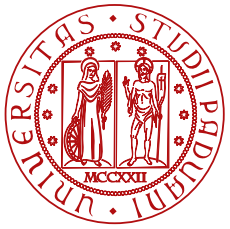


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Computer Engineering

Towards a Unified Platform for Integrating Documentation and Big Data Access in the RFX-mod2 Fusion Experiment

Master Candidate

Nguyen Tho The Cuong

Student ID 2106235

Supervisor

Prof. Rigoni Garola Andrea

University of Padova

Co-supervisor

Ing. Gianluca Moro

Consorzio RFX

Academic Year
2024/2025

To my grandparents
and parents

Abstract

The RFX-mod experiment, now evolving into RFX-mod2, is an operational magnetically confined plasma physics experiment, located at Consorzio RFX premises, focused on studying the physics of fusion plasmas and magnetic confinement in Reversed Field Pinch (RFP) configuration.
During the experiment, several significant improvements were made to the experimental setup, including mechanical modifications, updates to the electrical configuration, and enhancements to the CODAS (Control, Data Access, and Communication Systems). These changes, designed to increase the efficiency and reliability of the machine, will also provide researchers with an increasing amount of diagnostics, opening a view of the plasma phenomena with an unprecedented level of detail. This comes with a new challenge of handling such a large amount of data.
This thesis presents the implementation of a unified platform to streamline experimental data access and documentation. The primary objective is to facilitate advanced data analytics by integrating a web-based application for managing semantic information from the experiment logbook with a query engine for efficient scientific data access. By providing a more accessible and integrated framework for data analysis, this unified system aims to improve the efficiency and reliability of the overall research workflow.

Sommario

L'esperimento RFX-mod, attualmente in evoluzione nella configurazione RFX-mod2, è un impianto operativo di fisica del plasma a confinamento magnetico situato presso il Consorzio RFX, che ha come obiettivo principale lo studio della fisica dei plasmi da fusione e il confinamento magnetico in configurazione Reversed Field Pinch (RFP).
Nel corso dell'esperimento sono stati apportati numerosi miglioramenti significativi alle modalità di sperimentazione, tra cui interventi di natura meccanica, aggiornamenti dell'impianto elettrico e miglioramenti del sistema CODAS (Control, Data Access, e Communication Systems). Questi interventi, mirati ad aumentare l'efficienza e l'affidabilità del sistema sperimentale, aiuteranno anche i ricercatori a ottenere una quantità maggiore di strumenti diagnostici, permettendo loro di osservare i fenomeni relativi al plasma con un livello di dettaglio senza precedenti. Ciò comporta però una nuova sfida: la gestione di elevate quantità di dati di natura eterogenea.
Questa tesi tratta l'implementazione di una piattaforma unificata per la semplificazione dell'accesso ai dati sperimentali e alla documentazione. L'obiettivo principale è quello di facilitare l'analisi avanzata dei dati, integrando in un unico sistema: una applicazione web per la gestione delle informazioni semantiche provenienti dal logbook sperimentale e un motore di estrapolazione efficiente dei dati scientifici. Con la creazione di un framework più accessibile e integrato per l'analisi dei dati, questo sistema unificato mira a migliorare l'efficienza e l'affidabilità dell'intero flusso di lavoro della ricerca.

Contents

List of Figures

List of Tables

List of Code Snippets

List of Acronyms

API Application Programming Interface

CODAS Control, Data Access, and Communication Systems

CPU Central Processing Unit

CRUD Create - Read - Update - Delete

DAG Directed Acyclic Graph

ER Entity Relationship

IAM Identity and Access Management

JVM Java Virtual Machine

MHD Magnetohydrodynamic

OIDC OpenID Connect

ORM Object Relational Mapping

RDD Resilient Distributed Dataset

REST Representational State Transfer

RFP Reversed Field Pinch

SC Scientific Coordinator

SCADA Supervisory Control and Data Acquisition

SL Session Leader

SQL Structured Query Language

SSO Single Sign-On

TR Technical Responsible

UI User Interface

URL Uniform Resource Locator

Introduction

RFX, RFX-mod, and RFX-mod2 are the names that have marked the evolution over time of one of the largest experiment for the study of plasma physics in the Reversed Field Pinch (RFP) magnetic configuration located in Padua and held by RFX Consortium[1].
Given the inherent complexity of a fusion device, executing an experiment in such an environment requires a comprehensive set of tools designed to accurately record and manage data from each experimental attempt, as well as to support users in analyzing the large volumes of data generated.
The phases of an experimental campaign – and the associated recording of data and documentation – are numerous and heterogeneous. Consequently, a wide variety of tools are needed to capture the different aspects involved. The description of the experimental setup – including the documentation outlining the experiment's rationale, design specifications, and parameter configurations – typically relies on the use of a document management system and an experimental logbook. Parameter configuration, control system definition, and data acquisition are generally handled by an integrated Supervisory Control and Data Acquisition (SCADA) system. Finally, data post-processing and result synthesis are carried out using a dedicated summary code.
This thesis proposes the development of an integrated system for logbook data recording, access to experimental data, and automatic generation of summary reports.
Such an integrated tool for data access provides a solid foundation for ensuring the correlation of all information sources associated with the experimental

[1]RFX achieved its first plasma in 1997, progressed to RFX-mod operational between 2004-2016, updating now to RFX-mod2 https://www.igi.cnr.it/en/research/magnetic-confinement-research-in-padova/rfx-mod2/unveiling-the-evolution-upgrades-from-rfx-mod-to-rfx-mod2.

campaigns to be conducted on RFX-mod2. This is essential for maintaining a consistent interpretation of experimental data and for enabling broader contextualization, particularly in the perspective of future studies employing Big Data methodologies.

This thesis is organized into five chapters, each addressing a crucial aspect of the unified platform for experimental data access and documentation. Below is a brief overview of each chapter's content and purpose.

Chapter 1: Data Management in Magnetic Confined Nuclear Fusion Experiments This chapter presents a detailed explanation of the motivation, objectives, and scope of the thesis. It discusses the challenges involved in managing data generated by fusion experiments and in documenting the processes and outcomes of large-scale scientific research. To address these challenges, the thesis proposes the development of a unified platform designed to support efficient data management and to ensure comprehensive, consistent documentation throughout the experimental workflow.

Chapter 2: Experiment Logbook Management This chapter describes the design and implementation of the Experiment Logbook Management system, a web-based application for managing semantic information related to experiments. It emphasizes the importance of effective documentation for reproducibility and collaboration in scientific research.

Chapter 3: Scientific Data Access This chapter focuses on the design and implementation of the Query Engine system, an integrated framework for efficient access to, retrieval of, and processing of large volumes of experimental data stored in MDSplus. It describes how the system streamlines data retrieval and processing tasks and supports reusable and extensible data analytics workflows.

Chapter 4: Results and System Evaluation This chapter presents the results of evaluating the Query Engine system. It details the system setup, performance testing methodology, and experimental results, analyzing the system's functionality, scalability, and the impact of different configurations.

Chapter 5: Conclusions and Future Works The final chapter summarizes the findings and contributions of the work, discusses the system's overall impact on experimental research workflows, and proposes potential directions for further development and improvements.

1

Data Management in Magnetic Confined Nuclear Fusion Experiments

Nuclear fusion is the process by which two atomic nuclei combine to form a heavier nucleus, releasing a significant amount of energy. In thermonuclear fusion – used in experimental fusion research – high temperatures are employed to increase the kinetic energy of the particles, thereby enhancing the frequency and effectiveness of nuclear collisions. At the typical temperatures used in such experiments, matter exists in the plasma state – that is, a highly ionized gas composed of free electrons and atomic nuclei. In this state, the thermal energy is sufficient to overcome the electrostatic forces that normally bind electrons to nuclei, allowing the charged particles to move freely and interact through electromagnetic and nuclear forces. The most efficient fusion reactions involve light nuclei, with particular focus on the fusion of hydrogen isotopes such as deuterium (^{2}H) and tritium (^{3}H). These isotopes are widely used in fusion experiments due to their favorable reaction cross-section and energy yield. The primary reaction of interest is given by [8]:

$$^2\mathrm{H} + {}^3\mathrm{H} \rightarrow {}^4\mathrm{He}\ (3.5\ \mathrm{MeV}) + n\ (14.1\ \mathrm{MeV}) \tag{1.1}$$

The goal of any fusion reactor is to satisfy the so-called "Lawson criterion", that is, to reach an operational state in which the reaction rate is sufficient to self-

sustain the energy spent to keep the plasma ignited. By considering specific energy loss parameters, such as bremsstrahlung radiation or collisional and turbulent phenomena, the reactor's performance is expressed through a single value called the "triple product" [2]:

$$n\tau_E T_i \geq 3 \cdot 10^{21} m^{-3} s \cdot keV \tag{1.2}$$

The plasma ignition curve that satisfies the Lawson criterion shows a minimum value at $T_i \simeq 20keV$: this is the temperature to be achieved in the energy balance, by varying the density (n) and the confinement time (τ_E). Due to the high temperature, no material can contain the reaction without contaminating it. A solution studied for over 60 years is the **magnetic confinement**, where the plasma is contained trapped within a magnetic field gradient[1]. The design of each fusion machine reflects the chosen configuration of its field. The **toroidal** configuration seemed the most topologically advantageous: the torus is the only orient-able, compact surface where a continuous vector field can be defined without critical points[2]. The field configuration then consists of a composition of toroidal and poloidal magnetic components. A usual toroidal reference system is proposed with the following coordinate notation:

- r = radial coordinate
- ϑ = poloidal angle
- φ = toroidal angle

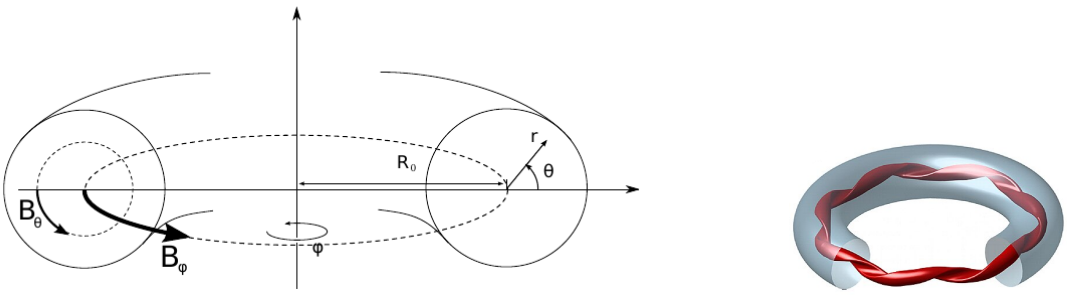
In Figure 1.1a, a sketch of the toroidal coordinates notation is shown for a better comprehension of the main geometry. Within this topological structure, the confinement machines can be also further divided into three main categories:

- *Tokamak*: Both fields are present, with a dominant toroidal field that stabilizes the plasma.
- *Stellarator*: A complex coil structure twists the plasma column.
- *Reversed Field Pinch (RFP)*: The toroidal and poloidal fields are comparable, but stability is more critical. This configuration offers higher efficiency.

[1] Another method is **inertial confinement (ICF)**, achieved by imploding a spherical shell with the D-T mixture, compressed by laser or high-energy ion beams.

[2] This allows the field lines to recombine, minimizing energy and maximizing stability.

