many slow queries, this can generate very large log files.
- **log_min_duration_sample (integer)**
  - **Purpose**: To sample and log the duration of slow queries, reducing logging overhead in high-traffic environments.
  - **Method**: This works in conjunction with log_statement_sample_rate. It defines a minimum duration for a query to be considered for sampling.
  - **Default**: -1 (disabled).
- **log_statement_sample_rate (floating point)**
  - **Purpose**: To determine the percentage of qualifying statements that will be logged.
  - **Method**: A floating-point value between 0.0 and 1.0. For example, a value of 0.1 means 10% of the statements exceeding log_min_duration_sample will be logged.
  - **Default**: 1.0.
- **log_transaction_sample_rate (floating point)**
  - **Purpose**: To log all statements from a randomly sampled fraction of transactions.
  - **Method**: A floating-point value from 0.0 to 1.0. A value of 0.01 means 1% of transactions will be logged in their entirety. This helps reconstruct the full context of a transaction, not just a single slow query.
  - **Default**: 0 (disabled).

3. Monitoring Startup Progress and Other Events ⚙️

- **log_startup_progress_interval (integer)**
    - **Purpose**: To log messages about long-running startup operations.
    - **Method**: Sets the time interval (in milliseconds) after which a message is logged for a startup operation that is still in progress. This can be helpful for diagnosing why a server takes a long time to start up or recover.
    - **Default**: 10s (10,000 milliseconds). Setting to 0 disables this feature. For example, if a data sync takes 25 seconds, with the default setting, a message would be logged at 10 seconds and again at 20 seconds.

📊 Overview: What Are Message Severity Levels?

PostgreSQL categorizes messages into severity levels to help developers and administrators understand the importance and urgency of each message. These levels determine:

- Whether a message is sent to the client.
- Whether it is logged to the server.
- How external logging systems like syslog or Windows eventlog interpret the message.

🔍 Severity Levels Breakdown

Severity	**Purpose**	**Syslog Level**	**Windows Eventlog Level**
DEBUG1–DEBUG5	Developer-level diagnostics; increasingly verbose with higher numbers.	DEBUG	INFORMATION
INFO	Informational messages requested by the user (e.g., VACUUM VERBOSE).	INFO	INFORMATION
NOTICE	Helpful messages for users (e.g., identifier truncation warnings).	NOTICE	INFORMATION

Severity	**Purpose**	**Syslog Level**	**Windows Eventlog Level**
WARNING	Alerts about likely issues (e.g., COMMIT outside a transaction block).	NOTICE	WARNING
ERROR	Command-level failure; aborts the current SQL command.	WARNING	ERROR
LOG	Administrative messages (e.g., checkpoint activity, autovacuum logs).	INFO	INFORMATION
FATAL	Session-level failure; aborts the current database session.	ERR	ERROR
PANIC	System-level failure; forces all sessions to terminate.	CRIT	ERROR

🧠 How PostgreSQL Uses These Levels

- **Client Visibility**: Controlled by client_min_messages. Messages below this threshold are not sent to the client.
- **Server Logging**: Controlled by log_min_messages. Messages below this threshold are not written to the server log.
- **RAISE Statement**: In PL/pgSQL, you can use RAISE to emit messages at any severity level:

```
RAISE NOTICE 'Processing job ID: %', job_id;

RAISE EXCEPTION 'Invalid user ID: %', user_id USING HINT = 'Check your input';
```

🛠️ Practical Use Cases

- **DEBUG**: Used during development to trace internal logic or performance bottlenecks.
- **INFO/NOTICE**: Useful for tracking routine operations or user-facing feedback.
- **WARNING/ERROR**: Critical for identifying and resolving operational issues.
- **LOG/FATAL/PANIC**: Essential for system administrators to monitor system health and crash recovery.

🔧 Configuration Tips

To fine-tune logging behavior:

```
# postgresql.conf
log_min_messages = warning
client_min_messages = notice
```

This setup ensures that only warnings and above are logged, while clients receive notices and above.

📃 PostgreSQL Logging & Debugging Parameters Explained

🔷 application_name

- **Purpose**: Identifies the client application connecting to PostgreSQL.
- **Usage**: Set via connection string, environment variable (PGAPPNAME), or ALTER SYSTEM.
- **Visibility**: Appears in pg_stat_activity, CSV logs, and optionally in regular logs via log_line_prefix.
- **Constraints**: Must be printable ASCII, max 64 characters. Non-ASCII replaced with C-style escapes.
- More details

🔷 Debugging Parameters

Parameter	**Description**
debug_print_parse	Logs the parse tree of each query.
debug_print_rewritten	Logs the output of the query rewriter.
debug_print_plan	Logs the execution plan of each query.
debug_pretty_print	Formats the above logs with indentation for readability (default: on).

- **Log Level**: All emit messages at LOG level.
- **Use Case**: Ideal for development and query analysis.
- Reference

🔷 Autovacuum & Checkpoints

Parameter	**Description**
log_autovacuum_min_duration	Logs autovacuum actions exceeding the threshold (ms). 0 logs all, -1 disables.
log_checkpoints	Logs checkpoint and restartpoint activity with buffer stats.

- **Use Case**: Helps track vacuum efficiency and I/O pressure.
- Reference

🔷 Connection Lifecycle

Parameter	**Description**
log_connections	Logs each connection attempt and authentication.
log_disconnections	Logs session termination and duration.

- **Use Case**: Useful for auditing and tracking client behavior.
- Reference

🔷 Query Duration & Verbosity

Parameter	**Description**
log_duration	Logs duration of every completed statement.
log_error_verbosity	Controls detail level: TERSE, DEFAULT, VERBOSE.

- **Use Case**: Performance profiling and error diagnostics.
- Reference

🔷 Hostname & Log Prefix

Parameter	**Description**
log_hostname	Logs hostname instead of IP (may impact performance).
log_line_prefix	Adds metadata to each log line using printf-style escapes.

- **Common Prefix**: %m [%p] %u@%d [%a] → timestamp, PID, user, DB, app name.

- Reference

🛠 PostgreSQL Log Analysis Script

Here's a basic shell script to parse and summarize PostgreSQL logs:

```
#!/bin/bash

LOGFILE="/var/log/postgresql/postgresql.log"


echo "🔍 Top 10 longest queries:"

grep "duration:" "$LOGFILE" | \

awk -F'duration: ' '{print $2}' | \

sort -nr | head -10


echo -e "\n📊 Error Summary:"

grep "ERROR:" "$LOGFILE" | \

awk -F'ERROR:' '{print $2}' | \

sort | uniq -c | sort -nr


echo -e "\n👥 Connection Summary:"

grep "connection authorized" "$LOGFILE" | \

awk '{print $NF}' | sort | uniq -c | sort -nr


echo -e "\n🧹 Autovacuum Events:"

grep "autovacuum" "$LOGFILE" | \

cut -d':' -f2- | sort | uniq -c | sort -nr
```

📈 Monitoring Tips

- Use **Grafana + Prometheus** to visualize log metrics.
- Enable **CSV or JSON logging** for structured ingestion.
- Track:

- Query durations
- Autovacuum frequency
- Connection churn
- Error types and frequency

🔧 Top Log Analysis Tools

Tool Name	Description	Type	Highlights
Elastic Stack (ELK)	Combines Elasticsearch, Logstash, and Kibana for log ingestion, search, and visualization.	Open-source	Real-time search, dashboards, ML-based anomaly detection
Grafana Loki	A log aggregation system designed to work seamlessly with Grafana.	Open-source	Efficient log querying, tight Grafana integration
Prometheus + Grafana	While Prometheus is more metric-focused, it can be extended for log monitoring.	Open-source	Alerting, visualization, time-series analysis
Graylog	Centralized log management with powerful search and alerting capabilities.	Open-source / Enterprise	Stream-based processing, user-friendly UI
Splunk	Enterprise-grade log analysis and SIEM platform.	Commercial	Real-time analytics, security insights, machine learning
Datadog Logs	Cloud-native observability platform with integrated log management.	Commercial	Unified metrics, traces, and logs; anomaly detection
New Relic	Full-stack observability with log correlation and performance monitoring.	Commercial	Deep integration with APM and infrastructure monitoring

Tool Name	**Description**	**Type**	**Highlights**
Better Stack	SQL-like log querying, real-time alerts, and collaborative dashboards.	Freemium	Fast search, ClickHouse backend, Grafana integration
Uptrace	Combines logs, traces, and metrics for full observability.	Open-source	OpenTelemetry support, trace correlation
Rsyslog	Lightweight and fast log forwarding and processing tool.	Open-source	Highly configurable, supports various protocols

🧠 Choosing the Right Tool

- **For PostgreSQL-heavy environments**: ELK, Grafana Loki, or Graylog work well with structured logs.
- **For cloud-native stacks**: Datadog, Better Stack, or New Relic offer seamless integrations.
- **For budget-conscious setups**: Rsyslog + Grafana or ELK provide robust open-source solutions.

🧩 Logging Prefix & Metadata

🔷 log_line_prefix

- **Purpose**: Adds metadata to each log line using escape sequences.
- **Default**: '%m [%p] ' → timestamp and process ID.
- **Common Escapes**:
  - %u = user, %d = database, %a = application name
  - %r = remote host/port, %e = SQLSTATE, %x = transaction ID
  - %Q = query ID (requires compute_query_id)
- **Tip**: End with a space or punctuation for readability.
- More on log_line_prefix

🔐 Lock & Recovery Monitoring

🔷 log_lock_waits

- **Logs**: When a session waits longer than deadlock_timeout for a lock.
- **Use Case**: Detect blocked queries and performance bottlenecks.
- Enable lock wait logging

🔷 **log_recovery_conflict_waits**

- **Logs**: When recovery is delayed due to conflicts (e.g., long-running queries on standby).
- **Use Case**: Diagnose replication lag and WAL apply delays.
- Details

🧵 **Parameter Logging**

🔷 **log_parameter_max_length**

- **Controls**: Max length of bind parameters in non-error logs.
- **Values**:
    - -1 = full value (default)
    - 0 = disabled
    - N = truncate to N bytes
- Documentation

🔷 **log_parameter_max_length_on_error**

- **Controls**: Max length of bind parameters in error logs.
- **Use Case**: Helps debug failed queries without exposing sensitive data.

📃 **Statement & Replication Logging**

🔷 **log_statement**

- **Options**: none, ddl, mod, all
- **Use Case**: Audit query activity; mod is common for tracking data changes.
- **Note**: Doesn't log syntax errors or failed parse/planning steps.
- Reference

🔷 **log_replication_commands**

- **Logs**: Each replication command (e.g., START_REPLICATION, slot changes).

- **Default**: Off
- Details

🧊 Temp Files & Timezone

🔷 log_temp_files

- **Logs**: Temp file name and size when deleted.
- **Values**:
    - -1 = disabled (default)
    - 0 = log all
    - N = log files ≥ N KB
- Guide

🔷 log_timezone

- **Sets**: Time zone for timestamps in logs.
- **Default**: GMT, but usually overridden during initdb.
- Reference

Using CSV-Format Log Output

This option emits log lines in comma-separated-values (CSV) format, with these columns: time stamp with milliseconds, user name, database name, process ID, client host:port number, session ID, per-session line number, command tag, session start time, virtual transaction ID, regular transaction ID, error severity, SQLSTATE code, error message, error message detail, hint, internal query that led to the error (if any), character count of the error position therein, error context, user query that led to the error (if any and enabled by log_min_error_statement), character count of the error position therein, location of the error in the PostgreSQL source code (if log_error_verbosity is set to verbose), application name, backend type, process ID of parallel group leader, and query id. Here is a sample table definition for storing CSV-format log output:

```
CREATE TABLE postgres_log

(

 log_time timestamp(3) with time zone,
```

```
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  backend_type text,
  leader_pid integer,
  query_id bigint,
  PRIMARY KEY (session_id, session_line_num)
);
```

To import a log file into this table, use the COPY FROM command:

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

It is also possible to access the file as a foreign table, using the supplied module.

There are a few things you need to do to simplify importing CSV log files:

1. Set log_filename and log_rotation_age to provide a consistent, predictable naming scheme for your log files. This lets you predict what the file name will be and know when an individual log file is complete and therefore ready to be imported.
2. Set log_rotation_size to 0 to disable size-based log rotation, as it makes the log file name difficult to predict.
3. Set log_truncate_on_rotation to on so that old log data isn't mixed with the new in the same file.
4. The table definition above includes a primary key specification. This is useful to protect against accidentally importing the same information twice.
   The COPY command commits all of the data it imports at one time, so any error will cause the entire import to fail. If you import a partial log file and later import the file again when it is complete, the primary key violation will cause the import to fail. Wait until the log is complete and closed before importing. This procedure will also protect against accidentally importing a partial line that hasn't been completely written, which would also cause COPY to fail.

🧠 What Is the Process Title?

The **process title** is the string shown in system tools like ps, top, or Windows Process Explorer to describe what each PostgreSQL backend process is doing. It helps DBAs and developers monitor activity without querying the database.

🔷 cluster_name (string)

- **Purpose**: Assigns a name to the PostgreSQL instance (cluster).
- **Visibility**: Appears in the process title of all server processes.
- **Use Case**: Distinguish between multiple PostgreSQL clusters running on the same host.
- **Constraints**:
  - Max 64 characters (NAMEDATALEN).

  - Only printable ASCII; others are escaped.
  - Default is empty (''), meaning no name is shown.
- **Set Location**: Only in postgresql.conf or at server start.
- More on cluster_name

🔷 **update_process_title (boolean)**

- **Purpose**: Controls whether PostgreSQL updates the process title with each new SQL command.
- **Default**:
  - on for Unix-like systems.
  - off for Windows (due to performance overhead).
- **Effect**:
  - When on, process titles reflect current activity (e.g., postgres: walreceiver streaming).
  - When off, titles remain static (e.g., just postgres: walreceiver).
- **Set Location**: Can be changed by superusers or users with SET privilege.
- More on update_process_title

🧪 **Example: Viewing Process Titles**

# On Linux

```
ps -u postgres -o pid,cmd
```

With update_process_title = on, you'll see detailed activity like:

```
postgres: user@db SELECT waiting
postgres: walreceiver streaming 7/C12BF730
```

With it off:

```
postgres: walreceiver
```

🧩 **Practical Tip**

If you're running multiple clusters (e.g., dev, staging, prod) on the same host, setting cluster_name helps you instantly identify which process belongs to which environment—especially useful during performance debugging or crash analysis.

Correlates **process titles** from the OS with **query activity** from pg_stat_activity, perfect for real-time insight into what each backend is doing.

🛠 Script: Correlate Process Titles with pg_stat_activity

This Bash + SQL combo script:

1. Uses ps to list PostgreSQL backend processes.
2. Queries pg_stat_activity for matching PIDs and SQL activity.
3. Outputs a joined view of OS-level process titles and database-level query info.

🔍 Bash + psql Script

```
#!/bin/bash

# Customize these

PGUSER="postgres"

PGDATABASE="your_db"

PGHOST="localhost"

PGPORT="5432"


echo "PID | Process Title | DB | User | State | Query"

echo "------------------------------------------------------------"

# Get active PostgreSQL backend PIDs and titles

ps -u postgres -o pid,command | grep "postgres:" | while read -r pid title; do

 # Query pg_stat_activity for matching PID

 query=$(psql -U "$PGUSER" -d "$PGDATABASE" -h "$PGHOST" -p "$PGPORT" -Atc \

  "SELECT datname, usename, state, query FROM pg_stat_activity WHERE pid = $pid;")


 if [ -n "$query" ]; then

  IFS='|' read -r db user state sql <<< "$query"

  echo "$pid | $title | $db | $user | $state | ${sql:0:50}..."

 fi
```

done

🧪 Sample Output

```
PID   | Process Title              | DB    | User     | State   | Query
------------------------------------------------------------------------
12345 | postgres: user@db SELECT      | mydb   | app_user | active  | SELECT * FROM orders WHERE status = 'pending'...
12346 | postgres: walreceiver streaming|       |          |         |
```

🧩 Notes

- This script assumes update_process_title = on for meaningful titles.
- You can run it periodically or integrate it into a Grafana panel via node_exporter + custom scripts.
- For deeper correlation, you could also join with pg_locks or pg_stat_statements.

