UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

DATABASE WORKLOAD EXTRACTION AND ANALYSIS

ROZHINA AHMADI

Thesis supervisor
ALBERTO ABELLO GAMAZO (Department of Service and Information System Engineering)

Thesis co-supervisor
WAFAA RADWAN

Degree
Bachelor's Degree in Informatics Engineering (Software Engineering)

Bachelor's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

01/07/2025

DATABASE WORKLOAD EXTRACTION AND ANALYSIS

ROZHINA AHMADI

Thesis supervisor
ALBERTO ABELLO GAMAZO (Department of Service and Information System Engineering)

Thesis co-supervisor
WAFAA RADWAN

Degree
Bachelor's Degree in Informatics Engineering (Software Engineering)

Bachelor's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

01/07/2025

Abstract

This bachelor’s thesis presents the design and implementation of an automated, modular tool for SQL workload extraction and analysis, tailored to the Sloan Digital Sky Survey (SDSS) logs. The tool aims to streamline and automate processes that were previously performed manually in academic studies, through a processing pipeline that includes fetching, parsing, cleaning, and grouping SQL queries. Its primary objective is to identify patterns within historical SQL logs to support more effective server performance optimization based on real workload behaviour.

The backend is developed in Python using FastAPI, enabling efficient API creation, query processing, and structured output generation. Both command-line and graphical interfaces have been implemented to support users with varying technical backgrounds. The frontend is built with React, offering an intuitive interface. Key features of the tool include clause normalization, query shape detection, and semantic grouping based on structural similarity and query modifiers.

The tool was developed iteratively using agile principles and validated through manual quality assurance, case studies, and continuous feedback. The result is an extendable tool that simplifies SQL workload processing and supports research on large-scale database optimization. Overall, the tool balances automation and flexibility to support workload-aware database optimization.

Resum

Aquest treball de final de grau presenta el disseny i la implementació d'una eina modular i automatitzada per a l’extracció i anàlisi de càrregues de treball SQL, adaptada específicament als registres del Sloan Digital SkySurvey (SDSS). L’eina té com a objectiu optimitzar i automatitzar processos que anteriorment es feien manualment en estudis acadèmics, mitjançant una flux de processament que inclou la recuperació, l'anàlisi sintàctica, la neteja i l'agrupació de consultes SQL. L’objectiu principal és identificar patrons dins dels registres històrics per afavorir una optimització més eficaç del rendiment del servidor en funció del comportament real de la càrrega de treball.

El backend està desenvolupat amb Python utilitzant FastAPI, fet que permet una creació eficient d’APIs, el processament de consultes i la generació estructurada de resultats. S’han implementat interfícies tant per línia de comandes com gràfiques per adaptar-se a perfils d’usuari amb diferents nivells tècnics. El frontend està construït amb React i ofereix una interfície intuïtiva. Les funcionalitats clau inclouen la normalització de clausules, la detecció de l’estructura de la consulta i l'agrupació semàntica basada en la similitud estructural i els modificadors.

L’eina ha estat desenvolupada de manera iterativa seguint principis àgils i validada mitjançant control de qualitat manual, estudis de cas i feedback continu. El resultat és una eina extensible que simplifica el processament de càrregues de treball SQL i dona suport a la investigació per a l’optimització de bases de dades a gran escala. En conjunt, l’eina combina automatització i flexibilitat per afavorir una optimització conscient del comportament de les consultes.

Resumen

Este trabajo de fin de grado presenta el diseño e implementación de una herramienta modular y automatizada para la extracción y el análisis de cargas de trabajo SQL, adaptada específicamente a los registros del Sloan Digital SkySurvey (SDSS). La herramienta tiene como objetivo agilizar y automatizar procesos que anteriormente se realizaban manualmente en estudios académicos, mediante un flujo de procesamiento que incluye la obtención, el análisis sintáctico, la limpieza y la agrupación de consultas SQL. Su objetivo principal es identificar patrones dentro de los registros históricos para optimizar de forma más eficaz el rendimiento del servidor en función del comportamiento real de las cargas de trabajo.

El backend está desarrollado en Python utilizando FastAPI, lo que permite una creación eficiente de APIs, procesamiento de consultas y generación estructurada de resultados. Se han implementado interfaces tanto de línea de comandos como gráficas para usuarios con distintos niveles técnicos. El frontend está construido con React, ofreciendo una interfaz intuitiva. Las funcionalidades clave de la herramienta incluyen la normalización de cláusulas, la detección de la estructura de las consultas y la agrupación semántica basada en similitudes estructurales y modificadores de consulta.

La herramienta fue desarrollada de manera iterativa siguiendo principios ágiles, y validada mediante control de calidad manual, estudios de caso y retroalimentación continua. El resultado es una solución extensible que simplifica el procesamiento de cargas de trabajo SQL y respalda investigaciones orientadas a la optimización de bases de datos a gran escala. En conjunto, la herramienta equilibra automatización y flexibilidad para favorecer una optimización consciente de las cargas de trabajo.

Table of Contents

List of Tables

List of Figures

1. Introduction

Large-scale databases generate vast amounts of query logs, requiring efficient processing and analysis. The Sloan Digital Sky Survey (SDSS) has accumulated a vast volume of SQL queries recorded in its SkyServer database logs. Analysing these SQL logs provides valuable insights into query patterns, system performance, and workload evolution.

A prior study in [1] examines how changing query workloads impact NoSQL schema optimization, emphasizing the need for continuous database adaptation. However, their research primarily addresses schema refactoring rather than the initial process of workload extraction and filtering.

This project aims to fill this gap by automating the extraction of SQL workload data from SDSS logs, eliminating manual processing and improving scalability. By building on prior research on NoSQL database refactoring [1], it implements a structured approach to query log analysis, enhancing efficiency and supporting further research into SQL workload evolution and optimization.

1.1 Context

The Sloan Digital Sky Survey (SDSS) is a long-term astronomical survey that has collected extensive datasets over multiple decades [2]. These datasets are stored in the SDSS SkyServer, which logs SQL query traffic through the SqlLog database. Over time, the number of SQL queries processed by SDSS SkyServer has increased exponentially, from 50,333 queries in 2003 to 3,623,390 queries in 2023, making manual filtering and grouping infeasible [3].

The increasing volume and complexity of database workloads require efficient tools for automated processing and analysis. This project builds on prior research on database workload evolution and schema optimization [1], which studied how changing workloads affect database performance and structure. While that study focused on NoSQL schema refactoring, this project specifically focuses on automating SQL workload extraction from SDSS logs, eliminating manual processes and improving scalability.

The current method for extracting SQL workloads from SkyServer requires executing queries individually, downloading results, filtering invalid entries, and manually grouping them. Due to the high query volume, this manual approach becomes inefficient and lacks scalability.

1.2 Concepts and Definitions

Database Query: A database query is a request to retrieve, insert, update, or delete data from a database.

SQL Logs: A record of executed SQL queries stored by a database system, often including metadata like timestamps, user IDs, and execution details. These logs track user interactions and database activity.

Workload: A set of database queries and operations executed over a period, grouped for analysis, optimization, and performance evaluation.

Workload Extraction: The process of retrieving and filtering SQL queries from database logs to analyse query patterns, detect trends, and optimize database performance. This process often involves preprocessing steps such as removing invalid queries, categorizing operations, and extracting key metadata.

1.3 Stakeholders

Stakeholders are the key individuals or groups affected by the project and its development. Their involvement is essential, as they help define system requirements based on their needs. The better these stakeholders are identified, the more accurately the system can be designed to meet its objectives.

Researchers and Data Scientists: Researchers and data scientists rely on workload analysis to extract useful insights from large datasets. The system developed in this project will automate the extraction and processing of SQL logs, reducing manual effort and improving the efficiency of database analysis. This tool will help researchers identify query patterns, optimize system performance, and analyse trends in workload evolution.

Software Engineer and Developer: The development and maintenance of this system require software engineers to implement key functionalities. The author of this project takes on the role of developer, designing and implementing the tool to ensure it meets efficiency, scalability, and usability requirements. The main responsibilities include automating data extraction, structuring the analysis process, and ensuring the system is adaptable to future needs.

Thesis Supervisor and Co-Supervisor: The project is supervised by Professor Alberto Abelló Gamazo, from the Department of Service and Information System Engineering at UPC, and co-supervised by Wafaa Radwan. They provide academic and technical guidance, ensuring that the project follows best practices and meets the required standards. Regular meetings and feedback sessions help keep the development process aligned with the objectives. Their expertise helps refine methodologies, improve the project’s structure, and ensure the final system is both functional and well-documented.

2. Justification

The development of an automated system for SQL workload extraction is necessary due to the inefficiency of manual methods and the lack of existing solutions tailored to SDSS SkyServer logs. In this section I compare some key features offered by the existing solutions with the proposed tool.

While several tools exist for SQL workload monitoring and analysis, they are limited to specific database management systems. Existing SQL log analysers, such as SQL Profiler [4] and Percona Query Analytics [5], are limited to specific database management systems and do not support SkyServer's web-based SQL log extraction. Similarly, current SQL parsing tools like `sqlparse` and Apache Calcite can analyse syntax but lack semantic classification capabilities. Commercial query analysis solutions are often proprietary and inflexible. In contrast, this research will develop a customizable tool that can be adapted for broader SQL workload analysis beyond SkyServer.

The key differentiating features of the proposed system, compared to existing solutions, are as follows:

Supports SDSS SkyServer SQL Log Extraction: Unlike existing tools, the proposed system is designed specifically to extract SQL logs from SkyServer, automating the extraction process directly from its server.

Automated Query Extraction: The proposed system eliminates the need for manual query execution and data filtering, making the process more efficient and scalable.

Query Categorization: Unlike SQL Profiler and Percona Query Analytics, which offer limited categorization, this system will classify queries based on workload analysis.

Workload Evolution Tracking: Tracking query usage over time allows for trend analysis and system performance improvements, a feature missing in most existing tools.

Scalability for Large Query Logs: Since SkyServer handles large datasets with millions of queries, the system is optimized for large-scale SQL log processing, ensuring efficiency in handling high query volumes.

Customization & Open-Source: Unlike commercial tools, which are often restrictive, the proposed system is fully customizable and open-source, allowing adaptation to various SQL workload analysis needs.

This research develops an automated tool that streamlines workload extraction, preprocessing, and analysis, reducing manual effort. Unlike existing solutions, it will be scalable, adaptable, and optimized for SkyServer, while also being extendable for broader

SQL workload analysis. Automating this process will ensure faster and more accurate extraction, reduce human effort, and allow for large-scale data processing.

Given the need for efficiency, scalability, and domain-specific optimization, this project justifies the development of a dedicated system over repurposing existing solutions.

The following table provides a comparison between existing tools and the proposed system:

Feature	SQL Profiler (Microsoft) [4]	Percona Query Analytics [5]	sqlparse (Python) [6]	Apache Calcite [7]	Proposed System
1	No	No	No	No	Yes
2	Manual	Limited	No	Yes	Fully Automated
3	Basic	Advanced	No	Advanced	Tailored for Workload Analysis
4	No	No	No	Yes	Yes
5	Limited	Moderate	No	High	Optimized for Large Datasets
6	No	No	Yes	Yes	Fully Customizable

Table 1: Comparison between Existing Tools and the Proposed Tool (Own Work)

3. Project Scope and Requirements

To ensure a clear project structure, this section defines the primary objective, sub-objectives, system requirements, and potential risks that may affect implementation.

3.1 Objectives and Sub-Objectives

The primary objective is to design and implement an automated tool that extracts, processes, and organizes SQL workload data from SDSS logs, replacing the current manual process. The system will generate structured outputs to support query analysis and database performance optimization.

The project will address the following sub-objectives:

1. Improve database workload management by automating SQL log retrieval from SDSS SkyServer, reducing manual intervention.
2. Enhance query optimization and data structuring by extracting and processing key metadata, such as tables, attributes, query types, and selection predicates.
3. Analyse and categorize queries based on their structure and usage patterns.
4. Support workload evolution studies by analysing query patterns over multiple database releases and generating comparative statistics.
5. Ensure structured data storage to facilitate further analysis and scalability of the extracted workload information.
6. Develop a scalable and reusable system for future database workload tasks.

3.2 Requirements

To achieve the expected outcomes, both functional and non-functional system requirements must be clearly defined in line with the project's objectives. Given the agile methodology adopted for development, as outlined in Section 4, these requirements may be revised or adapted throughout the process based on time constraints and system limitations.

3.2.1 Functional Requirements

The system must:

1. Retrieve SQL logs automatically from SDSS SkyServer.
2. Filter out unsuccessful queries and those involving customer user tables.
3. Parse and categorize queries, extracting key components such as table names, attributes, conditions, and query types.
4. Support exporting processed workloads in JSON, CSV, or a database.
5. Enable users to visualize and filter queries based on extracted features.

6. Ensure an intuitive user interface for easy interaction.

3.2.2 Non-Functional Requirements

The system should:

1. Ensure efficiency by processing large datasets with minimal latency.
2. Support future extensions and additional databases.
3. Follow a modular architecture, allowing easy integration of new features and reuse for different workload extractions.
4. Provide a clear and intuitive graphical user interface.

3.3 Potential Obstacles and Risks

During the development of this project, the following challenges and risks may arise that could impact the efficiency and feasibility of the system. These risks can be categorized into technical risks and project-related risks.

3.3.1 Technical Risks

Data Access Limitations: The SDSS SkyServer may have restrictions on data access and lacks a dedicated API for structured extraction, making SQL log retrieval complex.

Data Volume Challenges: The system must efficiently process millions of SQL queries without slowing down. Optimized storage, indexing, and filtering techniques will be required to ensure smooth performance.

3.3.2 Project Risks

Timeline Constraints: Given the iterative nature of development using Kanban, adapting to unexpected challenges may delay progress.

Dependency on External Feedback: Validation with stakeholders may take longer than anticipated, impacting iterative refinements.

4. Methodology

This project follows an agile development approach, ensuring flexibility, iterative improvements, and continuous feedback throughout the development cycle. Given its self-organized nature, the project requires a methodology that allows for ongoing adjustments rather than rigid planning.

The Kanban methodology has been chosen as it provides a continuous workflow, which allows tasks to be addressed dynamically instead of in predefined sprints. Unlike Scrum, which follows fixed-length iterations (sprints), Kanban offers real-time task tracking without strict deadlines, making it more suitable for projects with ongoing workload processing and refinement [8]. A Kanban board will be used to track progress, updating tasks in real time as they move through different stages (To Do → In Progress → Completed). This methodology enhances transparency, adaptability, and task prioritization.

Improvement is a continuous process, so collecting feedback and integrating it into the system is essential. Kanban is an effective approach for enabling real-time adjustments based on findings and ongoing refinements. After each completed task, the initial plan will be reviewed and updated to maintain efficiency and alignment with project objectives. This approach ensures structured yet flexible task management, making it well-suited for an evolving system.

4.1 Version Control with Git

To manage the versions of the generated code, Git will be used. Git is a widely adopted version control system that allows developers to track, manage, and collaborate on changes efficiently. It provides a structured way to handle modifications, enabling rollback, history tracking, and seamless integration across development environments.

I will specifically use GitHub, which is a cloud-based version control platform built on Git [9]. GitHub organizes the code into repositories. Within these repositories, branches are used to manage different versions of the code. In this project, the following branching strategy will be followed:

Main Branch: The stable version of the project containing production-ready code.

Frontend Branch: Dedicated to the development of the user interface and front-end components.

Backend Branch: Handles the logic, data processing, and workload automation functionalities.

Feature Branches: Temporary branches for developing new features, enhancements, or bug fixes before merging into the main branches.

4.2 Project Management with Trello

Trello will be used as a Kanban-based project management tool to track and organize tasks throughout the project. Trello provides a visual and flexible way to manage progress by using boards, lists, and cards to represent different tasks and workflow stages [10].

The project board will consist of the following columns or lists:

Backlog: Tasks that need to be done but are not yet prioritized.

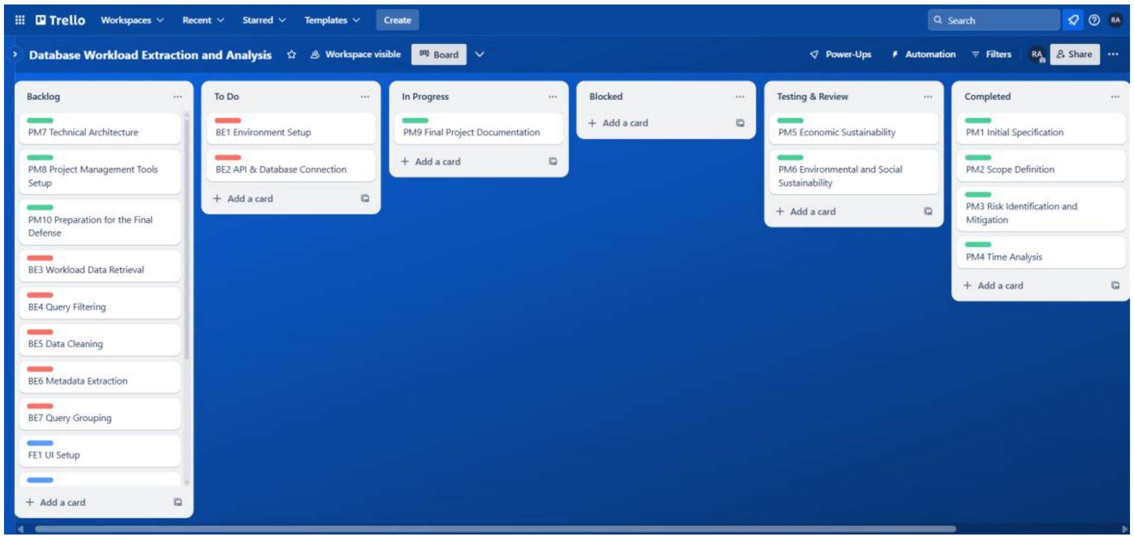
To Do: High-priority tasks that are scheduled for execution.

In Progress: Tasks that are currently being worked on.

Blocked: Tasks that are temporarily blocked due to external dependencies, missing information, or unresolved issues.

Testing and Review: Tasks pending validation or feedback.

Completed: Successfully finished tasks.



Backlog
- PM7 Technical Architecture
- PM8 Project Management Tools Setup
- PM10 Preparation for the Final Defense
- BE3 Workload Data Retrieval
- BE4 Query Filtering
- BE5 Data Cleaning
- BE6 Metadata Extraction
- BE7 Query Grouping
- FE1 UI Setup

To Do
- BE1 Environment Setup
- BE2 API & Database Connection

In Progress
- PM9 Final Project Documentation

Blocked

Testing & Review
- PM5 Economic Sustainability
- PM6 Environmental and Social Sustainability

Completed
- PM1 Initial Specification
- PM2 Scope Definition
- PM3 Risk Identification and Mitigation
- PM4 Time Analysis

Figure 1: Trello Project Management Board (Own Work)

Each task card will include descriptions, labels and attachments, ensuring clear task allocation and progress tracking.

Trello will enable continuous workflow management, facilitating real-time updates, and adaptability to changing priorities.

4.3 Documentation with Microsoft 365

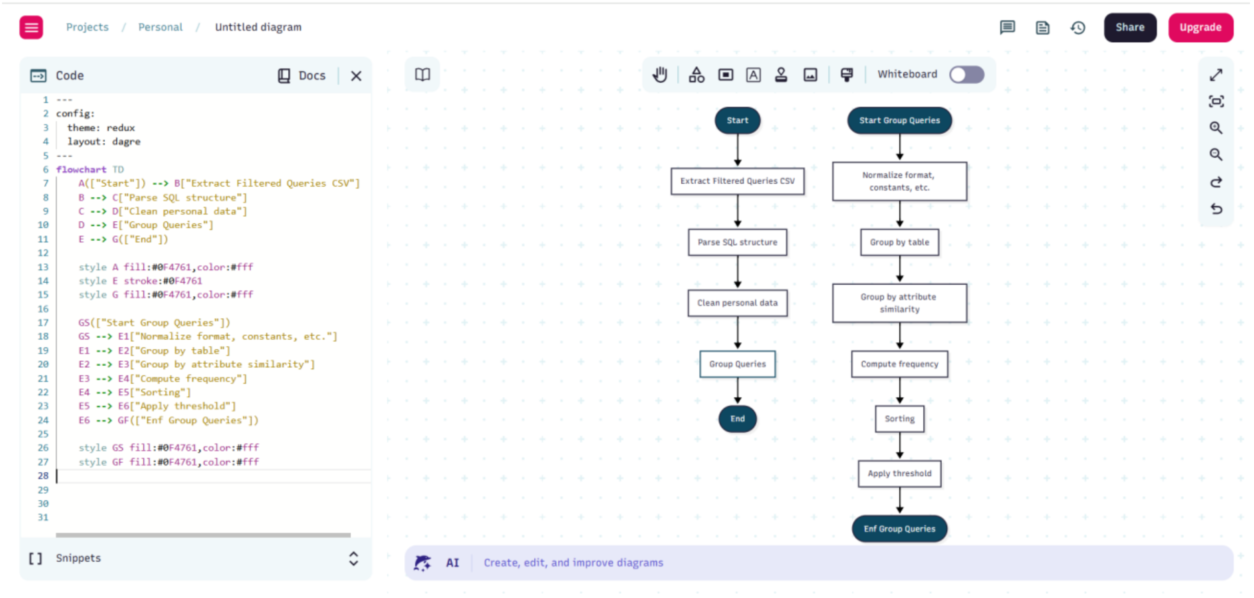
Microsoft Word and Excel will be used for creating and maintaining project documentation. Microsoft Word will be utilized for drafting and editing reports, user manuals, and other textual documents. Microsoft Excel will be employed for managing data, creating charts, and tracking project metrics.

In addition to Word, Markdown was used to author the CLI and GUI user manuals. Markdown provides lightweight syntax, version control compatibility, and native support for syntax-highlighted code blocks. The manuals were converted to .doc using `Pandoc`, enabling consistent academic formatting and integration into the final documentation. This hybrid approach ensures both professional appearance and maintainability of technical content.

4.4 Diagram Generation with AI

To improve visual clarity and maintain a consistent design throughout the documentation, AI-powered diagram generation tools were used to create both flowcharts and Gantt charts of the project. These tools allowed for rapid prototyping of diagrams based on textual prompts and mermaid-style syntax, enabling precise control over structure, layout, and visual styling. Custom prompts and configuration details (e.g., colour codes, layout direction, and node shapes) were provided to the tool to accurately reflect the logic and sequence of operations described in the thesis [10][11].