The RFX experiment, based in Padova at the Consorzio RFX, aim at studying the latter solution, namely the Reversed Field Configuration [16].
RFX and the subsequent modifications are characterized by a toroidal circular shaped geometry, with minor radius of 0.46 m and major radius of 2 m. The toroidal field can reach a maximum field of 0.7 T with a typical plasma pulse duration that spans along 0.5 s. The experiment evolved in 2004 in RFX-mod which was able to achieve a plasma current up to 2 MA. This high currents has been shown to be crucial for achieving the magnetic field reversals typical of RFP devices. The experiment uses a sophisticated system of magnetic coils to create and control the magnetic fields within the torus. These include: the toroidal field coils, winded around the torus and generating a magnetic field parallel to the torus's circular axis, and poloidal field coils, that shape the plasma equilibrium and control its stability by generating a magnetic field that runs in loops inside.
In addition to the standard configuration RFX exploits also the so called "saddle coils" that, injecting radial field through the vessel, can actively control Magnetohydrodynamic (MHD) instabilities. These coils are crucial for stabilizing the plasma and improving the confinement. They are named "saddle" coils due to their shape, which allows them to be placed around the torus to exert localized magnetic influence on the plasma.
The RFX-mod experiment has contributed significantly to the understanding of plasma confinement, stability, and the behavior of RFP systems. With its advanced control system it played a key role in exploring the high current regimes. The plasma in a such configuration, when the current exceeds a certain threshold value, shows to organize itself spontaneously assuming the shape of a helix: in this condition the temperature is higher and the confinement is more stable. This discovery, that has been initially observed in the RFX laboratories, became the principal matter of investigation of such kind of experiments since 2009 [15].
A pictorial representation of the main helix structure is shown in Figure 1.1b.
The overall plasma equilibrium is obtained trough the study of the plasma dynamics using the model called Magnetohydrodynamic (MHD) system. In the MHD model, the equations of classical electromagnetism are combined with those of the fluid motion [24]. Typically, the analysis of fluid plasma involves the generalized movements of each ionic species; however, for simplicity it is possible to consider each pair of values relating to ions and electrons in a single



(a) (b)

Figure 1.1: Typical notation for the toroidal coordinates reference system (a), and a pictorial representation of the self-organized helical single mode of a RFP (b).

parameter and thus obtain a single fluid trend.

Each charge is subjected to a forces that comes from pressure and electromagnetic interaction with fields. In the static equilibrium condition, the model assume a particularly simple shape and is described by the equations:

$$\nabla \cdot \vec{B} = 0$$
$$\nabla \times \vec{B} = \mu_0 \vec{J}$$
$$\vec{J} \times \vec{B} = \nabla p$$

Instabilities can develop around this equilibrium condition, and due to the system's strict toroidal periodic geometry, these perturbations can be effectively described as deviations from equilibrium expressed in the time-space frequency domain:

$$\tilde{\psi}(\vec{r}, t) = \sum_k \tilde{\psi}_k(r) e^{i(\vec{k}\cdot\vec{r} - \omega t)} = \sum_k \tilde{\psi}_k e^{i(m\vartheta + n\varphi - \omega t)} \tag{1.3}$$

Here, m and n denote the poloidal and toroidal mode numbers, respectively, corresponding to the number of oscillation periods in the poloidal (ϑ) and toroidal (φ) directions. These mode numbers characterize the spatial structure of the fluctuations within the plasma.

A gradient in the poloidal magnetic field creates a barrier that traps and pinches ions toward the center of the plasma column that forms inside the torus. In the same way, the perturbation wave numbers are closely related to this magnetic field configuration, which appears in the equilibrium condition as a series of nested magnetic surfaces – or "shells" – with increasing field lines pitch moving

outward in the radial axis, as illustrated in Figure 1.3a.
Thus, having parallel shells with the same winding number corresponds to surfaces where field lines attract each other, seeding a new possible instability caused by a growing de-structured internal field configuration. For this reason, a good prerequisite for stability is obtained keeping a winding number (and so the m/n ratio) monotonically changing along the radial section of the plasma column, thus ensuring that surfaces that are close to each other keep different winding number of the field lines. The usual quantity to observe for this behavior is the parameter called "*safety factor*" that in this case needs to be monotonically shaped.

$$q(r) = \frac{rB_\varphi(r)}{R_0 B_\vartheta(r)} = \frac{m}{n}(r) \tag{1.4}$$

The RFP configuration is also affected by the fact that, for q values that result rational numbers, a single shell present field lines that reconnect with them selves after m/n periods, creating the so called "resonant" surface. Many instabilities grow in such condition, making RFP generally unstable and prone to evolve in a chaotic configuration. However, as before stated, in high current regimes the system showed to relax in a more stable condition where a single mode dominates the others – the mode (1,-7) for the RFX and RFX-mod geometry[3] – a state of the plasma that appears convenient for the overall confinement.
RFX-mod helped to improve our understanding in this RFP self-organization, observing the formation of high transport barrier and the growth of convective cells. It can be also operated in the Tokamak configuration, in which it helped to explore shallow safety factor profiles (that are usually avoided by Tokamaks), with experiments reaching the edge of the safety factor values as low as q(a) = 1.2 (where a is the radial position of the wall).
It also contributed to the study of the so called tearing modes mitigation[4] and the stabilization of resistive wall modes using both poloidal and radial sensors. Controlling these may help to understand in a general experiment how to actively prevent disruptions, that are particularly dangerous in Tokamaks, and how to control the safety factor to actively move the plasma in the desired

[3]the n modes will be presented to be negative values hereafter to match the orientation of the reference that has been used in RFX

[4]Tearing modes are instabilities caused by the direct reconnection of the field lines from different magnetic shells, forming a chaotic configuration called magnetic island, that are very common in RFP due to the low q factor and plasma resistivity.

configuration [19].
To observe the inherent complexity of hot plasmas with their interaction with the surrounding matter, and to validate the related inherent models, RFX-mod has been equipped with a very large number of different diagnostics. In the Figure 1.2a, a brief schematic of the diagnostic components localization has been reported, due to space allocation around the torus the diagnostics are usually marked with their angular position as they observe a particular phase of the plasma perturbations.
In Figure 1.2b, some of the most representative observed quantities have been reported from the shot #31477. The first plot from the top presents the rapid increase of the plasma current profile followed by a stationary saturation period. The second plot shows the evolution of the magnetic structure of the plasma over time, where it can be easily appreciated the self organization behavior of the plasma that starts when the current reach a value of about 1.5 T. At that level the chaotic exchange of energy among different mode perturbation relaxes into a single "dominant" one characterized by the wave number (1,-7), this configuration is not completely stable though and the helix gets lost in time leaving space for a chaotic state of multi-modes interplay. The third plot shows the actual electron density of the plasma with respect to a particular density reference, called Greenwald limit, where other kind of perturbations grows [11].
In Chapter 4 both the current profile and the dominant mode ratio will be used as an example of RFX data access routines, specifically to extract from this signals some possible pulse summary quantities, such as the maximum of achieved current and the ratio of ordered single-elicity time over the chaotic state.
RFX-mod2 will be built thanks to the results obtained in RFX-mod, the RFP experiment in operation until 2016, when the design of its upgrades started. The RFX-mod2 machine is designed to complete and enhance the study of the properties of both the RFP configuration in the plasma current regimes up to 2MA and a variety of Tokamak equilibria [25]. To this end, modifications of some components of the RFX-mod machine are underway[5].
The substantial modification of the toroidal complex of the RFX experiment represents the second major modification since its original design [20]. The challenge of this new machine is to be able to produce a plasma with much

[5] https://www.igi.cnr.it/en/research/magnetic-confinement-research-in-padova/rfx-mod2/

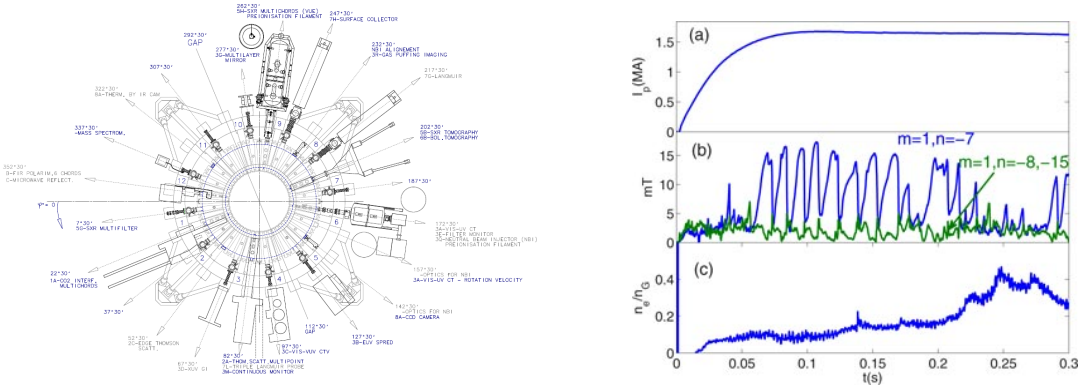


Figure 1.2: RFX map of diagnostics with related toroidal position (a). A RFP discharge time evolution for pulse #31477 of plasma current, dominant $m = 1, n = -7$ mode vs secondary $m = 1, n = -8, -15$ modes, and obtained Greenwald fraction for electron density (b)

improved parameters compared to those of RFX-mod and to clarify whether the RFP configuration can represent a valid alternative to the tokamak design in the perspective of a future fusion reactor.
In RFX-mod2, the internal inconel vacuum chamber will be removed, and the plasma will be directly surrounded by the copper shell (protected by 2016 graphite tiles). This increase in the conductivity of the first conductive material surrounding the plasma will allow to observe it without stationary localized interaction, even at high current regimes (Figure 1.3b).
It is expected that this will enable the production of better-confined plasmas with more stationary helical states. It should be emphasized that these results will be very important for understanding whether the RFP configuration can indeed have potential as a future reactor.
The purpose of these changes is twofold:

- Significantly decrease the resistivity of the first conducting structure surrounding the plasma.
- Bring the plasma closer to the copper stabilizing shell

Thanks to these modifications, firstly, it will be possible to more effectively control the energy losses connected to the interaction of the plasma with the chamber walls that contain it, obtaining more stationary Single Helicity states [5] and warmer and better confined plasmas. Secondly, the new machine will have a

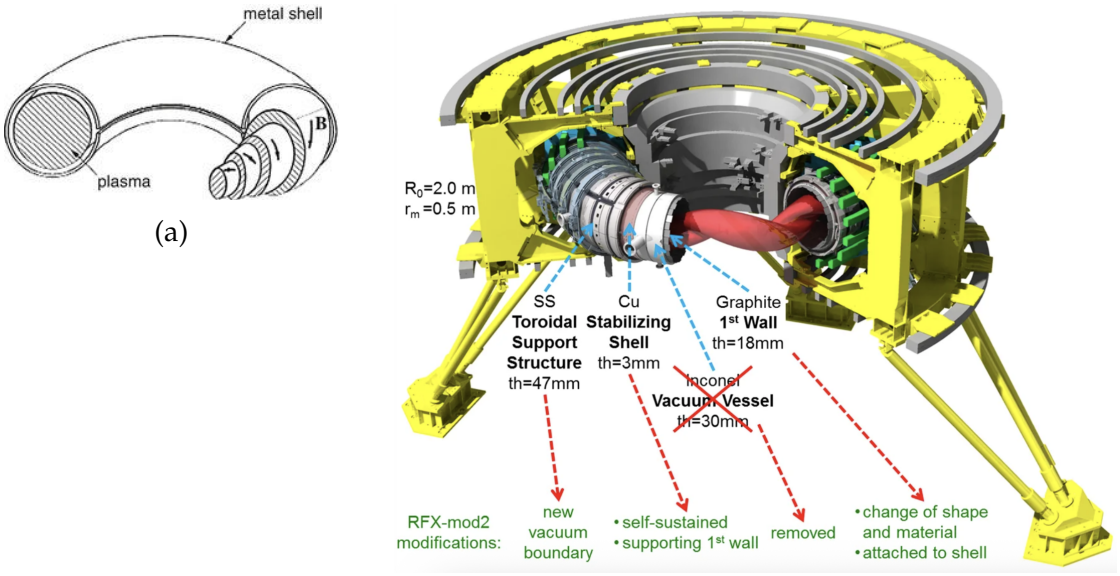


Figure 1.3: A reversed field pinch internal magnetic shells, showing how the direction of the magnetic field varies with radius (a). The current ongoing modification of the RFX-mod experiment into RFX-mod2 showing the structural changes and the removal of the inner vacuum vessel component (b).

larger plasma volume which is also closer to the control systems. The sophisticated magnetic instability control system already installed on the RFX-mod experiment is one of the flagships of the laboratory, being the most advanced currently operating on fusion experiments.[6].

