PostgreSQL Monitoring: A Complete Guide to Database Observability

Effective monitoring is crucial for maintaining healthy PostgreSQL databases. This comprehensive guide covers everything you need to know about monitoring PostgreSQL, from basic metrics to advanced monitoring strategies.

Table of Contents

- Essential PostgreSQL Metrics
- Monitoring Tools
- Query Performance Monitoring
- System Resource Monitoring
- Alerting Strategies
- Best Practices

Essential PostgreSQL Metrics {#essential-metrics}

1. Connection Metrics

```sql
-- Monitor active connections
SELECT count(*), state FROM pg_stat_activity GROUP BY state;

-- Check connection utilization

SELECT current_setting('max_connections')::int as max_conn, count(*) as current_conn,
(count(*)::float/current_setting('max_connections')::int * 100)::int as conn_percent FROM
pg_stat_activity;
```

2. Database Size and Growth

```sql
-- Monitor database sizes


SELECT     datname,    pg_size_pretty(pg_database_size(datname)) as size,
pg_size_pretty(pg_database_size(datname) -        coalesce(previous_size, 0)) as growthFROM
pg_database;
```

```
-- Table growth monitoring

SELECT     schemaname,    relname,    n_live_tup as row_count,
pg_size_pretty(pg_total_relation_size(schemaname || '.' || relname)) as total_sizeFROM
pg_stat_user_tablesORDER BY pg_total_relation_size(schemaname || '.' || relname) DESC;
```

3. Transaction Metrics

```
-- Transaction statistics
SELECT     datname,    xact_commit,    xact_rollback,    (xact_rollback::float / (xact_commit +
xact_rollback) * 100)::numeric(10,2)     as rollback_ratioFROM pg_stat_database;
```

Monitoring Tools {#monitoring-tools}

1. Prometheus with PostgreSQL Exporter

Example `prometheus.yml` configuration:

```
scrape_configs:
  - job_name: 'postgresql'
    static_configs:
      - targets: ['localhost:9187']
    metrics_path: '/metrics'
```

Example queries:

```
# Query rate of transactions
rate(pg_stat_database_xact_commit{datname="mydb"}[5m])

# Database size
pg_database_size_bytes{datname="mydb"}

# Connection utilization
pg_stat_activity_count / pg_settings_max_connections * 100# Query rate of
transactionsrate(pg_stat_database_xact_commit{datname="mydb"}[5m])# Database sizepg_database_size_bytes{datname="mydb"}#
Connection utilizationpg_stat_activity_count / pg_settings_max_connections * 100
```

PROMQL

2. Grafana Dashboard Setup

Example dashboard JSON:

```
{
 "panels": [
  {
   "title": "Active Connections",
   "targets": [
    {
     "expr": "pg_stat_activity_count{datname='mydb',state='active'}"
    }
   ]
  },
  {
   "title": "Transaction Rate",
   "targets": [
    {
     "expr": "rate(pg_stat_database_xact_commit{datname='mydb'}[5m])"
    }
   ]
  }
 ]
}
```

3. pgMonitor Configuration

Example pgMonitor setup:

```
# Install pgMonitor
git clone https://github.com/CrunchyData/pgmonitor.git
cd pgmonitor/exporter
./install-pg-exporter.sh

# Configure metrics collection
cat << EOF > /etc/pgmonitor/exporter/postgres/queries.yml
pg_replication:
 query: "SELECT * FROM pg_stat_replication"
 metrics:
  - write_lag_bytes
  - replay_lag_bytes



EOF
```

Query Performance Monitoring {#query-performance}

1. Slow Query Analysis

```
-- Create extension if not exists
CREATE EXTENSION pg_stat_statements;

-- Find slow queries
SELECT
    substring(query, 1, 50) as short_query,
    round(total_time::numeric, 2) as total_time,
    calls,
    round(mean_time::numeric, 2) as mean_time,
    round((100 * total_time / sum(total_time::numeric) over ())::numeric, 2) as percentage
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;
```

2. Index Usage Monitoring

```
-- Monitor index usageSELECT    schemaname,    tablename,    indexname,    idx_scan as number_of_scans,    idx_tup_read as tuples_read,    idx_tup_fetch as tuples_fetchedFROM pg_stat_user_indexesORDER BY idx_scan DESC;
```

SQL

System Resource Monitoring {#system-resources}

1. CPU Usage Monitoring

```
- Check CPU-intensive queries


SELECT    pid,    datname,    usename,    query_start,    state,    queryFROM pg_stat_activityWHERE state = 'active'ORDER BY query_start;
```