Flowcharts were generated to represent the backend data processing pipeline and detailed grouping logic, while the Gantt chart was used to visualize the project timeline, task dependencies, and workload distribution.



```
---
config:
  theme: redux
  layout: dagre
---
flowchart TD
    A(["Start"]) --> B["Extract Filtered Queries CSV"]
    B --> C["Parse SQL structure"]
    C --> D["Clean personal data"]
    D --> E["Group Queries"]
    E --> G(["End"])

    style A fill:#0F4761,color:#fff
    style E stroke:#0F4761
    style G fill:#0F4761,color:#fff

    GS(["Start Group Queries"])
    GS --> E1["Normalize format, constants, etc."]
    E1 --> E2["Group by table"]
    E2 --> E3["Group by attribute similarity"]
    E3 --> E4["Compute frequency"]
    E4 --> E5["Sorting"]
    E5 --> E6["Apply threshold"]
    E6 --> GF(["Enf Group Queries"])

    style GS fill:#0F4761,color:#fff
    style GF fill:#0F4761,color:#fff
```

Figure 2: Diagram Generation using Mermaid.com (Own Work)

4.5 Validation

Throughout the development of the project, direct and regular communication will be maintained with the Final Degree Project supervisor, Alberto Abelló Gamazo, and the co-supervisor, Wafaa Radwan, to ensure that the established objectives are met and that the quality of the work is maintained.

Weekly meetings will be scheduled to review project progress, discuss potential issues, and make necessary decisions regarding the development process. Additionally, testing and validation of the code will be conducted to ensure proper functionality and maintain the integrity of the developed solution.

The primary communication channel will be Google Gmail, with most meetings conducted online using Google Meets and scheduled via Google Calendar

5. Task Breakdown and Description

Planning is one of the most important aspects of ensuring the success of a project. This document outlines the different tasks that need to be completed to successfully develop and deliver the final thesis project.

The total academic workload for this final degree project is 18 credits, which includes 3 credits from the project management course and 15 credits for the main project. According to the regulations of the Barcelona Faculty of Computer Science, each credit is estimated to require 30 hours of work [13]. Based on this, the total workload for the project is approximately 540 hours, out of which 450 hours are dedicated solely to project development, excluding the project management course.

The project officially began with the Project Management course on February 19, 2025, and is expected to be completed by June 25, 2025, with the oral defence expected to the last week of June 2025. These dates are approximate and subject to change if necessary.

Key Dates and Duration are:

Project Dates: February 19, 2025 – June 19, 2025 (including the Project Management Course February 19, 2025 – March 19, 2025)

Thesis Defence Date: July 1, 2025

Total Duration Estimated: 565 hours (Section 6)

Weekly Work Hours: The planning of the tasks to be carried out during the execution of this project is a key element in ensuring its success. This section outlines the required tasks, covering project management, technical development, risk assessment, and deployment.

5.1 Project Management and Documentation

PM1. Initial Specification: The first step is defining the project objectives and drafting the initial backlog which includes identifying the core functionalities.

PM2. Scope Definition: A market study will be conducted to analyse existing solutions and determine the project's unique approach. Risks will be assessed, and an appropriate methodology will be selected to guide development.

PM3. Risk Identification and Mitigation: Identify potential risks such as API rate limits, data inconsistencies, system crashes, and cloud service failures. Define a mitigation plan early in the development cycle to ensure system stability.

PM4. Time Analysis: The entire project timeline will be structured, estimating task durations and defining dependencies.

PM5. Budget and Economic Sustainability: A cost analysis will determine the total budget, considering human resources, infrastructure, and potential overhead costs.

PM6. Environmental and Social Sustainability: The project's impact on resource consumption, data privacy, and ethical considerations will be evaluated.

PM7. Technical Architecture: The system's architecture will be designed, defining data flow, modularity, and integration points. This includes specifying the technology stack, setting up infrastructure, and establishing the environment for development and testing.

PM8. Project Management tools setup: Configure project management tools, including GitHub and Trello, to track progress and collaboration.

PM9. Final Project Documentation: A complete report detailing all phases of the project, including methodologies, technical choices, and outcomes, will be prepared for submission. This document includes a user manual of the tool as an appendix.

PM10. Preparation for the Final Defence: The final presentation and supporting materials will be developed, summarizing key findings and demonstrating the functionality of the developed system.

5.2 Backend Development

This section includes all backend-related tasks to ensure logical progression from setup to workload processing, optimization, and testing.

5.2.1 Setup and Environment Configuration

BE1. Environment Setup: Install dependencies, configure the project structure, and set up logging/debugging tools.

BE2. API & Database Connection: Establish a connection to SDSS SkyServer to retrieve SQL logs via API requests or direct SQL queries.

5.2.2 Workload Extraction and Preprocessing

BE3. Workload Data Retrieval: Retrieve raw query logs from the SDSS SkyServer Traffic Log (SqlLog table) and store the results in a structured format (CSV) for further analysis.

BE4. Query Filtering: Exclude failed queries from the dataset to ensure only relevant logs are processed.

BE5. Data Parsing and Metadata Extraction: Parse each SQL statement to extract structural features such as involved tables, selected columns, join conditions, filter predicates, and aggregation functions. The results are normalized and saved in JSON format for downstream analysis.

BE6. Query Cleaning: Remove parsed queries that reference private or user-specific databases. This ensures that only public and reproducible queries are retained for workload grouping and modelling.

5.2.3 Workload Grouping and Analysis

BE7. Query Grouping: Group queries based on table usage, column selection, and filtering conditions, ensuring a configurable threshold for meaningful patterns. This task will remove low-frequency queries that fall below the specified threshold and do not contribute significantly to workload patterns.

5.3 Frontend Development

FE1. UI Setup: Build a simple interface using HTML, JavaScript, or a lightweight frontend framework.

FE2. UI/UX Design: Design and develop a simple interface layout to provide an intuitive experience for viewing workload insights and interacting with optimization settings.

FE3. CLI Interface Implementation Develop a standalone Command-Line Interface (CLI) using Python's argparse module. This interface provides users with an alternative way to interact with the tool by running commands to fetch, parse, and analyse workloads directly from the terminal.

FE4. API Integration with GUI: Connect the frontend React application with the backend API using Axios (planned), enabling real-time interaction between the graphical interface and the backend logic.

5.4 Testing

TD1. Backend Performance Evaluation: Compare query execution times before and after optimizations to quantify performance improvements.

TD2. Frontend Testing and Validation: Evaluate UI responsiveness, API response times, and usability to ensure a smooth experience.

TD3. Final Quality Assurance and Bug Fixes: A final system-wide test will be conducted to identify and resolve defects before the tool is released. Ensure a stable connection between backend APIs and the frontend interface, verifying that data exchanges occur in real time.

6. Time Estimation and Gantt Chart

6.1 Initial Time Estimation

All tasks exceeding 50 hours have been broken down into smaller, more manageable components. However, project documentation has been allocated 90 hours because it spans the entire duration of the project and is essential for ensuring a well-structured, maintainable, and reproducible workflow. Unlike other tasks that focus on specific development aspects (such as backend, frontend, or testing), documentation requires continuous effort from the project's inception to its completion. Since documentation is not a discrete task that can be completed in a single phase, it evolves alongside the project.

The table below presents the distribution of these hours across all the defined project tasks.

Code	Task Title	Time (hours)	Dependencies
PM1	Initial Specification	20	
PM2	Scope Definition	20	
PM3	Risk Identification and Mitigation	15	
PM4	Time Analysis	30	PM1, PM2
PM5	Budget and Economic Sustainability	15	PM4
PM6	Environmental and Social Sustainability	15	
PM7	Technical Architecture	25	PM1, PM2
PM8	Project Management Tools Setup	10	
PM9	Final Project Documentation	90	
PM10	Preparation for the Final Defence	30	PM9
BE1	Environment Setup	10	
BE2	API & Database Connection	20	BE1
BE3	Workload Data Retrieval	25	BE2
BE4	Query Filtering	25	BE3
BE5	Data Cleaning	25	BE3
BE6	Metadata Extraction	25	BE3, BE5
BE7	Query Grouping	35	BE3
FE1	UI Setup	10	
FE2	UI/UX Design	25	
FE3	API & CLI Integration	35	FE1
TD1	Backend Performance Evaluation	20	BE7
TD2	Frontend Testing and Validation	20	FE3
TD3	Final Quality Assurance and Bug Fixes	20	TD1, TD2
Total	**Total Hours**	**565**	

Table 2: Time Estimation Breakdown (Own Work)

6.1.1 Initial Gantt Diagram

Additionally, a Gantt chart has been created to provide a clearer timeline and task scheduling, as shown in Figure 3. The red vertical line marks the date on which the diagram was generated, providing context for the project's progress at that point in time.

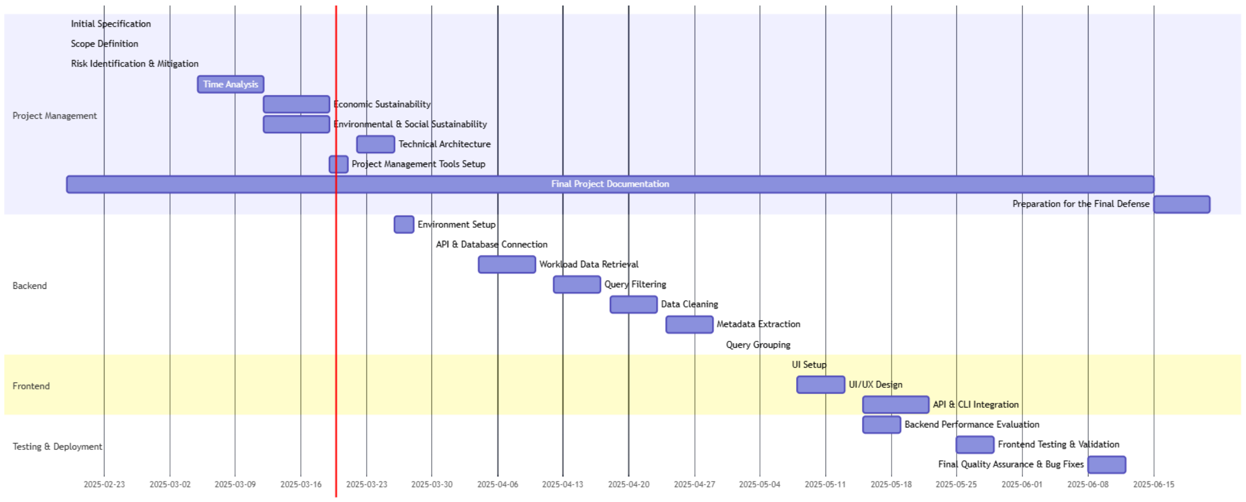


Figure 3: Initial Gantt Diagram (Own Work)

6.2. Final Timeline

Throughout the development of this project, several adjustments were made to the initial plan based on practical challenges and findings during implementation. These changes reflect an adaptation of the project timeline in response to technical complexity and the need for additional supporting features.

6.2.1 Justification of Timeline Changes and Task Adjustments

API & Database Connection (BE2) and **Workload Data Retrieval (BE3)** took significantly more time than initially estimated. This was due to the technical difficulties in connecting to the SDSS SkyServer and the variability in the time required to fetch large volumes of SQL logs. In response, two new subtasks, Workload Retrieval Time Benchmarking and Progress and Estimated Wait Time Visualization were added. These additions help future users of the tool better understand and manage delays associated with data fetching.

Despite delays and uncertainties in the implementation of the data retrieval module, development of other components continued in parallel. During the time I was exploring different methods to access the SDSS logs, resolving connection issues, and benchmarking query response times, I relied on a basic data-fetching method and previously retrieved logs. This decision allowed development to proceed with parsing, other stages, and the frontend development without being blocked by the risks identified in BE2 and BE3. This decision minimized idle time and helped keep the overall project timeline on track.

Data Parsing and Metadata Extraction (BE5) required more effort than expected due to the complexity of reliably extracting all relevant structural features from raw SQL logs. This module was later extended to support query normalization and feature parsing for use in grouping logic, which increased its scope and duration.

Query Cleaning (BE6) was initially planned before parsing, but it had to be rescheduled to occur afterwards, as parsing was required to extract table names and other contextual information necessary for identifying private or user-specific databases like `Mud`. As a result, this step was completed in less time than anticipated, and its complexity was lower than expected once integrated post-parsing.

Query Filtering (BE4) was originally implemented as a separate module but later merged into the workload extraction step (BE3) for efficiency. Since the filtering logic was straightforward (e.g., removing queries with errors or zero results), it consumed less time in total.

Overall, while some tasks required more time due to technical challenges, others were streamlined or adjusted logically based on dependencies discovered during implementation. These changes have been fully reflected in the updated time estimation table and final Gantt diagram.

CLI Interface Implementation (FE3) was originally grouped with GUI/API integration. This task was separated into its own standalone task. This is because it was developed in parallel with backend modules instead of waiting for the frontend or the backend to be finalized. This decision allowed for faster debugging, flexible experimentation, and early-stage functionality testing.

Query Grouping (BE7) was initially estimated at 35 hours. However, as the development progressed, this module required more work than anticipated, ultimately justifying an increase to 45 hours in the updated time estimation. The added complexity stemmed from several factors. First, the original plan focused only on grouping queries by referenced tables. During implementation, I decided to enhance the grouping mechanism by incorporating additional criteria such as selected columns, normalized WHERE conditions, and SQL clause flags like `DISTINCT, TOP, ORDER BY`, and `GROUP BY`. These extensions were necessary to capture more meaningful workload patterns.

To support these advanced grouping criteria, it was also necessary to revise the SQL parsing logic in BE5 to extract the required metadata. Moreover, defining what constitutes a "group" required the introduction of custom rules and thresholds, such as merging patterns with high column overlap (based on Jaccard similarity), and discarding low-frequency query shapes below a configurable threshold.

The following table presents the final updated breakdown of all project tasks, their estimated durations, and dependencies.

Code	Task Title	Time (hours)	Dependencies
PM1	Initial Specification	20	
PM2	Scope Definition	20	
PM3	Risk Identification and Mitigation	15	
PM4	Time Analysis	30	PM1, PM2
PM5	Budget and Economic Sustainability	15	PM4
PM6	Environmental and Social Sustainability	15	
PM7	Technical Architecture	25	PM1, PM2
PM8	Project Management Tools Setup	10	
PM9	Final Project Documentation	90	
PM10	Preparation for the Final Defence	30	PM9
BE1	Environment Setup	10	
BE2	API & Database Connection	**25**	BE1
BE3	Workload Data Retrieval	**45**	BE2
BE4	Query Filtering	**15**	BE3
BE5	Data Parsing and Metadata Extraction	**30**	BE3
BE6	Query Cleaning	**5**	BE3, BE5
BE7	Query Grouping	**45**	BE3, **BE5**
FE1	UI Setup	10	
FE2	UI/UX Design	25	
FE3	CLI Interface Implementation	**15**	
FE4	API Integration with GUI	**20**	FE2
TD1	Backend Performance Evaluation	20	BE7
TD2	Frontend Testing and Validation	20	FE3
TD3	Final Quality Assurance and Bug Fixes	20	TD1, TD2
Total	**Total Hours**	**620**	

Table 3: Final Time Estimation Breakdown (Own Work)

6.2.2 Final Gantt Diagram

The following Gantt diagram shows the final timeline of the project according to the adjustments. The red vertical line indicates the date the diagram was generated.

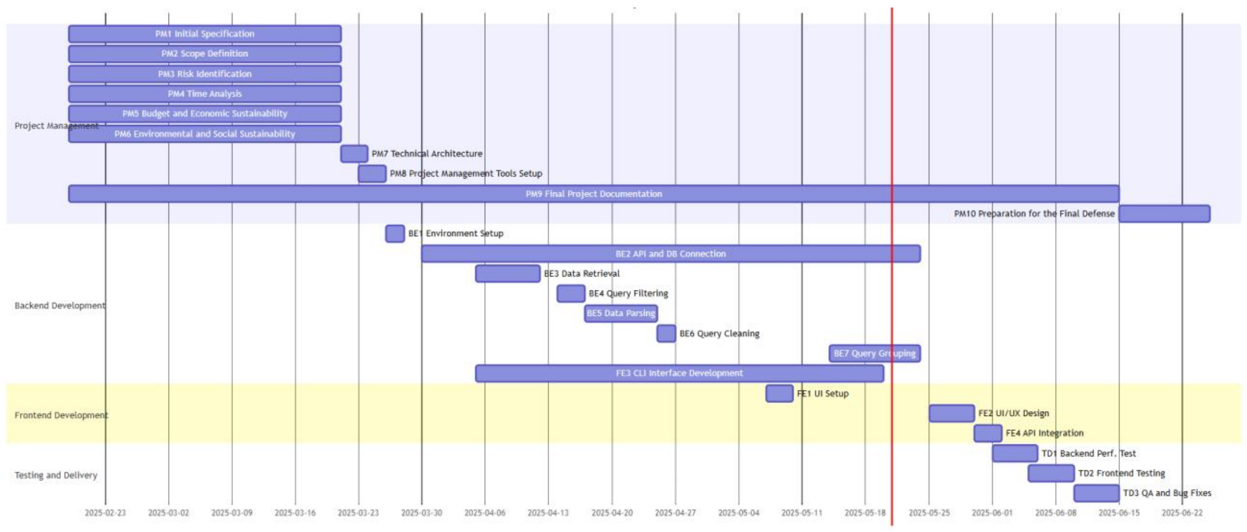


Figure 4: Final Gantt Diagram (Own Work)

7. Risk Management

As mentioned in previous sections, unexpected situations may arise that require modifications to the project plan. To minimize their impact, each identified risk has been evaluated in terms of probability, estimated delay, and mitigation strategies. This section outlines the necessary actions to be taken and alternative plans in case one or more of the documented risks occur.

The following table summarizes the risks, their impact, and mitigation strategies:

Category	Risk	Probability	Estimated Delay	Mitigation Plan
Technical	Data Access Limitations	Medium	20 hours	Develop custom scripts for data extraction and implement caching mechanisms.
Technical	Data Volume Challenges	High	30 hours	Use Big Data technologies like pySpark
Project	Timeline Constraints	High	40 hours	Prioritize critical tasks, allocate buffer time, and re-evaluate dependencies.
Project	Dependency on External Feedback	Medium	15 hours	Schedule early validation meetings and prepare fallback designs to minimize delays.

Table 4: Risk Identification and Evaluation (Own Work)

7.1 Risk Identification and Evaluation

This project faces both technical and project-related risks that could impact efficiency and feasibility. The following section presents mitigation strategies and alternative plans to ensure project continuity in case of unexpected issues.

7.1.1 Technical Risks

Data Access Limitations: The SDSS SkyServer may impose restrictions on data access and does not provide a dedicated API for structured data extraction. This makes retrieving SQL logs more complex.

- Probability: Medium
- Estimated Impact: 20 hours

- Mitigation Strategy: Develop custom scripts to extract logs efficiently and implement caching mechanisms to reduce redundant requests.

Data Volume Challenges: The system must efficiently handle and process millions of SQL queries without slowing down. Poorly optimized storage and query execution could lead to performance bottlenecks.

- Probability: High
- Estimated Impact: 30 hours
- Mitigation Strategy: Use Big Data technologies like pySpark to optimize query storage and ensure proper workload distribution

7.1.2 Project Risks

Timeline Constraints: Given the iterative development approach using Kanban, unexpected challenges or technical difficulties may delay progress.

- Probability: High
- Estimated Impact: 40 hours
- Mitigation Strategy: Prioritize critical tasks, allocate buffer time in the schedule, and re-evaluate task dependencies to ensure flexibility in execution.

Dependency on External Feedback: Validation with stakeholders or external parties may take longer than expected, delaying iterative refinements and system improvements.

- Probability: Medium
- Estimated Impact: 15 hours
- Mitigation Strategy: Schedule early validation meetings, set clear deadlines for feedback, and prepare fallback designs to avoid dependency bottlenecks.

7.2 Alternative Plans and Risk Mitigation Strategies

To counteract potential risks, alternative approaches have been considered:

Allocating Extra Time: Some tasks have been slightly overestimated to allow flexibility in case of delays.

Defining Backup Tasks (Plan B): In case of major issues, additional workaround solutions have been designed but will not be included in the standard project planning or Gantt chart. These contingency measures will only be activated if necessary to ensure that the project finishes on time.

Scope Reduction: As a last resort, if significant delays occur, some non-essential features may be postponed or excluded to focus on delivering a functional core system.

8. Budget

Proper budgeting ensures that the project stays within the allocated resources while accounting for potential risks and unforeseen costs. The budget is structured into human resources, software and infrastructure costs, amortization, and contingencies.

8.1 Identification of Costs

The budget is divided into the following categories:

Category	Description
Staff Costs	Development, testing, and documentation
Software and Data Provider	API access, storage and documentation tools
Amortization	Laptop (over 6 months)
Contingencies (15%)	Covers cloud resource needs & development delays

Table 5: Cost Categories (Own Work)

8.2 Cost Estimates

The following table shows a summary of Cost Estimates:

Category	Estimated Cost (€)
Staff Costs	€ 12834.25
Software and Data Provider	€ 49.50
Amortization	€ 183.20
Total Before Contingency	€ 13,066.95
Contingencies (15%)	€ 1,960.04
Total Cost	**€ 15,026.99**

Table 6: Cost Estimates (Own Work)

Staff Costs: The main expense in this project is the time spent on development, testing, and documentation. To allocate personnel costs for each task, we first define the salary of each role involved in the project. The salary estimates have been obtained from GlassDoor, providing average salaries for similar roles in the project's development area [14]. I have calculated the hourly wages for each role based on their base annual salaries and a full-time schedule (40 hours per week, 52 weeks per year). I have calculated the cost per hour for each role, including social security contributions, using a multiplier of 1.35. The gross hourly salary and the actual cost per hour (including social security contributions) are as follows:

Role	Gross Hourly Salary (€)	Cost per Hour (€)
Project Manager	21.20	28.62
Software Architect	26.44	35.69
UX Designer	15.87	21.42
Developer	12.02	16.23
Tester	11.54	15.58

Table 7: Hourly Wages and Cost per Role (Own Work)

I have assigned roles to each task and using the cost per hour values of each role, we can determine the total cost of each task.

Code	Time (hours)	Role	Cost per Hour	Total Cost
PM1	20	Project Manager	28.62	572.4
PM2	20	Project Manager	28.62	572.4
PM3	15	Project Manager	28.62	429.3
PM4	30	Project Manager	28.62	858.6
PM5	15	Project Manager	28.62	429.3
PM6	15	Project Manager	28.62	429.3
PM7	25	Software Architect	35.694	892.35
PM8	10	Project Manager	28.62	286.2
PM9	90	Project Manager	28.62	2575.8
PM10	30	Project Manager	28.62	858.6
BE1	10	Developer	16.227	162.27
BE2	20	Developer	16.227	324.54
BE3	25	Developer	16.227	405.675
BE4	25	Developer	16.227	405.675
BE5	25	Developer	16.227	405.675
BE6	25	Developer	16.227	405.675
BE7	35	Developer	16.227	567.945
FE1	10	UX Designer	21.4245	214.245
FE2	25	UX Designer	21.4245	535.6125
FE3	35	Developer	16.227	567.945
TD1	20	Tester	15.579	311.58
TD2	20	Tester	15.579	311.58
TD4	20	Tester	15.579	311.58
Total	**565**			**12,834.25**

Table 8: Total Project Staff Costs by Task (Own Work)

Software and Data Provider: The data extraction from SkyServer will be done using free API access, ensuring there are no costs associated with retrieving SQL logs.