The modifications, designed to enhance the efficiency and reliability of the machine, will not only improve its performance but also provide researchers with a further increased amount of diagnostics. These elements together will finally offer a detailed view of plasma phenomena at an unprecedented level of granularity. However, this improvement introduces another challenge on managing this even larger volume of data, underscoring the necessity for a robust data acquisition system and effective methods to monitor experiment sessions within this complex experimental framework.

Figure 1.4 briefly illustrates the general life cycle of an experimental session, from initial setup to data elaboration and summary. The workflow begins with Experiment Setup, proceeds through Parameter Configuration, and continues

[6] https://www.igi.cnr.it/en/research/magnetic-confinement-research-in-padova/rfx-mod2/unveiling-the-evolution-upgrades-from-rfx-mod-to-rfx-mod2/

with the execution of the experiment. This is followed by Data Collection and Data Elaboration stages. Control Systems interface directly with the experimental setup to manage and monitor various aspects of the experiment.
For each of such stages a proper software tool is devoted to document and gather data. The Logbook component captures semantic information related to the experiment, including detailed descriptions, diagrams, version-controlled software repositories, and both pre-experiment and post-experiment observations. Experimental data from each session is stored in a hierarchical structure using MDSplus[7], where each session is uniquely identified by a Shot Number (or Pulse Identifier). After data collection and elaboration, critical results are distilled and summarized in the Summary section, which highlights key findings and insights.
The logbook plays a critical role in enriching raw data by providing essential context of what we was aiming to achieve and how it was configured, while collecting only sensor data offers a limited view, often confined to a single pulse. As a result, integrating that data with logbook information enables a broader experimental perspective, allowing researchers to identify meaningful correlations across entire experimental campaigns.

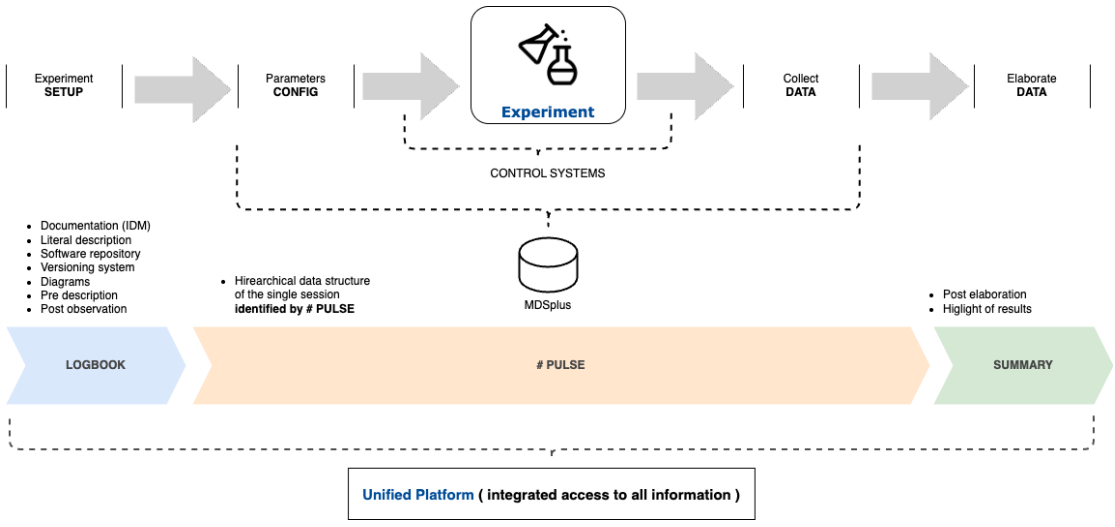


Figure 1.4: Unified Platform

In present-day scientific workflows, managing the vast amount of information generated during experiments remains a significant challenge. The complexity

[7] https://www.mdsplus.org/index.php/Introduction

of keeping track of all documentation – spanning from configuration parameters to contextual and observational data – makes it extremely difficult to relate information that originates from different experimental campaigns or from evolving stages within a single experiment. As a result, scientists frequently create localized databases tailored to specific configurations or setups. These databases typically serve immediate analytical needs but are developed independently, often retained privately by individual researchers or small teams.

Such localized solutions introduce several limitations. They are rarely shared among researchers, making collaboration difficult and limiting reproducibility. These databases are often developed in a way that makes them difficult to port to other analysis systems, thereby impeding integration with advanced analytical environments. Moreover, they tend to fall out of sync with the experimental data pipeline, particularly when updates, calibrations, or corrections are made to the raw data over time. As a result, important insights can become disconnected from the context in which the data was originally produced, reducing their relevance and reliability for future analysis.

At the same time, the scientific community is increasingly leveraging machine learning and data-driven methodologies to uncover complex and correlations across diverse datasets. These correlations can emerge not only within a single experimental campaign but also across multiple setups with different goals or even between entirely different experiments. However, the effectiveness of such techniques critically depends on the availability of comprehensive, well-integrated metadata and semantic context. Without it, the capacity of these methods to produce meaningful results is constrained.

This thesis presents the implementation of a unified platform to streamline experimental data access and documentation. The primary objective is to facilitate advanced data analytics by integrating a web-based application for managing semantic information from the experiment logbook with a query engine for efficient scientific data access. By providing a more accessible and integrated framework for data analysis, this unified system aims to improve the efficiency and reliability of the overall research workflow.

2

Experiment Logbook Management

Effective documentation plays an important role in the context of scientific research. In complex experimental environments, maintaining an accessible information of procedures, observations, and results is critical for ensuring reproducibility, enabling collaboration, and supporting subsequent data analysis. This chapter presents the design and implementation of the Experiment Logbook Management component within the unified platform. This system is developed to manage semantic information through a web-based application, providing researchers with an intuitive interface for recording, manipulating, and retrieving experimental data.

2.1 Problem Clarification

In the context of experimental data management, two important concepts underpin the organization of data: Shots and Runs. A shot refers to a single, discrete instance of data acquisition during an experimental event. Each shot is uniquely identified by an integer known as the shot number. During a typical experimental operation, a wide variety of diagnostic systems collect signals. These signals are archived together under the corresponding shot number, ensuring that all data pertaining to a specific experiment are cohesively stored and easily retrievable. Thus, a shot can be viewed as a complete snapshot of the experimental conditions and measurements at a particular point in time. In contrast, a run generally denotes a collection of shots, grouped based on a shared context such

as a particular day of operations, or a consistent experimental configuration. While the concept of a shot is formalized within MDSplus through its core data structures, the notion of a run is more informal and is typically managed through external metadata, directory conventions, or naming schemes. Runs facilitate the logical organization and analysis of a series of related experiments. By maintaining a clear distinction between individual shots and groups of related shots (runs), it enables researchers to efficiently manage and analyze large volumes of experimental data, thereby supporting reproducibility, traceability, and detailed post-experiment analysis.
The experiment logbook management component is designed to fulfill several essential requirements to support efficient and effective data documentation within the experimental workflow. The core system requirements are outlined as follows:

- **Hierarchical Structure of Logbook Organization** The logbook is structured hierarchically, primarily in terms of runs and shots. This structure can be extended to incorporate higher-level groupings such as campaigns and experiments, allowing for flexible categorization and management of experimental data.

- **Data Manipulation Operations for Shots and Runs** The logbook supports standard Create, Read, Update, and Delete (CRUD) operations on both runs and shots. These operations are applicable at all stages of the experimental campaign, ensuring adaptability to a wide range of experimental procedures and data recording requirements.

- **Authentication and Authorization** User access to the logbook is controlled through authentication and role-based authorization mechanisms. Only authenticated users are allowed to perform operations, and permissions are granted based on predefined roles. This approach ensures secure, controlled collaboration and preserves data integrity. To streamline user access, the system integrates Single Sign-On (SSO) as an efficient authentication and authorization solution.

2.2 Technical Background

2.2.1 System Design

The system is structured as a modern web-based application with a clear separation between the frontend, backend, authentication server, and database layers. As shown in Figure 2.1, users interact with the system through a browser-based frontend built using React and Material UI. The frontend communicates with the backend, which is developed in Go using the Gin framework and GORM as the Object Relational Mapping library. User authentication and authorization are managed by Keycloak which is an external authentication server. The backend handles logics, verifies authentication, and interacts with the PostgreSQL database for persistent data storage. This architecture ensures modularity, scalability, and secure data flow between all system components.

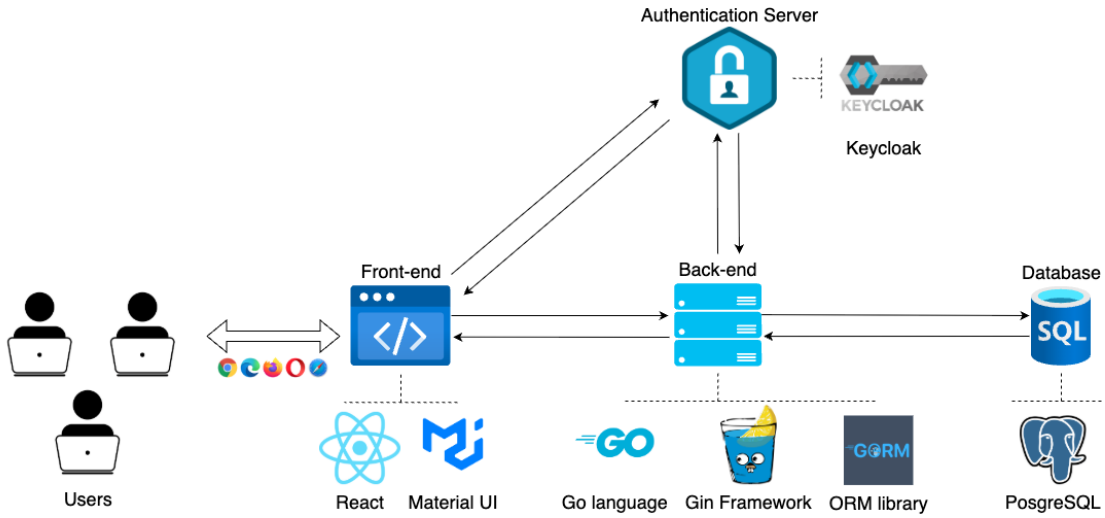


Figure 2.1: Logbook Management System Design

Before going to the implementation details, we will get better understanding about the involved technologies in the system.

2.2.2 Back-end Technologies

The Back-end of a software application is responsible for managing and processing data, ensuring that all components of the system work together smoothly. It forms the core of the application, handling logic, database interactions, authentication, and ensuring overall system performance and scalability.

In this section, we will explore fundamental technologies including databases, programming languages, frameworks, and APIs, each of which plays a crucial role in managing the flow of data, providing efficient performance, and ensuring reliability and scalability. Specifically, we will delve into PostgreSQL, Golang, Gin, GORM, and REST APIs, offering a closer look at how these technologies contribute to the development and deployment of Back-end systems.

