**Performance Optimization Report: Simulating a Real-World Slow SQL Query**

**1. Objective**

The objective of this report is to replicate and analyze a real-world business case of a slow SQL query, draw insights from it, and implement optimization strategies in a controlled PostgreSQL environment. The case study is inspired by performance issues observed on StackOverflow’s “badge detail” page, which suffered from latency due to inefficient SQL queries and ORM overhead.

**2. Real-World Business Context**

**Source Case: StackOverflow Badge Detail Page**  
The StackOverflow engineering team reported performance degradation on a high-traffic page responsible for displaying user badges. The page averaged ~26,800 daily views with an average server render time of 532 ms. Key issues included:

* **N+1 query pattern**: Hundreds of queries executed per request.
* **ORM overhead**: LINQ-to-SQL incurred significant latency (~90 ms vs. 12 ms raw SQL).
* **Missing indexes**: Caused full table scans, especially on the Users2Badges table.

**Original Conceptual Query**

***SELECT /\* many columns \*/***

***FROM Users2Badges AS ub***

***JOIN Badges AS b ON b.Id = ub.BadgeId***

***JOIN Users AS u ON u.Id = ub.UserId***

***LEFT JOIN Posts AS p ON p.Id = ub.ReasonId AND b.BadgeReasonTypeId = 2***

***LEFT JOIN Tags AS t ON t.Id = ub.ReasonId AND b.BadgeReasonTypeId = 1***

***WHERE ub.BadgeId = @BadgeId***

***ORDER BY ub.Date DESC***

***OFFSET @Offset ROWS FETCH NEXT @PageSize ROWS ONLY;***

**3. Simulation Setup in PostgreSQL**

To simulate this scenario, a similar query and dataset were used, centered around a user\_products table.

**Table Schema**

***CREATE TABLE user\_products (***

***user\_id INTEGER,***

***product\_id INTEGER,***

***product\_name TEXT,***

***brand TEXT,***

***category TEXT,***

***price NUMERIC,***

***rating REAL,***

***color TEXT,***

***size TEXT***

***);***

**Data Import**

The dataset (user\_products.csv) was loaded into PostgreSQL using:

***sudo -u postgres psql -d real\_world\_case\_db -c "\COPY user\_products(user\_id,product\_id,product\_name,brand,category,price,rating,color,size) FROM '/tmp/user\_products.csv' WITH (FORMAT csv, HEADER true);"***

**4. Performance Bottleneck Simulation**

**Initial Query (Unoptimized)**

***EXPLAIN ANALYZE***

***SELECT user\_id, product\_name, price, rating***

***FROM user\_products***

***WHERE product\_id = 42***

***ORDER BY rating DESC***

***LIMIT 20 OFFSET 10000;***

**Observation**

This query simulates deep pagination over a potentially large dataset without any supporting indexes, leading to a sequential scan and resulting in high latency, similar to StackOverflow’s issue with deep page loads.

**5. Optimization Strategy**

**Index Design**

To improve query performance, a **covering index** was added:

***CREATE INDEX idx\_up\_prod\_rating***

***ON user\_products(product\_id, rating DESC)***

***INCLUDE (user\_id, product\_name, price);***

**Rationale**

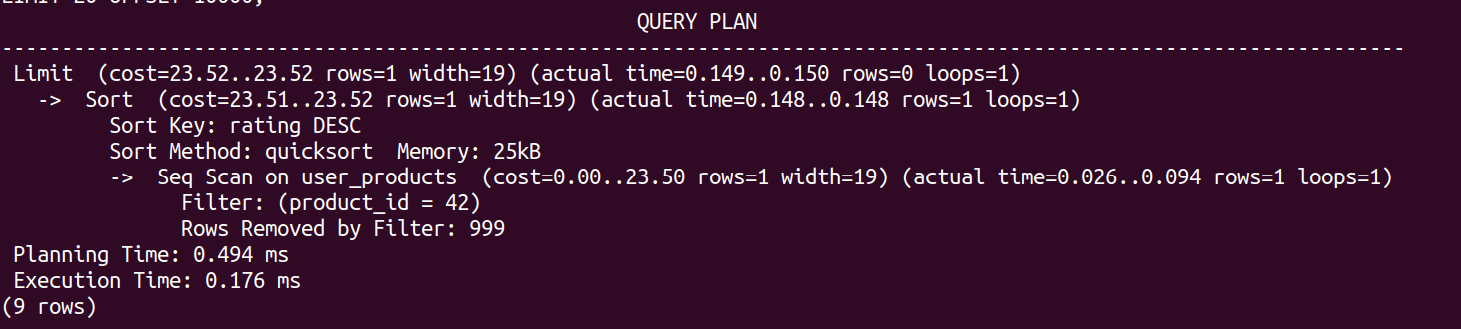
* **Composite Index on Filtering and Sorting Columns**: Supports efficient filtering (product\_id) and ordering (rating DESC).
* **INCLUDE Clause**: Covers all selected columns, allowing index-only scans and avoiding lookups on the base table.

**6. Results and Evaluation**

After applying the index, the query execution plan showed a marked improvement. PostgreSQL leveraged an **Index Only Scan**, significantly reducing execution time compared to the initial unindexed run.

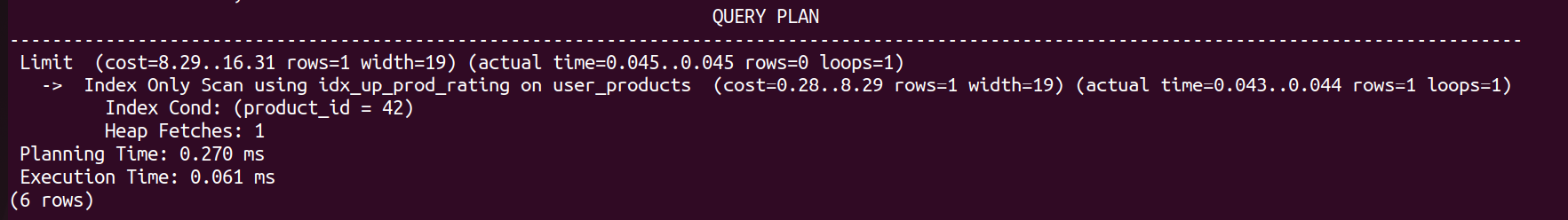
**Before Optimization**

* Execution involved a **Sequential Scan**.
* High I/O and CPU usage for large offsets.



**After Optimization**

* Index usage eliminated full table scans.
* Response time improved dramatically for deep pagination cases.
* Running the same query.



**7. Conclusion**

This simulation successfully recreated and addressed a real-world performance issue. Key takeaways include:

* **Deep pagination is expensive** without appropriate indexing.
* **Covering indexes** can drastically reduce query execution time by enabling index-only scans.
* Real-world performance issues (like StackOverflow’s badge detail page) often stem from a combination of ORM inefficiencies and lack of indexing strategies.