For storage, the project will use free services such as GitHub and Google Cloud, eliminating any expenses related to storing logs or project documentation.

However, I will be using Microsoft 365 Personal for documentation. This subscription costs €99 per year, but since the project lasts 6 months, I will only account for half of the annual cost. This means the Microsoft 365 expense for this project is €49.50.

Amortization: During the development of this project, I am using my personal laptop, which I purchased for €916. Since this is a long-term asset, its cost is distributed over its expected useful life. Based on standard depreciation methods, I estimate the total useful life of the laptop to be 2.5 years.

To determine the portion of this cost allocated to the project, I calculated the depreciation over 6 months. The laptop's value is distributed evenly over 2.5 years, meaning its annual depreciation is €366.40 per year. Since the project lasts 6 months, I account for only half of the yearly depreciation, resulting in an amortization cost of €183.20.

Contingencies: A 15% contingency fund is included in the budget to cover unexpected technical challenges and development delays. This provides flexibility in case of unforeseen technical issues requiring additional software tools and delays in development, which could extend resource usage beyond the planned timeline.

If the project approaches its budget limit, adjustments will be made by prioritizing essential tasks and postponing non-critical features if needed. This contingency plan ensures that the project remains financially controlled and can adapt to any unexpected circumstances.

8.3 Management Control

To ensure efficient use of resources and prevent overspending, the project will follow a structured budget management approach. This involves tracking expenses, monitoring financial risks, and adjusting when necessary to stay within budget.

The budget will be reviewed monthly, comparing planned and actual expenses. If spending deviates from estimates, corrective actions will be taken to avoid unnecessary costs while ensuring the project remains on track.

To measure financial efficiency, the following Key Performance Indicators (KPIs) will be tracked:

Budget Variance (%): This tracks the difference between actual spending and the planned budget. If significant deviations occur, corrective actions will be taken.

$$Budget\ Variance = \frac{\text{Actual Cost} - \text{Planned Cost}}{Planned\ Cost} x100$$

Burn Rate (€): This measures the rate at which project funds are being used. A higher-than-expected burn rate may indicate a need for cost-saving measures.

$$\text{Burn Rate} = \frac{\text{Total Expenses}}{Project\ Duration}$$

To prevent cost overruns, the 15% contingency fund will provide flexibility for potential challenges. If expenses approach budget limits, the project scope may be optimized to focus on core functionalities while postponing non-essential features.

By implementing regular budget reviews, financial tracking, and contingency planning, the project will remain financially controlled and efficient, ensuring that all expenses are justified and well-managed.

9. Sustainability Analysis

Sustainability is important to ensure that the project is efficient, cost-effective, and beneficial in the long run. This report looks at sustainability in three areas: environmental, economic, and social. The analysis follows the Sustainability Matrix from the Sustainability Report Guidelines (2018) [15] and considers the impact during development, future use, and possible risks.

9.1. Environmental Impact

This project is developed and tested locally and will not be deployed, meaning it does not contribute to the ongoing energy consumption of a cloud-based system. During development, cloud services are used only for storage. While cloud storage services generally have environmental costs due to energy and water consumption, companies like Google are actively working to mitigate these effects. Google Cloud has been carbon-neutral since 2007 and, since 2017, has matched 100% of its global electricity consumption with renewable energy [16].

The main environmental factor is electricity consumption. A personal computer typically uses around 0.2 kWh per hour, depending on its power consumption and usage patterns [17], meaning the total energy use for this project is minimal compared to larger-scale cloud-based systems

There are no significant environmental risks. By reducing unnecessary data processing and manual processes, this project enhances computational efficiency, leading to lower CPU and memory usage which minimizes energy consumption. Over time, these improvements can lead to significant reductions in operational costs and resource consumption and therefore, a positive environmental impact.

9.2 Economic Impact

This project has been designed to be cost-efficient, covering primary costs such as staffing, software and data provider expenses, infrastructure amortization, and contingencies.

Since the project is developed using free cloud services such as Google Cloud and GitHub, there are no infrastructure costs except for the personal laptop with an amortization cost of €183.20 justified previously. Using free and open-source tools makes this project more affordable economically.

To ensure financial stability, a contingency fund has been included in case additional software or development time is needed. The total cost remains manageable, and no ongoing costs are expected after development.

In conclusion, the long-term financial impact is minimal, as the system relies on free storage and software for continued use, ensuring that maintenance costs remain low.

9.3 Social Impact

This project aims to help researchers and data analysts by making SQL workload analysis easier and faster. By automating the extraction and processing of data, it saves time and reduces manual work, improving productivity.

The tool is designed to be easy to use, with clear documentation and a user-friendly interface, making it accessible even to those with little technical experience.

Since the project does not store or process personal data, there are no data privacy concerns. Users will only interact with SQL workload logs, ensuring that all information remains neutral and non-sensitive.

Overall, this project is expected to have a positive social impact by improving efficiency in data analysis and helping researchers work more effectively.

10. Tools and Technologies

This section outlines the software tools, frameworks, libraries, and interfaces used during the development of the system. Each component was selected based on familiarity and suitability for the project's goals. A full environment setup guide, dependency list, command-line manual and GUI Manual are provided in the README file on the GitHub repository and in the appendix of this document.

10.1 Development Environment

The main development was carried out using Visual Studio Code (VS Code). Its integrated terminal, extension ecosystem (Python, Git, etc.), and performance made it a practical choice. VS Code offered a unified workspace that allowed seamless switching between Python scripts and React components, streamlining the development workflow. PyCharm was evaluated for its deep integration with Python tools, but it was discarded due to heavier resource usage and more complex setup.

Version control was managed using Git, with GitHub as the remote repository. GitHub was selected for its ease of use, built-in collaboration features, and strong integration with project management tools. GitLab was explored and considered as an alternative for its integrated CI/CD support but was not chosen mainly due to developer's familiarity with GitHub [9].

10.2 Backend Technologies

For the backend of this project, Python was used along with a modern web framework and several essential libraries to handle API communication, data processing, and SQL parsing. This section describes the main technologies and configuration files used to build the backend.

FastAPI was chosen as the core web framework due to its high performance and suitability for building RESTful APIs that handle data-intensive tasks [18]. Although FastAPI supports asynchronous endpoints, this project uses primarily synchronous workflows, including the use of the `requests` library [19] to interact with the SDSS SkyServer.

FastAPI also provides automatic generation of interactive API documentation via Swagger and ReDoc [20], and uses Pydantic for type validation, significantly reducing runtime bugs and improving development speed.

Feature	FastAPI	Flask	Django
Performance	Very High (async)	High	Moderate
Async Support	Yes	Requires extra setup	No
Auto Docs	Yes (Swagger & ReDoc)	No	No
Type Validation	Yes (with Pydantic)	No	No
Best For	APIs, microservices, fast response apps	Simple web apps	Full-stack web apps

Table 9: Comparison of Python Web Frameworks for Backend Development (Own Work)

The backend is organized into modular components under a clear directory hierarchy to ensure maintainability, scalability, and separation of concerns. The main entry point is `main.py`, which initializes the backend logic or triggers the full pipeline. The `main_api.py` file launches the FastAPI server and configures the routes defined in `api/workload.py`. Core functionalities such as data fetching, cleaning, SQL parsing, and query grouping are implemented as separate modules inside the `modules/` folder. All dependencies are listed in the `requirements.txt` file, and setup instructions are included in the user manual and repository. The primary libraries used are listed below.

Library	Purpose
`fastapi`	Main web framework used to define and serve the REST API.
`uvicorn`	ASGI server used to run the FastAPI application in development and production environments.
`requests`	Handles HTTP requests to external services, such as the SDSS SkyServer SQL Weblog interface.
`pandas`	Processes and analyses tabular data retrieved from workload logs.
`sqlparse`	Parses and extracts structural components from raw SQL statements.

Table 10: Primary Backend Libraries Used (Own Work)

10.3 Frontend Technologies

The frontend of the project was built using React [21], a widely adopted JavaScript library for user interface development. React was selected due to the prior experience, its component-based architecture, and its strong ecosystem. Compared to alternatives like Angular or Vue, React offered the most balanced trade-off between simplicity, flexibility, and long-term maintainability.

To build and run the project, Vite [22] was used instead of Create React App (CRA), a toolchain traditionally used to bootstrap React applications. Vite offers faster development speeds, better performance, and easier configuration. Moreover, the code is organized into folders such as `components`/ and `pages`/ following standard frontend project structure conventions. This organization helps maintain clean, modular code and makes the project easier to navigate and scale.

The frontend was built using a modern and lightweight technology stack focused on maintainability, performance, and ease of development:

1. **TypeScript** [23] was integrated to add static typing. This helps prevent bugs by catching type-related errors during development and makes the codebase more understandable and maintainable, especially as the project grows.
2. **React Router** [24] was implemented to manage navigation between different pages of the application, such as the homepage and dashboard. It enables smooth transitions without reloading the entire page, improving user experience.
3. **CSS** was used for styling, giving full control over layout and design. Libraries like Tailwind CSS or Chakra UI were avoided to maintain flexibility and reduce dependency overhead.
4. **Axios** [25] was used to enable communication between the frontend and backend. It sends HTTP requests to the FastAPI backend once the API endpoints.

Overall, the frontend stack was designed to be simple, maintainable, and suitable for continuous expansion.

10.4 Command-Line Interface (CLI)

Alongside the web interface, a command-line interface (CLI) was developed to provide users with flexible and scriptable access to the system's core functionality. Through the CLI, users can perform all the key tasks such as workload extraction and grouping directly from the terminal. This allows for greater automation and integration into custom workflows.

The CLI is implemented in Python using the `argparse` module and is integrated into the `main.py` file, which serves as the application's central entry point. From there, the appropriate backend modules are invoked depending on the selected command (e.g., `fetch`, `parse`, or `group`). These scripts can be executed independently or combined depending on user needs.

For detailed instructions, command usage, and examples, please refer to the user manual in the attached Appendices and the GitHub repository.

10.5 API Documentation

FastAPI automatically generates two live API documentation views:

1. Swagger UI (/`docs`) for interactive endpoint testing
2. ReDoc: (/`redoc`) for structured API reference

These tools were extensively used during development to test requests and ensure backend consistency.

11. Backend Development

The backend workflow for this project follows a multi-stage pipeline:

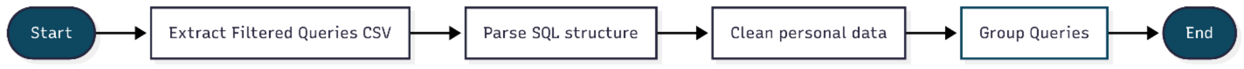


Figure 5: Backend Pipeline for SQL Workload Extraction and Analysis (Own Work)

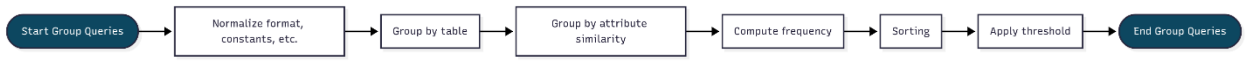


Figure 6: Detailed Breakdown of Query Grouping Stages (Own Work)

11.1 Workload Extraction and Filtering

The first and most critical step in the pipeline involves extracting SQL workload data from the Sloan Digital Sky Survey (SDSS). Several approaches were evaluated, each with different degrees of automation, access privileges, and technical constraints. The goal was to automate the retrieval of SQL query logs for parsing and analysis.

11.1.1 Workload Extraction Methods

As part of the project, access was obtained to both the SciServer and CasJobs platforms. Communication was established with the system administrators, who provided support and relevant API documentation. Four main methods for accessing SQL query logs were evaluated. The current system uses the SkyServer `x_sql.asp` endpoint, while the remaining approaches are outlined below as potential alternatives for future development or extension.

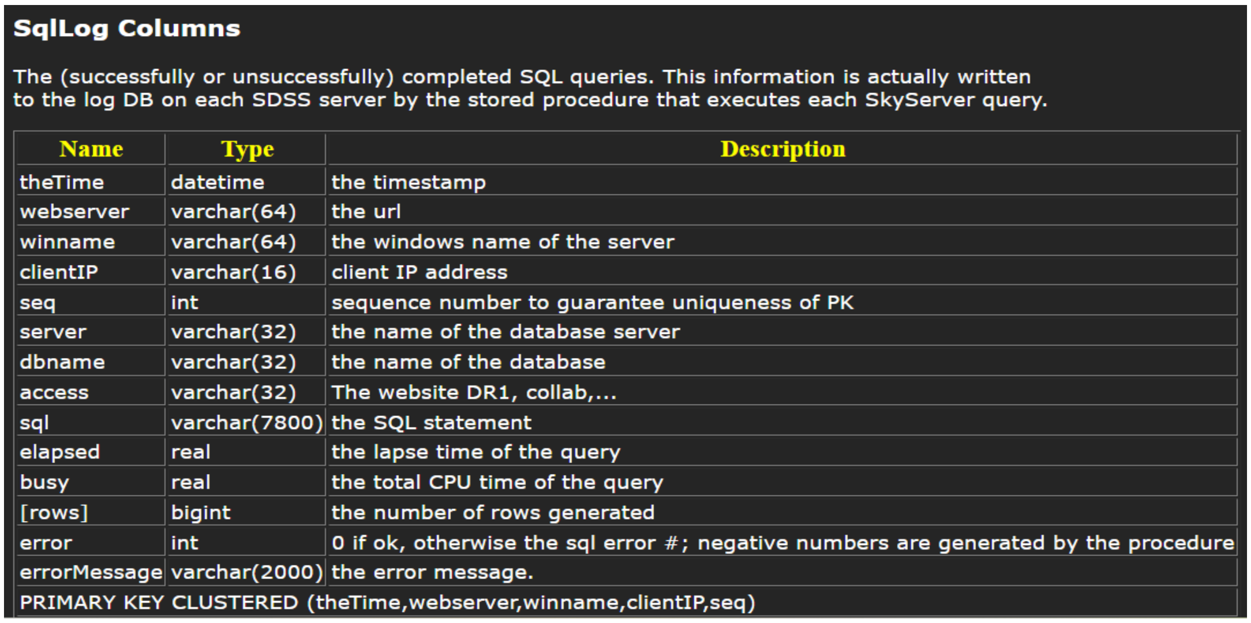
Current Implementation: SkyServer SQL Weblog Interface

On the SkyServer SQL Search platform [27], two primary log tables are exposed for analysis: Weblog and SqlLog. Each serves a distinct role in capturing SkyServer usage data:

Weblog: This table stores general HTTP request records such as client IP, requested URLs, and request types (e.g., GET, POST). It is primarily used for traffic and access analysis of the SkyServer website.

SqlLog: This table, on the other hand, captures detailed information about all SQL queries executed through the SkyServer platform. It includes the query text, execution time, CPU time, number of rows returned, and a timestamp. For this project, the fields `statement` (query text), `theTime` (execution timestamp), `elapsed` (execution duration), and `rows` (result size) are of particular importance. These fields are used during parsing, filtering, and analysis.

It is important to note that while the official SkyServer documentation refers to the SQL log column as `sql`, in practice the correct column name is `statement`. Attempting to query the table using the field name `sql` results in an "Invalid column name" error, indicating a discrepancy between documentation and implementation.


SqlLog Columns

The (successfully or unsuccessfully) completed SQL queries. This information is actually written to the log DB on each SDSS server by the stored procedure that executes each SkyServer query.

Name	Type	Description
theTime	datetime	the timestamp
webserver	varchar(64)	the url
winname	varchar(64)	the windows name of the server
clientIP	varchar(16)	client IP address
seq	int	sequence number to guarantee uniqueness of PK
server	varchar(32)	the name of the database server
dbname	varchar(32)	the name of the database
access	varchar(32)	The website DR1, collab,...
sql	varchar(7800)	the SQL statement
elapsed	real	the lapse time of the query
busy	real	the total CPU time of the query
[rows]	bigint	the number of rows generated
error	int	0 if ok, otherwise the sql error #; negative numbers are generated by the procedure
errorMessage	varchar(2000)	the error message.
PRIMARY KEY CLUSTERED (theTime,webserver,winname,clientIP,seq)		

Figure 7: SqlLog Table Schema (Sloan Digital Sky Surve [19])

In terms of implementation, it is essential to understand the distinction between the two web endpoints used to access this data:

sql.asp: This is the interactive HTML page where users manually enter SQL queries and submit them via a form. It is designed for human interaction. Sending a `POST` request to this page returns an HTML page, not structured data.

x_sql.asp: This is the underlying endpoint that processes SQL queries submitted via the form. When accessed programmatically, it accepts POST requests with parameters such as the SQL command and the desired output format (CSV, XML, HTML). This endpoint is used to retrieve structured query results, making it suitable for automation.

Therefore, all programmatic interactions in this project use `x_sql.asp` as the target URL to submit and retrieve query results in CSV format for further processing and parsing.

This method requires no authentication, and it is publicly accessible, making it easy to automate using Python's `requests` library.

Alternative Methods

Although the current solution is functional, several alternative methods remain under consideration for future development.

Option 1: SciServer Compute (Jupyter-Based Environment)

SciServer provides a Jupyter notebook interface within a cloud-based environment where users can submit SQL jobs to CasJobs. This method allows access to the `SdssWeblogs` context and supports writing query results to a personal storage area (`MyDB`). Data can then be exported manually. This approach is semi-automated and suitable for exploratory data access, but it requires manual export steps and cannot be integrated directly into the backend of this system.

Option 2: CasJobs Web Interface

CasJobs also allows users to submit SQL jobs through a web interface. Queries can be written using the SQL editor, executed in the `SdssWeblogs` context, and saved into `MyDB`. From there, users can manually download the output files for further analysis.

While this method is intuitive and provides complete access to the required logs, it is not suitable for backend automation because it requires manual user interaction through the browser.

Option3: CasJobs REST API (Preferred Long-Term Solution)

The CasJobs REST API offers the most complete and professional solution for automated data access. It allows authenticated users to submit SQL queries, check job statuses, and download results programmatically. The API supports all contexts, including `SdssWeblogs`, and enables integration with Python scripts via HTTP requests.

However, this method requires an access token that is only issued to users with permission to access the relevant context. At the time of writing, I have submitted a request to be added to the `SdssWeblogs` access group. Once approved, I intend to transition the system from using `x_sql.asp` to the CasJobs REST API.

A request, along with appropriate headers including the authentication token, can be sent to the API endpoint to fetch query logs in a structured format. The CasJobs API supports asynchronous jobs, meaning that once submitted, the system must check job status and retrieve the output once it is complete.

Comparative Overview of Available Methods

The following table provides a comparison of the different data access methods discussed:

Option	Automation	Data Access	Authentication	Comments
x_sql.asp Endpoint	Full	Error for some dates	No	Fast prototyping
SciServer Compute	Manual	Full	Yes	Reliable semi-automation
CasJobs Web Interface	Manual	Full	Yes	Interactive and manual
CasJobs REST API	Full	Full	Yes (Token needed)	Fully integrated backend

Table 11: Data Extraction Methods Comparison (Own Work)

In conclusion, the system currently uses the SkyServer `x_sql.asp` endpoint as the primary method for retrieving SQL logs.

11.1.2 Query Filtering

To ensure the quality, relevance, and consistency of the extracted SQL workload, a filtering step is implemented. During the early stages of implementation, this step was implemented as a standalone module executed after extraction. However, to improve performance and reduce the amount of I/O, this logic was later integrated directly into the SQL queries executed during extraction.

This approach discards irrelevant or noisy queries early on, making the dataset lighter and more representative before it reaches the parsing stage.

The following filtering criteria are now embedded directly in the SQL WHERE clause when retrieving logs from the SkyServer `SqlLog` table:

Successful execution: Only queries with `error = 0` are retained, ensuring that failed queries (due to syntax issues, timeouts, or runtime errors) are excluded.

Non-zero execution time: Queries with `elapsed <= 0` are discarded, as they typically indicate logging anomalies or system artifacts.

Non-empty results: Queries with `rows <= 0` are excluded, since they did not return data and are less useful for workload analysis.

These conditions are applied during the request to the `x_sql.asp` endpoint, making the filtering step efficient and transparent. As a result, the system avoids writing or processing unnecessary data, and the output is immediately ready for parsing and structural analysis.

11.1.3 Workload Retrieval Time Benchmarking

To provide users with an estimate of how long it will take to retrieve SQL workload data from the Sloan Digital Sky Survey (SDSS) SkyServer SQL Weblog, a performance benchmarking procedure was carried out.

Methodology

The benchmark was implemented using the `fetch` module developed as part of this project. To assess system *behaviour* under different workloads, queries were executed for increasing values of `TOP N`. Each query size was repeated three times to account for variability due to network conditions and backend load. The average execution time for each case was used in the analysis.

Three distinct years were selected:

1. December 2003: Early-stage system logs.
2. December 2013: Mid-life data reflecting a more mature platform.
3. December 2019: Considered for analysis but excluded from model training due to irregular performance.

While 2019 and 2023 were initially included in the analysis, significant performance anomalies were observed. In the case of 2023, most fetching attempts failed due to repeated timeouts, which are likely due to server-side limitations, or temporary unavailability. 2019 exhibited counterintuitive behaviour. The retrieval times did not follow a consistent pattern with the increase in data size. For example, smaller queries such as `TOP 10` took longer than larger ones.

Such behaviour may result from backend caching effects, queuing delays, or connection overhead dominating execution time for smaller requests. These irregularities disqualified 2019 and 2023 from inclusion in the regression model, as they could distort the fit and reduce generalizability.

Total query count

Before conducting the tests, the total number of available logs for December in each selected year was obtained using the `count` module. This contextualizes the workload volume and illustrates the increase in system usage over time.

Year	Month	Total Rows
2003	December	50,333
2013	December	1,488,670
2019	December	1,905,418
2023	December	3,623,390

Table 12: SQL Log Volume (Own Work)

Raw Timing Results (in seconds)

Each query size was executed three times to smooth out transient fluctuations. The following tables report the raw results.

TOP N Logs	Run 1	Run 2	Run 3
10	2.21	1.77	2.96
100	4.28	4.95	4.28
1,000	6.33	3.78	6.33
10,000	7.97	4.46	4.94
50,000	169.59	165.99	156.84

Table 13: Raw Timing Results December 2003 in seconds (Own Work)

Note 1: An early outlier (1.32s for 10,000 logs) was excluded due to probable caching effects. A more representative fourth run (7.07s) was used instead.

Note 2: The results for 50,000 rows show signs of being outliers and were excluded from the regression model to improve the accuracy and linearity of the fitted model.

TOP N Logs	Run 1	Run 2	Run 3
10	1.29	1.93	0.67
100	1.84	4.2	3.89
1,000	1.01	4.53	1.9
10,000	4.3	6.17	0.75
50,000	28.24	21.11	11.9
100,000	46.56	23.48	22.94
1,000,000	332.63	331.35	285.14

Table 14: Raw Timing Results December 2013 in seconds (Own Work)

TOP N Logs	Run 1	Run 2	Run 3
10	69.03	23.89	58.88
100	23.9	60.22	23.8
1000	23.8	29.02	24.23
10000	5.91	4.15	4.43
100000	46.73	38.48	23.9
1000000	285.68	263.71	306.49

Table 15: Raw Timing Results 2019 in seconds (Own Work)

Note: December 2019 demonstrates high variance and nonlinear scaling.