Postgresql

PostgreSQL [23] is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads. The origins of PostgreSQL date back to 1986 as part of the POSTGRES project at the University of California at Berkeley and has more than 35 years of active development on the core platform.
PostgreSQL has earned a strong reputation for its proven architecture, reliability, data integrity, robust feature set, extensibility, and the dedication of the open source community behind the software to consistently deliver performant and innovative solutions. PostgreSQL runs on all major operating systems, has been ACID-compliant since 2001, and has powerful add-ons such as the popular PostGIS geospatial database extender. It is no surprise that PostgreSQL has become the open source relational database of choice for many people and organisations.
PostgreSQL comes with many features aimed to help developers build applications, administrators to protect data integrity and build fault-tolerant environments, and help you manage your data no matter how big or small the dataset. In addition to being free and open source, PostgreSQL is highly extensible[1].

Golang

The Go (Golang) programming language was conceived in late 2007 as an answer to some of the problems the developers were seeing developing software infrastructure at Google. The computing landscape today is almost unrelated to the environment in which the languages being used, mostly C++, Java, and Python, had been created. The problems introduced by multicore processors,

[1] `https://www.postgresql.org/about/`

networked systems, massive computation clusters, and the web programming model were being worked around rather than addressed head-on. Moreover, the scale has changed: today's server programs comprise tens of millions of lines of code, are worked on by hundreds or even thousands of programmers, and are updated literally every day. To make matters worse, build times, even on large compilation clusters, have stretched to many minutes, even hours.
Go was designed and developed to make working in this environment more productive. Besides its better-known aspects such as built-in concurrency and garbage collection, Go's design considerations include rigorous dependency management, the adaptability of software architecture as systems grow, and robustness across the boundaries between components [18].

Gin

Gin is a web framework written in Go (Golang), it is the fastest full-featured web framework for Go[2].
The main features of the Gin framework include:

- **Fast** Radix tree based routing, small memory foot print. No reflection. Predictable API performance.
- **Middleware support** An incoming HTTP request can be handled by a chain of middlewares and the final action. For example: Logger, Authorization, GZIP and finally post a message in the DB.
- **Crash-free** Gin can catch a panic occurred during a HTTP request and recover it. This way, your server will be always available. As an example - it's also possible to report this panic to Sentry!
- **JSON validation** Gin can parse and validate the JSON of a request - for example, checking the existence of required values.
- **Routes grouping** Organize your routes better. Authorization required vs non required, different API versions... In addition, the groups can be nested unlimitedly without degrading performance.

[2] `https://gin-gonic.com/en/docs/introduction/`

- **Error management** Gin provides a convenient way to collect all the errors occurred during a HTTP request. Eventually, a middleware can write them to a log file, to a database and send them through the network.
- **Rendering built-in** Gin provides an easy to use API for JSON, XML and HTML rendering.
- **Extendable** Creating a new middleware is so easy, just check out the sample codes.

GORM

GORM is a powerful and feature-rich Object Relational Mapping (ORM) [13] library designed for the Go programming language. ORM is a technique that allows developers to interact with databases using an object-oriented programming paradigm, effectively bridging the gap between object-oriented languages and relational databases. Instead of writing SQL queries, ORMs enable developers to manipulate data through objects, simplifying database interactions and potentially boosting development speed. GORM emphasizes developer productivity and ease of use, making it a popular choice within the Go ecosystem[3]. The key features of GORM include:

- Full-Featured ORM
- Associations (Has One, Has Many, Belongs To, Many To Many, Polymorphism, Single-table inheritance)
- Hooks (Before/After Create/Save/Update/Delete/Find)
- Eager loading with `Preload`, `Joins`
- Transactions, Nested Transactions, Save Point, RollbackTo to Saved Point
- Context, Prepared Statement Mode, DryRun Mode
- Batch Insert, FindInBatches, Find/Create with Map, CRUD with SQL Expr and Context Valuer

[3] `https://gorm.io/docs/`

- SQL Builder, Upsert, Locking, Optimizer/Index/Comment Hints, Named Argument, SubQuery
- Composite Primary Key, Indexes, Constraints
- Auto Migrations
- Logger
- Extendable, flexible plugin API: Database Resolver (Multiple Databases, Read-/Write Splitting) / Prometheus...
- Every feature comes with tests
- Developer Friendly

Several features made GORM a particularly suitable choice for this project. Its ability to simplify database operations by abstracting away raw SQL enabled faster development and reduced the likelihood of syntax errors. GORM's database-agnostic design allowed flexibility in choosing or switching between supported databases such as MySQL or PostgreSQL, which was beneficial during the prototyping and testing phases. The model-driven development approach ensured consistency between the application's data structures and the underlying database schema, making the codebase easier to maintain and reason about. Additionally, automatic migrations significantly reduced overhead by synchronizing database schema changes directly from Go struct definitions. Finally, GORM's expressive query building API allowed for the construction of complex queries in a more readable and maintainable way, improving both development speed and code clarity.

REST API

A REST API (also called a RESTful API or RESTful web API) is An Application Programming Interface (API) that conforms to the design principles of the representational state transfer (REST) architectural style [6]. REST APIs provide a lightweight way to build web APIs and are commonly used to facilitate data exchange between applications, web services and databases, and to connect components in microservices architectures.

At the most basic level, an API is a mechanism that enables an application or service to access a resource within another application or service. The application or service that accesses resources is the client, and the application or service that contains the resource is the server. Some APIs, such as SOAP or XML-RPC, impose a strict framework on developers. But developers can develop REST APIs using virtually any programming language and support a variety of data formats. The only requirement is that they align to the following six REST design principles, also known as architectural constraints.

- **Uniform interface** All API requests for the same resource should look the same, no matter where the request comes from. The REST API should ensure that the same piece of data, such as the name or email address of a user, belongs to only one uniform resource identifier (URI). Resources shouldn't be too large but should contain every piece of information that the client might need.

- **Client-server decoupling** In REST API design, client and server applications must be completely independent of each other. The only information that the client application should know is the URI of the requested resource; it can't interact with the server application in any other ways. Similarly, a server application shouldn't modify the client application other than passing it to the requested data via HTTP.

- **Statelessness** REST APIs are stateless, meaning that each request needs to include all the information necessary for processing it. In other words, REST APIs do not require any server-side sessions. Server applications aren't allowed to store any data related to a client request.

- **Cacheability** When possible, resources should be cacheable on the client or server side. Server responses also need to contain information about whether caching is allowed for the delivered resource. The goal is to improve performance on the client side, while increasing scalability on the server side.

- **Layered system architecture** In REST APIs, the calls and responses go through different layers. As a rule of thumb, don't assume that the client, and server applications connect directly to each other. There may be a number of different intermediaries in the communication loop. REST APIs need to be designed so that neither the client nor the server can tell whether it communicates with the end application or an intermediary.

- **Code on demand (optional)** REST APIs usually send static resources, but in certain cases, responses can also contain executable code (such as Java applets). In these cases, the code should only run on-demand.

REST APIs communicate through HTTP requests to perform standard database functions like creating, reading, updating and deleting records (also known as CRUD) within a resource. For example, a REST API would use a GET request to retrieve a record. A POST request creates a new record. A PUT request updates a record, and a DELETE request deletes one. All HTTP methods can be used in API calls. A well-designed REST API is similar to a website running in a web browser with built-in HTTP functionality.
The state of a resource at any particular instant, or timestamp, is known as the resource representation. This information can be delivered to a client in virtually any format including JavaScript Object Notation (JSON), HTML, XLT, Python, PHP or plain text. JSON is popular because it's readable by both humans and machines and it is programming language-agnostic.
Request headers and parameters are also important in REST API calls because they include important identifier information such as metadata, authorizations, uniform resource identifiers (URIs), caching, cookies and more. Request headers and response headers, along with conventional HTTP status codes, are used within well-designed REST APIs[4].

2.2.3 Front-end Technologies

Front-end technologies are essential for building dynamic and interactive web applications, ensuring an optimal user experience across different devices. These tools enable developers to create responsive, high-performance interfaces with ease. This section highlights two widely-used technologies in modern web development: React and Material UI. React, a JavaScript library, offers a declarative, component-based approach to building user interfaces, while Material UI provides a set of prebuilt components based on Google's Material Design, helping developers create consistent, visually user interfaces quickly and efficiently.

[4] https://www.ibm.com/think/topics/rest-apis

React

React[5] is a widely-used JavaScript [7] library developed by Facebook for building user interfaces. It promotes a modular and declarative paradigm for front-end development, facilitating the construction of complex, maintainable applications.

- **Declarative:** React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes. Declarative views make your code more predictable, simpler to understand, and easier to debug.
- **Component-Based:** Build encapsulated components that manage their own state, then compose them to make complex UIs. Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep the state out of the DOM.
- **Learn Once, Write Anywhere:** React doesn't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code. React can also render on the server using Node and power mobile apps using React Native.

Material UI

Material UI[6] is an open-source React component library that implements Google's Material Design [10]. It includes a comprehensive collection of prebuilt components that are ready for use in production right out of the box and features a suite of customization options that make it easy to implement your own custom design system on top of our components.
The key advantages of Material UI include:

- **Ship faster:** Over 2,500 open-source contributors have poured countless hours into these components. Focus on your core business logic instead of reinventing the wheel.
- **Beautiful by default:** Developers have been meticulous about the implementation, ensuring that every Material UI component meets the highest standards

[5] https://github.com/facebook/react?tab=readme-ov-file
[6] https://mui.com/material-ui/getting-started/

of form and function, but diverge from the official spec where necessary to provide multiple great options.

- **Customizability:** The library includes an extensive set of intuitive customizability features. The templates in our store demonstrate how far you can go with customization
- **Cross-team collaboration:** Material UI's intuitive developer experience reduces the barrier to entry for back-end developers and less technical designers, empowering teams to collaborate more effectively. The design kits streamline your workflow and boost consistency between designers and developers.
- **Trusted by thousands of organizations:** Material UI has the largest UI community in the React ecosystem. It's almost as old as React itself – its history stretches back to 2014. You can count on the community's support for years to come (for example Stack Overflow[7]).

2.2.4 Identity and Access Management

Identity and Access Management (IAM) is essential for controlling and securing user access to systems, applications, and data. It ensures that only authorized users can access the resources they need while maintaining security and compliance. IAM solutions provide mechanisms for user authentication, access control, and identity verification. This section focuses on key IAM concepts, including Signle Sign-On (SSO), authentication and authorization flows, and the role of Keycloak as an open-source solution.
Authentication and authorization are particularly critical in the context of the Logbook Management System and Scientific Data Access. Within a typical session, users take on various roles – such as Scientific Coordinator (SC), Technical Responsible (TR), and Session Leader (SL) – each with specific responsibilities and permissions. Similarly, for Scientific Data Access, IAM enables users to define visibility levels – sharing data publicly, restricting it to specific groups, or keeping it private. It also governs access to computational resources for data processing and analysis, ensuring that only users with the right permissions can initiate tasks. Implementing robust authentication and authorization

[7] `https://stackoverflow.com/questions`

mechanisms is therefore fundamental to maintaining secure, collaborative, and efficient research workflows.

Single Sign-On (SSO)

SSO provides a seamless experience for users when using your applications and services. Instead of having to remember separate sets of credentials for each application or service, users can simply log in once and access your full suite of applications[8].
Single Sign-on and Single Logout are possible through the use of sessions. There may be up to three different sessions for a user with SSO:

- Local session maintained by the application
- Authorization Server session, if SSO is enabled
- Identity Provider (IdP) session, if the user chose to log in through an Identity Provider (such as Google, Facebook, or an enterprise SAML Identity Provider)

Authorization Server, defined in OAuth 2.0 RFC 6749 [3], is the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization. With SSO, a central domain performs authentication and then shares the session with other domains.

Authentication and Authorization Flows

OpenID Connect (OIDC) is an authentication protocol commonly used in consumer-facing SSO implementations. The OIDC protocol handles authentication through JSON Web Tokens and a central identity provider.
With OIDC:

1. A user requests access to an application.
2. The application redirects the user to the identity provider for authentication.
3. The identity provider verifies the user, and if successful, prompts the user to grant data access to the application.

