

# Comprehensive Technical Report: Database Query Performance Analyzer

## 1. Executive Summary

The **Database Query Performance Analyzer** is an advanced, full-stack analytical tool designed to bridge the gap between raw PostgreSQL execution metrics and actionable database insights. It provides an intuitive platform for developers and database administrators to submit SQL queries, visualize internal query execution plans in real-time, and automatically generate schema optimization recommendations. Built with a scalable Python backend (FastAPI, SQLAlchemy) and an interactive frontend dashboard (Streamlit), the application turns complex `EXPLAIN ANALYZE` outputs into clear visualizations and actionable `CREATE INDEX` suggestions.

---

## 2. System Architecture

The application is architected utilizing a modern decoupled full-stack design. It separates concerns between the user interface, the API orchestration layer, and the database engine.

### 2.1 Backend Service (FastAPI)

The backend is a high-performance REST API built with **FastAPI** and served via the **Uvicorn** ASGI server.

- **API Routing (`api/router.py`):** Exposes three primary endpoints:
  - `POST /analyze`: Receives a SQL string, executes it with `EXPLAIN (ANALYZE, FORMAT JSON)`, and persists the results.
  - `GET /history`: Retrieves the execution history from the local monitoring tables.
  - `GET /index_advisor`: Scans the history to provide smart indexing suggestions.
- **Core Configuration (`core/config.py`):** Utilizes `python-dotenv` and plain `@dataclass` classes to load and manage sensitive PostgreSQL credentials (`DB_USER`, `DB_PASSWORD`, `DB_HOST`, `DB_PORT`, `DB_NAME`). It natively supports multiple application profiles: `dev`, `staging`, and `prod`.
- **Database Connection Layer (`core/database.py`):** Employs **SQLAlchemy** to manage a dynamic dictionary of PostgreSQL connection engines.  
`pool_pre_ping=True` is enabled to ensure connection vitality before query execution.

### 2.2 Frontend Dashboard (Streamlit)

The user interface is built with **Streamlit**, providing a reactive and data-rich dashboard.

- **Query Analyzer Tab:** A workspace where users input SQL strings. It displays summary metrics (Total Cost, Execution Time in ms) extracted from the root plan node.
- **Execution Plan Visualization (`components/plan_visualizer.py`):** Uses the **Graphviz** library to recursively crawl the nested JSON execution plan returned by

PostgreSQL and render it as a directed acyclic graph (DAG). This allows users to easily visualize the flow of data from scanning to joining and aggregation.

- **Dashboard History Tab:** A tabular view powered by **Pandas** that presents a chronological log of previously executed queries, their profiles, and latency data.
- **Index Advisor Tab:** An interface that surfaces the AI-like insights generated by the backend optimization module.

## 2.3 Persistence Layer (PostgreSQL)

The target database itself acts as both the engine to analyze queries and the storage layer for the tool's history.

- **Monitoring Schema (`core/models.py`):** An SQLAlchemy model named `QueryHistory` automatically creates the `query_history` table in the target database. It tracks:
    - `id, profile, query_text`
    - `execution_time_ms, total_cost`
    - `plan_json` (Stores the raw Postgres JSON output in a native JSON column)
    - `timestamp`
- 

# 3. Core Operational Workflows

## 3.1 The Query Analysis Pipeline

When a user clicks "Analyze Query," the following sequence triggers:

1. The frontend sends the raw SQL to the backend via a REST POST payload.
2. The `api/router.py` validates the query string to prevent empty submissions.
3. The SQL is wrapped in a Postgres directive: `EXPLAIN (ANALYZE, COSTS, VERBOSE, BUFFERS, FORMAT JSON) <query>`.
4. The SQLAlchemy session executes this command against the selected profile's database. Postgres optimizer runs the query, respects the `ANALYZE` flag to gather actual row times, and outputs a highly detailed JSON structure instead of raw text.
5. Extracting the root node, the backend computes the top-level execution metrics.
6. A new `QueryHistory` record is generated and committed to the database.
7. The JSON plan is returned to Streamlit, where `graphviz` renders the nested operations tree.

## 3.2 The Smart Index Advisor Engine

The core intelligence of the application resides in `optimization/index_advisor.py`. This module acts as an automated DBA by analyzing historical JSON execution plans.

It functions via a recursive tree-walking algorithm:

- **Sequential Scans (Seq Scan):** If a scan occurs and contains a `Filter` clause (e.g., `(users.age > 18)`), a regex engine parses the clause to extract the column names (`users.age`). It suggests a B-Tree index on these columns to convert the slow sequential scan into an Index Scan.
- **Join Operations (Hash Join, Merge Join, Nested Loop):** The engine inspects `Hash Cond`, `Merge Cond`, and `Join Filter` attributes within the JSON node. It

identifies the exact keys joining the two tables and recommends indexing both sides of the condition.

- **Data Sorting (Sort):** Inspects the `Sort Key` array. Unoptimized sorts can spill to disk and severely bottleneck performance. An index covering the sort keys allows for pre-sorted retrieval.
- **Aggregations (Aggregate):** Inspects the `Group Key` to recommend indexing columns used in `GROUP BY` clauses.
- **Stopword Filtering:** The regex engine maintains a set of SQL stopwords (AND, OR, NOT, IS, NULL, INT, DATE) to filter out syntax noise, ensuring that only valid table columns are returned in the recommendations.

---

## 4. Technical Specifications & Dependencies

The project relies on specific versions of modern Python libraries, managed via `pip` and a local virtual environment:

| Category           | Library   | Purpose  |
|--------------------|---|--|
| Backend & Routing  | <code>fastapi</code> , <code>uvicorn</code>                                   | High performance ASGI web server                 |
| Database & ORM     | <code>sqlalchemy</code> , <code>psycopg2-binary</code> , <code>asyncpg</code> | PostgreSQL communication and ORM                 |
| Data Validation    | <code>pydantic</code> , <code>python-dotenv</code>                            | Schema enforcement and <code>.env</code> parsing |
| Frontend UI        | <code>streamlit</code>  | Reactive data dashboard                          |
| Data Visualization | <code>graphviz</code> , <code>plotly</code> , <code>pandas</code>             | DAG graph generation and dataframes              |
| PDF Generation     | <code>markdown-pdf</code>   | Report compilation                               |

---

## 5. Security & Configuration Best Practices

- **Credential Management:** Avoids hardcoding database passwords. Database strings are dynamically generated via `os.getenv` utilizing the `dotenv` library. A top-level `.env` file securely stores the `postgres` username, `admin123` password, and host parameters.
  - **Environment Isolation:** The `DatabaseProfile` architecture permits complete isolation between Development (`dev`), Staging (`staging`), and Production (`prod`) data sources.
  - **Authentication Resilience:** Designed to handle secure PostgreSQL connection topologies, including modifying `pg_hba.conf` temporarily for credential recovery if necessary.
- 

## 6. Future Enhancements

Potential roadmap features to expand the platform's capabilities:

1. **Schema Migration Integration:** Integrate with `Alembic` for programmatic application of the `CREATE INDEX` recommendations.

2. **Alerting System:** Implement an alerting threshold if average query execution time on `prod` exceeds  $N$  milliseconds.
3. **Caching:** Integrate a Redis caching layer for the `/history` endpoint to handle large scale dashboard refreshes.