The mean values for 2003 and 2013 (used in the model), and their combined averages are as follows:

TOP N Logs	2003 Average	2013 Average	Combined Average
10	2.31	1.29	1.805
100	4.50	3.31	3.90
1,000	5.48	2.48	3.98
10,000	5.49	5.17	5.33
50,000	Excluded	24.2	24.2
100,000	-	22.94	22.94
1,000,000	-	308.24	308.24

Table 16: Average Retrieval Time in seconds (Own Work)

Estimated Wait Time Model

Based on the average timing results, a linear regression model was constructed to estimate query execution time as a function of the number of returned rows. The model takes the form:

$$T(\mathrm{n}) = a.n + b$$

Where:

1. **T(n)** is the estimated time (in seconds)
2. **n** is the number of rows returned
3. **a** (slope) and **b** (intercept) are coefficients fitted to the data. a is the rate at which time increases per additional row. If a = 0.000015, it means each extra row adds 0.000015 seconds to the total. b is the base time the request takes even when asking for 0 rows.

To build this model, the `scikit-learn` library was used to fit a linear regression line over the combined average query times from December 2003 and 2013. These years were selected due to their consistent performance across query sizes, minimizing noise introduced by backend variability such as caching or request limits. To improve readability given the exponential increase in data size, the x-axis of the model visualization was plotted on a logarithmic scale. To complement the linear model, the x-axis of the visualization was plotted on a logarithmic scale. The resulting log-log plot reveals a near-linear trend.

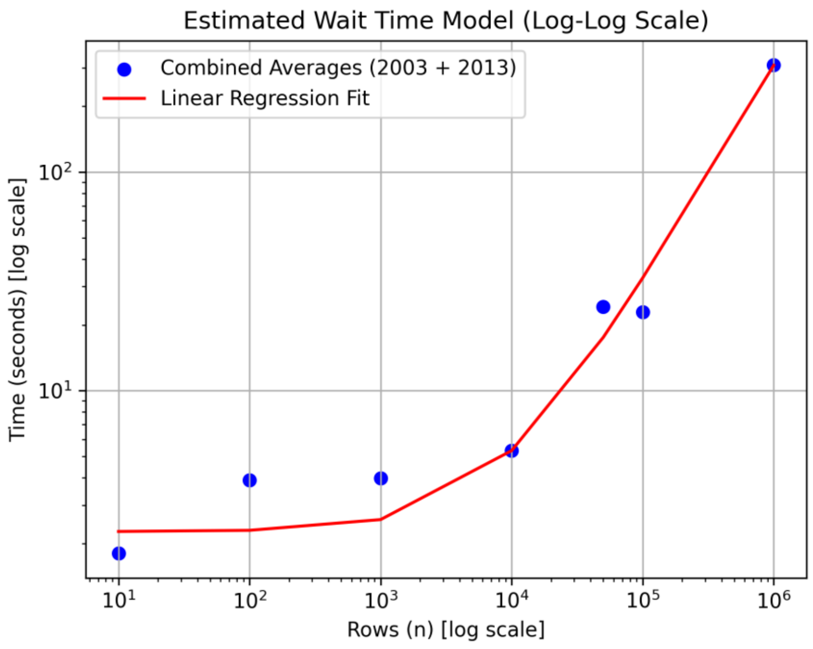


Figure 8: Linear regression model fitted to average retrieval times from 2003 & 2013 using a logarithmic x-axis (Own Work)

The resulting function is:

$$T(\mathrm{n}) = 0.00030533 * n + 2.27$$

This model achieves an R^2 of 0.9981, indicating an excellent fit. It demonstrates a strong linear correlation between the number of rows and the response time, which can be used to estimate user waiting time for future queries.

This model captures both the baseline latency and the expected growth in retrieval time as the number of rows increases. Minor deviations exist likely due to internal optimizations or system state. However, the linear approximation is sufficiently accurate for estimating user-facing wait times and improves transparency in the data download interface.

11.1.4 Progress and Estimated Wait Time Visualization

To inform the user about the progress, the Python library `tqdm` has been utilized. This library displays progress bars in loops, making it especially useful when processing large datasets, downloading files, or performing any task that takes a noticeable amount of time.

To use this library effectively, it is generally necessary to know the total target. However, in this case, it is not possible to determine the number of queries extracted since they are all extracted at once. Therefore, the library relies on time estimates to show visual progress.

11.2 SQL Parsing and Metadata Extraction

Parsing is an important step to filter and understand SQL queries before any further processing can be done. The SkyServer weblog provides raw SQL queries in CSV format, which cannot be directly analysed without breaking them into meaningful components. Therefore, a parsing function is implemented to automatically extract relevant structural features from each query.

SQL parsing refers to the process of extracting relevant information from it, such as which tables are involved, which columns are selected, whether the query includes a filter (WHERE clause), joins, aggregation functions, and more. This structured information is crucial for further analysis, classification, and workload filtering.

Instead of treating each SQL statement as plain text, the parser analyses and identifies:

1. The columns selected in the query (project)
2. The tables involved (pattern)
3. The WHERE clause, normalized to a parameterized format (filter)
4. Whether the query uses DISTINCT or TOP
5. Whether it contains ORDER BY, GROUP BY, or nested subqueries

To achieve this, the `sqlparse` and `re` libraries were used, along with other libraries and regular expressions, to parse and extract structured metadata from each SQL statement. The function first detects and removes optional constructs like TOP N and DISTINCT (while recording their presence). It then tokenizes the SQL statement to separate the SELECT, FROM, and WHERE blocks, from which it extracts the relevant fields. The final output is a clean, minimal representation of the query's structure.

Library	Purpose
`sqlparse`	Tokenize and structure SQL queries
`re`	Detect patterns like `TOP`, `DISTINCT`, and subqueries
`csv`	Read logs from CSV files
`json`	Output parsed query metadata
`argparse`	Handle command-line arguments for input/output

Table 17: Overview of the Libraries Used for SQL Parsing (Own Work)

11.2.1 Feature Extraction and Dictionary Format

The parser returns a dictionary of the following features for each query:

1. original_query: The raw SQL query string exactly as it appears in the log. It is extracted directly from the CSV column containing the SQL statement, named `statement`. Although this field is retained in the output for reference and

inspection, it is not used for grouping or pattern analysis, as small variations in formatting or literal values could incorrectly split structurally similar queries.

2. pattern (previously tables): A list of all actual tables referenced in the SQL query. Table names are extracted from the FROM clause, ignoring aliases or fields that appear to be column references (e.g., strings with a dot like `x.up_id`). Aliases are not included, and entries such as function calls or subqueries (e.g., `fGetNearbyObjEq(...)`) are filtered out unless they behave like table expressions. This list is used as a grouping key.
3. project (previously select_columns): A list of all columns specified in the `SELECT` clause. It is extracted by locating the `SELECT` block and parsing identifiers until the FROM keyword, using `sqlparse` or custom logic to parse `IdentifierList` elements. During intra-group analysis, this field is used to identify query shapes. To account for slight variations, a configurable threshold based on the Jaccard similarity ($|A \cap B| / |A \cup B|$) between column sets is applied to decide whether two queries should be considered part of the same pattern group.
4. has_distinct: Indicates whether the query uses `SELECT DISTINCT`. It is extracted by checking for the presence of the DISTINCT keyword within the SELECT clause, typically using a case-insensitive check or token type analysis. This feature is also used in grouping, as queries with and without DISTINCT are treated as structurally different.
5. top_value: The number specified in a `TOP N` clause, or None if the clause is not used. It is extracted using a regular expression or by scanning tokens after SELECT for the keyword TOP and its numeric value. Queries with different TOP N values are grouped separately.
6. filter (previously where_clause): The text of the `WHERE` clause, if present. It is extracted by identifying the WHERE token and capturing the condition expression until the next clause or end of the query, using token iteration or slicing.
7. has_order_by: A Boolean indicating whether the query contains an ORDER BY clause. This feature is extracted by searching for the presence of the ORDER BY keyword in the raw SQL query using regular expressions. Initially, the full clause was extracted as a raw string (order_by_clause), but this was later simplified to a Boolean flag. This change was made because the specific columns involved in ordering are generally already part of the SELECT clause, and for grouping purposes, only the presence of sorting (not its details) affects performance and cost analysis.
8. has_nested_queries: Boolean indicating whether the query contains a subquery. It is extracted using regular expressions or by analysing the token structure for nested SELECT statements within parentheses. For grouping purposes, queries with

subqueries are treated similarly to those using JOINs, as subqueries also imply multi-relation access patterns.

9. has_group_by: Boolean indicating whether the SQL query includes a GROUP BY clause. This feature helps identify queries with additional computation due to grouping, which can impact execution cost and workload characteristics. It is used to distinguish between simple and aggregating queries, even though the specific grouped columns are not extracted. Initially a list or name of aggregation functions used in the query, such as COUNT, SUM, AVG was extracted. This `aggregation` feature was later simplified to `has_group_by`.

ID	Key	Description
1	`original_query`	The raw SQL query string
2	`pattern`	List of all tables referenced including aliases (originally named tables)
3	`project`	List of selected columns in the SELECT clause (originally `select_columns`)
4	`has_distinct`	Boolean indicating the presence of the DISTINCT keyword
5	`top_value`	The integer value in a TOP clause, if present
6	`filter`	The normalized WHERE clause with constants replaced by "?"(originally `where_clause`)
7	`has_order_by`	Boolean indicating presence of an ORDER BY clause
8	`has_nested_queries`	Boolean indicating presence of subqueries
9	`has_group_by`	Boolean indicating presence of a GROUP BY clause

Table 18: Summary of parsed SQL query features extracted (Own Work)

During early iterations, additional features such as `join_details`, `join_columns`, `query_length`, `is_select_all`, `is_selection`, `is_projection`, `join_count`, `table_count`, `aggregation`, `order_by_clause` and `theTime` were extracted. However, these were later excluded from the final output, as they were either redundant, not directly useful for grouping or analysis, or available through other metadata. Their removal helped simplify the parser and reduce unnecessary processing overhead.

Moreover, in earlier versions of the system, the keys `tables`, `select_columns`, and `where_clause` were renamed to `pattern`, `project`, and `filter` respectively. This renaming was initially only performed in the grouping stage. This led to inconsistency across modules. To resolve this, the renaming was moved directly into the parsing stage to ensure consistency and clarity throughout all later phases.

11.2.2 Special Cases and Improvements

Several challenges were identified during testing that required refining the parsing logic to better capture edge cases and improve output consistency.

*Replacing Empty Projections with [*] for Implicit SELECT * Queries*

When no columns are explicitly detected in the SELECT clause, the projection is initially recorded as an empty list ([]). To better reflect the actual intent of the query (i.e., selecting all columns), this value is now replaced with ["*"].

Removing Column Aliases (AS)

To improve the precision of extracted query features, the parser was extended to strip column aliases defined using the AS keyword. In SQL, aliases are often used to rename expressions in the SELECT clause, such as SELECT a.name AS username. However, for workload analysis and pattern identification, the actual column or expression (a.name) is more relevant than its alias (username). To address this, I introduced a check for the presence of AS using a case-insensitive regular expression and then split the string to retain only the left-hand side of the alias. This ensures that the column extraction focused on actual database columns or expressions. For example, `sp.psfmag_i` AS `mag_i` now correctly yields `sp.psfmag_i` as the extracted column, discarding the alias `mag_i`.

Splitting Algebraic Expressions into Individual Columns

The second improvement targeted SQL expressions in the SELECT clause that involve arithmetic operations such as addition, subtraction, multiplication, or division. Queries like `SELECT a.price + a.tax AS total` were previously stored as a single string, which made it difficult to analyse which columns were involved in computations. I implemented a regular expression-based split on operators (+, -, *, /) and filtered out constants, whitespace, and parentheses. Only valid identifiers such as `a.price` or `a.tax` were retained and added individually to the `select_columns` list. This approach made the parser sensitive to multi-column expressions and provided a more granular view of the data usage in each query. For instance, `sp.psfmag_i - sp.extinction_i AS mag_i` now results in both `sp.psfmag_i` and `sp.extinction_i` being recorded separately.

Replacing Table Aliases with Full Table Names

Lastly, I addressed the issue of table aliases masking actual table names in query columns. In SQL, it is common to write queries using aliases for readability, such as `FROM Spec Photo AS sp`, and then refer to columns like `sp.psfmag_i`. To recover the full semantic meaning, I added functionality to map each alias to its corresponding full table name during the parsing of the FROM clause. When processing SELECT columns, this mapping was used to substitute any alias-based column references with their fully qualified form.

For example, `sp.psfmag_i` was transformed into `SpecPhoto_psfmag_i`. This replacement is essential for grouping and analysing workload patterns by table, as it removes ambiguity introduced by aliases and standardizes the representation of accessed columns across queries.

11.2.3 Query Normalization and Cleaning

Normalizing `WHERE` Clauses and Improvements

In the early stages of the project, it was observed that many SQL queries in the workload were very similar, differing only in the values used in their WHERE clause. For example: `WHERE objID = 12345` and `WHERE objID = 67890`. Even though the values are different, the structure of the query is the same. However, without handling this, they would be treated as two completely different queries.

To address this, a normalization step is implemented that replaces all constant values in the WHERE clause, like numbers or strings, with a placeholder `?`. This is a common convention in parameterized SQL queries. So, both above examples become "`WHERE objID = ?`". This normalization allows us to group queries based on their structure rather than their specific values, which is much more useful for workload analysis.

Additionally, the WHERE keyword itself was removed from the normalized clause. Keeping the keyword in the output field (e.g., `"filter": "WHERE objID = ? AND z < ?"`), when it is already implied by the field name, was redundant.

As a result of the previous steps, the same example `SELECT * FROM PhotoPrimary WHERE objID = 12345 AND z < 0.3` is stored as `"filter": " objID = ? AND z < ?"`.

During later testing stages, it was observed that some queries produced filters with unnecessary newline characters or inconsistent spacing, such as `"zooS.p_cw > ?\n"` or `"z BETWEEN ? AND ?\n"`. These inconsistencies could negatively impact analysis and grouping. As a result, the normalization logic was updated to automatically strip all newline characters (\n) and collapse them into single spaces while I keep them in the `original_query`. This improved the quality and consistency of the extracted filters.

Finally, during testing, it was observed that some queries did not include a WHERE clause at all, which caused unnecessary empty "filter" fields in the output JSON. To fix this, the 'filter' field is only included when a valid condition is present. This refinement improves the consistency and usability of the output. In conclusion, this makes the output cleaner and directly usable for the next stages of automated grouping and analysis.

Handling Trailing Artifacts and Query Corruption

During testing, some queries included trailing noise such as `"filter": "zooS.p_cw > ? ? ? ?"`. This issue was traced back to parsing artifacts caused by extra numeric values at the end of certain CSV rows (e.g., confidence scores or timestamps) that were not part of the original SQL query. These values polluted the parsed output and distorted both the `filter` and `original_query` fields.

To resolve this, a trimming step was implemented to remove any trailing numeric tokens. These include floats and scientific notation values which are unlikely to be part of valid SQL syntax. This step ensures that only the intended SQL statement is retained when reconstructing the query string. Additionally, I applied a post-processing normalization step using a regular expression to eliminate trailing sequences of placeholder tokens (e.g., `? ? ? ?`) from the `filter` field. This refinement helps clean malformed `WHERE` clauses and improves the accuracy of structural analysis.

Overall, this approach not only corrects the `filter` field but also ensures that the `original_query` reflects a clean and valid SQL command, free from extraneous or corrupted content at the end.

Improved Table Detection (pattern)

During the final testing phase, it was observed that the pattern field was incorrectly including column references instead of only table names. For example, an entry such as `x.up_id` appeared in the pattern, even though `x.up_id` is a column reference, not a table. Additionally, table aliases such as x were mistakenly preserved, rather than being resolved to their original table names, such as `#upload`. As a result, some entries in the pattern field looked like this: `["x.up_id", "#upload", "photoPrimary"]`. This is inaccurate because `x.up_id` is a column reference and should not be included.

To address this, the parsing logic was refined to ensure that only valid table names are extracted. Column references, aliases, function calls, and other non-table expressions are now excluded from the pattern field. This correction improves the accuracy of table-level grouping and ensures the semantic integrity of the parsed query structure.

11.2.4 Output Structure and Example

To illustrate how the parser works, consider the following SQL query extracted from the SkyServer SQL logs:

CSV Input:

```
theTime,statement
12/31/2003 7:16:19 PM,"SELECT  top 1   p.objID, p.run, p.rerun,
p.camcol, p.field, p.obj,
   p.type, p.ra, p.dec, p.u,p.g,p.r,p.i,p.z,
   p.Err_u, p.Err_g, p.Err_r,p.Err_i,p.Err_z
   FROM PhotoPrimary p
   WHERE p.objID = 12345 AND z < 0.3 0.5 5E-3 100"
```

Initial Approach Output:

```
{
  "original_query": "SELECT top 1 p.objID, p.run, p.rerun,
p.camcol, p.field, p.obj, p.type, p.ra, p.dec, p.u, p.g, p.r,
p.i, p.z, p.Err_u, p.Err_g, p.Err_r, p.Err_i, p.Err_z FROM
PhotoPrimary p WHERE p.objID = 12345 AND z < 0.3 0.5 5E-3 100",
  "tables": ["fGetNearbyObjEq(180,-0.5,3) n", "PhotoPrimary
p"],
  "select_columns": ["p.objID", "p.run", "p.rerun", "p.camcol",
"p.field", "p.obj", "p.type", "p.ra", "p.dec", "p.u", "p.g",
"p.r", "p.i", "p.z", "p.Err_u", "p.Err_g", "p.Err_r",
"p.Err_i", "p.Err_z"],
  "has_distinct": false,
  "top_value": "1",
  "join_details": [],
  "join_columns": [],
  "where_clause": "WHERE n.objID=p.objID",
  "order_by_clause": "",
  "is_selection": true,
  "is_projection": true,
  "is_select_all": false,
  "has_nested_queries": false,
  "aggregation": null,
  "query_length": 235,
  "join_count": 0,
  "table_count": 2,
  "theTime": "12/31/2003 7:16:19 PM"
}
```

Final Approach Parsed Output Before Cleaning and Normalization:

```
{
  "original_query": "SELECT top 1 p.objID, p.run, p.rerun,
p.camcol, p.field, p.obj, p.type, p.ra, p.dec, p.u, p.g, p.r,
p.i, p.z, p.Err_u, p.Err_g, p.Err_r, p.Err_i, p.Err_z FROM
PhotoPrimary p WHERE p.objID = 12345 AND z < 0.3 0.5 5E-3 100",
  "pattern": ["PhotoPrimary p"],
  "project": ["p.objID", "p.run", "p.rerun", "p.camcol",
"p.field", "p.obj", "p.type", "p.ra", "p.dec", "p.u", "p.g",
"p.r", "p.i", "p.z", "p.Err_u", "p.Err_g", "p.Err_r",
"p.Err_i", "p.Err_z"],
  "has_distinct": false,
  "top_value": "1",
  "filter": "p.objID = ? AND z < ? ? ? ?",
  "has_order_by": false,
  "has_nested_queries": false,
  "has_group_by": false
}
```

Final Parsed Output After Cleaning and Normalization:

```
{
  "original_query": "SELECT p.objID, p.run, p.rerun, p.camcol,
p.field, p.obj, p.type, p.ra, p.dec, p.u, p.g, p.r, p.i, p.z,
p.Err_u, p.Err_g, p.Err_r, p.Err_i, p.Err_z FROM PhotoPrimary p
WHERE p.objID = 12345 AND z < 0.3",
  "pattern": ["PhotoPrimary"],
  "project": ["p.objID", "p.run", "p.rerun", "p.camcol",
"p.field", "p.obj", "p.type", "p.ra", "p.dec", "p.u", "p.g",
"p.r", "p.i", "p.z", "p.Err_u", "p.Err_g", "p.Err_r",
"p.Err_i", "p.Err_z"],
  "has_distinct": false,
  "top_value": "1",
  "filter": "p.objID = ? AND z < ?",
  "has_order_by": false,
  "has_nested_queries": false,
  "has_group_by": false
}
```

11.3 Query Cleaning

The cleaning stage is applied after parsing and focuses on excluding SQL statements that interact with private, user-specific databases such as MyDB. These queries typically represent personal data manipulation rather than interactions with the public SDSS schema and are therefore not representative of general system usage patterns.

To ensure that the cleaning step does not discard useful public query patterns, a selective filtering strategy is used:

1. Queries that reference MyDB in the FROM, JOIN, or UPDATE clauses are discarded, as they are reading or modifying data in private contexts.
2. However, queries that include SELECT INTO MyDB are retained, as they typically retrieve public data and store the result in a personal workspace. This behaviour still

reflects interaction with the public schema and is therefore considered relevant for workload analysis.

The filtering is implemented using case-insensitive regular expressions that detect references to MyDB in specific SQL clauses. The script loads parsed queries from a structured JSON file, inspects each query's original SQL string, and discards only those that match the defined exclusion patterns. A separate check is used to ensure that queries with SELECT INTO MyDB are explicitly preserved.

As a result, the cleaned dataset contains only queries that interact with the public SDSS schema.

11.4 Query Grouping

To identify common workload patterns, SQL queries must be grouped based on their structural similarities. This section describes the multi-phase grouping implemented to extract meaningful patterns from the logs. The objective is to reduce variability and noise in the data, while retaining enough structure to derive insights into typical query behaviour.

11.4.1 Grouping by Used Tables

The first stage of grouping is based on the set of tables accessed by each query. Following the cleaning phase, the tool extracts table names from the `FROM` clause of each query. Each query may use one or more tables. To simplify analysis, the grouping algorithm extracts these table names, normalizes them to lowercase, and removes any aliases (e.g., `PhotoPrimary AS p` becomes `photoprimary`).

While grouping the queries, I came across queries that contain function calls instead of table names, such as `fGetNearbyObjEq(...)`. These queries are excluded from grouping, as they are not standard table references. Additionally, they will all have unique names like `fGetNearbyObjEq (180, -0.5,3)`. Including these queries would create meaningless or repetitive groups.

Initially, queries were grouped individually by each table. For example, if a query accessed both `PhotoObjAll` and `SpecObjAll`, it was placed in two separate groups. However, this approach was replaced with a set-based grouping strategy, where queries are grouped based on the combination of all tables they access together. For example, both of the following queries:

1. `SELECT * FROM PhotoObjAll JOIN SpecObjAll`
2. `SELECT spec.ra, photo.u FROM SpecObjAll AS spec JOIN PhotoObjAll AS photo`

These queries are grouped under the key (`'photoobjall'`,`'specobjall'`).