🧠 What Are PostgreSQL Logs?

PostgreSQL logs are textual records of database activity, errors, configuration changes, and query execution. They're essential for:

- 🔍 Troubleshooting errors and crashes
- 📊 Performance analysis (e.g. slow queries)
- 🛡️ Auditing access and transactions
- 📈 Monitoring system health

⚙️ Key Logging Parameters in postgresql.conf

Parameter	**Description**
logging_collector	Enables log redirection from stderr to files
log_destination	Specifies output format: stderr, csvlog, jsonlog, syslog, eventlog
log_directory	Directory where logs are stored (when collector is enabled)

Parameter	**Description**
log_filename	Naming pattern for log files
log_min_messages	Minimum severity to log (e.g. warning, error, info)
log_statement	Controls which SQL statements are logged (none, ddl, mod, all)
log_min_duration_statement	Logs queries exceeding a time threshold (in ms)

You can view current log file paths using:

```
SELECT * FROM pg_current_logfile();
```

📁 Default Log Locations by OS

OS Type	**Default Log Path**
Debian-based	/var/log/postgresql/postgresql-x.x-main.log
Red Hat-based	/var/lib/pgsql/data/pg_log
Windows	C:\Program Files\PostgreSQL\<version>\data\pg_log

Use log_directory to customize this location.

🗒 Log Formats

1. stderr (Default)

- Simple text output
- Mixed with other stderr messages unless logging_collector is enabled

2. CSV Log

- Structured format for easy parsing
- Enable with:

```
log_destination = 'csvlog'

logging_collector = on
```

- You can create a foreign table to query logs directly:

```
CREATE FOREIGN TABLE pg_logs (

 log_time timestamp,
```

```
  user_name text,

  database_name text,

  ...

) SERVER file_fdw OPTIONS (filename '/path/to/log.csv', format 'csv');
```

3. JSON Log (PostgreSQL 15+)

- Ideal for structured logging and integration with tools like ELK or Loki

```
log_destination = 'jsonlog'
```

4. Syslog

- Sends logs to system daemon

```
log_destination = 'syslog'

syslog_facility = 'LOCAL0'

syslog_ident = 'postgres'

And in /etc/rsyslog.conf:

local0.* /var/log/postgresql.log
```

5. Windows Event Log

- For Windows environments:

```
log_destination = 'stderr, eventlog'

Register DLL:


regsvr32 pgsql_library_directory/pgevent.dll
```

🐳 **Logging in Docker & Kubernetes**

- Avoid editing logs inside containers directly.
- Use **ConfigMaps** to pass logging parameters at container startup.
- For Kubernetes, mount postgresql.conf or use environment variables via Helm charts.

⚠️ **Performance Considerations**

- Logging too much (e.g. log_statement = 'all') can degrade performance.

- logging_collector ensures no logs are lost but may block under high load.
- Consider using syslog or external log shippers (e.g. Fluentd, Vector) for high-throughput environments.

🔍 Best Practices

- ✅ Enable log_min_duration_statement to catch slow queries.
- ✅ Use csvlog or jsonlog for structured analysis.
- ✅ Rotate logs using log_rotation_age and log_rotation_size.
- ✅ Centralize logs with ELK, Loki, or Better Stack for observability.

PostgreSQL Log Locations and Management 📝

PostgreSQL's logging system is highly flexible, allowing you to configure where log files are stored and how they're managed. While the default setup is simple, advanced configurations are available to suit various needs, from local file management to centralized logging systems.

🧠 Why Logging Selectively Matters

Logging everything can lead to:

- 🚫 Noise that obscures critical issues
- 🐢 Performance degradation due to I/O overhead
- 💾 Excessive disk usage

Instead, focus on **high-value logs**: slow queries, errors, warnings, and key transaction events.

🔍 Key Logging Filters and Techniques

1. Slow Query Thresholds

Use log_min_duration_statement to log queries exceeding a time limit:

```
log_min_duration_statement = 1000  # Logs queries > 1 second
```

Or dynamically:

ALTER DATABASE your_db SET log_min_duration_statement = '1000ms';

This helps pinpoint bottlenecks and optimize query performance.

2. Statement Logging

Use log_statement to control which SQL statements are logged:

Value Description

none	No statements logged
ddl	Logs CREATE, ALTER, DROP
mod	Logs DDL + INSERT, UPDATE, DELETE
all	Logs every statement (can be noisy)

Example:

log_statement = 'mod'

Or dynamically:

ALTER DATABASE your_db SET log_statement = 'mod';

3. Sampling

Sampling reduces log volume while retaining visibility into problematic queries.

Parameter	**Purpose**
log_min_duration_sample	Logs sampled queries above this duration
log_statement_sample_rate	Fraction of statements to log (e.g., 0.1 = 10%)
log_transaction_sample_rate	Fraction of transactions to log

Example:

log_min_duration_sample = 500

log_statement_sample_rate = 0.05

⚠ **Caution**: Sampling may miss the exact query causing an issue. Use full logging during debugging.

🚦 PostgreSQL Log Levels

Control verbosity with log_min_error_statement:

Level	Use Case
DEBUG1-5	Developer-level detail
INFO	Verbose output for user actions
NOTICE	Informational messages (e.g., truncation)
WARNING	Potential issues
ERROR	Command-level failures
LOG	Admin-level events (e.g., checkpoints)
FATAL	Session termination
PANIC	System-wide failure

🔄 Mapping to External Systems

PostgreSQL	Syslog	Windows Event Log
DEBUG1–5	DEBUG	INFORMATION
INFO	INFO	INFORMATION
NOTICE	NOTICE	INFORMATION
WARNING	NOTICE	WARNING
ERROR	WARNING	ERROR
LOG	INFO	INFORMATION
FATAL	ERR	ERROR
PANIC	CRIT	ERROR

🧩 Best Practices

- ✅ Use log_min_duration_statement for performance tuning.
- ✅ Log mod or ddl statements, not all, to reduce noise.

- ✅ Enable sampling in high-throughput environments.
- ✅ Set log_min_error_statement to WARNING or ERROR in production.
- ✅ Use structured formats (csvlog, jsonlog) for parsing and analysis.

Types of PostgreSQL logs, categorized by their purpose and relevance to administrators and application users. This guide will help you understand what to monitor, where to find it, and how to configure it for effective debugging and auditing.

🧩 PostgreSQL Log Types Overview

PostgreSQL logs can be broadly divided into two categories:

1. 🛠 Admin-Specific Logs

These logs help database administrators manage the PostgreSQL server and diagnose system-level issues.

🔷 Startup Logs

- **Purpose**: Capture events during server startup.
- **Includes**: Configuration errors, missing files, permission issues.
- **Location**: Default log directory (e.g., /var/log/postgresql/, pg_log/).
- **Trigger**: Automatically generated when PostgreSQL starts.

🔷 Server Logs

- **Purpose**: Track runtime behavior of the PostgreSQL server.
- **Includes**: Background processes, checkpoints, autovacuum, replication.
- **Configuration**: Controlled via log_destination, logging_collector, and log_line_prefix.

2. 👥 Application-User-Specific Logs

These logs provide insight into user activity, query performance, and access patterns.

🔷 Query Logs

- **Purpose**: Record SQL statements executed by users.
- **Enable via**: log_statement = 'mod' or 'all'
- **Use Case**: Debugging, auditing, performance analysis.

🔷 Transaction Logs (WAL)

- **Purpose**: Maintain a record of all changes for crash recovery.
- **Format**: Binary, not human-readable.
- **Tools**: Use pg_receivexlog or pg_waldump to inspect.
- **Use Case**: Replication, PITR (Point-In-Time Recovery).

🔷 Connection Logs

- **Purpose**: Track client connections and disconnections.
- **Enable via**:

log_connections = on

log_disconnections = on

- **Use Case**: Security auditing, connection pool analysis.

🔷 Error Logs

- **Purpose**: Capture failed queries, syntax errors, permission issues.
- **Control via**: log_min_error_statement
- **Levels**: ERROR, FATAL, PANIC, etc.

🔷 Audit & Access Logs

- **Purpose**: Track who did what and when.
- **Enable via**:
  - log_statement
  - Third-party tools like pgAudit
- **Use Case**: Compliance, forensic analysis.

📁 Log Storage & Rotation

- **Default Locations**:
  - Debian: /var/log/postgresql/
  - Red Hat: /var/lib/pgsql/data/pg_log
  - Windows: C:\Program Files\PostgreSQL\<version>\data\pg_log
- **Custom Location**: Set via log_directory

- **Rotation**: Controlled by log_rotation_age and log_rotation_size

🧪 Tools for Log Analysis

- **pgBadger**: Fast, detailed PostgreSQL log analyzer with HTML reports.
- **pgAudit**: Adds detailed session and object-level audit logging.
- **ELK Stack / Grafana Loki**: Centralized log aggregation and visualization.
- **Better Stack**: Real-time log search and alerting.

✅ Best Practices

- Enable log_min_duration_statement to catch slow queries.
- Use log_statement = 'mod' for meaningful query logging.
- Rotate logs regularly to avoid disk bloat.
- Use structured formats like csvlog or jsonlog for parsing.
- Integrate with external tools for centralized monitoring.

🔄 PostgreSQL Log Rotation

PostgreSQL supports automatic log rotation based on **time** and **size**, helping prevent log files from growing indefinitely and consuming disk space.

🔧 Key Parameters

Parameter	**Description**
log_rotation_age	Time-based rotation. Value in minutes (e.g., 60 for hourly, 1440 for daily).
log_rotation_size	Size-based rotation. Value in kilobytes (e.g., 1048576 for 1 GB).
log_truncate_on_rotation	If on, overwrites the log file when rotated; if off, appends to it.
log_filename	Pattern for log file names. Supports strftime escapes like %a, %H, %d.

🧪 Example Configuration

logging_collector = on
log_directory = 'pg_log'
log_filename = 'postgresql-%a.log'   # rotates daily with weekday name
log_rotation_age = 1440   # rotate every 24 hours
log_rotation_size = 0   # disable size-based rotation
log_truncate_on_rotation = on   # overwrite old logs

You can also rotate logs hourly or weekly by adjusting log_rotation_age and using different filename patterns like %H (hour), %w (week of month), etc.

📃 Log Formatting

Formatting logs makes them easier to parse, analyze, and ingest into monitoring tools.

🔷 CSV Format

- Enable by setting log_destination = 'csvlog'
- Requires logging_collector = on
- Output is structured and ideal for importing into SQL tables or log analysis tools.

🔷 JSON Format

- Enable with log_destination = 'jsonlog'
- Useful for structured logging in modern observability stacks.

🔷 log_line_prefix

This parameter customizes the prefix of each log line. Example:

log_line_prefix = '%m [%p] %u@%d/%a '

This outputs:

- %m: Timestamp
- %p: Process ID
- %u: Username
- %d: Database name
- %a: Application name

You can include other tokens like %h (remote host), %x (transaction ID), %e (SQL state), etc. Full list is in PostgreSQL's documentation.

📊 Aggregation & Analysis Tools

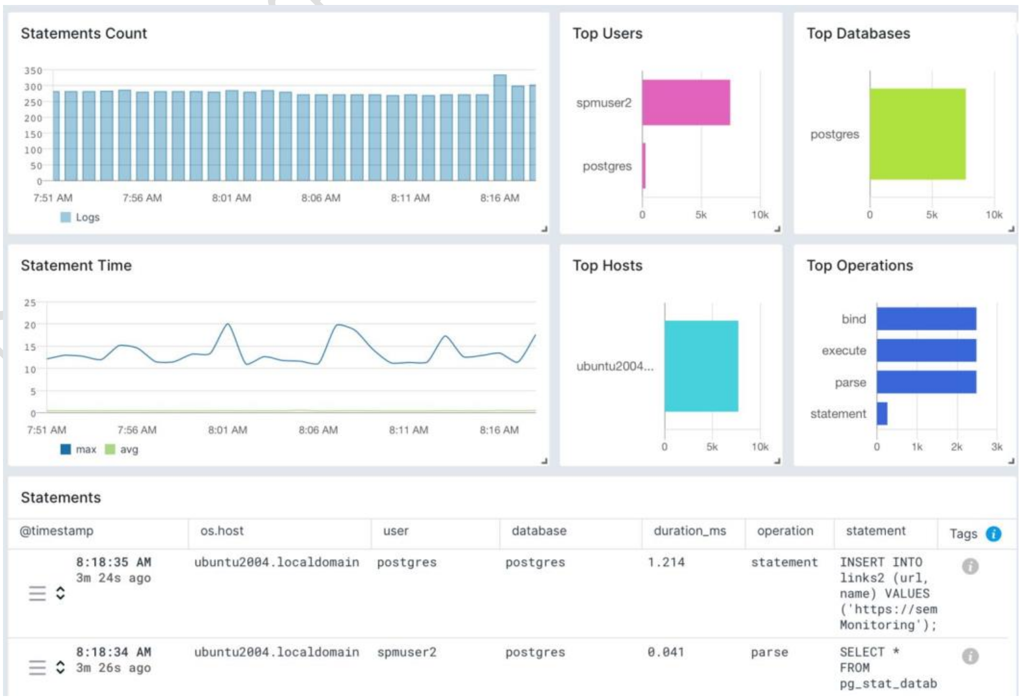
To avoid manually inspecting logs across servers, use centralized tools:

- **pgBadger**: Parses PostgreSQL logs and generates HTML reports.
- **Sematext**: Offers real-time log aggregation, search, and alerting.
- **ELK Stack (Elasticsearch, Logstash, Kibana)**: Popular open-source log analysis suite.
- **Grafana Loki**: Lightweight log aggregation with Grafana dashboards.

✅ Best Practices

- Use log_rotation_age and log_rotation_size together for robust control.
- Enable csvlog or jsonlog for structured output.
- Set log_line_prefix to include key identifiers for traceability.
- Rotate logs frequently and archive them for compliance and troubleshooting.
- Integrate with external tools for centralized visibility.

PostgreSQL Logging with Sematext



Statements

@timestamp	os.host	user	database	duration_ms	operation	statement	Tags
8:18:35 AM 3m 24s ago	ubuntu2004.localdomain	postgres	postgres	1.214	statement	INSERT INTO links2 (url, name) VALUES ('https://sem Monitoring');	
8:18:34 AM 3m 26s ago	ubuntu2004.localdomain	spmuser2	postgres	0.041	parse	SELECT * FROM pg_stat_datab	

PostgreSQL logging with Sematext

Sematext Logs is a log management and monitoring solution that lets you aggregate logs from various data sources across your infrastructure in one place for viewing and analysis.

Sematext features service auto-discovery so you just have to install the Sematext agent on your servers, perform some basic configuration, and your PostgreSQL logs will start flowing to Sematext and be presented via an intuitive, out-of-the-box dashboard. You can even easily create a custom dashboard, set up alerts, and send the alerts to different notification channels like Gmail, Slack, or PagerDuty.

PostgreSQL Database Monitoring With pg_stat_statements

pg_stat_statements is a built-in PostgreSQL extension that keeps track of SQL statements executed by the database server. It records details about query execution counts, total execution time, and I/O-related details.

What are long-running queries?

Long-running queries are simply queries that take a significant amount of time to execute, potentially causing performance issues or delays in database operations. These queries may consume excessive resources, such as CPU or memory, and can impact the overall responsiveness of the database system. Identifying and optimizing long-running queries is essential for maintaining the efficiency and reliability of the PostgreSQL database.