SQL

-

2. Memory Usage

```
-- Monitor shared buffer usage
SELECT    buffers_clean,    buffers_backend,    buffers_backend_fsyncFROM pg_stat_bgwriter;-- Check memory context usageSELECT    name,    setting,    unit,    contextFROM pg_settingsWHERE name LIKE '%memory%' OR name LIKE '%buffer%';
```

SQL

Alerting Strategies {#alerting}

1. Connection Alerts

```
-- Create alert function for connection threshold
CREATE OR REPLACE FUNCTION check_connection_threshold()
RETURNS trigger AS $$
BEGIN
  IF (SELECT count(*) FROM pg_stat_activity) >
     (current_setting('max_connections')::int * 0.8) THEN
    PERFORM pg_notify(
      'connection_alert',
      'Connection threshold exceeded: ' || count(*) || ' connections'
    );
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

2. Replication Lag Alerts

```
-- Monitor replication lag
SELECT
  client_addr,
  state,
  sent_lsn,
  write_lsn,
  flush_lsn,
  replay_lsn,
  pg_wal_lsn_diff(sent_lsn, replay_lsn) as lag_bytes
FROM pg_stat_replication
```

Best Practices {#best-practices}

1. Regular Maintenance

```
-- Monitor table bloat
WITH constants AS (
  SELECT current_setting('block_size')::numeric AS bs
)
SELECT
  schemaname,
  tablename,
  pg_size_pretty(bloat_size) as bloat_size,
  round(bloat_ratio::numeric, 2) as bloat_ratio
FROM (
  SELECT
    schemaname,
    tablename,
    bs*n_live_tup as actual_size,
    bs*reltuples as expected_size,
    bs*(reltuples-n_live_tup) as bloat_size,
    100.0 * (reltuples-n_live_tup)/reltuples as bloat_ratio
  FROM pg_stat_user_tables t
  JOIN pg_class c ON t.relname = c.relname
  CROSS JOIN constants
  WHERE reltuples > 0
) s
WHERE bloat_ratio > 10
ORDER BY bloat_size DESC;
```

2. Vacuum Monitoring

```
-- Monitor autovacuum activity


SELECT schemaname, relname, last_vacuum, last_autovacuum, vacuum_count,
autovacuum_countFROM pg_stat_user_tables;
```

SQL

}

3. Logging Configuration

```
# postgresql.conf logging settings
log_min_duration_statement = 1000  # Log queries taking more than 1 second
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
log_temp_files = 0
log_autovacuum_min_duration = 0
```

INI

Monitoring Automation Script

Here's a sample Python script for automated monitoring:

```
import psycopg2
import time
from datetime import datetime


def monitor_database():
    conn = psycopg2.connect("dbname=postgres user=postgres")
    cur = conn.cursor()

    # Collect metrics
    metrics = {
        'timestamp': datetime.now(),
        'connections': {},
        'performance': {},
        'storage': {}
    }

    # Connection metrics
    cur.execute("""
        SELECT count(*), state
        FROM pg_stat_activity
        GROUP BY state
    """)
    metrics['connections']['by_state'] = dict(cur.fetchall())

    # Performance metrics
    cur.execute("""
        SELECT datname, xact_commit, xact_rollback
        FROM pg_stat_database
        WHERE datname = current_database()
    """)
    metrics['performance']['transactions'] = cur.fetchone()

    # Storage metrics
    cur.execute("""
        SELECT pg_size_pretty(pg_database_size(current_database()))
    """)
    metrics['storage']['database_size'] = cur.fetchone()[0]

    return metrics

def main():
    while True:
        try:
            metrics = monitor_database()
            # Process metrics (e.g., send to monitoring system)
            print(f"Metrics collected at {metrics['timestamp']}")
            time.sleep(60)  # Collect metrics every minute
        except Exception as e:
            print(f"Error collecting metrics: {e}")
            time.sleep(5)


if __name__ == "__main__":
    main()
```

Conclusion

Effective PostgreSQL monitoring is essential for maintaining database health and performance. By implementing the monitoring strategies and tools discussed in this guide, you can:

- Proactively identify and resolve issues
- Optimize database performance
- Plan capacity effectively
- Ensure high availability
- Maintain data integrity

Remember to regularly review and adjust your monitoring setup as your database requirements evolve. The key is to find the right balance between comprehensive monitoring and system overhead.