The grouping key is constructed by applying the following transformations:

1. `set ()` is used to eliminate any duplicate table references.
2. `sorted ()` ensures the group is order-insensitive and the same tables in any order belong to the same group.
3. `tuple ()` makes the result hashable, so it can be used as a dictionary key.

The final grouping key looks like this in code:

```
tuple (sorted (set (t. lower (). split () [0] for t in
query["tables"] if "(" not in t)))
```

The grouped queries are stored using Python's `defaultdict`(list) structure, which allows for efficient and clean accumulation without the need to check key existence. The resulting dictionary maps each normalized tuple of table names (optionally including modifiers) to a list of queries that access that specific combination.

Although this structure is useful for inspection and analysis, it can become memory-intensive when handling large datasets. Therefore, the implementation allows for this data to be optionally persisted depending on the context.

11.4.2 Grouping by Query Shape

Once queries are grouped by their accessed tables, the next step is to classify them by query shape, which refers to the set of columns selected in the `SELECT` clause. Two queries are considered to have the same shape if they project the same columns, regardless of the order in which those columns appear.

To normalize query shapes, the column list of each query is transformed using three steps:

1. `set` () removes duplicate columns (e.g., ra and ra).
2. `sorted` () orders the columns alphabetically to ensure consistency.
3. `tuple` () converts the list into a hashable structure for dictionary storage.

As a result, for example the sets ["A", "B", "C"], ["C", "A", "B"], and ["B", "C", "A"] are all treated as the same query shape: ("A", "B", "C").

This process facilitates the identification of common access patterns across semantically equivalent queries.

11.4.3 Merging Similar Shapes

After distinct shapes have been identified, the algorithm attempts to merge those that are not identical but share a high degree of similarity in their selected columns. For example, a shape like ["A", "B"] may be considered like ["A", "B", "C"] if the overlap between them is significant.

To determine whether the two shapes are similar enough to be merged, the Jaccard similarity coefficient is computed between their column sets. This index is defined as:

$$Jaccard(A, B) = \frac{A \cap B}{A \ \cup B}$$

Where:

1. $\boldsymbol{A} \cap \boldsymbol{B}$ is the number of elements common to both sets A and B (the intersection).
2. $\boldsymbol{A} \ \cup \boldsymbol{B}$ is the total number of unique elements across both sets A and B (the union).

The Jaccard index returns a value between 0 and 1. The value 1 means the sets are identical and 0 means they have no elements in common. Only query shapes with a similarity score above a configurable threshold (e.g., 0.8) are merged. In other words, If the similarity is above a threshold (e.g., 0.8), the shapes are merged, and their frequencies are summed. This merging step reduces fragmentation in the grouping process and helps reveal more generalizable and representative workload patterns.

11.4.4 Group by Query Clauses

As an additional grouping criterion, I have included 4 modifiers: DISTINCT, TOP, ORDER BY, and GROUP BY. The SDSS catalogue data is stored in Microsoft SQL Server [27]. Microsoft SQL Server processes queries using an execution plan sensitive to query modifiers. Therefore, the presence of certain SQL clauses such as DISTINCT, TOP, ORDER BY, and GROUP BY can significantly affect the performance characteristics and computational cost of a query.

Each one of these elements introduces different performance implications. For instance, a query using DISTINCT instructs the database engine to remove duplicate rows from the result set. This operation often increases processing time and memory usage. Similarly, ORDER BY requires the result set to be sorted, which can be computationally expensive, especially when working with large datasets that lack indexing support. The GROUP BY clause triggers aggregation, which not only modifies the shape of the result set but also adds to the overall computational load. Meanwhile, TOP limits the number of rows returned, often reducing runtime but potentially hiding the broader access patterns of the underlying query logic.

To capture this impact, these modifiers were explicitly parsed during the SQL parsing stage and integrated as optional flags in the grouping phase.

By including these flags in the grouping process, the tool can capture more nuanced distinctions between queries that otherwise share identical structures in terms of selected columns and accessed tables. For example, two queries selecting the same columns from the same table may behave quite differently in terms of execution cost if one uses DISTINCT and the other does not.

These modifiers are detected during the SQL parsing phase and added as optional flags in the grouping key. Their inclusion allows the tool to distinguish between structurally similar queries that have different runtime implications. For example, the following two queries project the same columns from the same table, but one applies `DISTINCT`. If the grouping includes modifiers, they will be placed in separate groups.

1. `SELECT ra, dec FROM PhotoObjAll`
2. `SELECT DISTINCT ra, dec FROM PhotoObjAll`

The decision to make these flags optional supports analytical flexibility. Fine-grained analyses may enable them, while more general summarizations may omit them for broader pattern identification.

11.4.5 Calculating Frequency

For each group of queries, the number of occurrences is counted. This frequency reflects the relative importance of each group in the overall workload and serves as a filter for prioritizing high-impact patterns. Patterns that occur only once or very rarely are considered less informative for optimization purposes.

11.4.6 Sorting

To facilitate analysis and reporting, the resulting groups are sorted in descending order of frequency. This ensures that the most frequently occurring query structures are presented first in the output files and visualizations.

11.4.7 Applying Frequency Threshold

To reduce noise and focus on statistically meaningful patterns, a frequency threshold is applied. Only those column shapes that appear in at least a configurable fraction of queries (e.g., 0.5%) are retained. This ensures that extremely rare or one-off queries do not skew the results or introduce irrelevant complexity.

By applying this filter, the output reflects only the most representative query structures for each table group.

11.4.8 Final Output

Each query group is annotated with a unique `group_id`, assigned sequentially. This identifier simplifies reference, logging, and downstream processing.

The filter field, which previously included the full `WHERE` clause with its keyword, is now cleaned to remove the "`WHERE`" prefix. This enables more consistent parsing and easier comparison of filter logic. For example, the field filter": "`WHERE ABS (s.ra -?) < ? AND ABS (s.z -?) <?`" will become "`filter": "ABS (s.ra -?) < ? AND ABS (s.z -?) < ?`".

The pattern field, which records the accessed tables, has been modified to explicitly exclude function calls and expressions, focusing exclusively on relational tables.

The final output consists of two files:

1. Grouped Queries File which contains a list of representative queries per group, with the following schema:

```
{
        "queries": [
                {
                  "group_id": 0,
                  "frequency": 5,
                  "original_query": "...",
                  "pattern": [...],
                  "project": [...],
                  "filter": "...",
                  "has_distinct": true/false,
                  "top_value": int or null,
                  "has_order_by": true/false,
                  "has_nested_queries": true/false,
                  "has_group_by": true/false
                },
            ...
  ]
}
```

2. Frequencies file captures the frequency distribution of column selection shapes within each group. The key is the combination of tables and modifiers, and the value is a mapping of column combinations to their respective counts. For example:

```
"specobjall, False, False, False, None": {
"s.ra, s.dec, s.z": 5
}
```

This indicates that five queries in that group projected the exact combination of s.ra, s.dec, and s.z.

These outputs support multiple use cases, including workload summarization, indexing recommendations, and synthetic benchmark construction.

11.5 Automation Pipeline Implementation

To fully automate the process of workload analysis described in the previous sections, a unified script has been developed that encapsulates all required steps into a single command-line interface (CLI). This script allows users to execute the entire pipeline from raw log retrieval to query grouping and frequency analysis, either by supplying a previously downloaded SQL log file or by dynamically fetching logs from the SDSS SkyServer SQL Weblog interface.

The pipeline supports two modes of operation. In the first, the manual mode, the user provides the name of a local CSV file that has already been downloaded from SDSS. The script then proceeds to parse, clean, group, and analyse this file without requiring an internet connection. In the second mode, referred to as fetch mode, the script automatically retrieves query logs from the server for a specified year and month, and optionally for a specific day. These logs are downloaded in CSV format, stored locally, and processed using the same sequence of steps as in manual mode.

11.6 API Design and Endpoint Exposure

To facilitate communication between the frontend and the backend logic, a RESTful API was developed using FastAPI, a modern Python web framework optimized for performance and automatic documentation generation.

The API exposes multiple endpoints that correspond to each of the major steps in the data pipeline, including:

1. `/fetch`: to retrieve raw SQL logs from SDSS based on year, month, and optionally day and limit.
2. `/count`: to count available logs.
3. `/parse`: to parse CSV logs into structured JSON queries.
4. `/clean`: to normalize and filter queries based on criteria.
5. `/group`: to group cleaned queries by target table and extract attribute frequencies.
6. `/pipeline`: to execute the full workflow in sequence with a single request.

Each endpoint was implemented in a modular and reusable way. The routes are defined in workload.py and registered under the /api prefix using `include_router` mechanism.

Additionally, a main_api.py file was used as the main entry point of the application. It configures CORS to allow frontend requests, mounts the data folder to serve generated files, and links the router.

This API allowed the frontend to initiate, monitor, and preview every stage of the pipeline directly via HTTP POST requests. The endpoints were designed to accept both file uploads and parameterized forms to support different usage modes (manual or automated).

11.7 Challenges and Solutions

Data Normalization and Regex

One of the challenges during backend development was the extraction and normalization of meaningful features from raw SQL queries. The raw logs included queries of varying complexity, errors and inconsistent formatting.

To address this, regular expressions (regex) were used to complement the SQL parsing process. Despite having limited prior experience with regex, I progressively incorporated it into the development workflow to identify specific SQL patterns such as the presence of WHERE clauses, use of TOP N, or DISTINCT keywords, and to normalize various expressions.

Data Extraction

Another challenge during the backend development emerged early in the Project which was the unpredictable behaviour when fetching SQL workload logs from the SDSS SkyServer. Although the logs are publicly accessible, the platform frequently exhibited unstable responses. Some requests timed out or failed without explanation. Specific late years such as 2023 consistently failed to return data, likely due to server-side limitations or temporary unavailability. Also, inconsistent delays were observed even when benchmarking. These issues made the initial benchmarking process especially difficult and blocked early analysis.

To overcome this, while I continued working on other parts of the pipeline using manually extracted data. I performed benchmarking on different days, progressively fetching more logs. This iterative and patient approach ensured that data collection progressed in the background without blocking the development of other core components.

12. Frontend Development

This section describes the user interface of the SQL Workload Analyzer, which was developed using React and TypeScript. The frontend is designed to provide an intuitive, modular interface that allows users to run individual pipeline stages or execute the full analysis flow in a single interaction. Its objective is to make SQL workload analysis accessible without requiring users to interact with the command line.

The application uses React Router for navigation and consists of several dedicated views: fetch, parse, clean, group, results, and a one-click pipeline runner. Each page presents a clear layout with form inputs, descriptive labels, and a submit button. The layout is styled using plain CSS (defined in App.css), and navigation highlights the active route for clarity. Files are uploaded to the backend via `POST` requests and stored persistently. Later stages reference these filenames to process data without the need to re-upload.

Note: For detailed usage instructions and interface visuals, refer to Appendix B.

12.1 Architecture and Structure

The frontend follows a modular architecture with a clear folder structure. Page-level components are stored in the `pages/` directory, with each file representing one phase of the pipeline (e.g., `FetchPage.tsx`, `GroupPage.tsx`). Reusable elements such as forms (e.g., `FetchForm.tsx`, `UploadForm.tsx`) are located under `components/`.

Routing is defined in `App.tsx`, where all views are mapped to specific paths using React Router. Navigation elements are implemented using NavLink. NavLink is a React Router component that highlights the active route and ensures consistent styling based on the user's current location. Table 18 summarizes the main routes of the application and the purpose of each view.

Route	Description
/	Home page with an overview of the system and the workload pipeline steps
/fetch	Retrieve SQL logs from SDSS by specifying year, month, and limit
/parse	Parse an existing file from the server or upload a new one
/clean	Clean parsed data by removing queries with personal database references
/group	Group cleaned queries and calculate frequency distributions
/Results	View the latest results from processing steps (no file download)
/pipeline	Run all steps in one go from a single input file

Table 19: Frontend Routing Summary (Own Work)

The entry point of the app is defined in `main.tsx`, where the BrowserRouter is initialized. This ensures consistent layout wrapping and global styling across all pages. Styling is handled in a single `App.css` file, which controls visual hierarchy, spacing, hover effects, and navbar design. The use of a single CSS file allows for easy maintenance and a unified aesthetic.

All interactions with the backend are performed via HTTP requests to a FastAPI server. Each button or form in the UI is mapped to a corresponding API route, and the resulting files (e.g., parsed queries or grouped JSON data) are previewed directly or downloaded by the user.

12.2 Component Behaviour and Backend Integration

Each page collects specific user input and sends it to a corresponding backend API. All requests are made using the Axios. Uploaded files are transmitted to the backend using `POST` requests with `multipart/form-data`.

The current implementation does not support downloading the files to the local computer. However, file generated or are stored in the `/backend/data` folder. As mentioned previously it is possible to reuse these files by referencing to their names across stages without requiring repeated uploads.

All forms include minimal validation (e.g., required fields, allowed file types) and show confirmation messages upon success or error.

12.3 Challenges and Solutions

One important debugging challenge during development was unexpected data loss when running the parse operation from the frontend. Although previously fetched CSV files were successfully saved to disk, calling the `/parse` endpoint would result in the file being silently overwritten with empty content.

This issue originated from the assumption in the backend that the request always included a newly uploaded file (`UploadFile = File(...)`). When the frontend only sent a filename referring to an existing file on the server, FastAPI still treated it as a file upload and created an empty file with the same name, overwriting the original.

To solve this, I modified the backend to accept either a file or a filename. It now checks which one is present and avoids overwriting files unless a real upload is provided. This debugging process helped prevent silent data loss and improved input flexibility in other steps like /`clean` and `/group`.

Other parts of the frontend development proceeded smoothly, thanks to prior experience with React and a clearly defined scope. The intentionally minimal interface helped keep the implementation focused, lightweight, and easy to maintain.

12.4 Frontend Limitations and Future Improvements

The current frontend does not support downloading files directly to the user’s computer, which limits its usability in offline or post-processing scenarios. Additionally, there is no persistent file preview before submission (e.g., showing a few rows of the CSV). Implementing a reusable file preview component could help reduce user error. Moreover, all filenames must be typed manually in some steps; adding an auto-complete or dropdown file selector would reduce friction and improve usability. Finally, deploying the frontend alongside the backend and serving both under the same origin would allow enabling browser file downloads, simplify integration, and make the tool accessible without manual setup.

13. System Testing and Validation

The objective of this project was to develop a tool capable of automating and simplifying the SQL workload analysis pipeline. Given the academic context and time constraints of a Final Degree Project, the testing approach was focused on demonstrating core functionality, identifying potential edge cases, and validating the logic and output at every key stage of the pipeline.

To validate the behaviour of the pipeline, a set of real-world SQL workload logs was fetched from the SDSS SkyServer. These logs were processed through the main modules with intermediate outputs examined and reviewed. This manual validation process also served to illustrate the modular architecture of the tool.

Furthermore, a showcase test was conducted using logs from several different days to evaluate the consistency of grouping behaviour across varying workloads. These results are documented in Appendix C, not only to support the validation process, but also to demonstrate the potential of the tool in use.

13.1 Backend Testing

Unit Testing of Components

Each core backend module was evaluated independently to confirm that it functioned as expected and produced reliable outputs:

Count and Fetch modules: These modules were tested using different inputs. The outputs and error handling when the server response is delayed or truncated were verified.

Parsing module: `parse.py` was tested to ensure that the original SQL query was preserved and that the fields pattern, project, and filter were extracted and structured consistently. Tests verified that:

1. The `original_query` string remained unchanged.
2. Fully qualified column and table names were captured in pattern and project.
3. filter was only included when a valid WHERE clause was detected.
4. Placeholder artifacts (e.g., `? ? ?)` were avoided.
5. Invalid or unresolved aliases did not pollute output fields.

Some advanced queries involving alias reuse, implicit joins, or derived subqueries could not be parsed perfectly within the scope of this delivery. These have been documented for future iterations and parser refinement.

Cleaning module: `clean.py` was tested to ensure that user-specific queries, those referencing `MyDB` schemas were excluded. The logic successfully detected and removed the queries as expected:

1. Ensured that remaining queries had no residual `MyDB` references.
2. Logged a clear summary of how many queries were retained.

Grouping module: `group.py` was tested using cleaned data to confirm that grouping logic worked as intended:

1. Queries with shared patterns were grouped correctly.
2. Grouping filters (e.g., --`distinct`, --`groupby`) functioned as expected.
3. The resulting JSON included meaningful group labels and clean structure.

Small Dataset Test Runs

Each module was tested on known subsets of data (e.g., top 100 queries from a log) to simplify debugging. Outputs were visually inspected to confirm correct behaviour and edge case handling. Some of these tests required correction and modifications of the module being tested or the previous modules in the pipeline. All these modifications are documented in the corresponding section to the module.

Edge Case Checks

Special attention was given to complex SQL statements, for example queries lacking `WHERE` clauses, subqueries and nested `SELECT` blocks and function calls in `FROM` clauses. I have created special SQL files to including these queries and repeated tests of each module and manually inspected the output of each module.

Performance Considerations

Runtime logs were recorded to measure parsing and grouping performance on larger datasets. Summary metrics (e.g., number of discarded or grouped queries) were also logged to assess scalability.

13.2 Frontend Testing

The frontend of this system was validated through an iterative process that combined manual testing, simulated user flows, and mock backend interactions. Due to the academic and temporal constraints of a final degree project, formal unit or end-to-end testing frameworks (e.g., Jest, Cypress) were not employed. Instead, emphasis was placed on functional validation, visual consistency, and overall user experience across the application's key components.

Each frontend page was tested in isolation during development. Initially, mock JSON responses were used to emulate backend behaviour, allowing for the verification of frontend logic before actual API integration. For instance, the Fetch page utilized mocked server replies for year, month, and day inputs to test form behaviour and loading states. Once the /fetch endpoint was implemented, the form was re-tested using live backend data to ensure full compatibility.

Additionally, error messages for invalid or missing input, and confirmation modals upon successful submission, were implemented to ensure usability, prevent silent failures, and give users clear feedback at each step.

Finally, integration testing of the frontend with the backend was conducted using the "Run All Steps" functionality. These tests confirmed correct data flow across components, proper error handling in edge cases (e.g., malformed files or server unavailability), and the successful rendering of analysis results. Overall, this multi-layered testing approach ensured that the frontend met functional expectations and provided a smooth user experience despite the limited testing framework.

13.3 Integration Testing

End-to-end functionality was validated using the full pipeline, from frontend interaction to backend processing and file generation.

End-to-End Flow

1. A CSV file was uploaded via the frontend interface.
2. The backend successfully received and parsed the file, producing intermediate JSONs.
3. Cleaned and grouped query data was saved to disk.
4. The frontend displayed appropriate success/error messages based on backend response.

API Contract Validation

The input and output formats of the backend were tested against frontend expectations. JSON structures returned by the backend were checked to ensure key fields (e.g., queries, pattern, project, filter) were present and well-formed.

File Output Verification

Files produced by the backend were opened and inspected manually to ensure content alignment with parsing logic.

13.4 Manual QA

While unit and integration testing provided automation, manual validation was the main approach to catch subtle issues.

In the backend, a sample of parsed outputs was reviewed manually to ensure that SQL features were interpreted correctly. This included checking filter normalization, correct alias resolution, and table/column naming consistency.

In the frontend, all user flows were tested from beginning to end, from uploading files to navigating results and interpreting response content.

Finally, system behaviour was cross-referenced with what was described in the thesis. Adjustments were made in the documentation at the same time whenever a change was made and there were inconsistencies between implementation and documentation.

13.5 Iterative Refinement Through Supervisor Feedback

Throughout development, weekly meetings were held to review progress and receive feedback from the thesis supervisor and co-supervisor. Feedback sessions were used to identify errors in the outputs and discuss improvements. These sessions helped iteratively improve both the implementation and the overall structure of the thesis. These sessions also helped to follow the requirement and adjust the project to the need of the scientist. Several changes including alias resolution improvements and filter normalization tweaks were made directly in response to these discussions.

13.6 Limitations and Future Work

Although the tool successfully automates SQL workload extraction and grouping in most scenarios, some complex or less frequent cases remain challenging to handle. These limitations were identified during the final stages of testing and supervisor review, at a point where the core pipeline was already stable. They were documented and deferred for future iterations due to time constraints.

Parser Refinement for Complex SQL Constructs

During testing and supervisor review, certain advanced SQL patterns were found to require deeper analysis:

Alias Reuse and Implicit Joins: Queries that reuse aliases or include implicit joins occasionally lead to ambiguous column resolution. Future improvements should enhance alias scoping and parse join conditions explicitly to improve column disambiguation.

Function Calls in FROM Clauses: Handling function calls in FROM clauses (e.g., `fGetNearbyObjEq(...))` remains a limitation. The current parser does not distinguish them from standard tables, which may lead to misleading alias-based conditions. Future work should detect these patterns and treat them as derived objects, optionally replacing them with normalized placeholders such as `SpecPhoto_ObjID =?.`

Deployment and Enhanced Output and Usability

An additional improvement would be to allow users to download the fetched data and resulting grouped files directly from the interface. Currently, only file uploads are supported. While feasible, this feature depends on deployment or a more advanced setup where the backend can serve file downloads independently.

Moreover, to maximize accessibility, future work could include packaging the tool as a deployable web app, including frontend hosting and backend. This would eliminate the need to run both servers manually and support a smoother user experience. These tasks represent meaningful yet scoped extensions to the current work. Their completion would increase the tool's generalizability and usability.

14. Conclusions

14.1 General Summary and Reflections

This thesis presents the design and development of a modular tool for automating SQL workload extraction and analysis, using data from the Sloan Digital Sky Survey (SDSS) SkyServer. The main goal was to replace manual and methods with a reliable and reusable pipeline that can process real SQL logs, group similar queries, and highlight common access patterns to help optimize the database.

Once the query logs are available, either by uploading a CSV or by fetching directly from SDSS, the data flows through a multi-stage pipeline that includes parsing, cleaning, grouping, and frequency analysis. The heart of the tool lies in how it groups similar queries: first by the tables accessed, then by the structure of the projected columns, and optionally by SQL modifiers such as DISTINCT, ORDER BY, GROUP BY, and TOP. This approach allows for flexible and meaningful classification of queries, helping to highlight repeated access patterns and typical workload shapes.

To make the results more useful, the tool filters out low-frequency patterns and merges similar query shapes using the Jaccard similarity index. This reduces noise and produces a compact yet informative summary of the workload. The final outputs include representative queries per group and frequency statistics for projected column combinations. These outputs can be used to support indexing decisions, query optimization, and synthetic benchmark creation.

Careful attention was also given to the software architecture. Each processing step is handled by its own backend module and the full pipeline can be executed with a single command or API call. At the same time, users can run individual steps independently when needed. The system can be operated through a command-line interface (CLI) or a minimal frontend built with reusable React components. These design choices ensure that the tool is not only functional but also extensible, maintainable, and user-friendly. It can serve as a foundation for future work.

Finally, use case study is included in Appendix C to show the practical utility of the tool for workload analysis. It demonstrates full pipeline execution across three real samples. A short comparison between modifier-based and default grouping shows how the system adapts to query diversity and highlights consistent patterns and how it can be used.