How to identify the long-running queries

You can use the following query to extract the top three longest-running queries in your PostgreSQL database:

```
SELECT
  userid :: regrole,
```

```
 dbid,
 mean_exec_time / 1000 as mean_exec_time_secs,
 max_exec_time / 1000 as max_exec_time_secs,
 min_exec_time / 1000 as min_exec_time_secs,
 stddev_exec_time,
 calls,
 query
from
 pg_stat_statements
order by
 mean_exec_time DESC limit 3;
```

```
userid  | dbid |  mean_exec_time_secs  | max_exec_time_secs | min_exec_time_secs
|  stddev_exec_time   | calls
|                                   query                          -----------------------------
--------------------------------------------------------------------

 semab   | 5   | 0.0018055465678913738 |     0.006023764 |      0.001152825
|  0.7903876933367537 |  3756 | SELECT name, setting, COALESCE(unit, $1),
short_desc, vartype FROM pg_settings WHERE vartype IN ($2, $3, $4) AND name != $5

 postgres | 5   |        0.001178044 |      0.001178044 |      0.001178044 |               0 |    1 |
SELECT pg_stat_statements_reset()

 semab   | 5   | 0.0007018039854898849 |      0.001922814 |      0.000597347 |
0.034553571651097015 | 15024 | SELECT pg_database_size($1)

(3 rows)
```

userid :: regrole: This syntax converts the **userid** column to the **regrole** data type, representing the executing role (user) of the SQL statement. Thus, it displays the actual **name** of the user instead of their **userid**, enhancing readability.

dbid: Represents the **database ID** where the SQL statement was executed.

mean_exec_time / 1000 as mean_exec_time_secs: it calculates the mean execution time(average execution time) of the SQL statement in seconds.

max_exec_time / 1000 as max_exec_time_secs: it calculates the maximum execution time of the SQL statement in seconds.

min_exec_time / 1000 as min_exec_time_secs: it calculates the minimum execution time of the SQL statement in seconds.

stddev_exec_time: It measures the amount of variation or dispersion in the execution times of the query. A higher **stddev_exec_time** indicates more variability in the query execution times, while a lower value suggests more consistency. This metric helps assess the stability and predictability of query performance.

calls: it indicates the number of times the SQL statement has been executed.

query: it represents the SQL query statement itself.

ORDER BY: it orders the result set based on the **mean_exec_time** column in descending order, meaning SQL statements with the longest mean execution times will appear first in the result set.

LIMIT 3: it limits the number of rows returned to three, ensuring only the top three SQL statements with the longest mean execution times are included in the result set.

What are I/O-intensive queries?

I/O-intensive queries are database operations that heavily rely on input/output operations, typically involving frequent reads or writes to disk. These queries often require significant disk access, leading to high disk I/O usage and potential performance bottlenecks.

How to identify I/O-intensive queries?

You can use the following query to identify queries that frequently access disk resources, indicating potential disk-intensive operations:

```
SELECT

 mean_exec_time / 1000 as mean_exec_time_secs,

 calls,

 rows,

 shared_blks_hit,

 shared_blks_read,

 shared_blks_hit /(shared_blks_hit + shared_blks_read):: NUMERIC * 100 as hit_ratio,

 (blk_read_time + blk_write_time)/calls as average_io_time_ms,

 query

FROM
```

```
  pg_stat_statements
where
  shared_blks_hit > 0
ORDER BY
  (blk_read_time + blk_write_time)/calls DESC;
```

shared_blks_hit: it retrieves the amount of data read from the shared buffer cache.

shared_blks_read: it retrieves the amount of data read from the disk.

shared_blks_hit /(shared_blks_hit + shared_blks_read):: NUMERIC * 100 as hit_ratio: It calculates the **hit ratio**, which represents the percentage of **shared blocks** found in the buffer cache compared to the total number of **shared blocks** accessed (both from cache and disk). You can calculate it as (**shared_blks_hit** / (**shared_blks_hit** + **shared_blks_read**)) * 100.

(blk_read_time + blk_write_time)/calls as average_io_time_ms: it calculates the average **I/O** time per call in milliseconds, which is the sum of block read and block write time divided by the number of calls.

WHERE: the filter shared_blks_hit > 0 retrieves the rows where the number of shared blocks in the buffer cache is greater than zero (0), focusing only on statements with at least one shared block hit.

ORDER BY: this filter **(blk_read_time + blk_write_time)/calls DESC** sorts the result in descending order based on the average **I/O** time per call.

PostgreSQL Database Monitoring With pg_stat_all_tables

pg_stat_all_tables is a system view in PostgreSQL that provides statistical information about all tables in the current database. It includes various metrics related to table access and usage, such as the number of **sequential** and **index** scans performed on each table, the number of **tuples inserted, updated,** and **deleted**, as well as information about **vacuum** and analysis operations.

What is a sequential scan?

A sequential scan refers to the process of scanning all the rows in a table sequentially, usually without using an index. It reads each row one by one from start to finish, which can be less efficient for large tables compared to using an index to access specific rows directly.

How to identify tables with the highest frequency of sequential scans

You can use the following query to retrieve the top three tables with the most sequential scans:

```
SELECT
  schemaname,
  relname,
  Seq_scan,
  idx_scan seq_tup_read,
  seq_tup_read / seq_scan as avg_seq_read
FROM
  pg_stat_all_tables
WHERE
  seq_scan > 0 AND schemaname not in (‘pg_catalog’,’information_schema’)
ORDER BY
  Avg_seq_read DESC LIMIT 3;
```

schemaname: the name of the schema containing the table.

relname: the name of the table.

seq_scan: the number of sequential scans initiated on the table.

idx_scan: the number of index scans initiated on the table.

seq_tup_read: the number of live rows fetched by sequential scans.

seq_tup_read / seq_scan as avg_seq_read: it calculates the average number of rows read per sequential scan.

seq_scan > 0: it selects only tables that have been sequentially scanned at least once.

schemaname not in ('pg_catalog', 'information_schema'): This clause in the SQL query filters out tables from the **pg_catalog** and **information_schema** schemas. These schemas contain system tables and views that are automatically created by PostgreSQL and are not typically user-created or user-managed.

ORDER BY: it orders the result set based on the calculated **avg_seq_read** column in descending order, meaning tables with the highest average sequential read rate will appear first in the result set.

LIMIT 10: it limits the number of rows returned to 10, ensuring only the top 10 tables with the highest average sequential read rates are included in the result set.

What are infrequently accessed tables?

Infrequently accessed tables in a database are those that are not frequently queried or manipulated. These tables typically have low activity and are accessed less frequently compared to other tables in the database. They may contain historical data, archival data, or data that is rarely used in day-to-day operations.

How to identify infrequently accessed tables?

The following query retrieves tables with low total scan counts (both **sequential** and **index** scans combined):

```
SELECT
 schemaname,
 relname,
 seq_scan,
 idx_scan,
 (COALESCE(seq_scan, 0) + COALESCE(idx_scan, 0)) as
total_scans_performed
FROM
 pg_stat_all_tables
WHERE
 (COALESCE(seq_scan, 0) + COALESCE(idx_scan, 0)) < 10
AND schemaname not in (‘pg_catalog’, ‘information_schema’)
ORDER BY
 5 DESC;
```

seq_scan: this column represents the number of **sequential** scans performed on the table.

idx_scan: this column represents the number of **index** scans performed on the table.

(COALESCE(seq_scan, 0) + COALESCE(idx_scan, 0)) as total_scans_performed: This expression calculates the total number of scans performed on the table by summing up the sequential scans and index scans. **COALESCE** function is used to handle **NULL** values by replacing them with 0.

(COALESCE(seq_scan, 0) + COALESCE(idx_scan, 0)) < 10: This condition filters the tables based on the total number of scans performed on each table. It selects tables where the total number of scans (**sequential scans** + **index scans**) is less than **10**.

total_scans_performed DESC: This clause orders the result set by the total number of scans performed on each table in **descending** order. Tables with the highest total number of scans appear first in the result set.

PostgreSQL Database Monitoring With pg_stat_activity

pg_stat_activity is a system view in PostgreSQL that provides information about the current activity of database connections. It includes one row per server process, showing details such as the username of the connected user, the database being accessed, the state of the connection (idle, active, waiting, etc.), the current query being executed, and more.

Finding long-running queries by time

The following SQL query retrieves information about currently running (**active**) database sessions in PostgreSQL that have been executing for longer than **one minute**.

```
SELECT
 datname AS database_name,
 usename AS user_name,
 application_name,
 client_addr AS client_address,
 client_hostname,
 query AS current_query,
 state,
 query_start,
 now() - query_start AS query_duration
FROM
 pg_stat_activity
WHERE
```

```
  state = 'active' AND now() - query_start > INTERVAL '10 sec'
ORDER BY
  query_start DESC;
```

now() - query_start AS query_duration: it calculates the duration of the query execution by subtracting the start time from the current time.

state = 'active': it filters the results to include only active queries currently running.

now() - query_start > INTERVAL '10 sec': it filters the results to include only queries that have been running for more than 10 seconds.

query_start DESC: it orders the results based on the query start time in descending order, showing the most recently started queries first.

Integrating PostgreSQL With Grafana and Prometheus for Database Monitoring

The PostgreSQL exporter (postgres_exporter) is a specialized tool designed to extract metrics and statistics from PostgreSQL database servers. It collects various performance-related data points and makes them available for monitoring and analysis through monitoring systems like Prometheus.

An open-source application used for event monitoring and alerting, Prometheus records metrics in a time-series database built using an HTTP pull model with flexible queries and real-time alerting.

Grafana is a versatile open-source analytics and interactive visualization tool accessible across multiple platforms. It enables the creation of charts, graphs, and alerts on the web, offering comprehensive insights when linked with compatible data sources.

For this integration, we are using an EC2 Ubuntu 22.04 instance where we are first setting up **PostgreSQL** and **Postgres exporter** and will then set up **Prometheus** and **Grafana.**

Ports information

PostgreSQL: 5432

Postgres exporter: 9100

Prometheus: 9090

Grafana: 3000

Set up PostgreSQL and the postgres_exporter

Install PostgreSQL

Install the latest version of PostgreSQL:

sudo sh -c 'echo "deb https://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'

wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -

sudo apt-get update

sudo apt-get -y install postgresql

Configure PostgreSQL

Modify the authentication method in the pg_hba.conf file to use scram-sha-256 instead of peer for local connections. This will allow you to use password-based authentication to the server.

```
Local    all    all    scram-sha-256
```

Restart the server:

sudo systemctl restart postgresql

Create a database user for monitoring and set a password for this user:

CREATE USER monitoring_user WITH PASSWORD test@1234 SUPERUSER;

Install the postgres_exporter

Download the latest release of the postgres_exporter binary:

wget https://github.com/prometheus-community/postgres_exporter/releases/download/v0.14.0/postgres_exporter-0.14.0.linux-amd64.tar.gz

Unzip the binary:

tar xzf postgres_exporter-0.14.0.linux-amd64.tar.gz

Move postgres_exporter binary to /usr/local/bin:

sudo cp postgres_exporter /usr/local/bin

Configure the postgres_exporter

Create a new directory under /opt to store connection information for the PostgreSQL server:

mkdir /opt/postgres_exporter

echo DATA_SOURCE_NAME="postgresql://monitoring_user:test@1234@localhost:5432/?sslmode=disable" > /opt/postgres_exporter/postgres_exporter.env

Create a service file for the postgres_exporter:echo '[Unit]

Description=Postgres exporter for Prometheus

Wants=network-online.target

After=network-online.target

[Service]

User=postgres

Group=postgres

WorkingDirectory=/opt/postgres_exporter

EnvironmentFile=/opt/postgres_exporter/postgres_exporter.env

ExecStart=/usr/local/bin/postgres_exporter --web.listen-address=localhost:9100 --web.telemetry-path=/metrics

Restart=always

[Install]|

WantedBy=multi-user.target' >> /etc/systemd/system/postgres_exporter.service

Since we created a new service file, it is better to reload the demon once so it recognizes the new file:

sudo systemctl daemon-reload

Start and enable the postgres_exporter service:

sudo systemctl start postgres_exporter

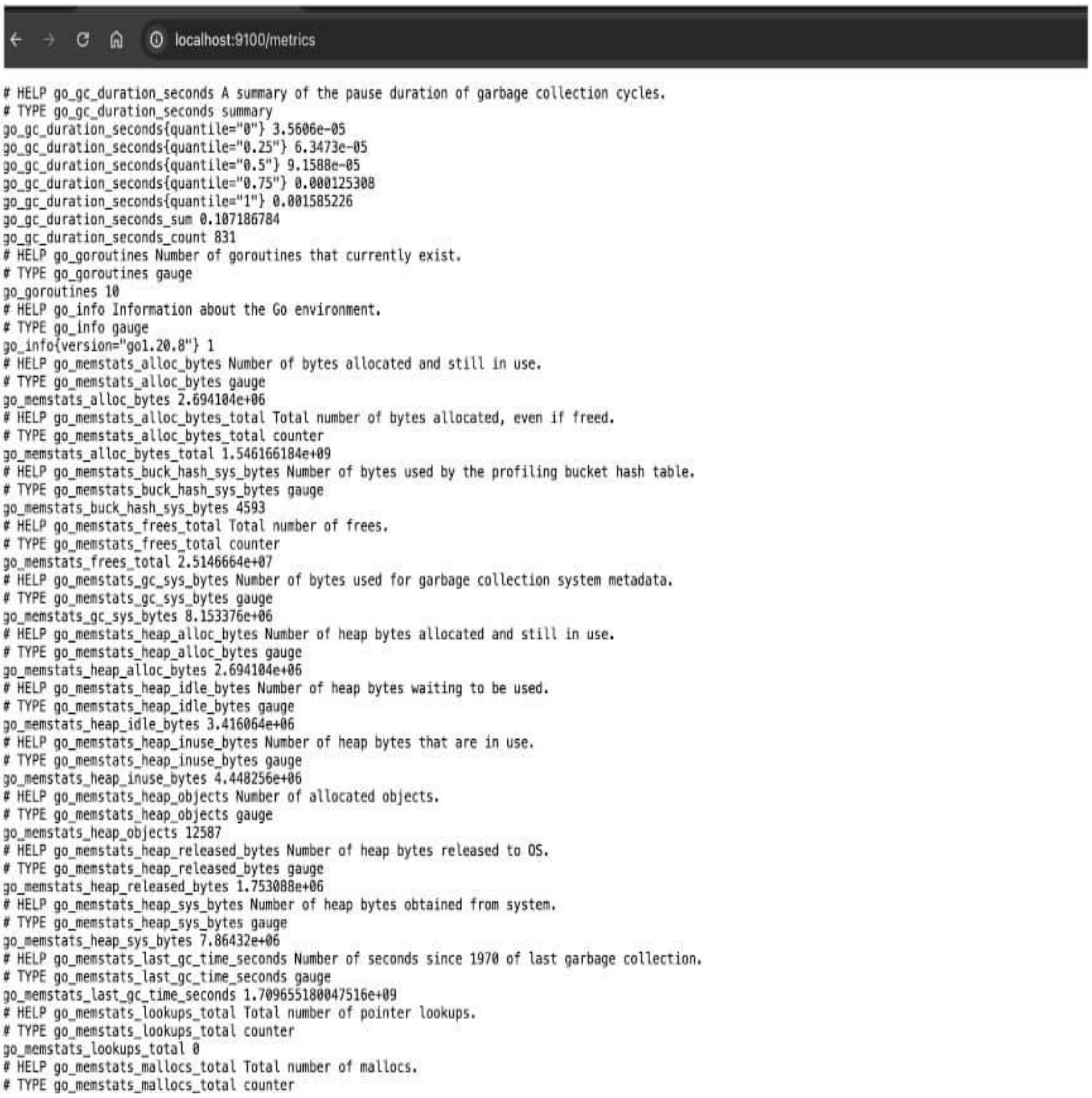
sudo systemctl enable postgres_exporter

Check the service status:

sudo systemctl status postgres_exporter

```
 postgres_exporter.service - Prometheus exporter for Postgresql
   Loaded: loaded (/etc/systemd/system/postgres_exporter.service; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2024-03-05 13:52:56 UTC; 2h 15min ago
  Main PID: 9438 (postgres_export)
   Tasks: 6 (limit: 9498)
```