[8] `https://auth0.com/docs/authenticate/single-sign-on`

4. If access is granted, the identity provider generates an ID Token, which contains user identity information that the application can consume.

5. The identity provider returns the user to the application.

According to the OIDC/OAuth 2.0 specifications [3], multiple authentication flows are defined. Within the scope of this project, the focus is on two specific flows: the Authorization Code Flow and the Authorization Code Flow with Proof Key for Code Exchange (PKCE).

Authorization Code Flow The Authorization Code Flow (defined in OAuth 2.0 RFC 6749, section 4.1 [3]), involves exchanging an authorization code for a token. This flow can only be used for confidential applications (such as back-end Service) because the application's authentication methods are included in the exchange and must be kept secure.
Figure 2.2 illustrates the steps involved in the Authorization Code Flow:

1. **User initiates login:** The user chooses to grant your application permissions via OAuth, such as by choosing "Log in with Keycloak" in your app.

2. **Authorization Code request:** The client requests an Authorization Code from the authorization server, including information about what the app is and what permissions it's requesting.

3. **Prompt for consent:** The authorization server asks the user to authenticate and provide consent for the app to access their resources.

4. **User authentication:** The user logs in to the authorization server and approves the requested permissions.

5. **Authorization code return:** The authorization server redirects the user back to your application with a temporary Authorization Code

6. **Token exchange:** Your application sends this code to the authorization server along with your app credentials.

7. **Access token issued:** The authorization server validates everything and returns ID and access tokens

8. **Resource access:** Your application uses the access token to request protected resources or API

9. **Resource response:** API responds with requested data.

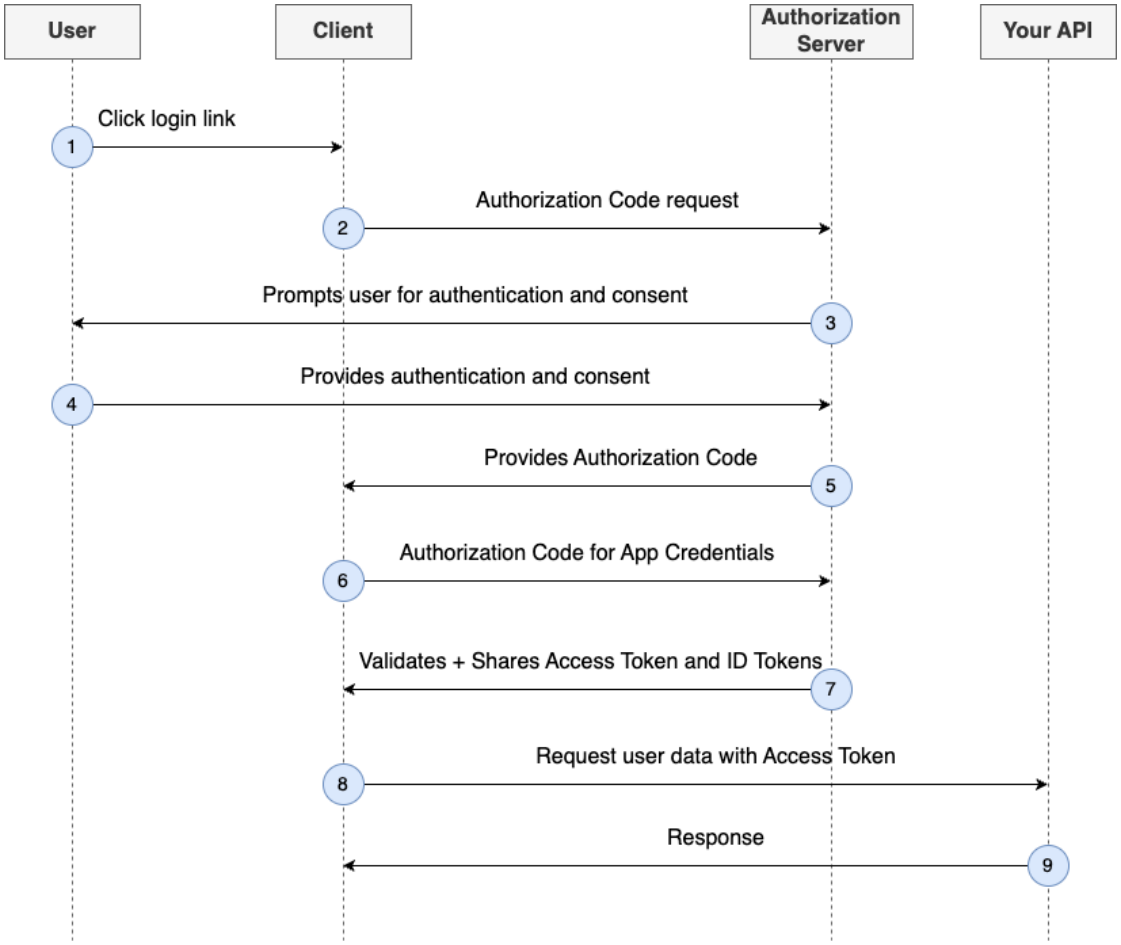


Figure 2.2: Authorization Code Flow

Authorization Code Flow with Proof Key for Code Exchange (PKCE) When public clients, such as e single-page applications, request access tokens, some additional security concerns are posed that are not mitigated by the Authorization Code Flow alone. This is because Single-page apps cannot securely store a Client Secret because their entire source is available to the browser.
Given these situations, OAuth 2.0 provides a version of the Authorization Code Flow which makes use of a Proof Key for Code Exchange (PKCE) (defined in OAuth 2.0 RFC 7636 [17]).
The PKCE-enhanced Authorization Code Flow introduces a secret created by the calling application that can be verified by the authorization server; this secret is called the Code Verifier. Additionally, the calling app creates a transform

value of the Code Verifier called the Code Challenge and sends this value over HTTPS to retrieve an Authorization Code. This way, a malicious attacker can only intercept the Authorization Code, and they cannot exchange it for a token without the Code Verifier.
Because the PKCE-enhanced Authorization Code Flow builds upon the standard Authorization Code Flow, the steps are very similar. Figure 2.3 demonstrates the steps involved in the Authorization Code Flow with Proof Key for Code Exchange.

1. **User initiates login:** Just like in a standard Authorization Code flow, the user initiates the process by selecting the prompt associated with granting your application permissions via OAuth, like "Log in with Keycloak".

2. **Code verifier creation:** Before starting the flow, your application generates a random secret. This is not the same as a client secret – it's a special component called a "code verifier". Client secrets can still be used alongside it. Your application also creates a "code challenge" by transforming the verifier, usually by hashing it (a one-way process that can't be reversed or decoded).

3. **Authorization code request:** The application requests an authorization code from the server and includes the code challenge (along with the hashing method, like SHA-256).

4. **Prompt for user consent:** The authorization server prompts the user to authenticate and provide consent for the requested permissions (same as standard flow).

5. **User authentication:** The user logs in and approves the permissions (same as standard flow).

6. Authorization code return: The authorization server redirects back to your application with the code.

7. **Token exchange (with verification):** Your application sends the code to the authorization server along with the original code verifier.

8. **Server verification:** The authorization server compares the code verifier to the code challenge before issuing any tokens. For example, if the method used was hashing with SHA-256, the server will also hash the code challenge and ensure it matches the verifier; the two strings should be the same.

9. **Access token issued:** If the code verifier matches up with the code challenge, the authorization server returns ID and access tokens.

10. **Resource access**: Your application can now use these tokens to request the necessary resources or API (as in the standard flow).

11. **Resource response:** API responds with requested data.

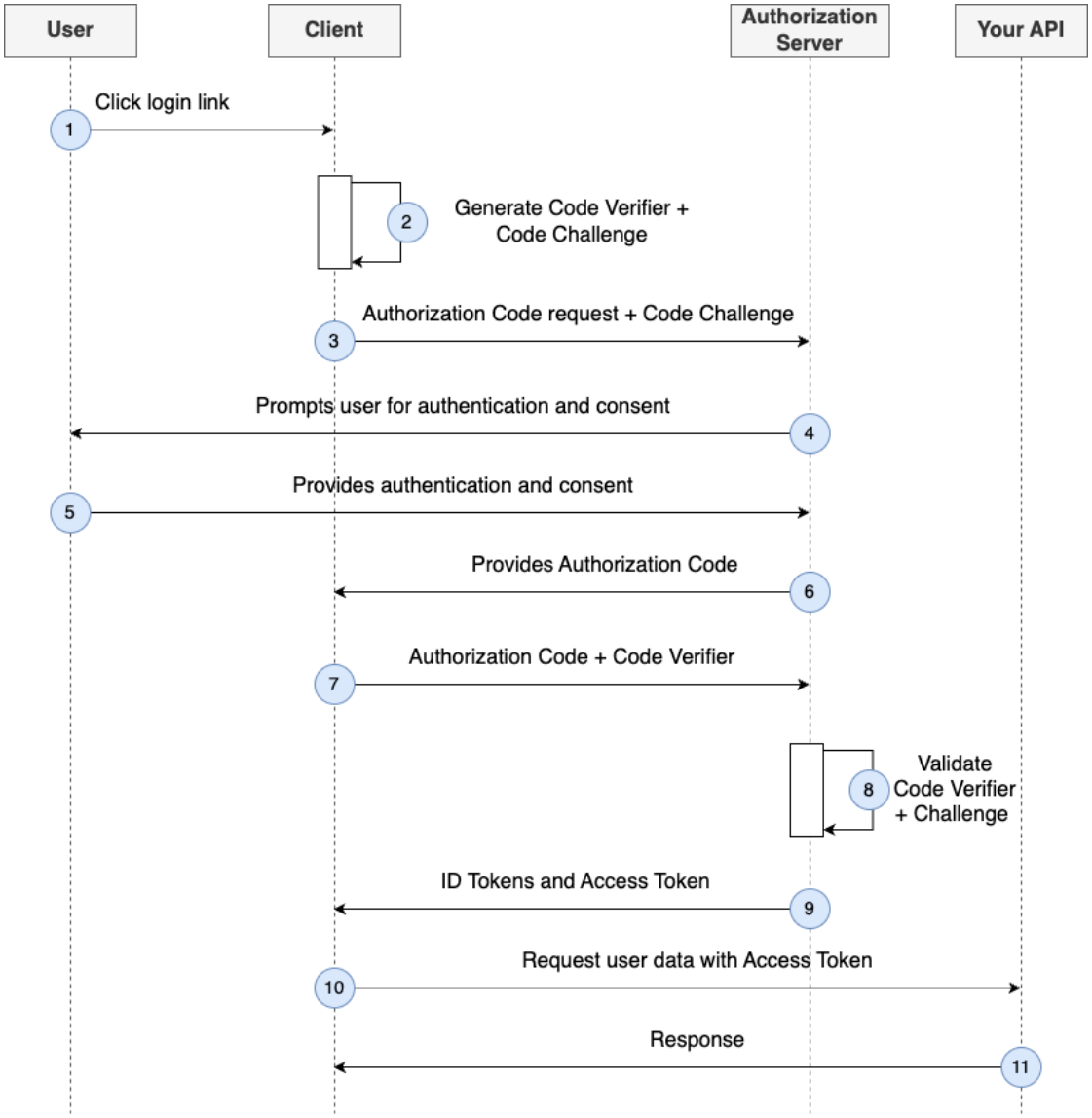


Figure 2.3: Authorization Code Flow with Proof Key for Code Exchange

Several Authorization Server solutions are available, including Auth0, Keycloak, and Okta. For this implementation, Keycloak was chosen due to its open-source

nature, extensive feature set, and strong support for modern identity protocols. The following subsection provides a deeper overview of Keycloak.