This project taught me that working with real systems requires flexibility. It is essential to adapt to unexpected server behaviours and communicate trade-offs clearly. Parsing and designing meaningful grouping strategies, and tuning thresholds showed me the complexity behind even simple looking data tasks. Weekly meetings with my supervisor and co-supervisor played a crucial role in shaping the technical direction, identifying bugs, and improving the structure of the final documentation. Their guidance helped me refine my communication and present technical processes in a clear, professional, and academic way.

In summary, this thesis delivers a working system for analysing SQL workloads in a structured and automated way. It contributes both a practical tool and a clear methodology for understanding and summarizing complex query logs, helping to extract meaningful insight from raw data and supporting future research and optimization efforts.

14.1 Justification of Achieved Technical Competencies

This subsection details how the technical competencies proposed for this project have been developed and applied:

CES1.1: To develop, maintain, and evaluate complex and/or critical software systems and services.

Expected Level: A little bit

I developed a full-stack tool from scratch and worked with real-world SQL workloads retrieved from the SDSS SkyServer, which involved navigating incomplete documentation, inconsistent formats, and constraints of the server interface. I designed and implemented a fully automated pipeline to process and analyse these logs in multiple stages. This demonstrates that I achieved the expected basic level of competence in developing and evaluating software systems.

CES1.5: To specify, design, implement, and evaluate databases.
Expected Level: Enough
While I did not create or manage a live database instance, I worked extensively with real SQL queries. I implemented logic to parse these queries, extract structure and metadata, and analyse them in a modular way. My implementation supports understanding and optimizing database behaviour without requiring physical database deployment.

CES1.6: To administrate databases (CIS4.3).
Expected Level: A little bit
Although database administration was not a core focus, I implemented basic pre-processing steps such as filtering out noisy queries, cleaning logs, and handling malformed input. These steps improve the quality and manageability of the dataset before analysis. These actions improved the quality and manageability of the dataset, demonstrating basic

administration capabilities. This aligns with the expected basic level of database design and evaluation.

CES1.7: To control the quality and design tests in software production.
Expected Level: In depth
I tested each module of the tool individually to verify its correctness. I used real data samples and manual inspection to test pipeline robustness, edge cases, and logic correctness. Although I did not implement automated testing, the systematic validation ensures a reliable system. This shows partial achievement of the expected in-depth level due to the lack of automated testing.

CES2.1: To define and manage the requirements of a software system.
Expected Level: Enough
I defined the system objectives clearly based on academic research papers and the needs of researchers using SDSS. These requirements were then translated into concrete technical steps, showing strong skills in both functional and technical specification. I achieved and slightly exceeded the expected level in requirements definition and management. This shoed enough maturity and autonomy in gathering, analysing, and defining the requirements of a software system.

CES2.2: To design adequate solutions in one or more application domains, using software engineering methods that integrate ethical, social, legal, and economic aspects.
Expected Level: Enough
The tool was designed specifically for the SDSS SkyServer context, taking into account domain-specific constraints such as limited automated access, inconsistent log formatting, and a high diversity of SQL queries. These challenges were addressed through tailored architectural and adaptability. Furthermore, as the solution revolves around parsing, analysing, and grouping large volumes of SQL queries, it also directly engages with the database domain. This shows an appropriate application of domain-specific solution design, as expected.

CES3.2: To design and manage a data warehouse.
Expected Level: In depth
While I did not design a full data warehouse, I processed a high volume of SQL queries to extract and group patterns. These results can support future data warehouse modelling or optimization. This shows partial fulfilment of the expected level, by supporting rather than fully implementing a data warehouse.

14.2 Relationship Between the Project and the Courses in the Specialization

This project is strongly connected to what I have learned throughout my degree and specialization.

In the Databases (BD) and Database Design (DBD) subjects, I was introduced to important concepts such as normalization, relational algebra, and how to write and analyse SQL queries. These topics helped me understand the foundations of this project and how to break down and process real SQL logs.

In the Web Applications and Services (ASW) subject, I was introduced to frontend and backend development, and how APIs work. It was the first time I programmed a backend in Python and a frontend in React. Later, in the Software Engineering Project (PES) subject, I practiced backend development further using Python and created APIs. I also worked on full-stack projects using agile methodologies. During my internship, I continued to improve my skills in React.

Finally, in the Software Project Management (GPS) course, I learned how to plan and manage projects. I applied tools such as Gantt charts, estimated task durations, and tracked my progress. This made the planning aspect of my thesis more structured and professional. I was also able to apply this knowledge during the PES course.

Overall, I believe this project demonstrates how all the knowledge and skills I have gained during my degree come together. Every subject I have studied has contributed in some way, and solving the challenges in this project was only possible thanks to that foundation.

References

[1] E. Gallinucci, M. Golfarelli, W. Radwan, G. Zarate, and A. Abelló, “Impact study of NoSQL refactoring in SkyServer database,” in *Proc. 27th Int. Workshop on Design, Optimization, Languages, and Analytical Processing of Big Data*, 2025.

[2] SDSS Collaboration, *SDSS SkyServer Documentation*, 2025.

[3] G. Zarate, "Optimizing a real-life database based on workload evolution: A comparative analysis of SQL and NoSQL approaches," 2024.

[4] Microsoft, "SQL Server Profiler", *Microsoft Documentation*. [Online]. Available: https://learn.microsoft.com/en-us/sql/tools/sql-server-profiler/.

[5] Percona, "Percona Query Analytics", *Percona Documentation*. [Online]. Available: https://www.percona.com.

[6] sqlparse, "sqlparse: A non-validating SQL parser for Python," *GitHub Repository*, [Online]. Available: https://github.com/andialbrecht/sqlparse.

[7] Apache Calcite, "Apache Calcite: Query optimization framework," *Apache Software Foundation*, [Online]. Available: https://calcite.apache.org.

[8] Atlassian, "Kanban vs. Scrum: Which agile are you?" *Atlassian*. [Online]. Available: https://www.atlassian.com/agile/kanban/kanban-vs-scrum.

[9] GitHub, Inc., *GitHub: Development platform for version control and collaboration*, 2008. [Online]. Available: https://github.com.

[10] Trello, "Trello project management tool", *Atlassian*. [Online]. Available: https://trello.com.

[11] Mermaid Chart, "Mermaid Live Editor – Diagramming and visualization tool". [Online]. Available: https://www.mermaidchart.com
[12] Diagram Generator, "AI-based Diagram Generation Tool". [Online]. Available: https://diagram-generator.com

[13] Facultat d’Informàtica de Barcelona, *Normativa del treball de final de grau en Enginyeria Informàtica de la Facultat d’Informàtica de Barcelona*. [Online]. Available:https://www.fib.upc.edu/sites/fib/files/documents/estudis/normativa-tfe-fib-ca.pdf.

[14] GlassDoor, "Average Salaries for IT Roles," Available at: https://www.glassdoor.com

[15] Sustainability Report English 2018, Universitat Politècnica de Catalunya.

[16] Google Cloud, "Sharing carbon-free energy percentage for Google Cloud regions," *Google Cloud Blog*, Mar. 18, 2021. [Online]. Available: https://cloud.google.com/blog/topics/sustainability/sharing-carbon-free-energy-percentage-for-google-cloud-regions.

[17] BillsWiz, "How Much Electricity Does a Computer Use?" BillsWiz.com, 2024. [Online]. Available: https://billswiz.com/computers-electricity-use/.

[18] S. Ramírez, FastAPI Documentation. [Online]. Available: https://fastapi.tiangolo.com/

[19] Requests: HTTP for Humans. [Online]. Available: https://requests.readthedocs.io

[20] Swagger: OpenAPI Specification. [Online]. Available: https://swagger.io/specification/

[21] Meta, React. [Online]. https://react.dev

[22] Vite. [Online]. https://vitejs.dev

[23] Microsoft, TypeScript Documentation. [Online]. Available: https://www.typescriptlang.org

[24] Remix, React Router Documentation. [Online]. Available: https://reactrouter.com

[25] Axios, Promise-based HTTP client for the browser and node.js. [Online]. https://axios-http.com

[26] SDSS SkyServer, “SkyServer SQL Weblog Interface,” [Online]. Available: https://skyserver.sdss.org/log/en/traffic/sql.asp

[27] *SDSS SkyServer,* The SDSS Catalog Data Model, [Online]. Available: https://skyserver.sdss.org/dr18/Support/intro

Appendix A: Command-Line Interface (CLI) User Manual

This appendix provides instructions for using the workload automation pipeline via the command-line interface (CLI). Each module of the workload pipeline can be executed independently through `main.py`. You can also run the complete pipeline using `pipeline.py`.

A.1 Environment Setup

To use the command-line interface (CLI), ensure that the backend environment is properly configured. The following steps are prerequisites:

1. Clone the repository from GitHub:

https://github.com/rozhinaahmadii/workload-automation

2. Navigate to the `backend/` directory and activate the Python virtual environment:

3. Install dependencies

Note: For complete environment setup instructions, including how to create the virtual environment, refer to the README file.

A.2 Using CLI Modules (main.py)

Run individual modules with:

```
python main.py <command> [--flag1 value1] [--flag2 value2].
```

When running `main.py`, you can use different commands to specify the module following necessary flags to configure them and customize the behaviour.

Command	Description
`Count`	Get total number of logs
`Fetch`	Download query logs
`Parse`	Extract structure and metadata
`Clean`	Remove user-specific queries
`Group`	Identify groups

Table A1: Main.py Commands (Own Work)

The CLI includes built-in help for each command. You can access it by running:

```
python main.py <command> --help
```

Note: Replace <command> with the actual command name (like fetch, parse, group, etc.).

Example:

```
python main.py fetch --help
```

A.2.1 Count Logs

Get the number of successful SQL queries for a specific month or day.

Syntax:

```
python main.py count --year <YYYY> --month <MM> [--day <DD>]
```

Arguments:

- `--year` (Required): Year of logs (e.g. 2023)
- `--month` (Required): Month of logs (e.g. 12)
- `--day` (Optional): Day of logs (1–31). If omitted, counts all logs in the month.

Example:

```
python main.py count --year 2003 --month 12
```

A.2.2 Fetch SQL Logs

Download the top N SQL logs for a specific date.

Syntax:

```
python main.py fetch --year <YYYY> --month <MM> [--day <DD>] [--limit
<N>]
```

Arguments:

- `--year` (Required): Year to fetch logs from
- `--month` (Required): Month to fetch logs from
- `--day (Optional):` Day (1–31). If omitted, fetches logs for the entire month.
- `--limit` (Optional): Max number of logs to fetch. Defaults to all logs.

Examples:

```
python main.py fetch --year 2023 --month 12 --day 17 --limit 500

python main.py fetch --year 2023 --month 12
```

A.2.3 Parse SQL Logs

Extract metadata and features from raw SQL queries.

Syntax:

```
python main.py parse --input <input_file.csv> --output
<output_file.json>
```

Arguments:

- `--input` (Required): Path to the input CSV file with logs
- `--output` (Required): Output JSON file for structured results

Example:

```
python main.py parse --input data/filtered_2023_12.csv --output data/p
arsed_2023_12.json
```

A.2.4 Clean Queries

Remove queries that use `MyDB` (except for `SELECT INTO MyDB`) to focus on public workloads.

Arguments:

- `--input` (Required): Name of the parsed JSON file
- `--output` (Required): Name of the cleaned JSON file

Example:

```
python main.py clean --input data/parsed_2023_12.json --output data/cl
eaned_2023_12.json
```

A.2.5 Query Grouping

Group queries by table and optionally by SQL modifiers.

Syntax:

```
python main.py group --input <cleaned_file.json> --freq_output
<summary.json> [--output <grouped_file.json>] --threshold <min_freq> -
-jaccard <similarity> [--modifiers <mod1> <mod2> ...]
```

Arguments:

- `--input` (Required): Cleaned JSON file with queries
- `--freq_output` (Required): Path to save query shape frequency summary
- `--output` (Required only if `--store_groups` is used): Path to save grouped queries.
- `--threshold` (Required): Minimum frequency (e.g. 0.005 keeps patterns in ≥0.5% of logs)

- `--jaccard` (Required): Similarity threshold for merging column patterns (default: 0.8)
- `--modifiers` (Optional): List of modifiers to include in the grouping. Options: `distinct`, `groupby`, `orderby`, `top`
- `--store_groups` (Optional): If specified, the grouped queries will be saved to the file provided via `--output`.

Examples:

```
python main.py group --input data/cleaned.json --output data/grouped.j
son --freq_output data/frequencies.json --threshold 0.005 --store_grou
ps
```

```
python main.py group --input data/cleaned.json --output data/grouped.j
son --freq_output data/frequencies.json --threshold 0.005 --store_grou
ps
```

A.3 Run the Full Pipeline (pipeline.py)

Instead of running steps individually, you can run the full pipeline with a single command. You can either fetch the csv and proceed with all steps (Option 1) or use an existing CSV file previously fetched (Option 2).

A.3.1 Option 1: Fetch and Process

Syntax:

```
python pipeline.py --fetch --csv <csv_name.csv> --year <YYYY> --month
<MM> [--day <DD>] [--limit <N>] --modifiers <modifiers> --threshold
<min_freq> --jaccard <similarity>
```

Example:

```
python pipeline.py --fetch --csv raw.csv --year 2004 --month 3 --limit
5000 --modifiers distinct --threshold 0.01 --jaccard 0.75
```

A.3.2 Option 2: Use an Existing CSV

Syntax:

```
python pipeline.py --csv <existing.csv> --modifiers <modifiers> --thre
shold <min_freq> --jaccard <similarity>
```

Example:

```
python pipeline.py --csv fetched_2024_12_top10000.csv --modifiers grou
pby orderby --threshold 0.01 --jaccard 0.75
```

Appendix B: Graphical User Interface (GUI) User Manual

B.1 Purpose and Scope

This appendix provides a complete walkthrough of the browser-based interface of the SQL Workload Analyzer. The interface is designed to allow users to interact with all core functionalities of the system without using the command line.

This guide is intended for end users and explains, step by step, how to use the tool, what inputs are expected on each page, and what kind of output is generated. Screenshots are included throughout to support understanding of the interface.

B.2 Environment Setup

To run the GUI locally, ensure both the backend and frontend are set up:

1. Clone the repository:

   https://github.com/rozhinaahmadii/workload-automation

2. Set up the backend Python environment and start the server.
3. Install frontend dependencies using `npm install` and start the frontend server.

Refer to the project README file for full instructions.

B.3 Uploading Input Files

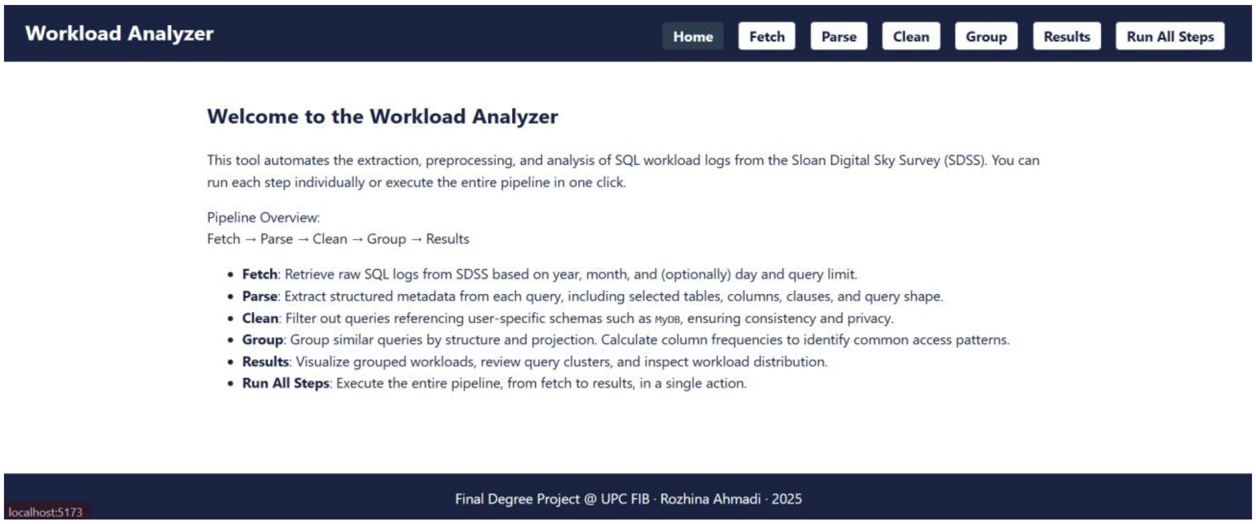
Throughout the application, users are often asked to provide an input file in `.csv` or `.json` format. The interface supports two methods of doing so:

1. Upload from local machine: Users can browse and select a file using a file input field. The selected file is immediately uploaded to the backend.
2. Use an existing file: If the file already exists on the backend server users can enter its filename directly into a text field.

This flexible input method is available on multiple pages including the parse, clean, group, and run-all steps. Output files are stored automatically in the `/data` directory in the backend. Direct download is not currently supported through the interface.

B.4 Home Page (`/`)

The home page serves as the starting point of the application. It presents an overview of the tool and introduces the sequence of pipeline steps available to the user. Each step is briefly described, and users are encouraged to navigate to the appropriate page using the top navigation bar. This page is purely informative.



Welcome to the Workload Analyzer

This tool automates the extraction, preprocessing, and analysis of SQL workload logs from the Sloan Digital Sky Survey (SDSS). You can run each step individually or execute the entire pipeline in one click.

Pipeline Overview:
Fetch → Parse → Clean → Group → Results

- **Fetch**: Retrieve raw SQL logs from SDSS based on year, month, and (optionally) day and query limit.
- **Parse**: Extract structured metadata from each query, including selected tables, columns, clauses, and query shape.
- **Clean**: Filter out queries referencing user-specific schemas such as `MyDB`, ensuring consistency and privacy.
- **Group**: Group similar queries by structure and projection. Calculate column frequencies to identify common access patterns.
- **Results**: Visualize grouped workloads, review query clusters, and inspect workload distribution.
- **Run All Steps**: Execute the entire pipeline, from fetch to results, in a single action.

Final Degree Project @ UPC FIB · Rozhina Ahmadi · 2025

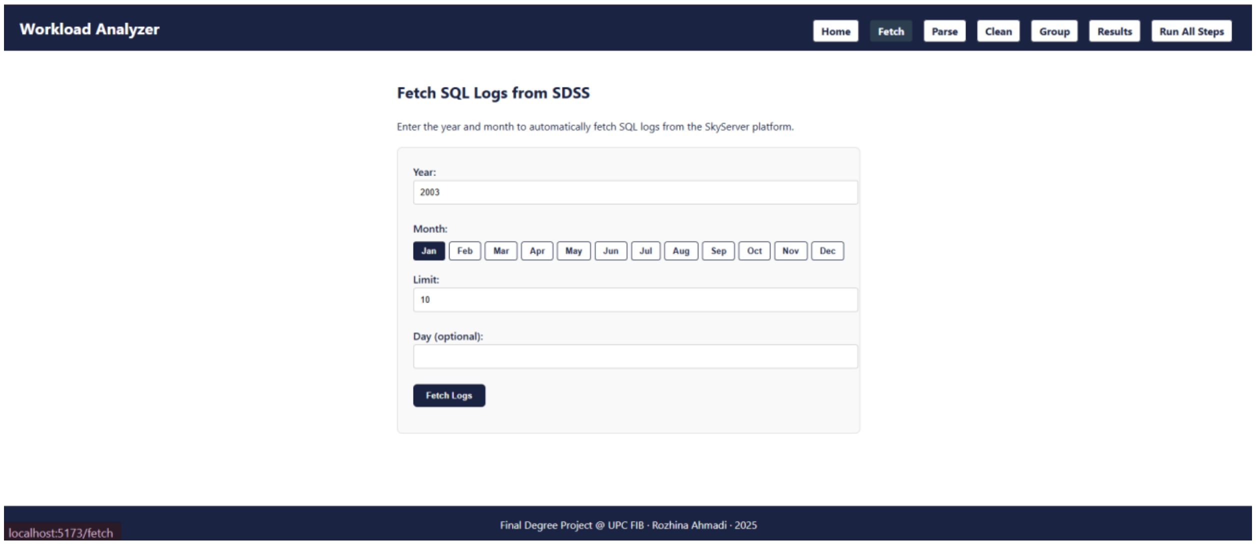
localhost:5173

Figure B1: Screenshot Home Page (Own Work)

B.5 Fetch Page (/fetch)

The fetch page allows users to retrieve raw SQL workload logs directly from SDSS. Users must specify the year and month they want to query and can optionally limit the number of queries fetched by entering a numeric value. Once submitted, the system sends a request to the backend, which retrieves the data and stores it server-side as a `.csv` file.

A confirmation message is displayed upon successful completion of the fetch operation.



Fetch SQL Logs from SDSS

Enter the year and month to automatically fetch SQL logs from the SkyServer platform.

Year: 2003

Month: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

Limit: 10

Day (optional):

Final Degree Project @ UPC FIB · Rozhina Ahmadi · 2025

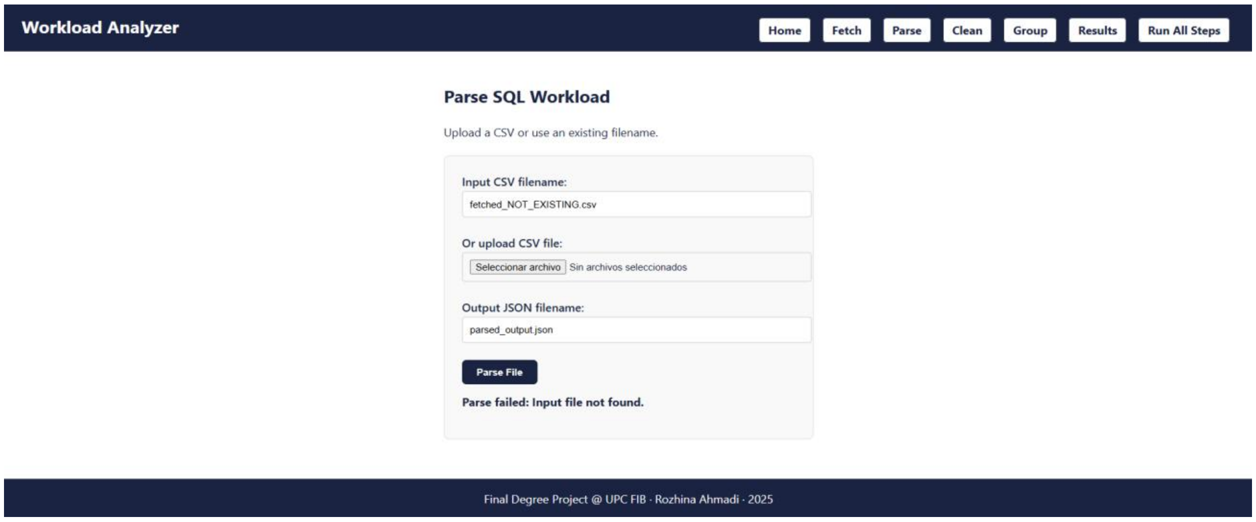
localhost:5173/fetch

Figure B2: Screenshot Fetch Page (Own Work)

B.6 Parse Page (/parse)

The parse page extracts structured features from a raw SQL log file. Users must provide the input filename (previously fetched or uploaded) and specify an output filename for the parsed JSON file. The output is a structured `.json` file stored on the backend. This parsed file will be used in the next cleaning stage.

