**Section 2.**

**Question 7**

**Write a Prometheus exporter in Python that connects to a specified RabbitMQ HTTP API(it’s the management plugin) and periodically reads…**

**Answer;**

[**https://github.com/belisky/lemonade\_devops/blob/main/prometheus\_exporter.py**](https://github.com/belisky/lemonade_devops/blob/main/prometheus_exporter.py)

**Question 8:**

**Write a script to restart the Laravel backend service if CPU usage exceeds 80%;**

**Answers;  
Python script version:** [**https://github.com/belisky/lemonade\_devops/blob/main/laravel\_restart\_python.py**](https://github.com/belisky/lemonade_devops/blob/main/laravel_restart_python.py)

**Bash script version:**

[**https://github.com/belisky/lemonade\_devops/blob/main/laravel\_restart\_bash.sh**](https://github.com/belisky/lemonade_devops/blob/main/laravel_restart_bash.sh)

**Question 9:**

**A Postgres query is running slower than expected. Explain your approach to troubleshooting it.**

**Answer;**

### **1. Understandin the Context**

* **Query Details**:
  + What does the query do (e.g., SELECT, INSERT, UPDATE)?
  + Is it part of an application workflow or a one-off query?
  + Are there specific inputs causing slowness?
* **History**:
  + Has the query always been slow, or is this a recent change?
  + Have there been recent changes to the database schema, indexes, or data volume?
* **Environment**:
  + Are other queries running slower, or is it isolated to this query?
  + Is the issue specific to certain times, workloads, or environments (e.g., development, production)?

**2. Reproduce and Measure**

* Run the query in an isolated environment (e.g., psql) to rule out external factors like network latency.
* Use EXPLAIN and EXPLAIN (ANALYZE) to inspect the query execution plan:  
  EXPLAIN ANALYZE SELECT \* FROM my\_table WHERE condition;
* Look for high-cost operations such as sequential scans, nested loops, or large sort operations.

### **3. Identify Performance Bottlenecks**

#### **a. Query Structure**

* Check for:
  + Inefficient JOINs or WHERE conditions.
  + Use of functions on indexed columns (e.g., WHERE LOWER(column) = 'value' prevents index usage).
  + Missing or redundant columns in SELECT (e.g., SELECT \* can fetch unnecessary data).

#### **b. Indexes**

* Verify if appropriate indexes exist for the query:

| SELECT \* FROM pg\_indexes WHERE tablename = 'my\_table'; |
| --- |

If a sequential scan is being used, consider adding an index to columns used in WHERE, JOIN, GROUP BY, or ORDER BY clauses.

#### **c. Table Size and Statistics**

* Check table size and the number of rows:

| SELECT pg\_size\_pretty(pg\_relation\_size('my\_table')); SELECT COUNT(\*) FROM my\_table; |
| --- |

Ensure table statistics are up to date:

| ANALYZE my\_table; |
| --- |

#### **d. Locks and Contention**

* Check for locks or contention from other queries:

| SELECT \* FROM pg\_stat\_activity; |
| --- |

* Look for long-running transactions or blocked queries.

#### **e. Hardware/Resource Issues**

* Monitor system resources (CPU, memory, disk I/O) on the database server.
* Use PostgreSQL's statistics views:

| SELECT \* FROM pg\_stat\_database WHERE datname = 'my\_database'; |
| --- |

### **4. Optimize the Query**

#### **a. Indexing**

* Add indexes for frequently queried columns

#### **b. Query Rewriting**

* Rewrite complex queries to use subqueries, CTEs, or temp tables for better clarity and execution.
* Limit data retrieved by the query using LIMIT or OFFSET where possible.

#### **c. Partitioning**

* For very large tables, consider partitioning:

#### **d. Avoid Unnecessary Operations**

* Remove unnecessary DISTINCT, ORDER BY, or GROUP BY clauses.

### **5. Use PostgreSQL Tools**

* **Auto-Tuning**: Use tools like pg\_stat\_statements to analyze query performance over time.

### **6. Monitor and Test**

* After making changes, test the query with realistic data and workloads.
* Compare query execution times before and after optimization.

### **7. Scale if Necessary**

If the issue is resource-related, consider:

* Upgrading hardware (e.g., more memory, faster storage).
* Implementing read replicas for read-heavy workloads.
* Using connection pooling (e.g., PgBouncer) to handle high concurrency.