Verify the postgres_exporter setup from the browser:



```
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 3.5606e-05
go_gc_duration_seconds{quantile="0.25"} 6.3473e-05
go_gc_duration_seconds{quantile="0.5"} 9.1588e-05
go_gc_duration_seconds{quantile="0.75"} 0.000125308
go_gc_duration_seconds{quantile="1"} 0.001585226
go_gc_duration_seconds_sum 0.107186784
go_gc_duration_seconds_count 831
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 10
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.20.8"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 2.694104e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 1.546166184e+09
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 4593
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 2.5146664e+07
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 8.153376e+06
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 2.694104e+06
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 3.416064e+06
# HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use.
# TYPE go_memstats_heap_inuse_bytes gauge
go_memstats_heap_inuse_bytes 4.448256e+06
# HELP go_memstats_heap_objects Number of allocated objects.
# TYPE go_memstats_heap_objects gauge
go_memstats_heap_objects 12587
# HELP go_memstats_heap_released_bytes Number of heap bytes released to OS.
# TYPE go_memstats_heap_released_bytes gauge
go_memstats_heap_released_bytes 1.753088e+06
# HELP go_memstats_heap_sys_bytes Number of heap bytes obtained from system.
# TYPE go_memstats_heap_sys_bytes gauge
go_memstats_heap_sys_bytes 7.86432e+06
# HELP go_memstats_last_gc_time_seconds Number of seconds since 1970 of last garbage collection.
# TYPE go_memstats_last_gc_time_seconds gauge
go_memstats_last_gc_time_seconds 1.709655180047516e+09
# HELP go_memstats_lookups_total Total number of pointer lookups.
# TYPE go_memstats_lookups_total counter
go_memstats_lookups_total 0
# HELP go_memstats_mallocs_total Total number of mallocs.
# TYPE go_memstats_mallocs_total counter
```

Set up the Prometheus and Grafana server

Install Prometheus

Create a system group named Prometheus:

```
sudo groupadd --system prometheus
```

Create a system user named Prometheus in a Prometheus group without an interactive login:

```
sudo useradd -s /sbin/nologin --system -g prometheus prometheus
```

Creating the required directory structure:

```
sudo mkdir /etc/prometheus

sudo mkdir /var/lib/prometheus
```

Download the Prometheus source:

```
wget
https://github.com/prometheus/prometheus/releases/download/v2.43.0/prometheus-2.43.0.linux-amd64.tar.gz
```

Decompress the source code:

```
tar vxf prometheus*.tar.gz
```

Set up proper permissions for the installation files:

```
cd prometheus*/

sudo mv prometheus /usr/local/bin

sudo mv promtool /usr/local/bin

sudo chown prometheus:prometheus /usr/local/bin/prometheus

sudo chown prometheus:prometheus /usr/local/bin/promtool

sudo mv consoles /etc/prometheus

sudo mv console_libraries /etc/prometheus

sudo mv prometheus.yml /etc/prometheus

sudo chown prometheus:prometheus /etc/prometheus

sudo chown -R prometheus:prometheus /etc/prometheus/consoles

sudo chown -R prometheus:prometheus /etc/prometheus/console_libraries

sudo chown -R prometheus:prometheus /var/lib/prometheus
```

Configure Prometheus

Add the PostgreSQL Exporter configurations inside the prometheus.yml file at the following location:

/etc/prometheus/prometheus.yml

```
scrape_configs:

 - job_name: "Postgres exporter"

  scrape_interval: 5s

  static_configs:

   - targets: [localhost:9100]
```

Create a new service file for Prometheus at the following location:/etc/systemd/system/prometheus.service

```
[Unit]

Description=Prometheus

Wants=network-online.target

After=network-online.target

[Service]

User=prometheus

Group=prometheus

Type=simple


ExecStart=/usr/local/bin/prometheus \

  --config.file /etc/prometheus/prometheus.yml \

  --storage.tsdb.path /var/lib/prometheus/ \

  --web.console.templates=/etc/prometheus/consoles \

  --web.console.libraries=/etc/prometheus/console_libraries

[Install]

WantedBy=multi-user.target
```

Reload **systemd** manager:

```
sudo systemctl daemon-reload
```

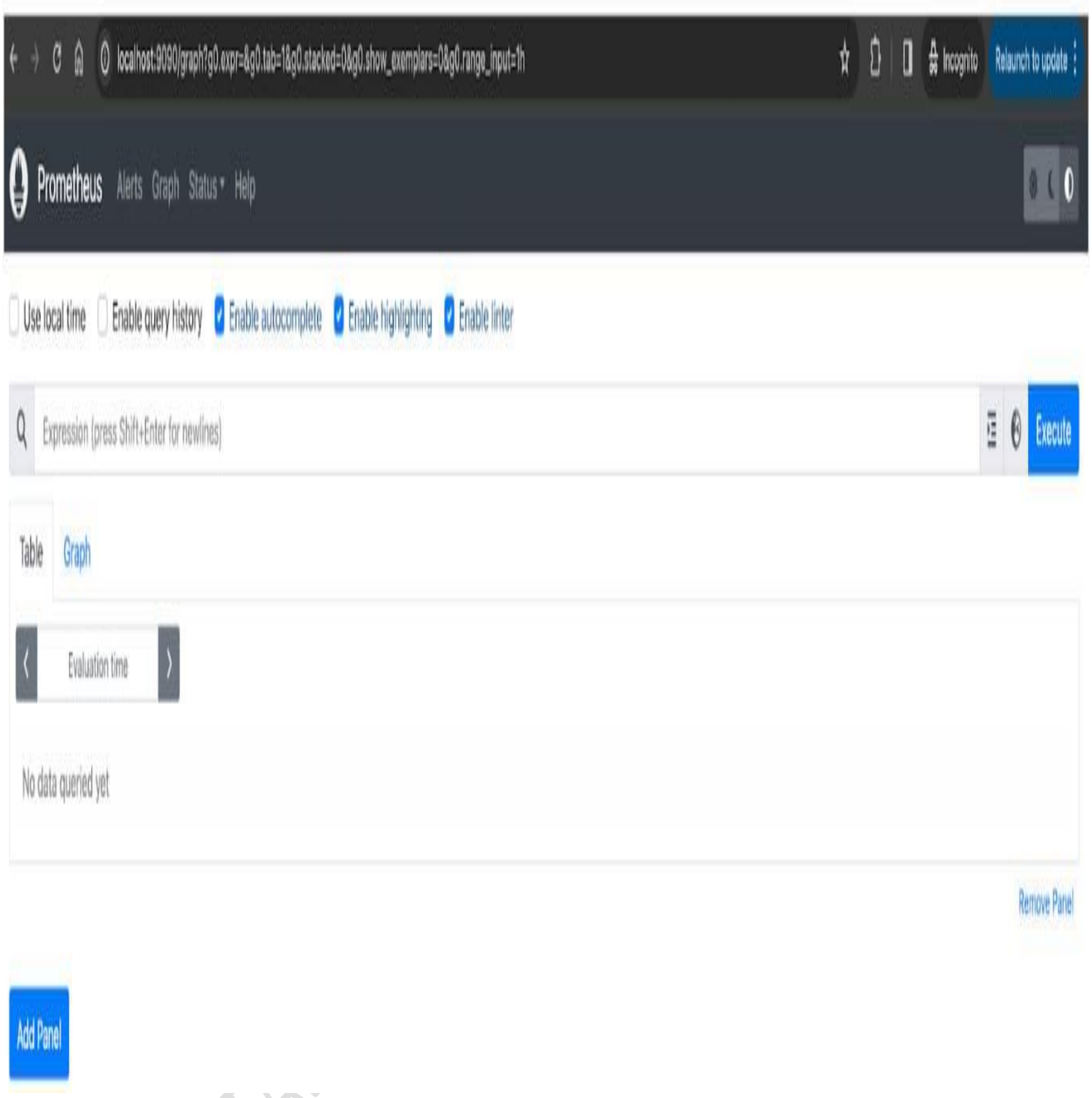
Start the Prometheus service:

```
sudo systemctl enable prometheus
sudo systemctl start prometheus
```

Verify the Prometheus service:

```
sudo systemctl status prometheus
 prometheus.service - Prometheus
   Loaded: loaded (/etc/systemd/system/prometheus.service; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2024-03-05 13:53:51 UTC; 2h 27min ago
  Main PID: 9470 (prometheus)
    Tasks: 8 (limit: 9498)
```

Verify the Prometheus setup from the browser:



Install Grafana

Install the pre-required packages:

```
sudo apt install -y apt-transport-https software-properties-common
```

Add the Grafana GPG key:

```
sudo mkdir -p /etc/apt/keyrings/
```

```
wget -q -O - https://apt.grafana.com/gpg.key | gpg --dearmor | sudo tee
/etc/apt/keyrings/grafana.gpg > /dev/null
```

Add Grafana's APT repository:

```
echo "deb [signed-by=/etc/apt/keyrings/grafana.gpg] https://apt.grafana.com stable main" | sudo tee -a /etc/apt/sources.list.d/grafana.list
```

```
sudo apt update
```

Install the Grafana package:

```
sudo apt install grafana
```

Start the Grafana service:

```
sudo systemctl start grafana-server
```

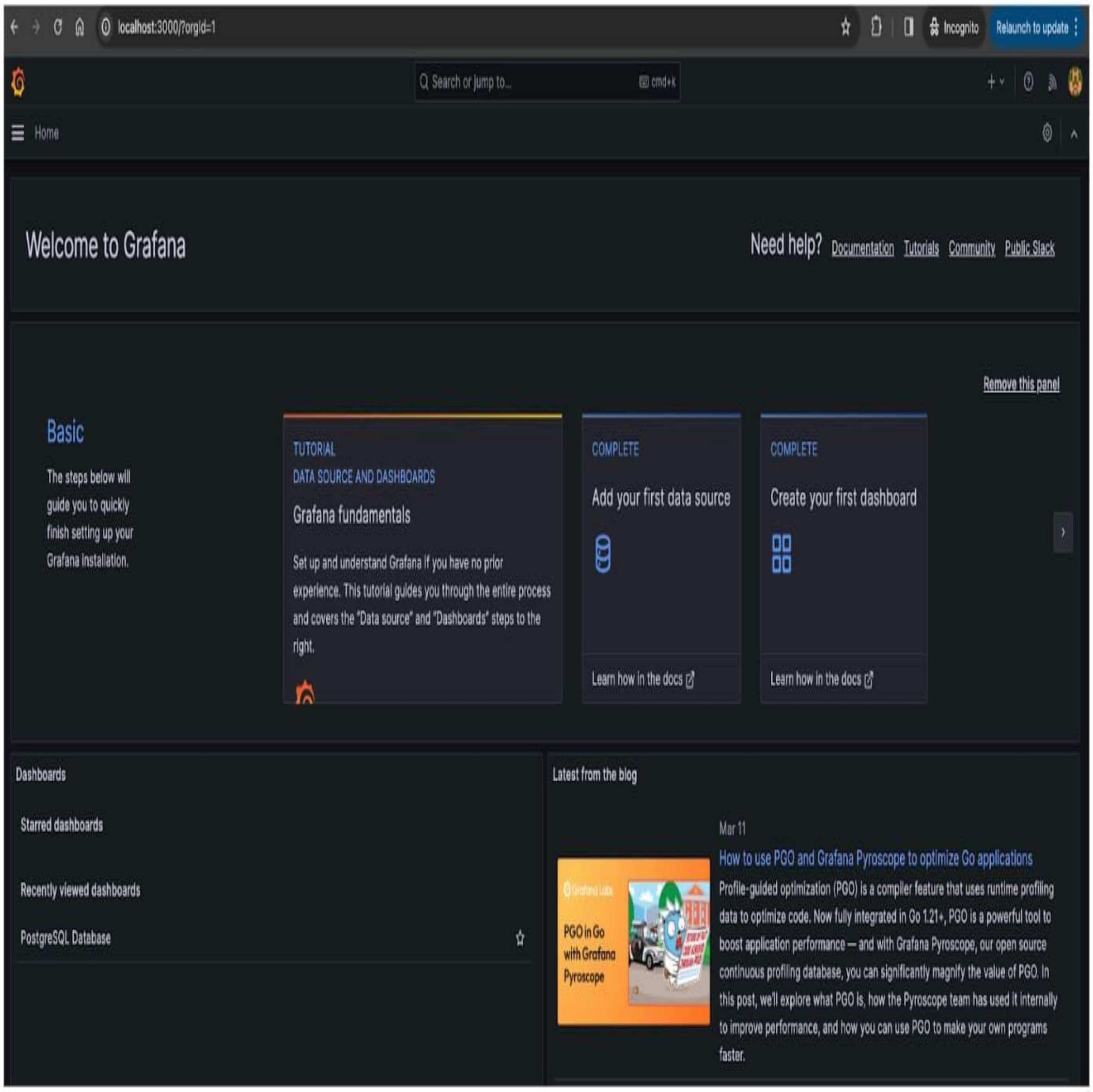
```
sudo systemctl enable grafana-server
```

Verify the Grafana service:

```
sudo systemctl status grafana-server
```

```
● grafana-server.service - Grafana instance
   Loaded: loaded (/lib/systemd/system/grafana-server.service; enabled; vendor preset: enabled)
    Active: active (running) since Tue 2024-03-12 05:45:23 UTC; 1h 29min ago
      Docs: http://docs.grafana.org
  Main PID: 719 (grafana)
```

Verify your Grafana setup from the browser:



Welcome to Grafana

Need help? Documentation Tutorials Community Public Slack

Remove this panel

Basic

The steps below will guide you to quickly finish setting up your Grafana installation.

TUTORIAL
DATA SOURCE AND DASHBOARDS
Grafana fundamentals

Set up and understand Grafana if you have no prior experience. This tutorial guides you through the entire process and covers the "Data source" and "Dashboards" steps to the right.

COMPLETE
Add your first data source
Learn how in the docs

COMPLETE
Create your first dashboard
Learn how in the docs

Dashboards
Starred dashboards
Recently viewed dashboards
PostgreSQL Database

Latest from the blog
Mar 11
How to use PGO and Grafana Pyroscope to optimize Go applications
Profile-guided optimization (PGO) is a compiler feature that uses runtime profiling data to optimize code. Now fully integrated in Go 1.21+, PGO is a powerful tool to boost application performance — and with Grafana Pyroscope, our open source continuous profiling database, you can significantly magnify the value of PGO. In this post, we'll explore what PGO is, how the Pyroscope team has used it internally to improve performance, and how you can use PGO to make your own programs faster.

