Why Grafana's Database Matters for Alerting Despite Prometheus Handling Time Series Data

You're right that **Prometheus handles the heavy lifting for time series data** through its TSDB, but there's a crucial distinction between **data querying** and **alerting operations** that explains why upgrading from SQLite to Postgres resolves your database lock issues.

The Two-Database Architecture

Prometheus TSDB: Time Series Data

- Stores and serves metric data via PromQL queries
- Handles the computational heavy lifting for data aggregation and analysis
- Grafana queries this via HTTP API calls

Grafana Internal Database: Metadata and State Management

- Stores dashboards, users, datasource configurations
- **Critically**: Manages alert rules, state, and execution history

Why SQLite Locks During Grafana Alerting

When you enable **Grafana Alerting**, your Grafana database becomes much more active because it needs to handle:

Alert Rule Management

- Storing alert rule definitions and configurations
- Managing alert rule state transitions (OK → Pending → Alerting → Resolved)
- Recording alert evaluation timestamps and results

Alert State Tracking

- Maintaining alert instance lifecycle data
- Storing alert annotations and metadata
- Managing alert group states and dependencies

Notification Processing

- Queuing and tracking alert notifications
- Managing notification channel configurations
- Recording delivery status and retry attempts

Alert History and Logging

- Storing alert evaluation logs
- Maintaining alert state change history
- Recording alert rule execution metrics

SQLite's Concurrent Write Limitations

SQLite's locking behavior becomes problematic during alerting because:

- **Single Writer Lock**: SQLite allows only one write operation at a time
- **Alert Evaluation Frequency**: Multiple alerts evaluating simultaneously create concurrent write demands
- **State Updates**: Each alert state change requires immediate database writes
- **Notification Logging**: Each notification attempt generates database activity

PostgreSQL Solves the Concurrency Problem

Postgres handles concurrent operations much better:

- **Multi-Version Concurrency Control (MVCC)**: Allows concurrent reads and writes without locking
- **Row-Level Locking**: Only locks specific rows being modified, not entire tables
- **Better Transaction Handling**: More sophisticated transaction management for concurrent operations
- **Connection Pooling**: Better handling of multiple simultaneous database connections

The Performance Paradox

While **Prometheus does the computational work**, **Grafana alerting creates intense metadata activity**:

```
Alert Evaluation Cycle:
1. Grafana reads alert rules from internal DB
2. Grafana sends PromQL query to Prometheus ← Heavy computation here
3. Prometheus returns results
4. Grafana evaluates conditions and updates alert state in internal DB ← Concurrent writ
5. Grafana logs results and manages notifications in internal DB ← More concurrent write
```

The bottleneck isn't the PromQL query execution—it's the **concurrent metadata management** that overwhelms SQLite's single-writer architecture.

Practical Impact

With SQLite: Alert evaluations queue up waiting for database locks, causing delays, timeouts, and missed alerts.

With Postgres: Multiple alert evaluations can write state changes simultaneously, maintaining responsive alerting performance.

This is why upgrading your Grafana database to Postgres resolves the issue even though the time series heavy lifting remains with Prometheus. The databases serve different purposes, and alerting creates concurrent demands specifically on Grafana's internal database that exceed SQLite's capabilities.

⁂