A confirmation message is displayed upon successful completion.



Parse SQL Workload

Upload a CSV or use an existing filename.

Input CSV filename: fetched_NOT_EXISTING.csv

Or upload CSV file: Seleccionar archivo Sin archivos seleccionados

Output JSON filename: parsed_output.json

Parse File

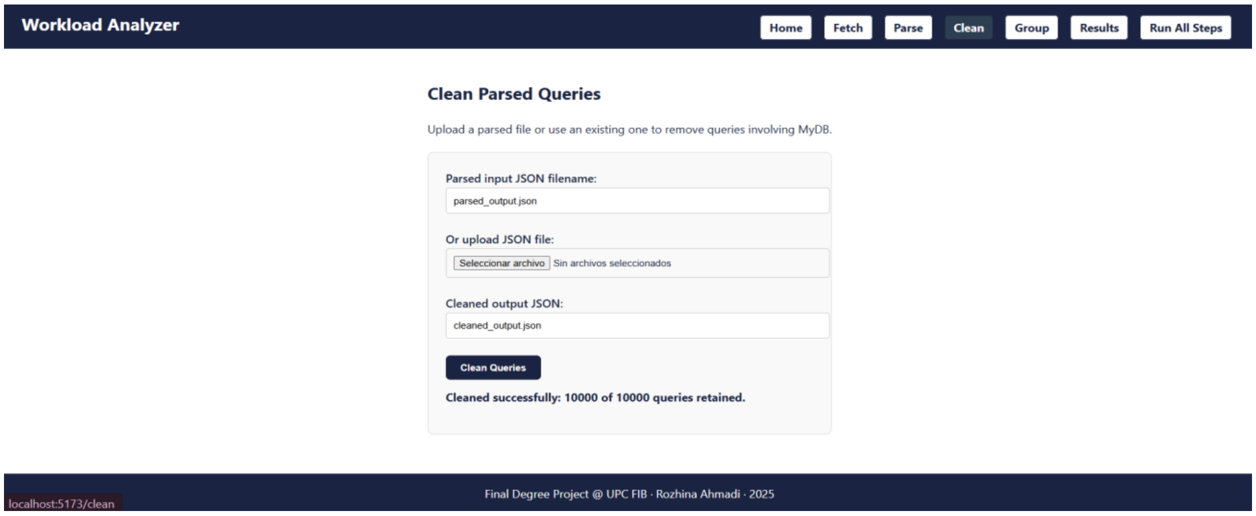
Parse failed: Input file not found.

Final Degree Project @ UPC FIB · Rozhina Ahmadi · 2025

Figure B3: Screenshot Parse Page Example Invalid Input (Own Work)

B.7 Clean Page (/clean)

This page removes non-generalizable queries from the parsed dataset. Users can either upload a parsed `.json` file or provide the name of one already stored on the server. The cleaned output is saved as a new `.json` file.



Clean Parsed Queries

Upload a parsed file or use an existing one to remove queries involving MyDB.

Parsed input JSON filename: parsed_output.json

Or upload JSON file: Seleccionar archivo Sin archivos seleccionados

Cleaned output JSON: cleaned_output.json

Clean Queries

Cleaned successfully: 10000 of 10000 queries retained.

Final Degree Project @ UPC FIB · Rozhina Ahmadi · 2025

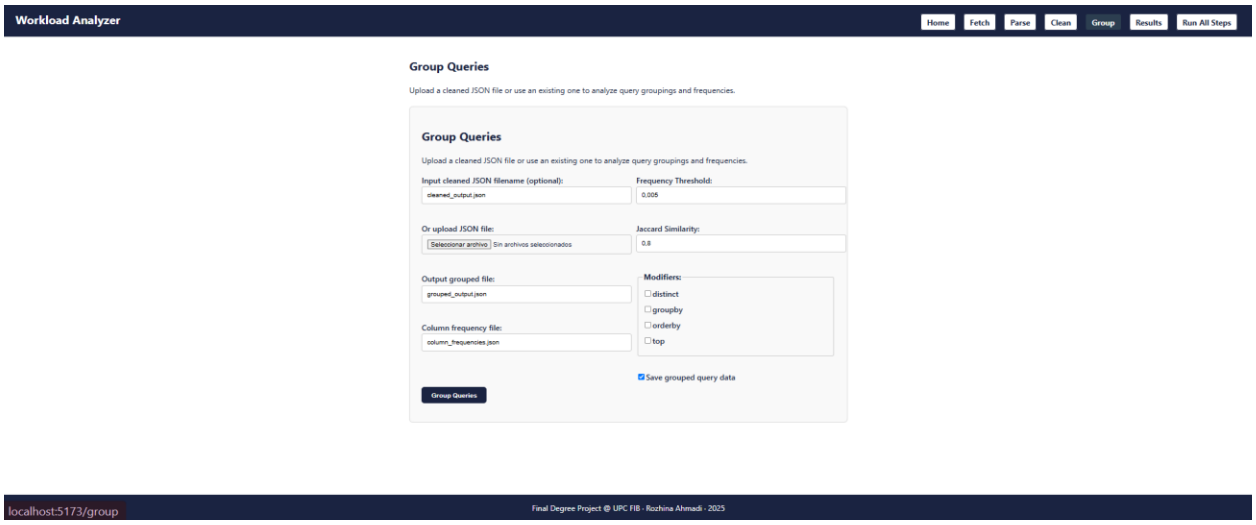
localhost:5173/clean

Figure B4: Screenshot Clean Page Example on Success (Own Work)

B.8 Group Page (/group)

On the group page, users can semantically group similar SQL queries based on their structure and usage patterns. The required input is the name of a cleaned `.json` file. Users may also specify optional parameters such as:

1. Threshold: minimum frequency required for a group to be included.
2. Jaccard threshold: controls similarity level for merging query shapes.
3. Modifiers: logic-based flags such as distinct, group by, order by and top.



Group Queries

Upload a cleaned JSON file or use an existing one to analyze query groupings and frequencies.

Group Queries

Upload a cleaned JSON file or use an existing one to analyze query groupings and frequencies.

Input cleaned JSON filename (optional): cleaned_output.json

Frequency Threshold: 0.005

Or upload JSON file: Seleccionar archivo Sin archivos seleccionados

Jaccard Similarity: 0.8

Output grouped file: grouped_output.json

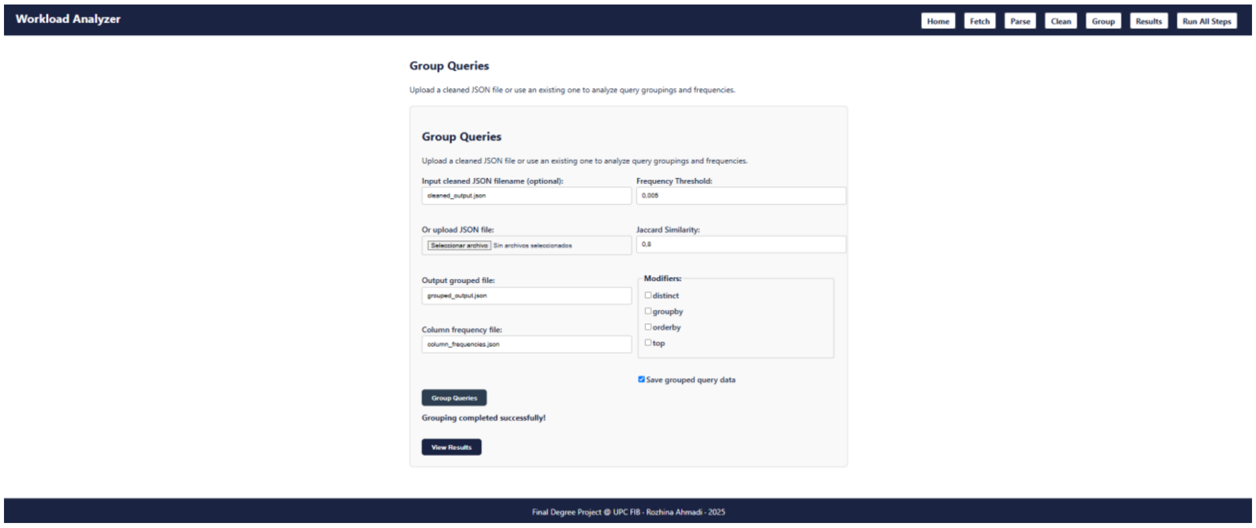
Modifiers:
☐ distinct
☐ groupby
☐ orderby
☐ top

Column frequency file: column_frequencies.json

☑ Save grouped query data

Figure B5: Screenshot Group Page before Clicking the Button (Own Work)

The output files are stored in the `/data` directory and cannot be downloaded via the interface. Once grouping is complete, the user can click the "View Result" button to be redirected to the results page with the grouped data automatically loaded.



Group Queries

Upload a cleaned JSON file or use an existing one to analyze query groupings and frequencies.

Group Queries

Upload a cleaned JSON file or use an existing one to analyze query groupings and frequencies.

Input cleaned JSON filename (optional): cleaned_output.json

Frequency Threshold: 0.005

Or upload JSON file: Seleccionar archivo Sin archivos seleccionados

Jaccard Similarity: 0.8

Output grouped file: grouped_output.json

Modifiers:
☐ distinct
☐ groupby
☐ orderby
☐ top

Column frequency file: column_frequencies.json

☑ Save grouped query data

Grouping completed successfully!

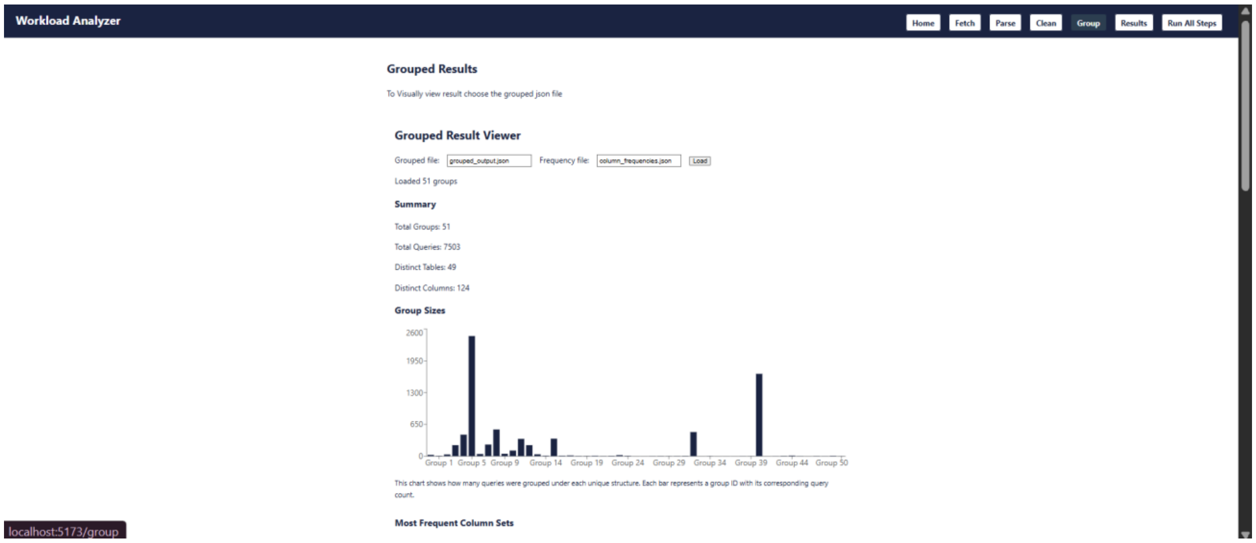
Figure B6: Screenshot Group Page after Grouping (Own Work)

B.9 Results Page (/results)

This page provides a visual summary of the grouped query workload. If accessed from the group page, filenames are preloaded; otherwise, the user can manually enter the name of the grouped output. The results page displays:

1. Overall statistics (total queries, number of groups, unique tables and columns)
2. A bar chart showing group sizes
3. Expandable sections showing query structures and sample queries per group

The output files are stored in `/data` directory and cannot be downloaded via the interface. This page supports pagination and toggle controls for easier exploration of large datasets.



Grouped Results

To Visually view result choose the grouped json file

Grouped Result Viewer

Grouped file: grouped_output.json Frequency file: column_frequencies.json Load

Loaded 51 groups

Summary

Total Groups: 51

Total Queries: 7503

Distinct Tables: 49

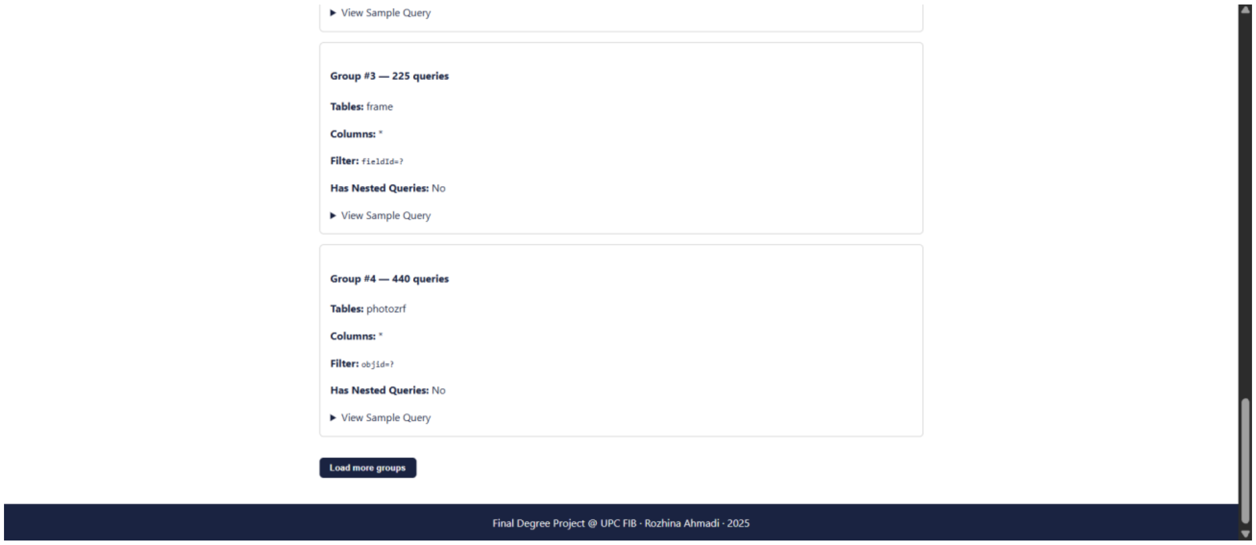
Distinct Columns: 124

Group Sizes

This chart shows how many queries were grouped under each unique structure. Each bar represents a group ID with its corresponding query count.

Most Frequent Column Sets

Figure B7: Screenshot Result Page (Own Work)



Group #3 — 225 queries

Tables: frame

Columns: *

Filter: `fieldId=?`

Has Nested Queries: No

Group #4 — 440 queries

Tables: photozrf

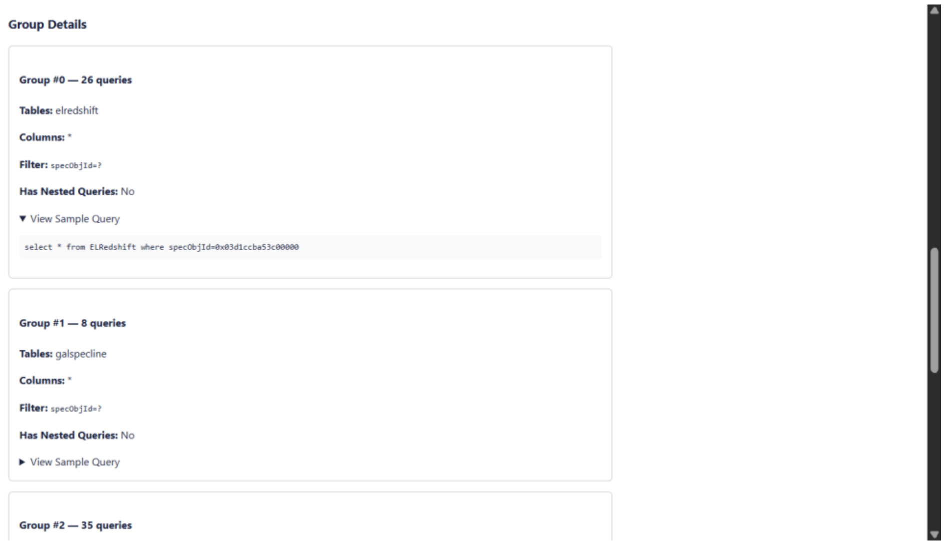
Columns: *

Filter: `objid=?`

Has Nested Queries: No

Final Degree Project @ UPC FIB · Rozhina Ahmadi · 2025

Figure B8: Screenshot Result Page Example Load More Groups button (Own Work)



Group Details

Group #0 — 26 queries

Tables: elredshift

Columns: *

Filter: `specObjId=?`

Has Nested Queries: No

▼ View Sample Query

```
select * from ELRedshift where specObjId=0x03d1ccba53c00000
```

Group #1 — 8 queries

Tables: galspecline

Columns: *

Filter: `specObjId=?`

Has Nested Queries: No

▶ View Sample Query

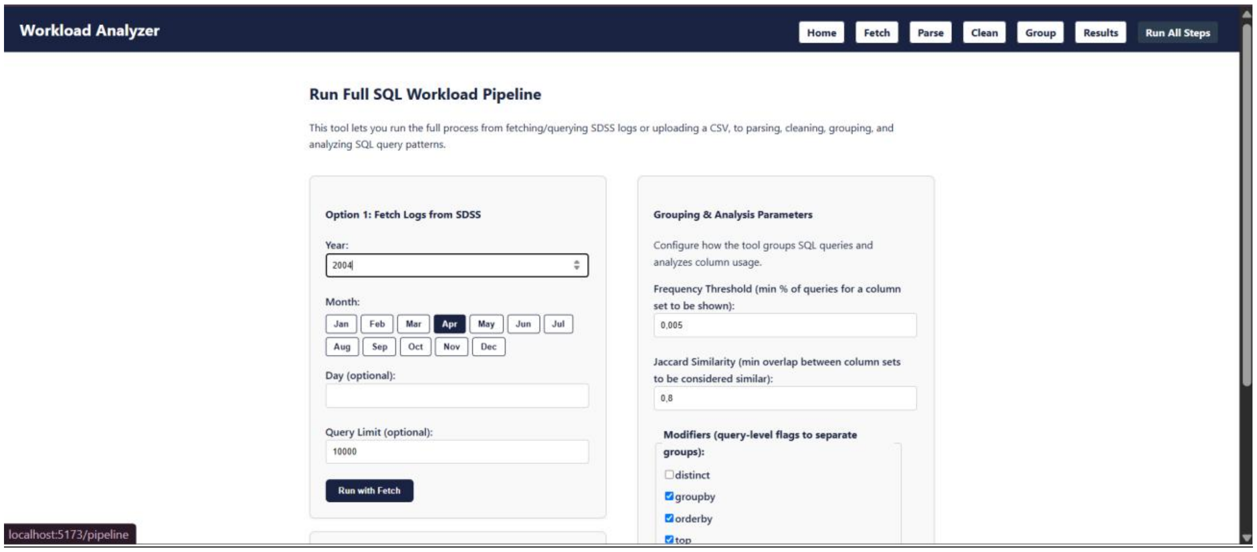
Group #2 — 35 queries

Figure B9: Screenshot Result Page Example View Sample Query (Own Work)

B.10 Run All Steps Page (/pipeline)

This page simplifies the process by allowing the user to execute the entire pipeline in one click. Users can upload a raw .csv file or fetch SQL logs directly from SDSS using the same parameters as on the fetch page. The system then automatically performs parsing, cleaning, and grouping using default settings. Progress is displayed after each step, and the user is finally redirected to the results page. This unified page is ideal for quick testing or when users prefer not to interact with each stage separately.

Scrolling down, users can see Option 2 where you can use and existing CSV file.



Run Full SQL Workload Pipeline

This tool lets you run the full process from fetching/querying SDSS logs or uploading a CSV, to parsing, cleaning, grouping, and analyzing SQL query patterns.

Option 1: Fetch Logs from SDSS

Year: 2004

Month: Jan Feb Mar [✓ Apr] May Jun Jul Aug Sep Oct Nov Dec

Day (optional):

Query Limit (optional): 10000

Grouping & Analysis Parameters

Configure how the tool groups SQL queries and analyzes column usage.

Frequency Threshold (min % of queries for a column set to be shown): 0,005

Jaccard Similarity (min overlap between column sets to be considered similar): 0,8

Modifiers (query-level flags to separate groups):

☐ distinct
☑ groupby
☑ orderby
☑ top

Figure B10: Run All Steps Page (Own Work)

Appendix C: Use Case Demonstration

C.1 Introduction

This appendix demonstrates the tool in a real-world scenario by running the complete SQL workload extraction and analysis pipeline using both the Command-Line Interface (CLI) and the Graphical User Interface (GUI). All processing steps (fetch, parse, clean, group) were performed via the CLI, while the GUI was used to display results through visual graphs. This dual-interface setup emphasizes the tool's modularity and supports user-friendly analysis.

C.2 Objective

The goal is to evaluate the tool's effectiveness in analysing real SQL logs from the Sloan Digital Sky Survey (SDSS) and to explore how query patterns vary across different datasets. This section includes:

1. Full pipeline execution: fetch → parse → clean → group
2. Comparison of SQL workload patterns across three samples
3. Impact of grouping modifiers on query similarity detection

Applying the same workflow across multiple datasets helps validate the tool's consistency and analytical capabilities in realistic conditions.

To simulate diverse workload scenarios, three non-consecutive days from 2022 were chosen:

1. 2022-01-15
2. 2022-03-10
3. 2022-06-25

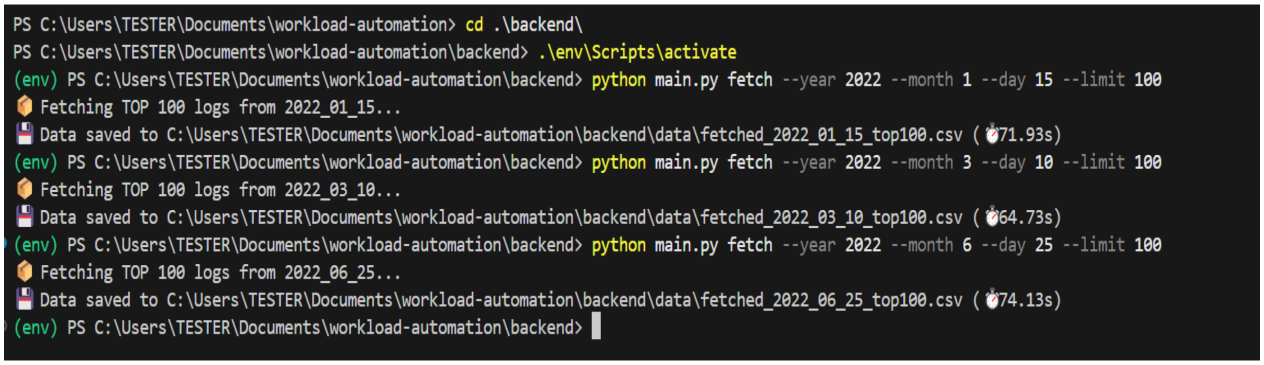
Each day includes the top 100 most frequent queries, allowing for seasonal variation analysis and consistent sample sizes.