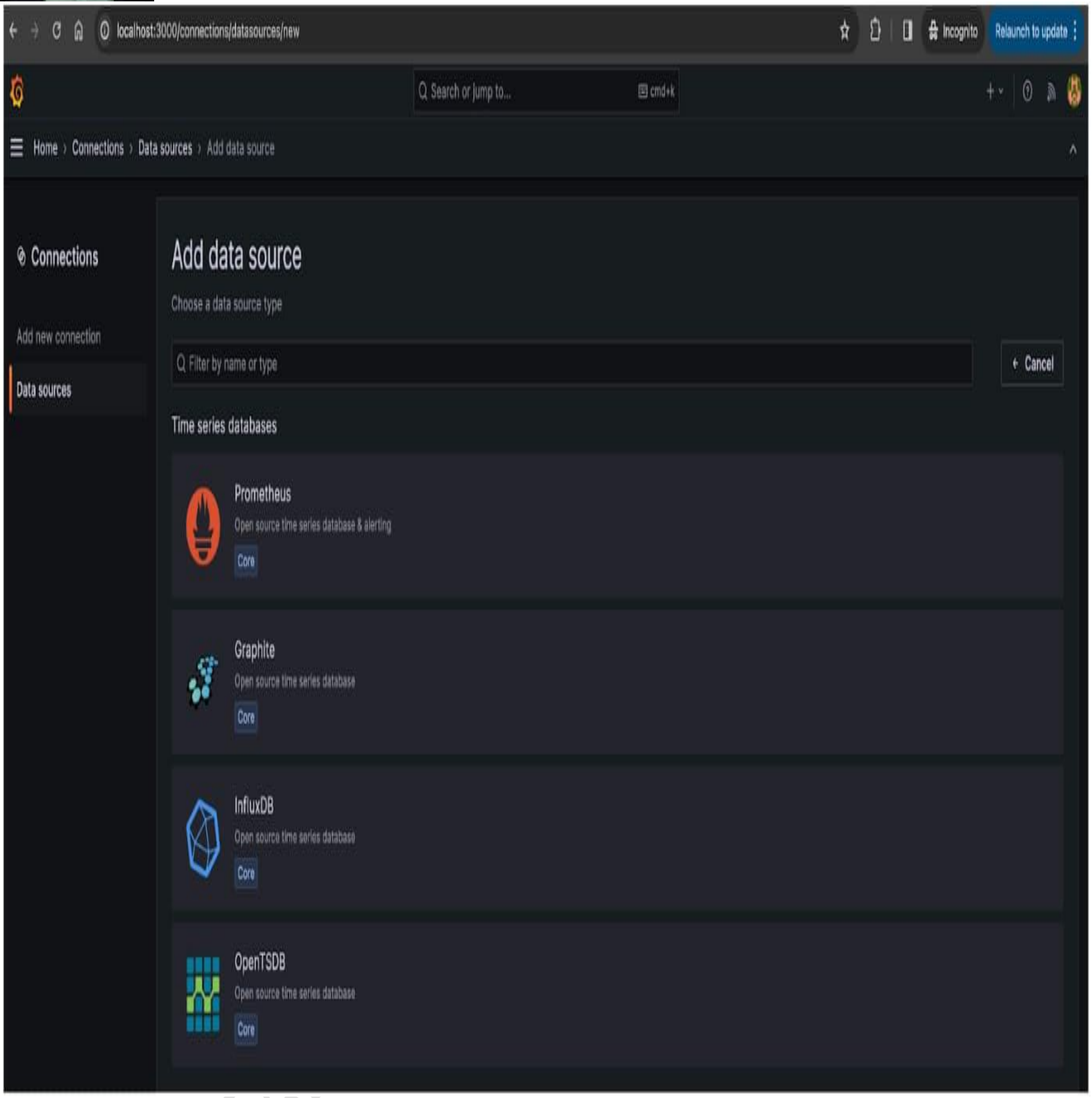
Note: if you see the login screen, log in by specifying the following credentials.

Username: admin

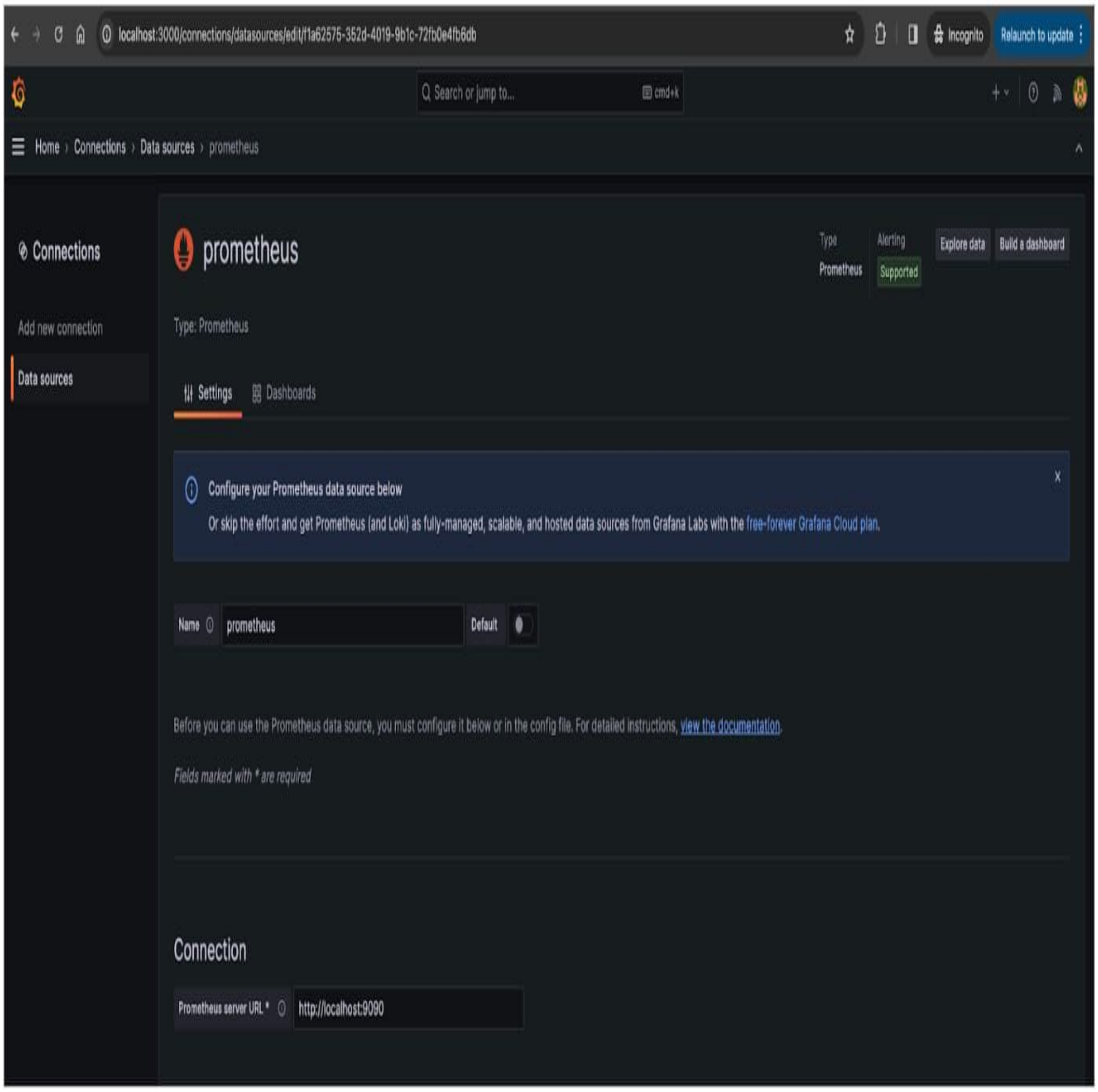
Password: admin

Integrate Prometheus server with Grafana

Add new data source for Prometheus:


localhost:3000/connections/datasources/new
Incognito
Relaunch to update
Search or jump to...
cmd+k
Home › Connections › Data sources › Add data source
Connections
Add new connection
Data sources
Add data source
Choose a data source type
Filter by name or type
Cancel
Time series databases
Prometheus
Open source time series database & alerting
Core
Graphite
Open source time series database
Core
InfluxDB
Open source time series database
Core
OpenTSDB
Open source time series database
Core

Configure the data source, click Save, and test.



Type: Prometheus

Configure your Prometheus data source below

Or skip the effort and get Prometheus (and Loki) as fully-managed, scalable, and hosted data sources from Grafana Labs with the free-forever Grafana Cloud plan.

Name: prometheus
Default

Before you can use the Prometheus data source, you must configure it below or in the config file. For detailed instructions, view the documentation.

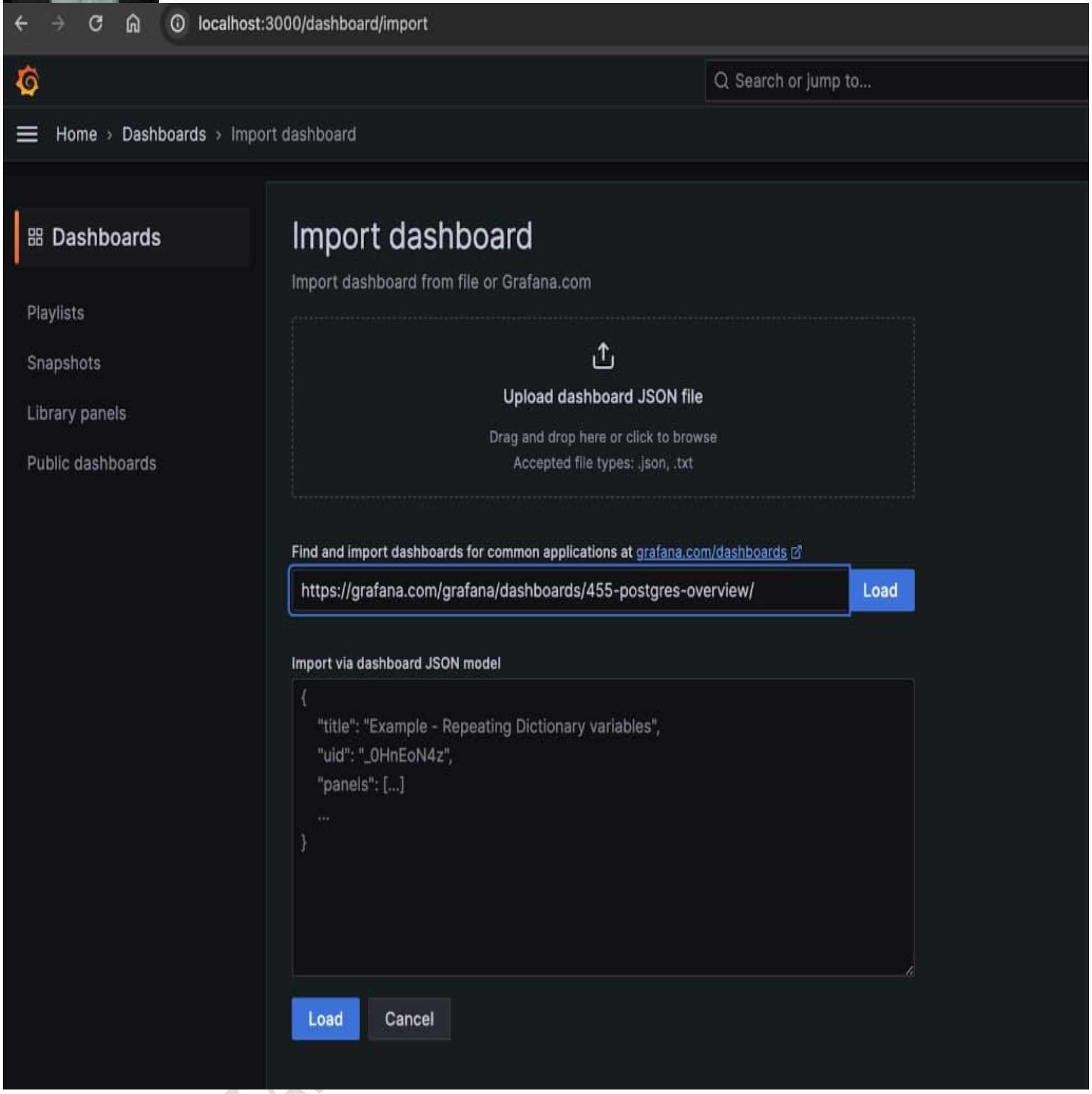
*Fields marked with * are required*

Connection

Prometheus server URL *: http://localhost:9090

Create a new dashboard

Find a PostgreSQL Grafana dashboard online and copy the URL of the dashboard that suits your preferences. Paste the URL and select Load.


localhost:3000/dashboard/import
Search or jump to...
Home › Dashboards › Import dashboard
Dashboards
Playlists
Snapshots
Library panels
Public dashboards
Upload dashboard JSON file
Drag and drop here or click to browse
Accepted file types: .json, .txt
Load
Import via dashboard JSON model
Load
Cancel

Import dashboard

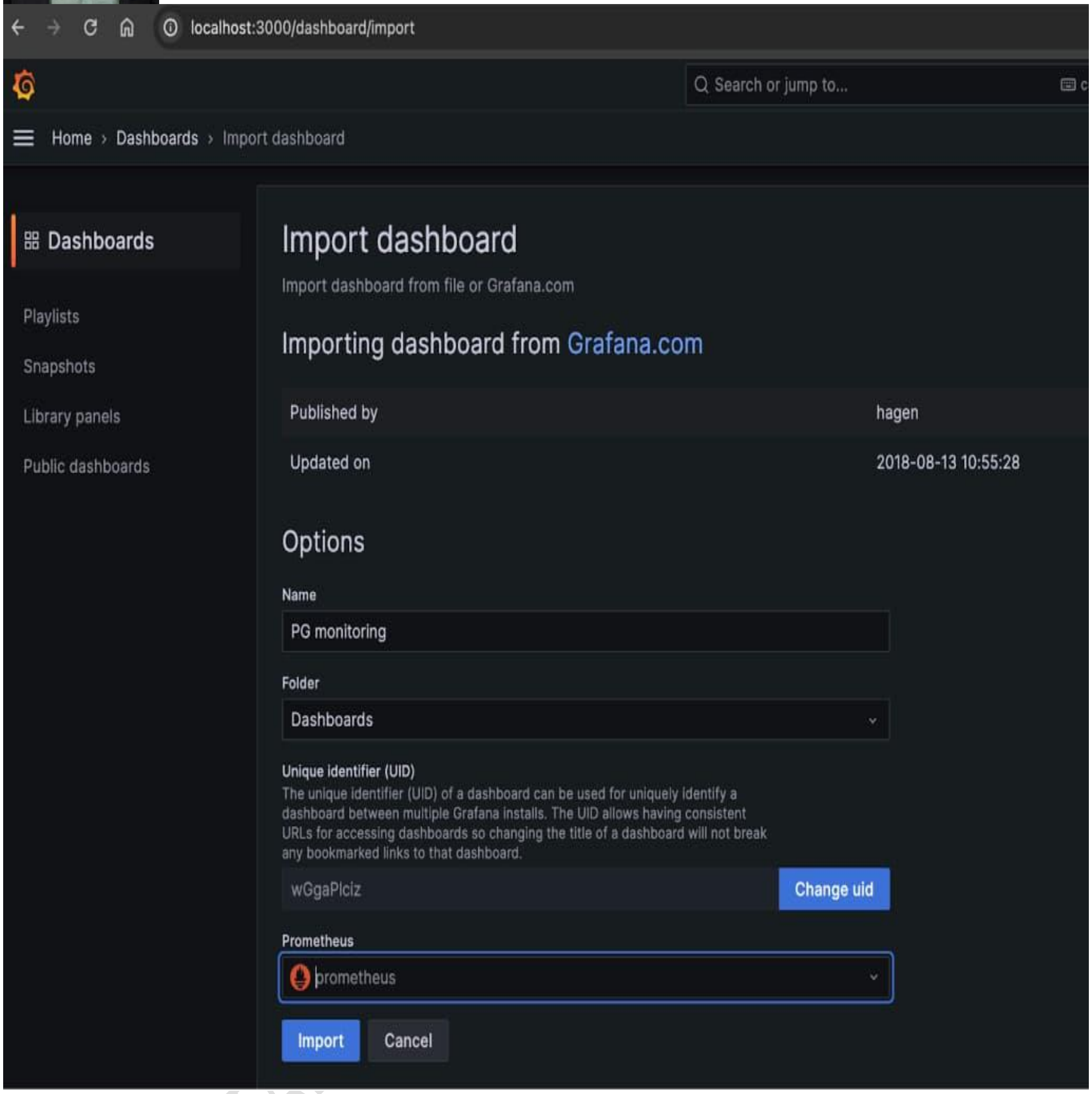
Import dashboard from file or Grafana.com

Find and import dashboards for common applications at grafana.com/dashboards

https://grafana.com/grafana/dashboards/455-postgres-overview/

```
{
  "title": "Example - Repeating Dictionary variables",
  "uid": "_0HnEoN4z",
  "panels": [...]
  ...
}
```

Fill out the required fields. Select Prometheus from the data source and click Import.



localhost:3000/dashboard/import

Home › Dashboards › Import dashboard

Import dashboard

Import dashboard from file or Grafana.com

Importing dashboard from Grafana.com

Published by	hagen
Updated on	2018-08-13 10:55:28

Options

Name: PG monitoring

Folder: Dashboards

Unique identifier (UID)
The unique identifier (UID) of a dashboard can be used for uniquely identify a dashboard between multiple Grafana installs. The UID allows having consistent URLs for accessing dashboards so changing the title of a dashboard will not break any bookmarked links to that dashboard.