Keycloak

Keycloak[9] is an open source Identity and Access Management, allowing Single Sign-On with identity and access management aimed at modern applications and services. Keycloak provides a wide range of features designed to support various authentication and authorization requirements.

- **Single-Sign On** Users authenticate with Keycloak rather than individual applications. This means that your applications don't have to deal with login forms, authenticating users, and storing users. Once logged-in to Keycloak, users don't have to login again to access a different application. This also applies to logout. Keycloak provides single-sign out, which means users only have to logout once to be logged-out of all applications that use Keycloak.
- **Identity Brokering and Social Login** Enabling login with social networks is easy to add through the admin console. It's just a matter of selecting the social network you want to add. No code or changes to your application is required. Keycloak can also authenticate users with existing OpenID Connect or SAML 2.0 Identity Providers. Again, this is just a matter of configuring the Identity Provider through the admin console.
- **User Federation** Keycloak has built-in support to connect to existing LDAP or Active Directory servers. You can also implement your own provider if you have users in other stores, such as a relational database.
- **Admin Console**Through the admin console administrators can centrally manage all aspects of the Keycloak server. They can enable and disable various features. They can configure identity brokering and user federation. They can create and manage applications and services, and define fine-grained authorization policies. They can also manage users, including permissions and sessions.
- **Account Management Console** Through the account management console users can manage their own accounts. They can update the profile, change

[9] `https://www.keycloak.org/`

passwords, and setup two-factor authentication. Users can also manage sessions as well as view history for the account. If you've enabled social login or identity brokering users can also link their accounts with additional providers to allow them to authenticate to the same account with different identity providers.

- **Standard Protocols** Keycloak is based on standard protocols and provides support for OpenID Connect, OAuth 2.0, and SAML.
- **Authorization Services** If role based authorization doesn't cover your needs, Keycloak provides fine-grained authorization services as well. This allows you to manage permissions for all your services from the Keycloak admin console and gives you the power to define exactly the policies you need.

2.3 Development Details

This section provides the necessary details on how each component is implemented and integrated to form the Exeperiment Logbook Management system.

2.3.1 Data Models Declaration

Figure 2.4 presents the Entity Relationship (ER) diagram illustrating the structure and relationships between the Run and Shot entities.

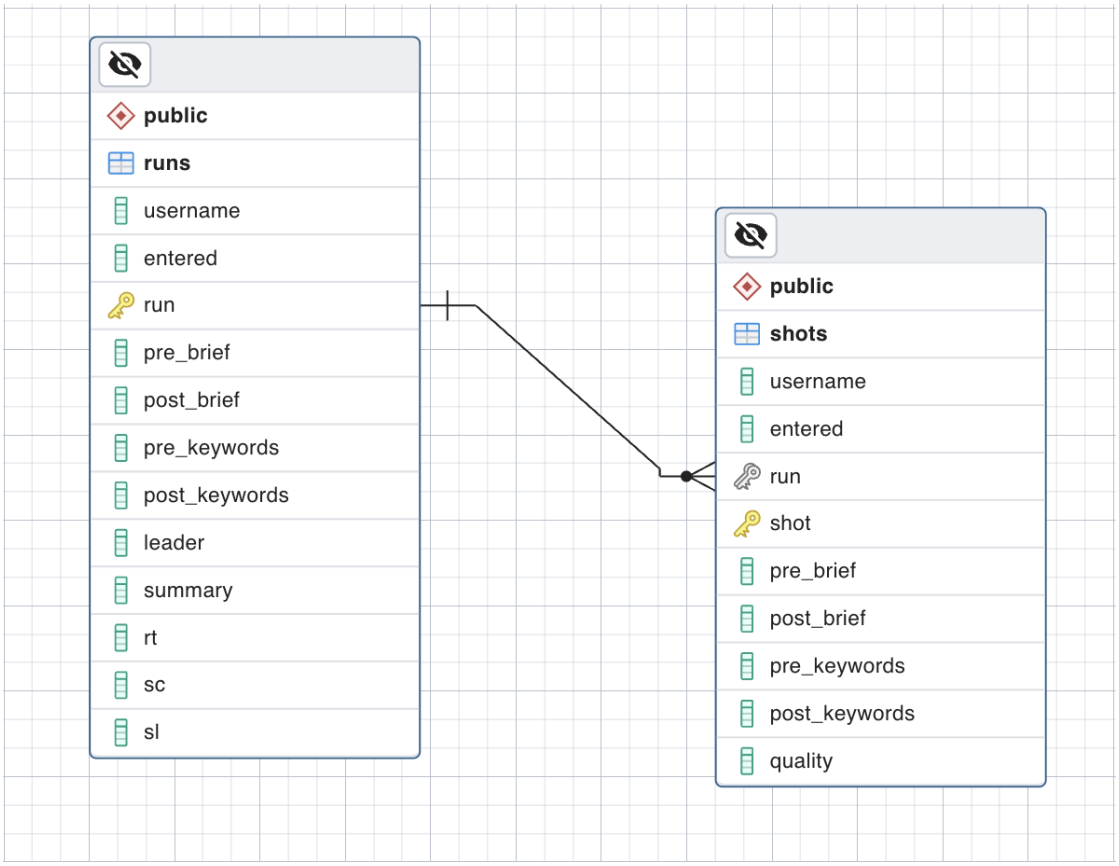


Figure 2.4: Entity Relationship Diagram of Runs and Shots

Using GORM, these entities are implemented in Go as shown in Code 2.1 and Code 2.2.

2.3.2 APIs Implementation

This section examines the implementation of a RESTful API using the Gin web framework in Go (Golang). Gin facilitates efficient development of high-

```
type Run struct {
  Username      string     `gorm:"type:varchar(14)"`
  Entered       time.Time  `gorm:"type:timestamp without time zone;
   default:CURRENT_TIMESTAMP"`
  Run           uint       `gorm:"primaryKey;"`
  PreBrief      string     `gorm:"type:varchar(300)"`
  PostBrief     string     `gorm:"type:varchar(300)"`
  PreKeywords   string     `gorm:"type:varchar(150)"`
  PostKeywords  string     `gorm:"type:varchar(150)"`
  Leader        string     `gorm:"type:varchar(150)"`
  Summary       string     `gorm:"type:varchar(256)"`
  Rt            string     `gorm:"type:varchar(256)"`
  Sc            string     `gorm:"type:varchar(256)"`
  Sl            string     `gorm:"type:varchar(256)"`
}
```

Code 2.1: Run Model Declaration with GORM

```
type Shot struct {
  Username      string     `gorm:"type:varchar(14)"`
  Entered       time.Time  `gorm:"type:timestamp without time zone;
   default:CURRENT_TIMESTAMP"`
  RunID         uint       `gorm:"column:run" json:"Run"`
  Shot          uint       `gorm:"primaryKey;"`
  PreBrief      string     `gorm:"type:varchar(300)"`
  PostBrief     string     `gorm:"type:varchar(300)"`
  PreKeywords   string     `gorm:"type:varchar(150)"`
  PostKeywords  string     `gorm:"type:varchar(150)"`
  Quality       string     `gorm:"type:varchar(2)"`

  RunRef Run `gorm:"foreignKey:RunID;references:Run;constraint:
   OnUpdate:CASCADE,OnDelete:CASCADE;" json:"-"`
}
```

Code 2.2: Shot Model Declaration with GORM

performance web services by offering a simple routing mechanism, middleware support, and seamless JSON handling.

In a typical RESTful API using Gin, HTTP methods such as GET, POST, PUT, and DELETE are mapped to route handlers that process client requests and return responses – commonly in JSON format – with appropriate HTTP status codes.

We describe the implementation in two main parts: setting up the router, and handling request-response logic with a focus on retrieving run data.

Setting up the Gin Router Code 2.3 demonstrates how to initialize the Gin engine and configure the application's API routes. The router setup includes

middleware for CORS and Keycloak-based authentication, ensuring secure access control.

```go
func SetupRouter() *gin.Engine {
  r := gin.Default()
  r.Use(middleware.Cors())
  r.Use(middleware.KeycloakAuthMiddleware())

  r.GET("/logbook/runs", Controllers.GetRuns)
  r.POST("/logbook/runs", Controllers.CreateRun)
  r.PUT("/logbook/runs/:id", Controllers.UpdateRun)
  r.DELETE("/logbook/runs/:id", Controllers.DeleteRun)

  r.GET("/logbook/shots", Controllers.GetShots)
  r.POST("/logbook/shots", Controllers.CreateShot)
  r.PUT("/logbook/shots/:id", Controllers.UpdateShot)
  r.DELETE("/logbook/shots/:id", Controllers.DeleteShot)

  return r
}
```

Code 2.3: Setting up the Gin Router

The defined routes follow RESTful conventions, mapping resource types (runs and shots) to appropriate HTTP methods. Each route is handled by a corresponding function in the `Controllers` package containing handlers for incoming requests.

Handling Request-Response logics The request handling for retrieving run entries is shown in Code 2.4. The handler parses pagination parameters from the query string, retrieves the relevant data using GORM, and returns a structured JSON response.
To retrieve the data, the handler relies on helper functions that interact with a PostgreSQL database via GORM. Code 2.5 shows these data access functions.
For example, sending the following HTTP request to retrieve the first run entry: `GET http://localhost:8081/logbook/runs?page=1&pageSize=1` produces a JSON response as shown in Code 2.6.

RESTful Request–Response Pattern Figure 2.5 presents a sequence diagram illustrating the flow of a typical RESTful API request and response handled by the Gin framework.

```go
func GetRuns(c *gin.Context) {
  // Parse pagination parameters from query string
  page, _ := strconv.Atoi(c.DefaultQuery("page", "1"))
  pageSize, _ := strconv.Atoi(c.DefaultQuery("pageSize", "100"))

  var totalRuns int64
  var runs []Models.Run

  // Retrieve total count and paginated Runs from the database
  err1 := Models.GetTotalRuns(&totalRuns)
  err2 := Models.GetRuns(&runs, page, pageSize)

  // Handle potential errors and return response
  if err1 != nil || err2 != nil {
    c.AbortWithStatus(http.StatusInternalServerError)
  } else {
    c.JSON(http.StatusOK, gin.H{
      "runs":      runs,
      "totalRuns": totalRuns,
    })
  }
}
```

Code 2.4: Handler implementation for retrieving runs

```go
// Queries the total number of Run records in the database
func GetTotalRuns(totalRuns *int64) (err error) {
  if err = Config.DB.Model(&Run{}).Count(totalRuns).Error; err != nil
    {
    return err
  }
  return nil
}

// Retrieves a paginated list of Run records from the database
func GetRuns(runs *[]Run, page int, pageSize int) (err error) {
  offset := (page - 1) * pageSize

  if err = Config.DB.Order("run").Limit(pageSize).Offset(offset).Find
    (runs).Error; err != nil {
    return err
  }
  return nil
}
```

Code 2.5: Querying data in PostgreSQL with GORM

```
% curl 'http://localhost:8081/logbook/runs?page=1&pageSize=1'

{
    "runs": [
        {
            "Username": "MDSPLUS",
            "Entered": "1996-03-28T17:45:03.23Z",
            "Run": 4,
            "PreBrief": "test 2",
            "PostBrief": "",
            "PreKeywords": "",
            "PostKeywords": "",
            "Leader": "",
            "Summary": "",
            "Rt": "",
            "Sc": "",
            "Sl": ""
        }
    ],
    "totalRuns": 2369
}
```

