

# Performance Tuning Report

## 1. Introduction

The operational database designed in Phase 1 models a healthcare system containing patient admissions, medical conditions, hospitals, doctors, insurance providers, and related attributes. In Phase 2, the focus shifts from OLTP operations to **analytical workloads**, where complex queries require efficient execution.

This report documents:

- Performance analysis of an analytical query related to **patient readmissions**
- Query profiling using **EXPLAIN ANALYZE**
- An indexing strategy designed to improve performance
- A comparison of **before vs. after** execution times
- Interpretation of the improvement and future scalability considerations

## 2. Analytical Query Profiled

### Business Question

*For each patient, identify admissions that occur within 30 days of their previous admission.*

This metric is commonly used in healthcare analytics to measure hospital performance, quality of care, and patient outcomes.

### SQL Query Analyzed

```

WITH ordered_admissions AS (
  SELECT
    a.admission_id,
    a.patient_id,
    a.date_of_admission,
    LAG(a.date_of_admission) OVER (
      PARTITION BY a.patient_id
      ORDER BY a.date_of_admission
    ) AS previous_admission_date
  FROM admission a
)
SELECT
  oa.patient_id,
  p.name AS patient_name,
  oa.admission_id,
  oa.date_of_admission,
  oa.previous_admission_date,
  EXTRACT(DAY FROM (oa.date_of_admission - oa.previous_admission_date)) AS
  days_since_last_admission
FROM ordered_admissions oa
JOIN patient p
  ON oa.patient_id = p.patient_id
WHERE oa.previous_admission_date IS NOT NULL
  AND oa.date_of_admission - oa.previous_admission_date <= INTERVAL '30 days'
ORDER BY oa.patient_id, oa.date_of_admission;

```

## Why This Query Is Computationally Expensive

- Uses a **window function (LAG)** that requires ordering rows within each patient group
- Operates over a dataset of ~55,000 admissions
- Requires joining with the patient table
- Needs filtering on date intervals
- Requires a final sorted output

This combination of operations makes it a strong candidate for performance optimization.

## 3. Baseline Performance (Before Indexing)

The query was first executed using:

```

EXPLAIN ANALYZE
WITH ordered_admissions AS ( ... same query ... )
SELECT ...;
```

### Execution Time (Before Index)

→ 44.088 ms

## Observation

From the query plan (not included here for brevity), PostgreSQL was forced to:

- Sequentially scan the admission table
- Perform sorting within each `patient_id` partition for `LAG()`
- Re-sort the final result for the `ORDER BY` clause
- Join via `patient_id` without index support for ordered access

This revealed the need for an appropriate composite index.

## 4. Optimization Strategy

### Index Implemented

```
CREATE INDEX idx_admission_patient_date  
ON admission (patient_id, date_of_admission);
```

### Rationale

The window function requires:

```
PARTITION BY patient_id  
ORDER BY date_of_admission
```

The critical part of the query is:

```
LAG(a.date_of_admission) OVER (  
    PARTITION BY a.patient_id  
    ORDER BY a.date_of_admission  
)
```

Thus, PostgreSQL benefits from an index that:

- Organizes rows by **patient\_id**
- Orders them inside each partition by **date\_of\_admission**

This eliminates or reduces sorting during window function execution and speeds up both filtering and joins using these fields.

## 5. Performance After Indexing

After applying the index, the same query was re-executed with EXPLAIN ANALYZE.

### Execution Time (After Index)

→ 33.802 ms

### Summary of Measurements

- Before index: 44.088 ms
- After index: 33.802 ms
- Improvement: 23.3% faster
- All measurements were taken using PostgreSQL (Docker) via pgAdmin Query Tool

## 6. Interpretation of Results

The composite index (patient\_id, date\_of\_admission) improved performance because:

### 1. Faster Processing of Window Function

The index provided pre-ordered rows for each patient group.  
PostgreSQL avoided expensive sorting during the LAG() operation.

### 2. Faster Filtering

Filtering based on:

```
oa.date_of_admission - oa.previous_admission_date <= INTERVAL '30 days'
```

is more efficient when the rows are already ordered and grouped.

### 3. Faster Join with Patient Table

Since rows were accessed in indexed order, the join on patient\_id benefited indirectly.

### 4. Scalable Improvement

Even though the dataset contains only ~55k rows, the improvement is noticeable.  
On a dataset with millions of rows, the performance gain would be much more significant.

## 7. Future Optimization Opportunities

If this system expands to production or larger datasets, additional optimizations could include:

### 1. Additional Indexes

- (condition\_id)
- (hospital\_id)
- (insurance\_id)
- (date\_of\_admission) for range queries

### 2. Table Partitioning

Partitioning admission by:

- year
- quarter
- month

can greatly improve performance on time-based queries.

### 3. Materialized Views

Examples:

- Monthly hospital readmission trends
- Condition-wise average length of stay
- Patient-level longitudinal visit summaries

### 4. dbt Integration

The star schema and fact table produced in dbt could include preprocessing of:

- readmission flags
- length of stay buckets
- insurance risk categories

## 8. Conclusion

This performance tuning exercise demonstrates:

- Proper use of **EXPLAIN ANALYZE**
- Identification of bottlenecks in analytical queries
- Correct index design based on access patterns
- Measurable performance gains (23.3% improvement)

The optimization aligns with principles of relational query tuning and enhances the analytical layer performance for the healthcare domain dataset.