C.3 Step-by-Step CLI Execution

The full pipeline was executed on each of the above datasets using the CLI developed during the project using the following commands.

C.3.1 Fetch SQL Logs from SDSS

```
python main.py fetch --year 2022 --month 1 --day 15 --limit 100
python main.py fetch --year 2022 --month 3 --day 10 --limit 100
python main.py fetch --year 2022 --month 6 --day 25 --limit 100
```



```
PS C:\Users\TESTER\Documents\workload-automation> cd .\backend\
PS C:\Users\TESTER\Documents\workload-automation\backend> .\env\Scripts\activate
(env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py fetch --year 2022 --month 1 --day 15 --limit 100
📦 Fetching TOP 100 logs from 2022_01_15...
💾 Data saved to C:\Users\TESTER\Documents\workload-automation\backend\data\fetched_2022_01_15_top100.csv (⏱71.93s)
(env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py fetch --year 2022 --month 3 --day 10 --limit 100
📦 Fetching TOP 100 logs from 2022_03_10...
💾 Data saved to C:\Users\TESTER\Documents\workload-automation\backend\data\fetched_2022_03_10_top100.csv (⏱64.73s)
(env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py fetch --year 2022 --month 6 --day 25 --limit 100
📦 Fetching TOP 100 logs from 2022_06_25...
💾 Data saved to C:\Users\TESTER\Documents\workload-automation\backend\data\fetched_2022_06_25_top100.csv (⏱74.13s)
(env) PS C:\Users\TESTER\Documents\workload-automation\backend>
```

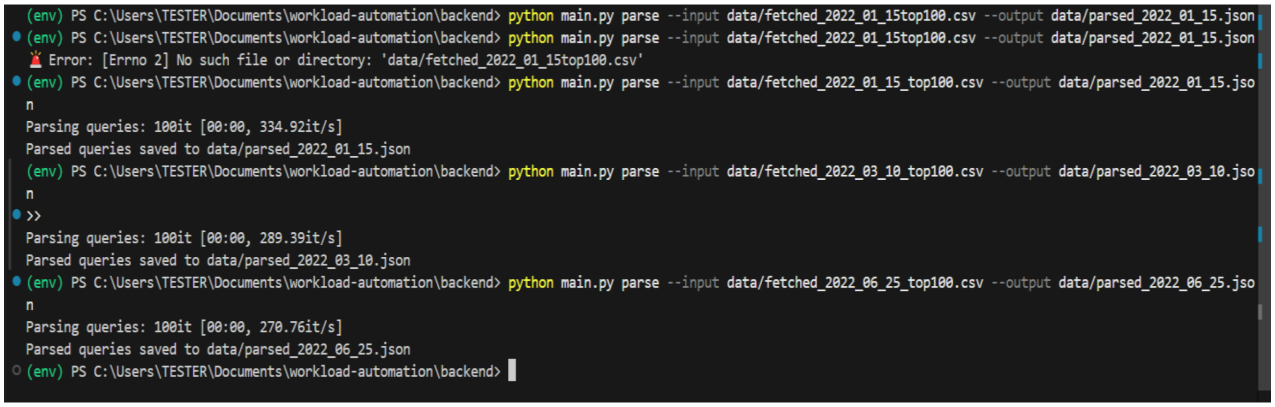
Figure C.1: CLI Fetched Queries (Own Work)

C.3.2 Parse Retrieved Logs

```
python main.py parse --input data/fetched_2022_01_15_top100.csv --outp
ut data/parsed_2022_01_15.json

python main.py parse --input data/fetched_2022_03_10_top100.csv --outp
ut data/parsed_2022_03_10.json

python main.py parse --input data/fetched_2022_06_25_top100.csv --outp
ut data/parsed_2022_06_25.json
```



```
(env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py parse --input data/fetched_2022_01_15top100.csv --output data/parsed_2022_01_15.json
(env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py parse --input data/fetched_2022_01_15top100.csv --output data/parsed_2022_01_15.json
🚨 Error: [Errno 2] No such file or directory: 'data/fetched_2022_01_15top100.csv'
(env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py parse --input data/fetched_2022_01_15_top100.csv --output data/parsed_2022_01_15.jso
n
Parsing queries: 100it [00:00, 334.92it/s]
Parsed queries saved to data/parsed_2022_01_15.json
(env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py parse --input data/fetched_2022_03_10_top100.csv --output data/parsed_2022_03_10.jso
n
>>
Parsing queries: 100it [00:00, 289.39it/s]
Parsed queries saved to data/parsed_2022_03_10.json
(env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py parse --input data/fetched_2022_06_25_top100.csv --output data/parsed_2022_06_25.jso
n
Parsing queries: 100it [00:00, 270.76it/s]
Parsed queries saved to data/parsed_2022_06_25.json
(env) PS C:\Users\TESTER\Documents\workload-automation\backend>
```

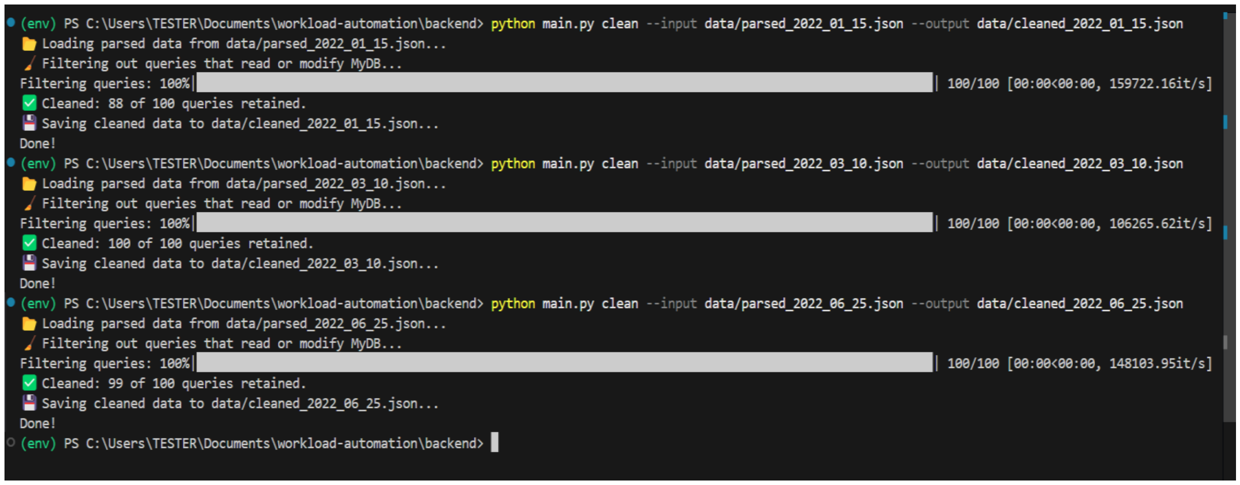
Figure C.2: CLI Parsed Queries (Own Work)

C.3.3 Clean Parsed Data

```
python main.py clean --input data/parsed_2022_01_15.json --output data
/cleaned_2022_01_15.json

python main.py clean --input data/parsed_2022_03_10.json --output data
/cleaned_2022_03_10.json

python main.py clean --input data/parsed_2022_06_25.json --output data
/cleaned_2022_06_25.json
```



```
● (env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py clean --input data/parsed_2022_01_15.json --output data/cleaned_2022_01_15.json
  📁 Loading parsed data from data/parsed_2022_01_15.json...
  🧹 Filtering out queries that read or modify MyDB...
  Filtering queries: 100%|████████████████████████████████████████| 100/100 [00:00<00:00, 159722.16it/s]
  ✅ Cleaned: 88 of 100 queries retained.
  💾 Saving cleaned data to data/cleaned_2022_01_15.json...
  Done!
● (env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py clean --input data/parsed_2022_03_10.json --output data/cleaned_2022_03_10.json
  📁 Loading parsed data from data/parsed_2022_03_10.json...
  🧹 Filtering out queries that read or modify MyDB...
  Filtering queries: 100%|████████████████████████████████████████| 100/100 [00:00<00:00, 106265.62it/s]
  ✅ Cleaned: 100 of 100 queries retained.
  💾 Saving cleaned data to data/cleaned_2022_03_10.json...
  Done!
● (env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py clean --input data/parsed_2022_06_25.json --output data/cleaned_2022_06_25.json
  📁 Loading parsed data from data/parsed_2022_06_25.json...
  🧹 Filtering out queries that read or modify MyDB...
  Filtering queries: 100%|████████████████████████████████████████| 100/100 [00:00<00:00, 148103.95it/s]
  ✅ Cleaned: 99 of 100 queries retained.
  💾 Saving cleaned data to data/cleaned_2022_06_25.json...
  Done!
○ (env) PS C:\Users\TESTER\Documents\workload-automation\backend> █
```

Figure C.3: CLI Cleaned Queries (Own Work)

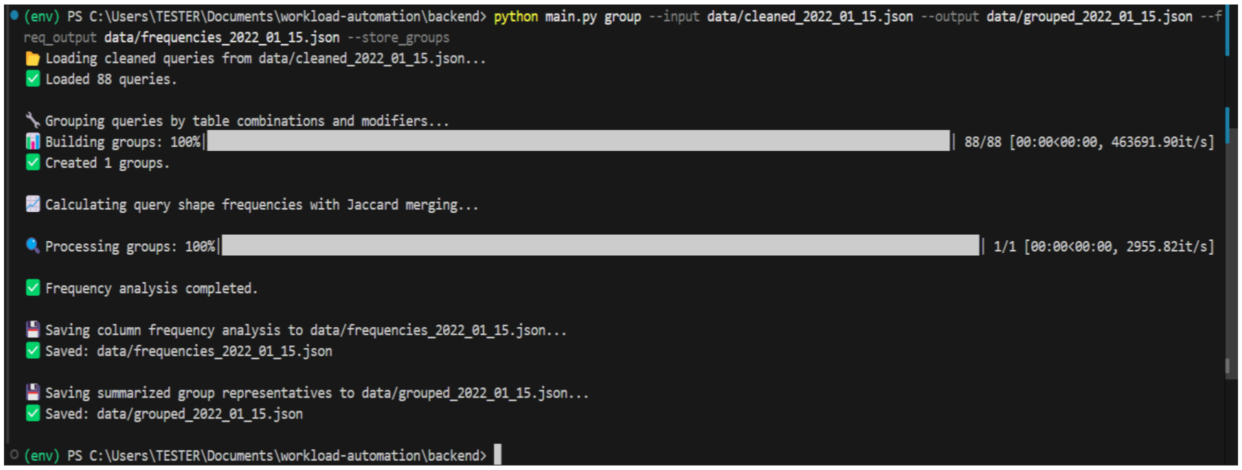
C.3.4 Group Queries without Modifiers

The grouping stage was executed using default parameters `--threshold 0.005` and `--jaccard 0.8` initially without modifiers.

```
python main.py group --input data/cleaned_2022_01_15.json --output dat
a/grouped_2022_01_15.json --freq_output data/frequencies_2022_01_15.js
on --store_groups

python main.py group --input data/cleaned_2022_03_10.json --output dat
a/grouped_2022_03_10.json --freq_output data/frequencies_2022_03_10.js
on --store_groups

python main.py group --input data/cleaned_2022_06_25.json --output dat
a/grouped_2022_06_25.json --freq_output data/frequencies_2022_06_25.js
on --store_groups
```



```
● (env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py group --input data/cleaned_2022_01_15.json --output data/grouped_2022_01_15.json --f
  req_output data/frequencies_2022_01_15.json --store_groups
  📁 Loading cleaned queries from data/cleaned_2022_01_15.json...
  ✅ Loaded 88 queries.

  🔧 Grouping queries by table combinations and modifiers...
  📊 Building groups: 100%|████████████████████████████████████████| 88/88 [00:00<00:00, 463691.90it/s]
  ✅ Created 1 groups.

  📈 Calculating query shape frequencies with Jaccard merging...

  🔍 Processing groups: 100%|████████████████████████████████████████| 1/1 [00:00<00:00, 2955.82it/s]

  ✅ Frequency analysis completed.

  💾 Saving column frequency analysis to data/frequencies_2022_01_15.json...
  ✅ Saved: data/frequencies_2022_01_15.json

  💾 Saving summarized group representatives to data/grouped_2022_01_15.json...
  ✅ Saved: data/grouped_2022_01_15.json

○ (env) PS C:\Users\TESTER\Documents\workload-automation\backend> █
```

Figure C.4: CLI Grouped Queries Example 2022_01_15 (Own Work)

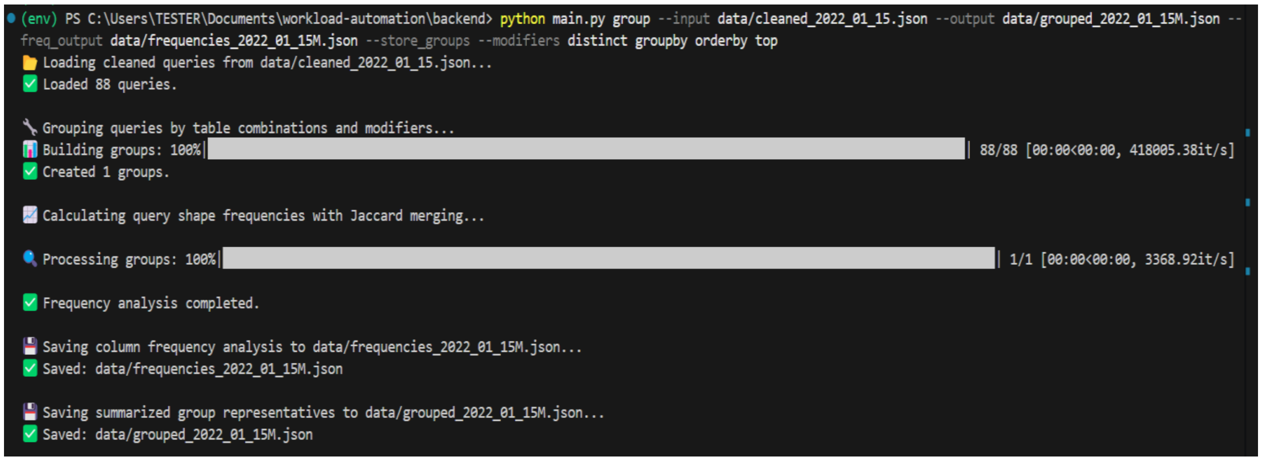
C.3.5 Group Queries with Modifiers

To assess the impact of grouping modifiers, the same datasets were grouped again using clause-based flags `distinct`, `groupby`, `orderby` and `top`.

```
python main.py group --input data/cleaned_2022_01_15.json --output dat
a/grouped_2022_01_15M.json --freq_output data/frequencies_2022_01_15M.
json --store_groups --modifiers distinct groupby orderby top

python main.py group --input data/cleaned_2022_03_10.json --output dat
a/grouped_2022_03_10M.json --freq_output data/frequencies_2022_03_10M.
json --store_groups --modifiers distinct groupby orderby top

python main.py group --input data/cleaned_2022_06_25.json --output dat
a/grouped_2022_06_25M.json --freq_output data/frequencies_2022_06_25M.
json --store_groups --modifiers distinct groupby orderby top
```



```
(env) PS C:\Users\TESTER\Documents\workload-automation\backend> python main.py group --input data/cleaned_2022_01_15.json --output data/grouped_2022_01_15M.json --
freq_output data/frequencies_2022_01_15M.json --store_groups --modifiers distinct groupby orderby top
Loading cleaned queries from data/cleaned_2022_01_15.json...
Loaded 88 queries.

Grouping queries by table combinations and modifiers...
Building groups: 100%|████████████████████████████████████████| 88/88 [00:00<00:00, 418005.38it/s]
Created 1 groups.

Calculating query shape frequencies with Jaccard merging...

Processing groups: 100%|████████████████████████████████████████| 1/1 [00:00<00:00, 3368.92it/s]

Frequency analysis completed.

Saving column frequency analysis to data/frequencies_2022_01_15M.json...
Saved: data/frequencies_2022_01_15M.json

Saving summarized group representatives to data/grouped_2022_01_15M.json...
Saved: data/grouped_2022_01_15M.json
```

Figure C.5: CLI Grouped Queries with Modifiers Example 2022_01_15 (Own Work)

C.4 Results and Observations

C.4.1 Results without Modifiers

The following table shows the grouping results and the highest frequency of repeated query shapes for three selected log samples, where grouping was performed without using any modifiers:

Date	Total Queries	Query Groups	Most Frequent Shape
2022-01-15	88	1	85
2022-03-10	100	7	73
2022-06-25	99	9	47

Table C.1: Grouping Result Summary (Own Work)

On January 15, the workload was extremely homogeneous, with 85 out of 88 queries sharing the same shape and accessing only the `PhotoPrimary` table with identical columns.

On March 10, although the most frequent query still belonged to the `PhotoPrimary` table, only 73% of queries matched that shape. Six additional groups were detected, revealing a broader workload diversity.

On June 25, the most frequent shape appeared only 47 times, and 8 additional groups were identified. Although `PhotoPrimary` remains dominant, we notice small structural variations (e.g., different `objID` casing or presence of URL artifacts) contributing to new groups. This demonstrates how even minor differences, if not normalized further, can cause query shape fragmentation.

C.4.2 Grouping with and without Modifiers Comparison

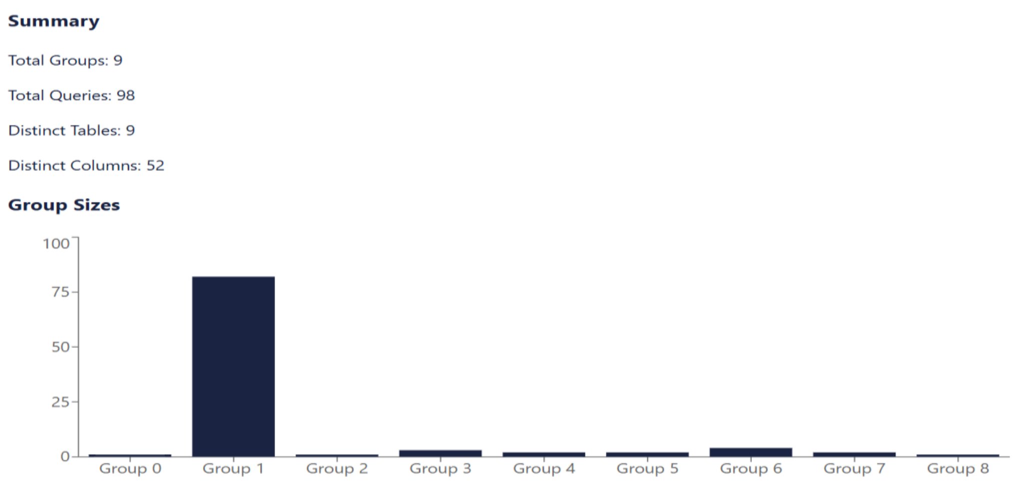
Date	Groups (No Modifiers)	Groups (Modifiers)	Top Group (No Modifiers)	Top Group (Modifiers)
15/01/2022	1	1	85	85
10/03/2022	7	7	73	73
25/06/2022	9	11	82	63

Table C.2: Grouping Comparison Summary (Own Work)

On 2022-01-15, modifiers had no effect. The queries were highly uniform, and grouping results remained identical with or without modifiers.

On 2022-03-10, modifiers also had no impact. The same group structure was preserved, indicating consistent grouping behaviour across both configurations.

On 2022-06-25, the number of groups increased when modifiers were applied. This indicates that modifiers successfully differentiated queries that were previously grouped together. As a result, the size of the top group decreased by 23.17%, highlighting enhanced detection of structurally similar but semantically distinct query shapes.


Summary

Total Groups: 9

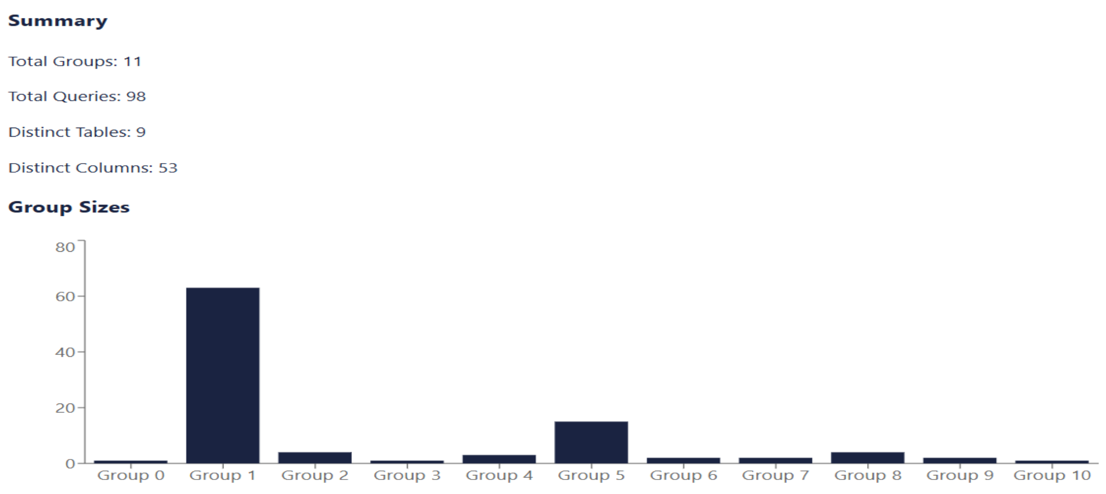
Total Queries: 98

Distinct Tables: 9

Distinct Columns: 52

Group Sizes

Figure C.6: GUI Results Summary Example 2022_01_15 (Own Work)


Summary

Total Groups: 11

Total Queries: 98

Distinct Tables: 9

Distinct Columns: 53

Group Sizes

Figure C.7: GUI Results Summary Example with Modifiers 2022_01_15 (Own Work)

C.5 Conclusion

The experimental results confirm that the developed tool performs effectively and meets the functional objectives of the project. Query grouping reveals significant insights into user behaviour, dominant access patterns, and structural diversity.

Moreover, the inclusion of modifiers enhances semantic granularity in query analysis. While not impactful in uniformly structured workloads, modifiers prove valuable in distinguishing more complex or noisy query sets.

This appendix demonstrates the practical relevance and scalability of the proposed tool, validating its utility for automated workload characterization and potential downstream optimization tasks.