Code 2.6: HTTP request to retrieve the first run entry

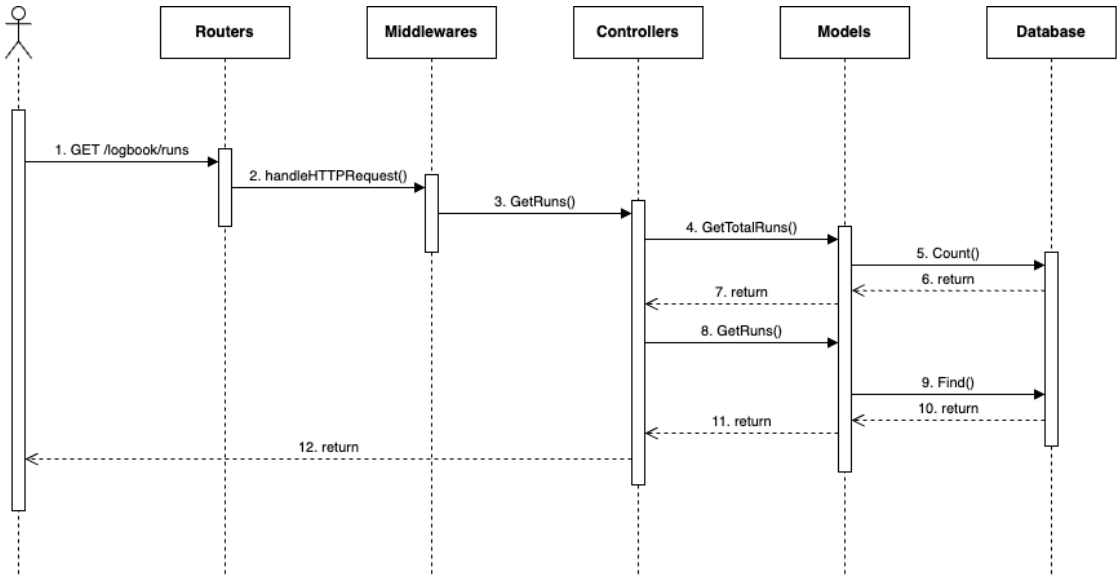
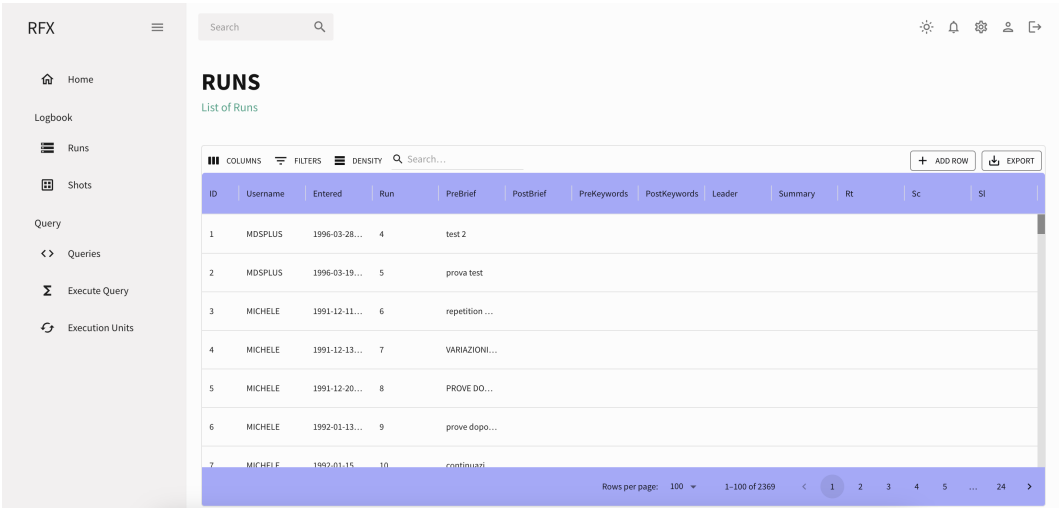


Figure 2.5: RESTful Request – Response Pattern with Gin

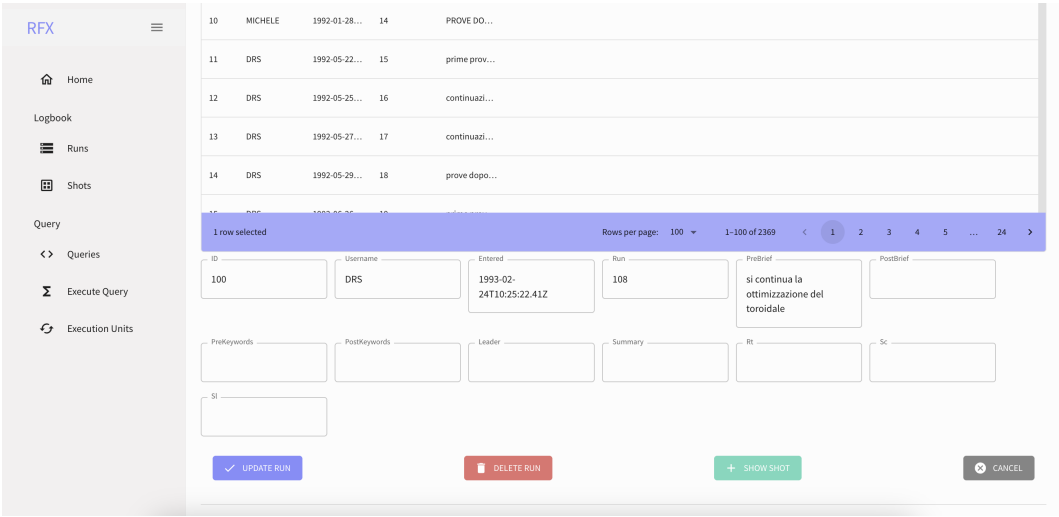
This pattern highlights the flow from incoming HTTP request, through routing, middleware, and controller, to database interaction, and finally the response return.

2.3.3 User Interface

Figure 2.6 presents the user interface designed for managing Runs. This interface is visually similar to the one used for Shots. It enables users not only to view existing data but also to interact with the system to add new entries, update existing records, delete items, and paginate through data seamlessly.



(a)



(b)

Figure 2.6: User Interface for managing Runs

The dashboard UI is constructed using several reusable components provided by the MUI framework, including `Box`, `DataGrid`, `Button`, `TextField`, among others. These components are well documented in the official MUI documentation[10], making development both efficient and consistent with modern UI design practices.
To enable interaction with the back-end service via REST APIs, we utilize the Axios library[11] for performing HTTP requests. Code 2.7 demonstrates how Axios is used to retrieve a paginated list of Runs, with authentication handled via a Bearer token obtained from Keycloak.

```
const getRuns = (page, pageSize, keycloak) => {
    return axios.get(API_URL, {
        params: {
            page: page,
            pageSize: pageSize,
        },

        headers: { Authorization: `Bearer ${keycloak.token}` },
    });
}
```

Code 2.7: Axios request to retrieve paginated list of runs

Figure 2.7 illustrates the Sign-In page, which is presented to users during the authentication process. Using prebuilt components such as `Dialog` and the `SignInPage`[12], the application supports integration with various sign-in methods. For this project, authentication is handled through Keycloak. The implementation details of the authentication process are discussed in the following subsection.

2.3.4 Single Sign-On

Figure 2.8 illustrates the authentication and authorization workflow based on Single Sign-On (SSO) using Keycloak. This setup is integrated into a system architecture that separates the front-end and back-end services, enhancing modularity and scalability. The flow proceeds as follows:

[10] https://mui.com/material-ui/all-components/
[11] https://axios-http.com/docs/intro
[12] https://next.mui.com/toolpad/core/react-sign-in-page/

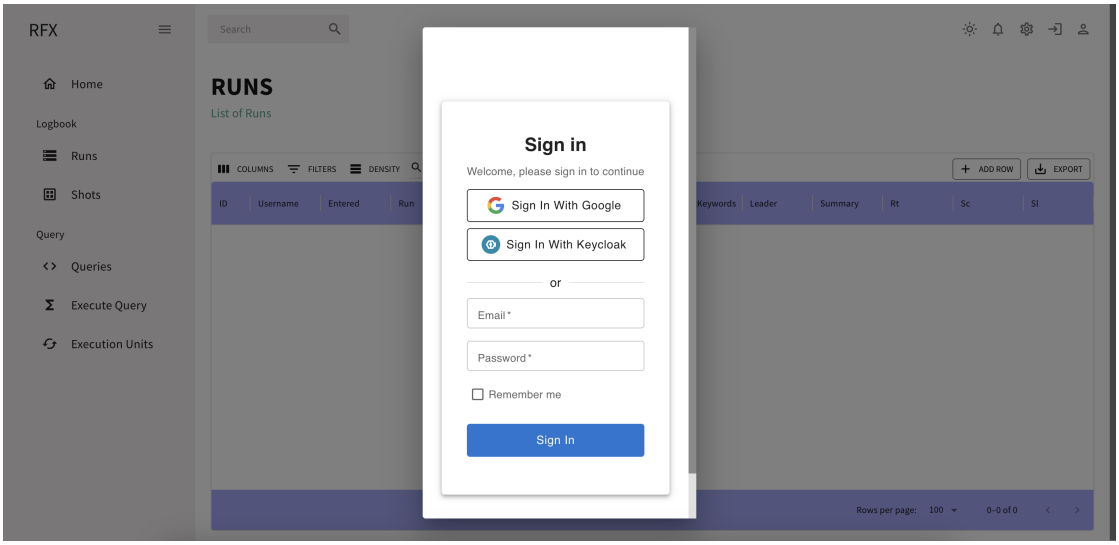


Figure 2.7: Sign-in Page

1. **Login with Keycloak:** User initiates the login process through the frontend application, which redirects the user to the Keycloak authentication server. Here, the user provides their credentials via a login interface managed by Keycloak.

2. **Get Access Token:** Upon successful authentication, Keycloak issues an access token and redirects the user back to the frontend. This token contains user identity and authorization information, enabling secure communication with back-end services.

3. **API call with Access Token:** The frontend includes the access token in the headers of API requests to the backend. This ensures that each request is authenticated and can be authorized appropriately.

4. **Request Access Token Verify:** The backend, upon receiving a request, forwards the access token to the Keycloak server for verification. This step ensures the token is valid, not expired, and has the necessary permissions. This logic is implemented within the Middleware layer of the Go backend.

5. **Allow/Deny Response:** Based on the verification result, the Keycloak server responds to the backend, indicating whether the request should be allowed or denied.

6. **Result of API:** If the token is verified and the request is authorized, the backend proceeds with the requested operation and returns the result to the

frontend, which then displays it to the user.

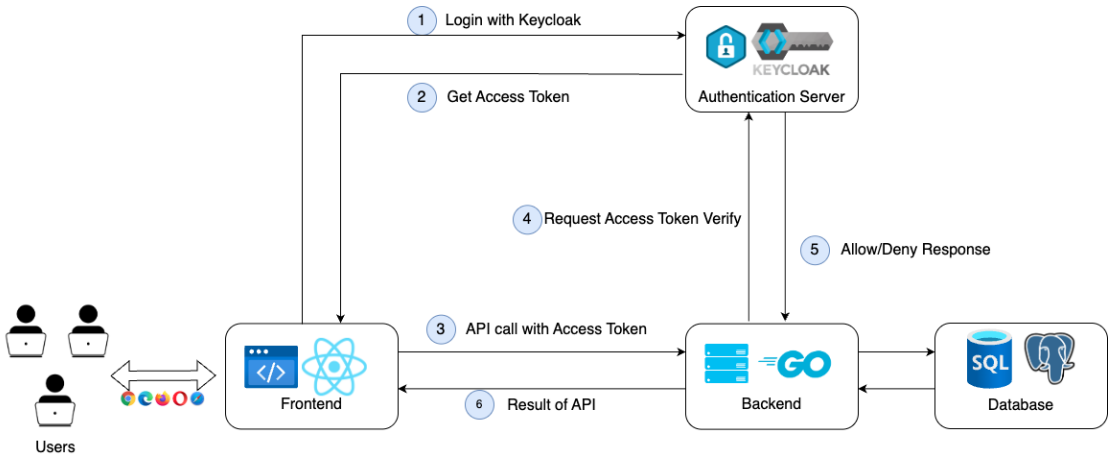


Figure 2.8: Single Sign-On with Keycloak Workflow

This workflow demonstrates how Keycloak acts as a centralized authentication server, enabling secure and streamlined access control across distributed components. In the following discussion, we will delve deeper into the implementation details for each step in this workflow.