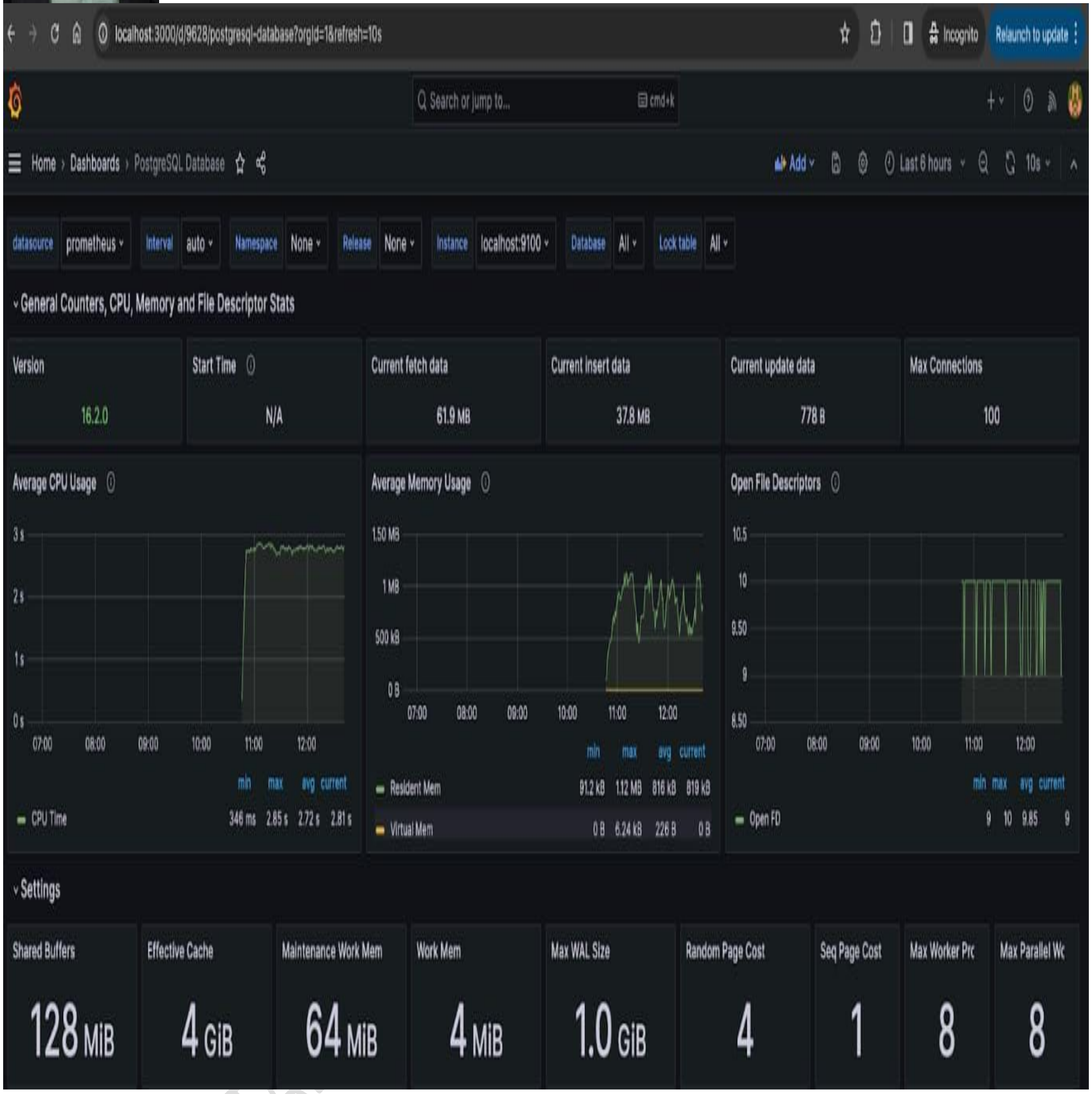
wGgaPlciz

Prometheus: prometheus

Your dashboard is ready!

Mukesh Chaurasia – SQL DBA

COURSE: POSTGRESQL DATABASE ADMINISTRATOR


localhost:3000/d/9628/postgresql-database?orgId=1&refresh=10s
Incognito
Relaunch to update
Search or jump to...
cmd+k
Home › Dashboards › PostgreSQL Database
Add
Last 6 hours
10s

datasource prometheus · Interval auto · Namespace None · Release None · Instance localhost:9100 · Database All · Lock table All

General Counters, CPU, Memory and File Descriptor Stats

Version	Start Time	Current fetch data	Current insert data	Current update data	Max Connections
16.2.0	N/A	61.9 MB	37.8 MB	778 B	100

Average CPU Usage

	min	max	avg	current
CPU Time	346 ms	2.85 s	2.72 s	2.81 s

Average Memory Usage

	min	max	avg	current
Resident Mem	91.2 kB	1.12 MB	816 kB	819 kB
Virtual Mem	0 B	6.24 kB	226 B	0 B

Open File Descriptors

	min	max	avg	current
Open FD	9	10	9.85	9

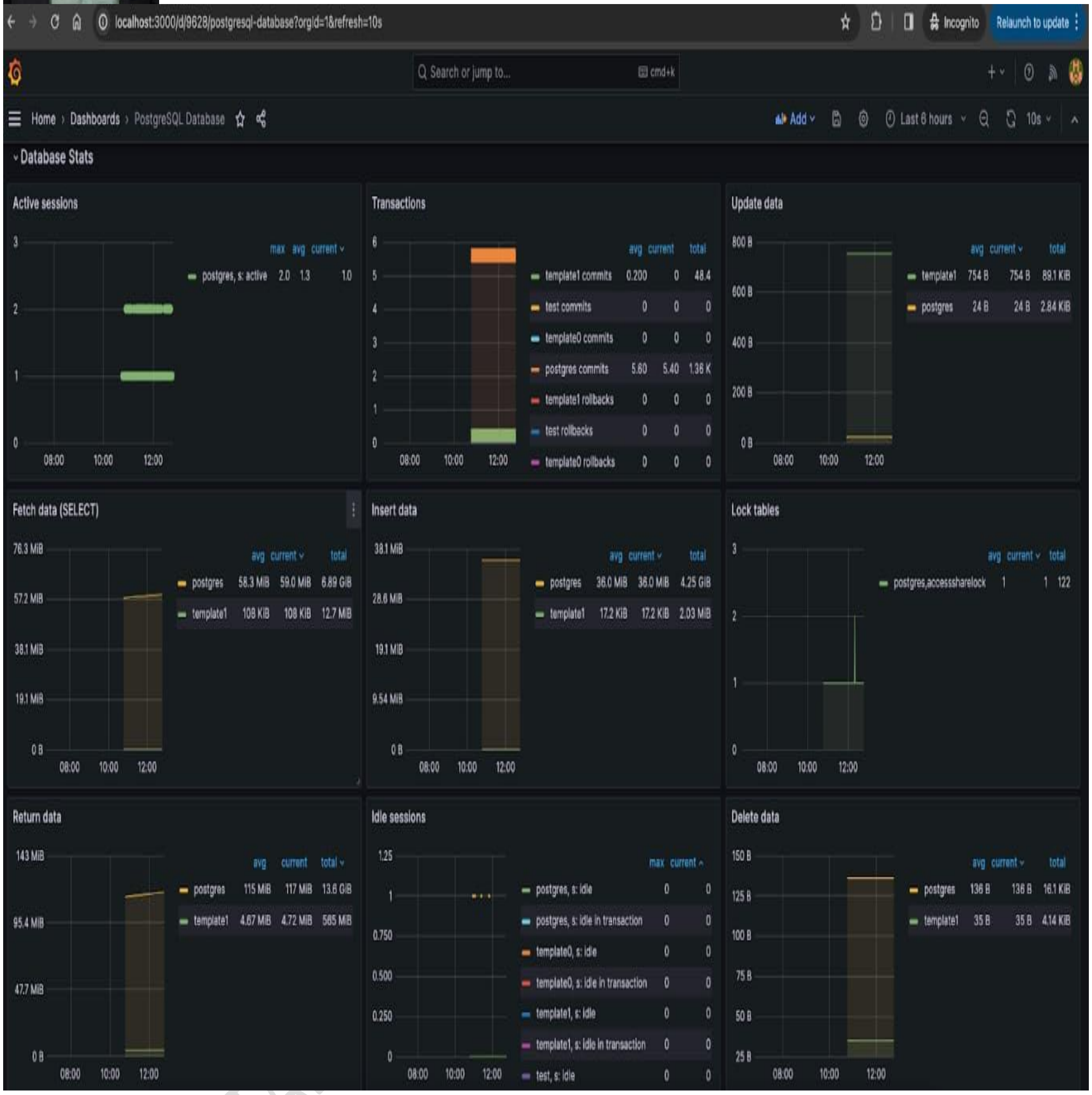
Settings

Shared Buffers	Effective Cache	Maintenance Work Mem	Work Mem	Max WAL Size	Random Page Cost	Seq Page Cost	Max Worker Prc	Max Parallel Wc
128 MiB	4 GiB	64 MiB	4 MiB	1.0 GiB	4	1	8	8

You can see the following properties straight away: average CPU usage, average memory usage, and a couple of important postgresql.conf properties:

- Shared buffers
- Effective cache
- Maintenance work memory
- Work memory, etc.

If you scroll down, you can see a more detailed version of database stats, including active/idle sessions, cache hit ratio, buffers, etc.


localhost:3000/d/9628/postgresql-database?orgId=1&refresh=10s
Incognito
Relaunch to update
Search or jump to...
cmd+k
Home › Dashboards › PostgreSQL Database
Add
Last 6 hours
10s
Database Stats

Active sessions

	max	avg	current
postgres, s: active	2.0	1.3	1.0

Transactions

	avg	current	total
template1 commits	0.200	0	48.4
test commits	0	0	0
template0 commits	0	0	0
postgres commits	5.60	5.40	1.36 K
template1 rollbacks	0	0	0
test rollbacks	0	0	0
template0 rollbacks	0	0	0

Update data

	avg	current	total
template1	754 B	754 B	89.1 KiB
postgres	24 B	24 B	2.84 KiB

Fetch data (SELECT)

	avg	current	total
postgres	58.3 MiB	59.0 MiB	6.89 GiB
template1	108 KiB	108 KiB	12.7 MiB

Insert data

	avg	current	total
postgres	36.0 MiB	36.0 MiB	4.25 GiB
template1	17.2 KiB	17.2 KiB	2.03 MiB

Lock tables

	avg	current	total
postgres,accesssharelock	1	1	122

Return data

	avg	current	total
postgres	115 MiB	117 MiB	13.6 GiB
template1	4.67 MiB	4.72 MiB	585 MiB

Idle sessions

	max	current
postgres, s: idle	0	0
postgres, s: idle in transaction	0	0
template0, s: idle	0	0
template0, s: idle in transaction	0	0
template1, s: idle	0	0
template1, s: idle in transaction	0	0
test, s: idle	0	0

Delete data

	avg	current	total
postgres	136 B	136 B	16.1 KiB
template1	35 B	35 B	4.14 KiB

PostgreSQL Database Monitoring With Metrics Dashboard and Insights

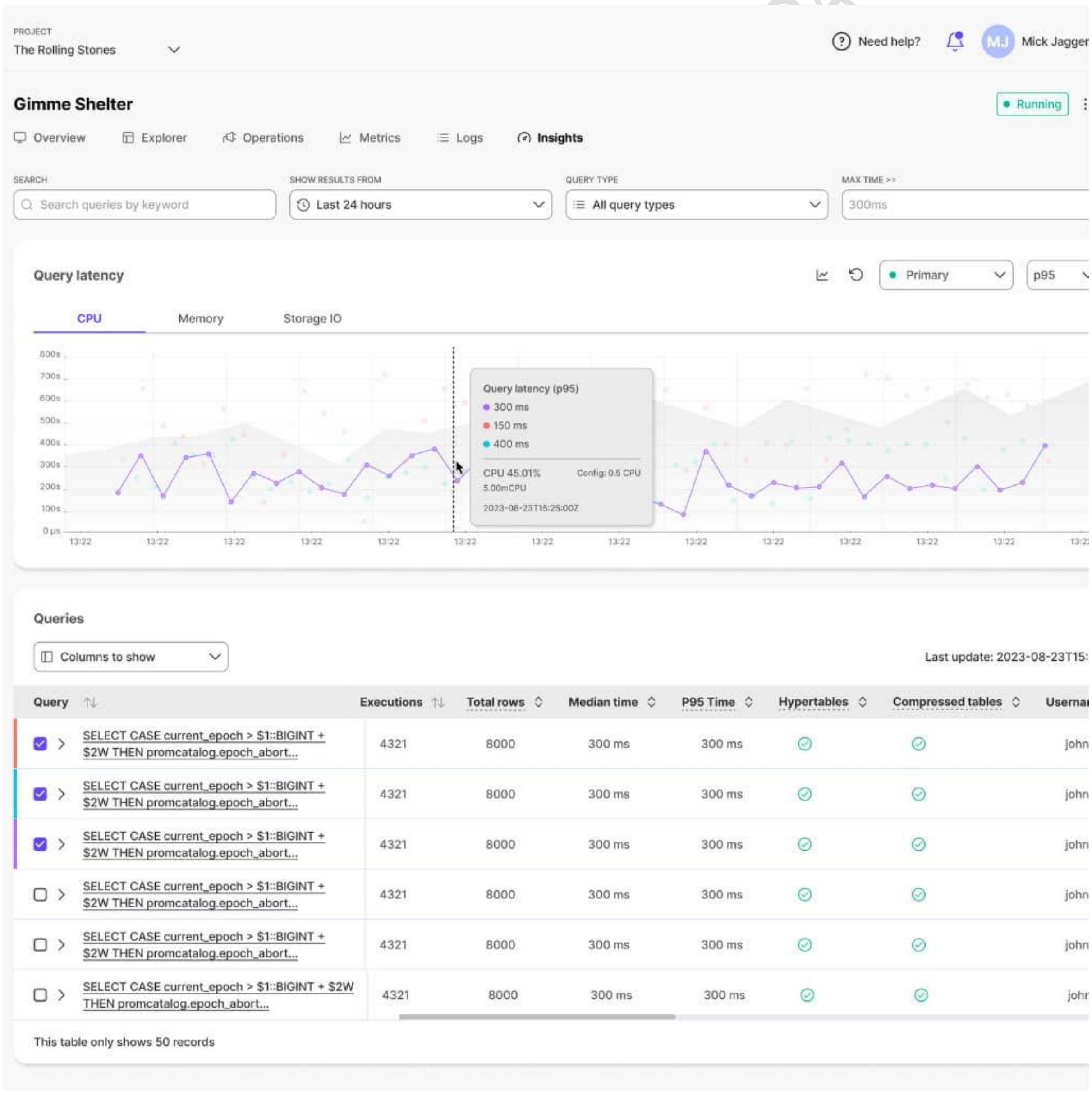
If you want to save yourself all of this work when it comes to database monitoring, Timescale (built on PostgreSQL) has a metrics dashboard that significantly improves the monitoring capabilities of your database services by providing detailed, service-level insights such as CPU, memory, and query-level statistics.

It offers support for various time ranges to view metrics, including the last hour, 24 hours, seven days, and 30 days, each with specific granularity. Furthermore, you can continuously monitor the health and resource consumption of your database services, enabling proactive management and optimization of database performance.

This feature facilitates the identification of trends, diagnosis of issues, and optimization of configurations to maintain optimal service health. It also includes pg_stat_statements by default.

However, we believe that pg_stat_statements has some limitations. So, to overcome them and provide a more in-depth query monitoring experience, you can utilize Insights, a tool we developed last year. Insights empowers you to closely examine the queries executed in your database within a specific timeframe, providing valuable statistics on timing/latency, memory usage, and more.

Upon identifying specific queries for further investigation, you can access additional information through a drill-down view to better understand the situation, such as whether your database performance has been deteriorating or improving (e.g., due to adding a new index).



