Database Query Performance Analyzer

1. Project Overview

The **Database Query Performance Analyzer** is a full-stack tool built to provide actionable insights into PostgreSQL query execution plans. It helps developers and database administrators (DBAs) understand how queries run under the hood, pinpoint performance bottlenecks, and discover missing indexes.

The platform provides a real-time monitoring and analysis dashboard with advanced capabilities such as **Query Plan Visualization**, **Execution History Tracking**, and a **Smart Index Advisor**.

---

2. Technology Stack

Backend Analytics Engine

- **Framework:** FastAPI (Python)
- **ASGI Server:** Uvicorn
- **Database Connection & ORM:** SQLAlchemy, psycopg2
- **Config Management:** pydantic, python-dotenv

Interactive Frontend Dashboard

- **Framework:** Streamlit
- **Data Manipulation:** Pandas
- **Visualization:** Graphviz (for query plan trees), Plotly

Database

- **Engine:** PostgreSQL 18
- **Core tracking objects:** Real-time query histories and JSON-formatted execution plans stored in a `query_history` ORM table.

---

3. Core Features & Architecture

3.1 Advanced Query Analyzer

The core feature of the application is the `AnalyzeQuery` module.

- **Workflow:** When a user inputs a raw SQL query via the Streamlit frontend, the backend receives it and wraps it in a Postgres `EXPLAIN (ANALYZE, COSTS, VERBOSE, BUFFERS, FORMAT JSON)` statement.
- **Metric Extraction:** The backend parses the resulting JSON structure returned by the PostgreSQL optimizer to extract critical performance metrics, such as:
    - **Actual Execution Time (ms)**

    - **Total Estimated Cost**

- **Visual Graph Generation:** Using the `graphviz` library on the frontend side, the parsed JSON is recursively translated into an interactive hierarchical tree graph. This visual representation allows users to immediately spot expensive nested loops, sequential scans, or excessive sorting times.

3.2 Smart Index Advisor

PostgreSQL execution plans contain detailed node-level operations. The Index Advisor acts as an expert DBA by traversing historical query plan JSON trees.

- **Rule-Based Engine:** The advisor specifically targets highly-inefficient node types in the execution plan tree:
    - **Sequential Scans (`Seq Scan`):** Identifies tables missing indexes and extracts column names from the `Filter` conditions to suggest covering indexes.
    - **Expensive Joins (`Hash Join`, `Merge Join`, `Nested Loop`):** Analyzes `Hash Cond` and `Join Filter` arguments to suggest indexing the join keys.
    - **Unoptimized Sorts (`Sort`):** Looks for missing indexes on `Sort Key`s to eliminate costly in-memory or on-disk disk merge sorts.
    - **Aggregations (`Aggregate`):** Looks for missing indexes on `Group Key`s.
- **Actionable Outputs:** Generates explicit `CREATE INDEX` recommendations mapped directly to the problematic SQL queries.

3.3 Multi-Profile Execution History

The platform stores a persistent record of all executed queries.

- **Data Persistence:** Every analyzed query is saved in the auto-generated `query_history` table via SQLAlchemy upon execution, complete with timestamps, execution duration, and raw JSON plans.
- **Profile Support:** The tool natively supports database profiles (`dev`, `staging`, `prod`) configured via `.env`. Users can seamlessly switch between profiles in the UI to measure query performance differences across environments.
- **History Dashboard:** A streamlined history view built with Pandas dataframes allows users to track performance regressions over time.

---

4. How the System Works (Data Flow)

1. **User Input:** User submits an SQL query through the Streamlit web interface.
2. **API Request:** Streamlit sends a `POST /api/v1/analyze` request to the FastAPI backend, payload containing the SQL string and the target profile (e.g., `dev`).
3. **Database Execution:** FastAPI establishes a Postgres connection via SQLAlchemy to the profile's DSN, appends the `EXPLAIN ANALYZE ... FORMAT JSON` wrapper, and executes it.
4. **Data Persistence:** The execution metrics and JSON plan are asynchronously persisted to the `query_history` schema table.
5. **UI Rendering:** The frontend consumes the standard `AnalyzeResponse` JSON, rendering the metrics (cost & time) natively, while passing the plan array to the local python `graphviz` adapter to draw the visual tree.

6. **Background Analysis:** Separately, the Index Advisor periodically pulls the top *N* slowest queries from history and runs its regex-based rule engine (`extract_columns_from_condition`) over the plan structures to identify optimization opportunities.