Query	Executions	Total rows	Median time	P95 Time	Hypertables	Compressed tables	Userna
☑ SELECT CASE current_epoch > $1::BIGINT + $2W THEN promcatalog.epoch_abort...	4321	8000	300 ms	300 ms	✓	✓	john
☑ SELECT CASE current_epoch > $1::BIGINT + $2W THEN promcatalog.epoch_abort...	4321	8000	300 ms	300 ms	✓	✓	john
☑ SELECT CASE current_epoch > $1::BIGINT + $2W THEN promcatalog.epoch_abort...	4321	8000	300 ms	300 ms	✓	✓	john
☐ SELECT CASE current_epoch > $1::BIGINT + $2W THEN promcatalog.epoch_abort...	4321	8000	300 ms	300 ms	✓	✓	john
☐ SELECT CASE current_epoch > $1::BIGINT + $2W THEN promcatalog.epoch_abort...	4321	8000	300 ms	300 ms	✓	✓	john
☐ SELECT CASE current_epoch > $1::BIGINT + $2W THEN promcatalog.epoch_abort...	4321	8000	300 ms	300 ms	✓	✓	johr

This table only shows 50 records

Insights offers an unmatched level of granularity in database observability, complementing and enhancing the monitoring and optimization capabilities provided by **pg_stat_statements** for PostgreSQL. This combination enables a comprehensive approach to query optimization and performance monitoring within the Postgres environment.

Add Postgres Alerts

Defining your alerts

The Prometheus resources allow you to add Prometheus instances and Prometheus rules. The prometheus rules are the resources to define your alerts using promql with some other required parameters. This is and example of a postgres rule:

```
- alert: PostgresInstanceDown
    expr: pg_up == 0
    for: 1m
    labels:
      severity: critical
      service: "PostgreSQL"
    annotations:
      summary: "Postgres server instance is down"
      description: "Postgres has not been responding for the past 1 minutes on {{
$labels.instance }}"
      title: "Postgres server instance {{ $labels.instance }} is down "
```

The expr parameter is the prometheus query to retreive the data for the alert in this example pg_up metric return 0 is the instance is down and 1 if up. The for parameter is the threshold time to evaluate the query. Then you can also add labels and annotations to describe more precisely your alert. So the above example means If the pg_up metric value is equal to zero for 1m then fire the alert. Check the alerts section below for the full alerts description and the yaml file definition.

These are some useful alerts you can add to your StackGres cluster.

Alert	**Severity**	**Threshod**	**Description**
PostgresExporterErrors	critical	5m	Check the last scrape from postgres-exporter
PostgresInstanceDown	critical	1m	Check is postgres service is up
PostgresSplitBrain	critical	1m	Check if is more than one Read/Write instance on the cluster
PostgresPromotedNode	critical	5m	Check if the replication leader was changed
PostgresInactiveReplicationSlots	warning	30m	Check if there is a inactive replication slot
PostgresReplicationLagSizeTooLarge	warning	5m	Check if the replication lag is increasing
PostgresTooManyDeadTuples	warning	30m	Check if there are to many dead tuples
PostgresTooManyConnections	warning	5m	Check if postgres connections are above 90% of max_connections
PostgresNotEnoughConnections	warning	5m	Check if postgres available connections are less than 5
PgBouncerWaitingClients	crititcal	1m	Check if pgbouncer has waiting clients
PgBouncerNotEnoughConnections	critical	5m	Check if pool size is not enough for the current connections
PgBouncerPoolFillingUp	warning	5m	Check if pgBouncer pool is filling up
PgBouncerAvgWaitTimeTooHigh	warning	5m	Check if time spent by clients waiting for a connections > 1s
PgBouncerQueryTimeTooHigh	warning	5m	Check if average query duration more than 5 seconds
DatabaseLowDiskAvailable	warning	15m	Check the database available disk size <= 20%

With the default Prometheus stack installation use the next yaml description to create a file stackgres-alerts.yaml:

```
apiVersion: monitoring.coreos.com/v1

kind: PrometheusRule

metadata:
```

```
 labels:
   app: kube-prometheus-stack
   release: prometheus-operator
 name: stackgres-rules
 namespace: monitoring
spec:
 groups:
 - name: StackGres
   rules:
   - alert: PostgresInstanceDown
     expr: pg_up == 0
     for: 1m
     labels:
       severity: critical
       service: "PostgreSQL"
       cluster: "StackGres"
     annotations:
       summary: "Postgres server instance is down"
       description: "Postgres has not been responding for the past 1 minutes on {{
$labels.instance }}"
       title: "Postgres server instance {{ $labels.instance }} is down "
   - alert: PostgresExporterErrors
     expr: pg_exporter_last_scrape_error == 1
     for: 10m
     labels:
       severity: critical
       service: "PostgreSQL"
       cluster: "StackGres"
```

```
    annotations:
      summary: "Postgres Exporter is down or is showing errors"
      description: "postgres-exporter is not running or it is showing errors {{
$labels.instance }}"
  - alert: PostgresReplicationLagSizeTooLarge
    expr: pg_replication_status_lag_size > 1e+09
    for: 5m
    labels:
      severity: critical
      service: "PostgreSQL"
      cluster: "StackGres"
    annotations:
      summary: "Postgres replication lag size is to large"
      description: "Replication lag size on server {{$labels.instance}}
({{$labels.application_name}}) is currently {{ $value | humanize1024}}B behind the
leader in cluster {{$labels.cluster_name}}"
  - alert: PostgresTooManyDeadTuples
    expr: ((pg_stat_user_tables_n_dead_tup > 1e+06) / (pg_stat_user_tables_n_live_tup +
pg_stat_user_tables_n_dead_tup)) >= 0.05
    for: 30m
    labels:
      severity: warning
      service: "PostgreSQL"
      cluster: "StackGres"
    annotations:
      summary: "PostgreSQL dead tuples is too large"
      description: "The dead tuple ratio of {{$labels.relname}} on database
{{$labels.datname}} is greater than 5% in cluster {{$labels.cluster_name}}"
  - alert: PostgresInactiveReplicationSlots
```

```
    expr: pg_replication_slots_active == 0
    for: 30m
    labels:
      severity: warning
      service: "PostgreSQL"
      cluster: "StackGres"
    annotations:
      summary: "There are inactive replications slots"
      description: "The are some inactive replication slots on {{$labels.instance}} in
cluster {{$labels.cluster_name}}"
  - alert: PostgresSplitBrain
    expr: count by(cluster_name) (pg_replication_is_replica == 0) > 1
    for: 1m
    labels:
      severity: critical
      service: "PostgreSQL"
      cluster: "StackGres"
    annotations:
      summary: "There are more than one instance in read-write mode"
      description: "Split Brain: too many postgres databases in cluster
{{$labels.cluster_name}} in read-write mode"
  - alert: PostgresTooManyConnections
    expr: sum by (datname) (pg_stat_activity_count{datname!~"template.*|postgres"}) >
pg_settings_max_connections * 0.9
    for: 5m
    labels:
      severity: warning
      service: "PostgreSQL"
```

```
        cluster: "StackGres"
      annotations:
        summary: Postgresql too many connections (instance {{ $labels.instance }} in cluster {{$labels.cluster_name}})
        description: "PostgreSQL instance has too many connections\n  VALUE = {{ $value }}\n  LABELS: {{ $labels }}"
    - alert: PostgresNotEnoughConnections
      expr: sum by (datname) (pg_stat_activity_count{datname!~"template.*|postgres"}) < 5
      for: 5m
      labels:
        severity: warning
        service: "PostgreSQL"
        cluster: "StackGres"
      annotations:
        summary: Postgresql not enough connections (instance {{ $labels.instance }} in cluster {{$labels.cluster_name}})
        description: "PostgreSQL instance should have more connections (> 5)\n  VALUE = {{ $value }}\n  LABELS: {{ $labels }}"
    - alert: PostgresPromotedNode
      expr: pg_replication_is_replica and changes(pg_replication_is_replica[1m]) > 0
      for: 5m
      labels:
        severity: warning
        service: "PostgreSQL"
        cluster: "StackGres"
      annotations:
        summary: "Postgresql promoted node (instance {{ $labels.instance }}, cluster {{ $labels.cluster_name }})"
        description: "Postgresql standby server has been promoted as primary node\n
  VALUE = {{ $value }}\n  LABELS: {{ $labels }}"

    # Connection Pooling alerts

    - alert: PgBouncerWaitingClients

      expr: pgbouncer_show_pools_cl_waiting > 0

      for: 5m

      labels:

        severity: warning

        service: "PgBouncer"

        cluster: "StackGres"

      annotations:

        summary: PgBouncer has waiting clients on instance {{ $labels.instance }} in cluster
  {{$labels.cluster_name}})

        description: "PgBouncer instance has waiting clients\n  VALUE = {{ $value }}\n
  LABELS: {{ $labels }}"

    - alert: PgBouncerNotEnoughConnections

      expr: (sum by (database,instance)
  (pgbouncer_show_pools_cl_active{database!~"template.*|postgres|pgbouncer"}) + sum
  by (database, instance)
  (pgbouncer_show_pools_cl_waiting{database!~"template.*|postgres|pgbouncer"}))  - on
  (database,instance)
  (pgbouncer_show_databases_pool_size{database!~"template.*|postgres|pgbouncer"})
  > 0

      for: 10m

      labels:

        severity: critical

        service: "PgBouncer"

        cluster: "StackGres"

      annotations:
      summary: PgBouncer pool size is not enough for the current connections on {{ $labels.instance }} in cluster {{$labels.cluster_name}})

      description: "PgBouncer is getting more connections than the pool size, extra connections = {{ $value }}"

   - alert: PgBouncerPoolFillingUp

     expr: (sum by (database,instance) (pgbouncer_show_databases_pool_size{database!~"template.*|postgres|pgbouncer"}) - on (database,instance) pgbouncer_show_databases_current_connections)  <= 15

     for: 5m

     labels:

      severity: warning

      service: "PgBouncer"

      cluster: "StackGres"

     annotations:

      summary: PgBouncer pool is filling up on {{ $labels.instance }} in cluster {{$labels.cluster_name}})

      description: "PgBouncer pool is filling up, remaining connections = {{ $value }}"

   - alert: PgBouncerAvgWaitTimeTooHigh

     expr: pgbouncer_show_stats_avg_wait_time > 1e+6

     for: 5m

     labels:

      severity: warning

      service: "PgBouncer"

      cluster: "StackGres"

     annotations:

      summary: PgBouncer time spent by clients waiting for a connections is too high on {{ $labels.instance }} in cluster {{$labels.cluster_name}})

      description: "PgBouncer wait for a server connections is too high = {{ $value }}"

   - alert: PgBouncerQueryTimeTooHigh
```

```
   expr: pgbouncer_show_stats_avg_query_time > 5e+6
   for: 5m
   labels:
    severity: warning
    service: "PgBouncer"
    cluster: "StackGres"
   annotations:
    summary: PgBouncer average query duration more than 5 seconds on {{ $labels.instance }} in cluster {{$labels.cluster_name}})
    description: "PgBouncer average query duration more than 5 seconds = {{ $value }}"
  - alert: DatabaseLowDiskAvailable
   expr: (1.0 - node_filesystem_avail_bytes{mountpoint="/var/lib/postgresql",fstype!=""} / node_filesystem_size_bytes{mountpoint="/var/lib/postgresql",fstype!=""}) * 100 >= 80
   for: 15m
   labels:
    severity: warning
    cluster: "StackGres"
   annotations:
    summary: Database disk is filling up currently have less than 20% available on {{ $labels.instance }} in cluster {{$labels.cluster_name}})
    description: "Database disk is filling up currently have less than 20%, currently occupied {{ $value }} %"
```

and deploy it to kubernetes:

```
kubectl apply -f stackgres-alerts.yaml
```

How Prometheus discover the alerts

With the default prometheus installation some deployment will be added:

```
$ kubectl get deployments.apps -n monitoring
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
prometheus-operator-grafana	1/1	1	1	25d
prometheus-operator-kube-p-operator	1/1	1	1	25d
prometheus-operator-kube-state-metrics	1/1	1	1	25d

This will create a Prometheus instance:

```
$ kubectl get pods -n monitoring -l app=prometheus

NAME                                              READY   STATUS    RESTARTS   AGE

prometheus-prometheus-operator-kube-p-prometheus-0  2/2    Running  0       25d
```

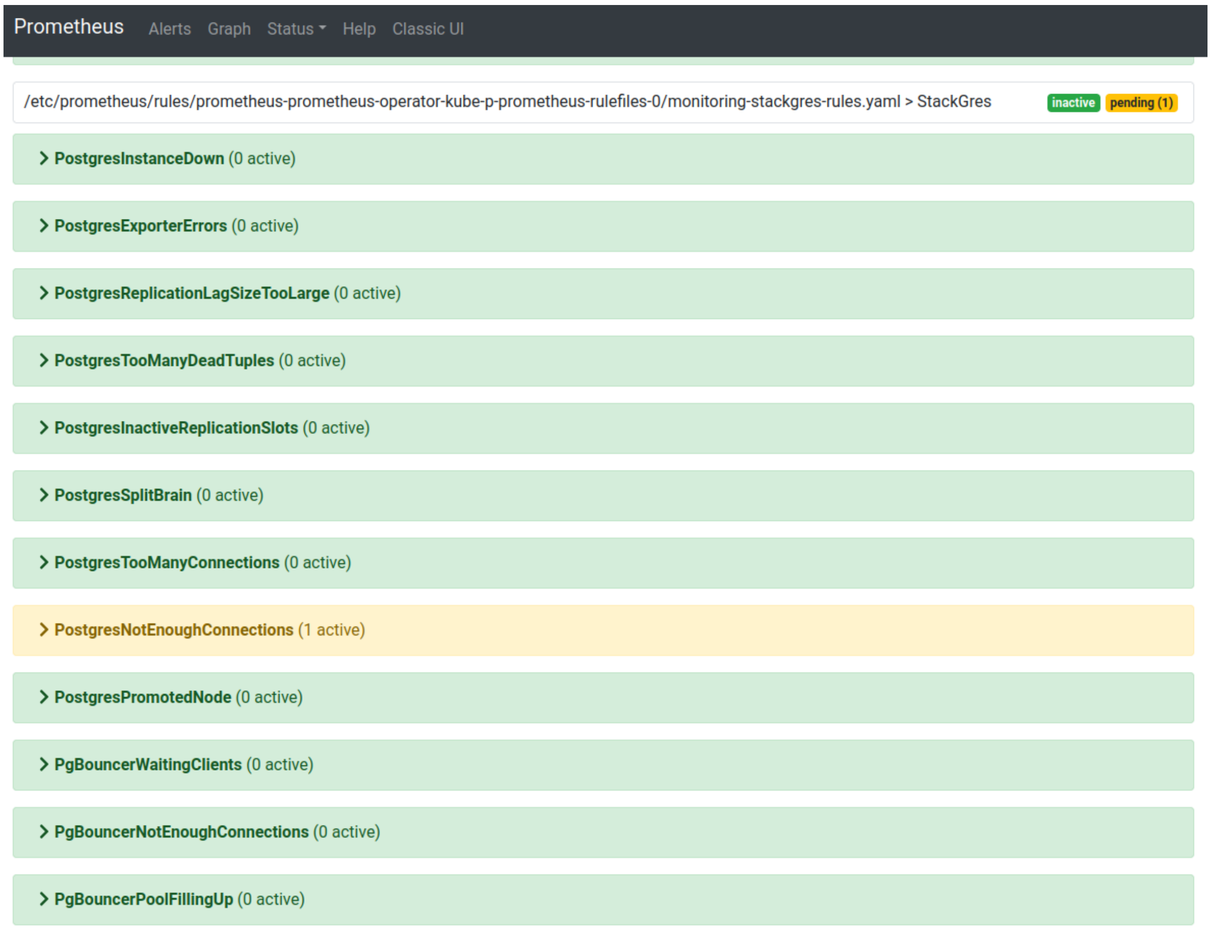
This instance has defined a ruleSelector section with some labels:

```
$ kubectl get -n monitoring prometheus prometheus-operator-kube-p-prometheus -o yaml | grep -A3 ruleSelector

 ruleSelector:

  matchLabels:

   app: kube-prometheus-stack

   release: prometheus-operator
```

You need to make sure the metadata labels from your Prometheus Rule match with the ruleSelector labels of your prometheus instance. In this case will be as is shown below:

```
apiVersion: monitoring.coreos.com/v1

kind: PrometheusRule

metadata:

 labels:

  app: kube-prometheus-stack

  release: prometheus-operator

 name: stackgres-rules

 namespace: monitoring

 ...

 ...
```

Prometheus will autodiscover the alerts according to the matching lables and add them to the instance:



/etc/prometheus/rules/prometheus-prometheus-operator-kube-p-prometheus-rulefiles-0/monitoring-stackgres-rules.yaml > StackGres

- PostgresInstanceDown (0 active)
- PostgresExporterErrors (0 active)
- PostgresReplicationLagSizeTooLarge (0 active)
- PostgresTooManyDeadTuples (0 active)
- PostgresInactiveReplicationSlots (0 active)
- PostgresSplitBrain (0 active)
- PostgresTooManyConnections (0 active)
- PostgresNotEnoughConnections (1 active)
- PostgresPromotedNode (0 active)
- PgBouncerWaitingClients (0 active)
- PgBouncerNotEnoughConnections (0 active)
- PgBouncerPoolFillingUp (0 active)

Email alerts with AlertManager

Once you have configured the alerts on Prometheus, depending on your infrastructure you can create push messages in an alert is fire. You could use, email, Slack, Pagerduty, Opsgenie and others. Check Alert Manager Configuration for more details. In this example you'll see how to configure alerts with email messages.

The Prometheus stack installation include an installation of AlertManager:

```
$ kubectl get pods -n monitoring -l app=alertmanager

NAME                                              READY   STATUS    RESTARTS   AGE

alertmanager-prometheus-operator-kube-p-alertmanager-0   2/2     Running   0          68m
```

The alerting configuration is added using a secret::

```
$ kubectl get secrets -n monitoring alertmanager-prometheus-operator-kube-p-alertmanager

NAME                                    TYPE    DATA   AGE
```

```
alertmanager-prometheus-operator-kube-p-alertmanager   Opaque   1     30d
```

To update the configuration you need to first create a yml file alertmanager.yaml with the next content(Replace the users and credentials with your settings):

```
global:
  resolve_timeout: 5m
route:
  receiver: 'email-alert'
  group_by: ['datname']

  routes:
  - receiver: 'email-alert'
    group_wait: 50s
    group_interval: 5m
    repeat_interval: 12h

receivers:
- name: email-alert
  email_configs:
  - to: email-group@ongres.com
    from: stackgres-alerts@ongres.com
    # Your smtp server address
    smarthost: mail.ongres.com:587
    auth_username: myuser@ongres.com
    auth_identity: myuser@ongres.com
    auth_password: XXXXXXX
```

Next you need to encrypt the content of the file running:

```
$ cat alertmanager.yaml | base64 -w0
```

This will return a string like:

Z2xvYmFsOgogIHJlc29sdmVfdGltZW91dDogNW0Kcm91dGU6CiAgcmVjZWl2ZXI6ICdlbWFpbC1hbGVydCcK

Edit the alertmanager secret with the string generated with the command above:

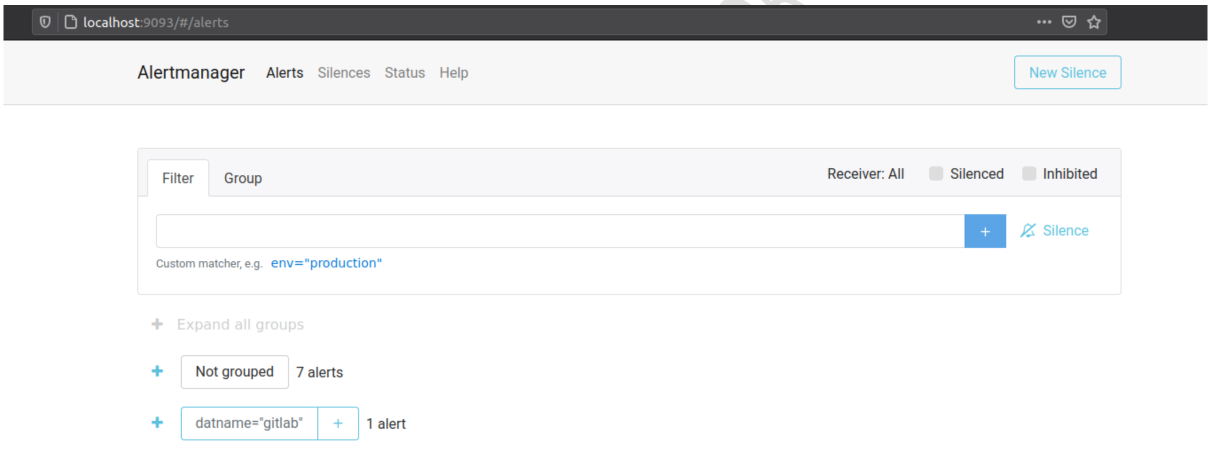
$ kubectl patch secret -n monitoring alertmanager-prometheus-operator-kube-p-alertmanager -p='{"data":{"alertmanager.yaml": "Z2xvYmFsOgogIHJlc29sdmVfdGltZW91dDogNW0Kcm91dGU6CiAgcmVjZWl2ZXI6ICdlbWFpbC1hbGVydCcK"}}' -v=1

After a few seconds, the new configuration will be applied, you can verify it by accessing the AlertManager console, with:

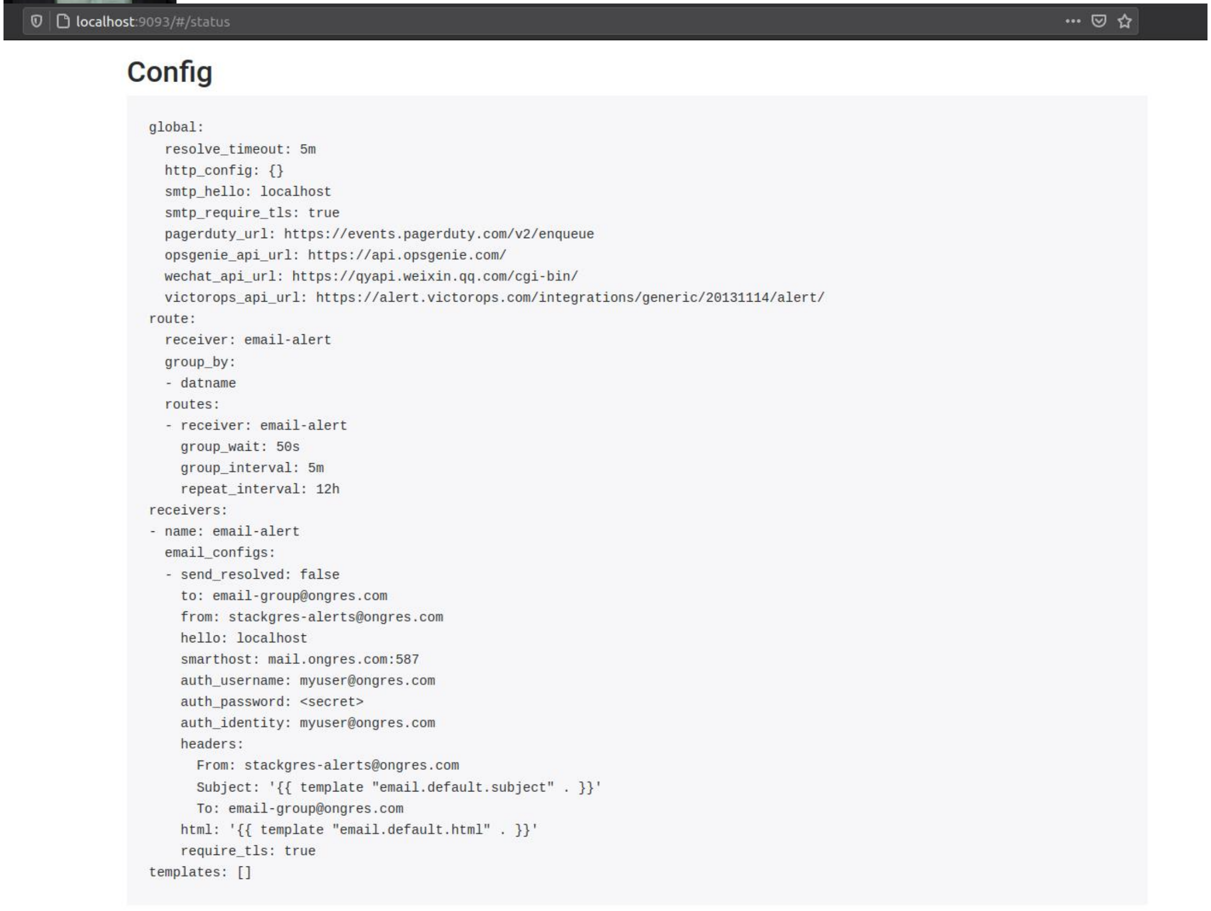
kubectl port-forward -n monitoring alertmanager-prometheus-operator-kube-p-alertmanager-0 9093:9093

Then open your web browser and type:

http://localhost:9093



Now go to the Status section and verify your changes were applied:

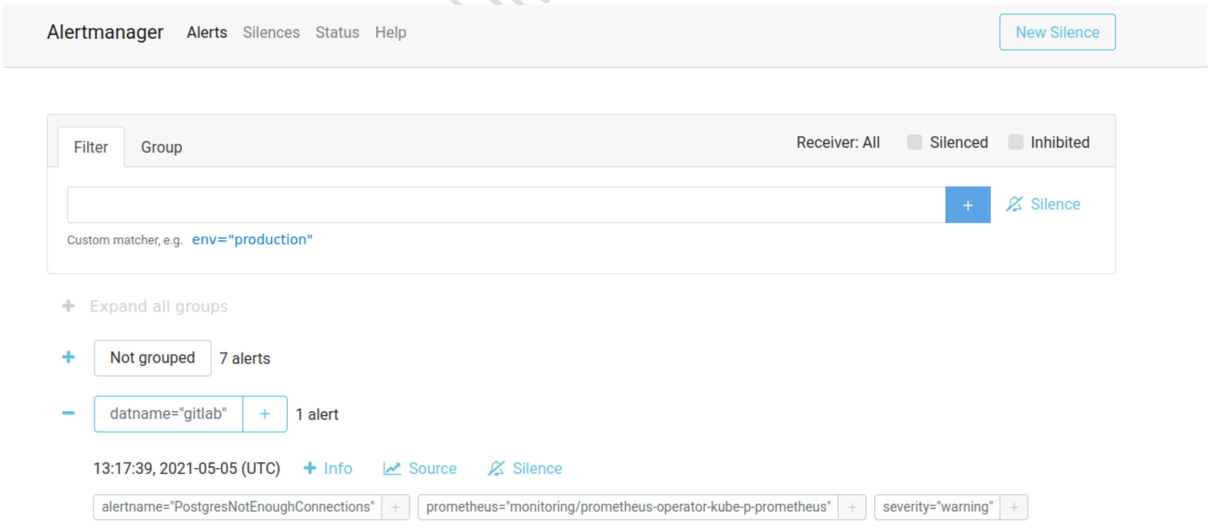


localhost:9093/#/status

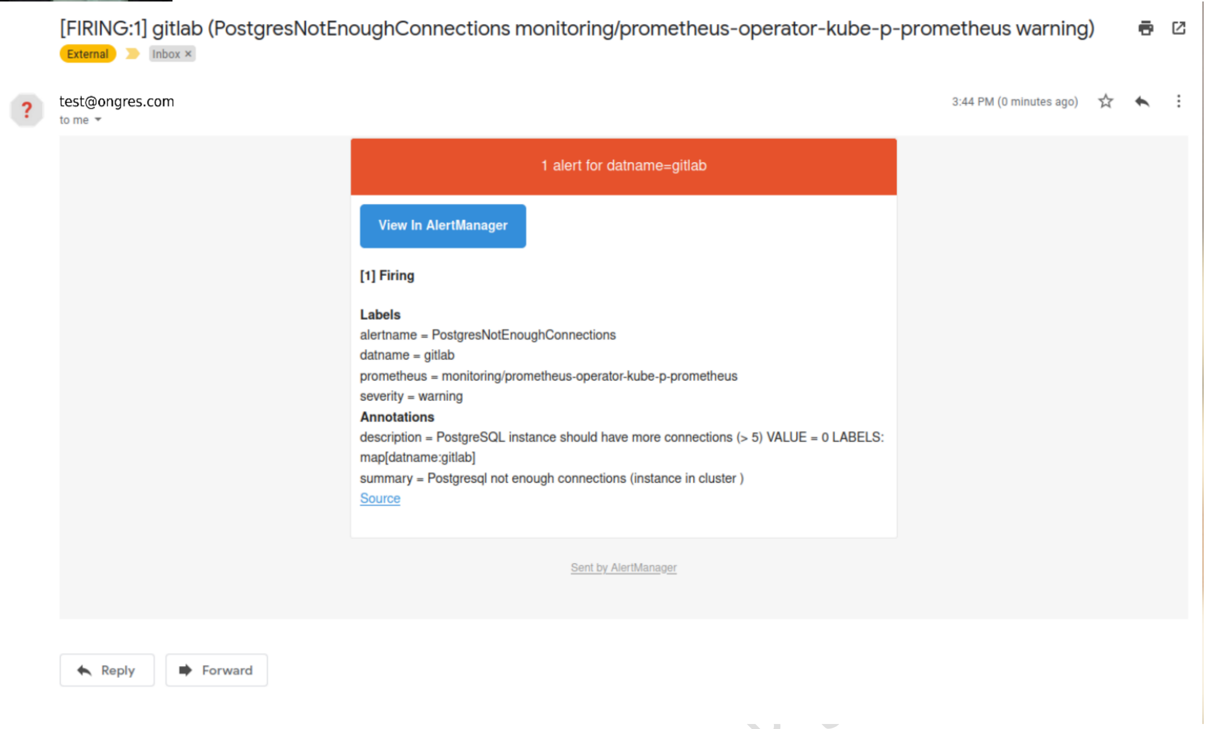
Config

```
global:
  resolve_timeout: 5m
  http_config: {}
  smtp_hello: localhost
  smtp_require_tls: true
  pagerduty_url: https://events.pagerduty.com/v2/enqueue
  opsgenie_api_url: https://api.opsgenie.com/
  wechat_api_url: https://qyapi.weixin.qq.com/cgi-bin/
  victorops_api_url: https://alert.victorops.com/integrations/generic/20131114/alert/
route:
  receiver: email-alert
  group_by:
  - datname
  routes:
  - receiver: email-alert
    group_wait: 50s
    group_interval: 5m
    repeat_interval: 12h
receivers:
- name: email-alert
  email_configs:
  - send_resolved: false
    to: email-group@ongres.com
    from: stackgres-alerts@ongres.com
    hello: localhost
    smarthost: mail.ongres.com:587
    auth_username: myuser@ongres.com
    auth_password: <secret>
    auth_identity: myuser@ongres.com
    headers:
      From: stackgres-alerts@ongres.com
      Subject: '{{ template "email.default.subject" . }}'
      To: email-group@ongres.com
    html: '{{ template "email.default.html" . }}'
    require_tls: true
templates: []
```

If there's an active alert on your cluster you should see it in the alert manager console:



And if the email configuration and credentials you provide are OK, you'll get the alert message:



[FIRING:1] gitlab (PostgresNotEnoughConnections monitoring/prometheus-operator-kube-p-prometheus warning)

test@ongres.com
to me

3:44 PM (0 minutes ago)

1 alert for datname=gitlab

[1] Firing

Labels
alertname = PostgresNotEnoughConnections
datname = gitlab
prometheus = monitoring/prometheus-operator-kube-p-prometheus
severity = warning
Annotations
description = PostgreSQL instance should have more connections (> 5) VALUE = 0 LABELS: map[datname:gitlab]
summary = Postgresql not enough connections (instance in cluster)
Source

Sent by AlertManager