Setting up Keycloak

Before integrating Keycloak into the system, it is important to understand several fundamental concepts that form the basis of its authentication and authorization model[13]. These core components define how users, applications, and security policies interact within the Keycloak system.

- **Realm** A realm manages a set of users, credentials, roles, and groups. A user belongs to and logs into a realm. Realms are isolated from one another and can only manage and authenticate the users that they control.
- **Client** Clients are entities that can request Keycloak to authenticate a user. Most often, clients are applications and services that want to use Keycloak to secure themselves and provide a single sign-on solution. Clients can also be entities that just want to request identity information or an access token so that they can securely invoke other services on the network that are secured by Keycloak.

[13] `https://www.keycloak.org/docs/latest/server_admin/index.html`

- **User** Users are entities that are able to log into your system. They can have attributes associated with themselves like email, username, address, phone number, and birthday. They can be assigned group membership and have specific roles assigned to them.

To run Keycloak in development mode and configure it for basic use, follow these initial setup steps. A more detailed guide is available on the official website[14].

1. **Create a Realm:** Create a new realm to isolate your authentication environment. This will serve as the container for users, clients, and roles.
2. **Create a User:** Create a user to the realm. This user can later be used to test authentication flows and token issuance.
3. **Create a Client with OpenID Connect:** Create new clients such as the frontend and back-end services, and configure them to use the OpenID Connect protocol.

Front-end Client

Keycloak comes with a client-side JavaScript library called `keycloak-js` that can be used to secure web applications. The adapter uses OpenID Connect protocol under the covers[15].
Code 2.8 demonstrates how to create and initialize a Keycloak client instance in JavaScript.

```
const keycloak = new Keycloak({
    url: API_ENDPOINTS.keycloakEndpoint,
    realm: 'myrealm',
    clientId: 'frontend-client',
});

keycloak.initPromise = keycloak.init({ onLoad: 'check-sso' });
```

Code 2.8: Create a new Keycloak client instance in Javascript

After initializing the Keycloak client, the authentication status and user information can be retrieved, as shown in Code 2.9. This example uses React hooks[16] to manage authentication state and user data.

[14] https://www.keycloak.org/getting-started/getting-started-zip
[15] https://www.keycloak.org/securing-apps/javascript-adapter
[16] https://react.dev/learn

```
    // Track authentication status
    const [authenticated, setAuthenticated] = useState(false);
    // Store authenticated user information
    const [userInfo, setUserInfo] = useState(null);
    // Initialize Keycloak instance and verify authentication
    useEffect(() => {
      keycloak.initPromise
          .then(authenticated => {
              setAuthenticated(authenticated);
              if (authenticated) {
                // Load user info if authenticated
                keycloak.loadUserInfo().then((user) => {
                  setUserInfo(user);
                });
              }
          })
          .catch(error => {
              // Log initialization errors
              console.error("Keycloak initialization failed", error);
          });
    }, []);
```

Code 2.9: Obtain the authentication information in Javascript

Once the user is authenticated, HTTP requests made from the front-end service, such as using the Axios library, must include the Keycloak-issued access token in the request headers. This ensures that the back-end services can verify the identity and permissions of the requester. Code 2.7 shows how to attach the token to an Axios request. This setup allows the frontend to securely communicate with protected back-end endpoints, leveraging Keycloak's identity and access management features without manually handling tokens or user session logic.

Back-end Client

In the back-end service, the logic for verifying access tokens is encapsulated in a middleware component. This approach ensures that authentication is handled consistently across protected endpoints before any business logic is executed. Code 2.10 illustrates the implementation of this middleware in Go programming language using the `gocloak` package, which provides Golang Keycloak API for interacting with the Keycloak server[17].

[17] `https://github.com/Nerzal/gocloak`

```go
func KeycloakAuthMiddleware() gin.HandlerFunc {
    return func(c *gin.Context) {
        // Get Authorization header
        authHeader := c.GetHeader("Authorization")
        if authHeader == "" {
            c.JSON(http.StatusUnauthorized,
                   gin.H{"error": "Authorization header missing"})
            c.Abort()
            return
        }
        // Extract the token from "Bearer <token>"
        tokenParts := strings.Split(authHeader, " ")
        if len(tokenParts) != 2 || tokenParts[0] != "Bearer" {
            c.JSON(http.StatusUnauthorized,
                   gin.H{"error": "Invalid Authorization header
    format"})
            c.Abort()
            return
        }
        tokenStr := tokenParts[1]
        // Decode and Validate Access Token
        ctx := context.Background()
        decodedToken, _, err :=
            client.DecodeAccessToken(ctx, tokenStr, realm)
        if err != nil || decodedToken == nil {
            c.JSON(http.StatusUnauthorized,
                   gin.H{"error": "Invalid or expired token"})
            c.Abort()
            return
        }

        c.Next()
    }
}
```

Code 2.10: Request Access Token Verify in Go

When a client makes a request to a protected API endpoint, it includes an `Authorization` header in the HTTP request. This header carries the access token in the form of a `Bearer` token, following the syntax:

`Authorization: Bearer <access_token>`

Here, `Bearer` is an authentication scheme that signals the server the client presents a token that must be validated. The access token itself is issued by the Keycloak authentication server, containing claims that prove the identity and privileges of the client.

For instance, a typical HTTP request to access a protected endpoint might look like the example shown in Code 2.11.

```
GET /api/protected/resource HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Accept: application/json
```

Code 2.11: Example HTTP Request with Bearer Token in Authorization Header

This middleware acts as a gatekeeper for protected API endpoints. It retrieves the `Bearer` token from the `Authorization` header of incoming HTTP requests, verifies its format, decodes, and validates the token using the Keycloak server. If the token is missing, malformed, invalid, or expired, the request is rejected with a `401 Unauthorized` response. Otherwise, the request proceeds to the next handler in the middleware chain.

This design promotes a clear separation of concerns: authentication logic is decoupled from the business logic of the API, enhancing maintainability, security, and readability in the back-end codebase.

This chapter detailed the development of the Experiment Logbook Management system, including problem analysis, system architecture, technology stack, and implementation strategies across front-end and back-end components. It also details the integration of Single Sign-On using Keycloak for authentication and authorization. With the foundational infrastructure for semantic experiment documentation now established, the next step involves integrating this platform with a dedicated Scientific Data Access system. This component focuses on enabling efficient, structured retrieval of experimental data through a specialized query engine. The following chapter explores the design and implementation of this data access system in detail.

3

Scientific Data Access

Efficient access to and processing of large volumes of diagnostic data are essential in experimental research. Experimental devices generate vast amounts of structured data across numerous shots, requiring researchers to search for effective tools to access, analyze, and extract meaningful insights from this information. The Query Engine system has been developed as an integrated component of the unified platform to provide a structured, extensible framework for managing and executing reusable data processing logic across multiple shots. This chapter presents the design and implementation of the Query Engine system.

3.1 Problem Clarification

Scientific research frequently involves analyzing large volumes of experimental data collected across numerous shots or runs. To derive meaningful insights, researchers often need to extract specific features, such as the maximum current, from these datasets. Manually implementing such analyses through custom scripts can be inefficient, error-prone, and difficult to reproduce. The Query Engine system, as illustrated in Figure 3.1, is designed to address these challenges by fulfilling several key functional and performance requirements essential for scalable, efficient, and collaborative scientific data analysis. The core system requirements are outlined as follows:

- Summary Publication Users can define and publish summaries (also referred as queries), which represent logical processing units. Each query consists of

a unique name, a textual description, dependencies on other queries, and an associated execution unit function. These queries are used to perform feature extraction across multiple shots.

- **Data Manipulation Operations** The system supports full lifecycle management of queries, including Create, Read, Update, and Delete. This enables users to iteratively develop and refine their analysis routines within a consistent framework.

- **Scalability Through Apache Spark** To enable efficient execution of queries across large numbers of experimental shots, the Query Engine leverages Apache Spark for distributed and parallel processing. This architecture allows queries to be executed concurrently over multiple shots, significantly reducing computation time. The use of Spark ensures high performance and scalability, making the system well-suited for data-intensive tasks such as long-term experimental campaigns and batch-processing workflows.

- **Result Caching Mechanism** To further optimize efficiency, the system incorporates a caching mechanism that stores the results of previously executed queries. When the same query is executed again with identical parameters, the system can serve results directly from the cache. This significantly reduces both response time and computational cost, especially for frequently used or computationally expensive queries.

- **User Interfaces and Client Package** Users interact with the Query Engine via a web-based interface or a Python client, providing flexibility for both graphical and programmatic access. These interfaces simplify publishing, executing, and managing queries without requiring detailed knowledge of the underlying infrastructure.

In summary, the Query Engine abstracts the complexity of working with shot-based scientific data by offering a standardized, high-performance, and user-friendly platform. It promotes reproducibility, collaboration, and efficiency in data-driven scientific research.

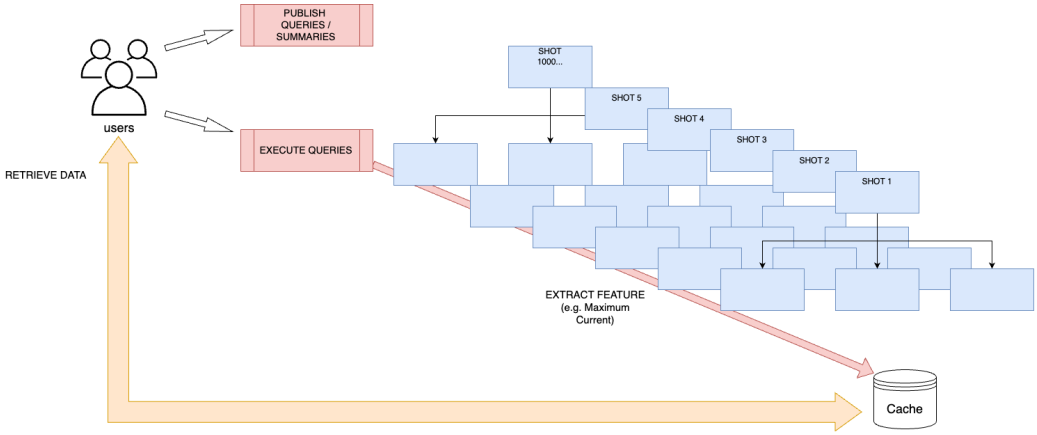


Figure 3.1: Query Engine requirements

3.2 Technical Background

3.2.1 System Design

Figure 3.2 illustrates the high-level architecture of the Query Engine system.

The Client component offers both a web-based user interface and a Python API. These interfaces enable researchers to create, manipulate, and execute queries either through an intuitive graphical interface or programmatically, supporting flexible workflows tailored to different user preferences and use cases.

At the core of the system is the Backend, implemented in Python, which acts as the central coordinator. It handles user requests, manages query parsing and manipulation, orchestrates query execution, and communicates with both the underlying database and the distributed computing cluster to perform data processing tasks efficiently.

The computational engine is a Spark Standalone Cluster comprising a Spark Master node and multiple Spark Worker nodes. This architecture facilitates scalable and parallel processing of large volumes of experimental data across multiple shots. Apache Spark provides fault tolerance and efficient concurrent, distributed task scheduling, which are essential for ensuring high performance and reliability in big data processing environments.

Experimental data is stored in MDSplus Storage Servers, which provide a hierarchical and efficient data management system tailored to experiments. Spark Workers act as Distributed Clients of MDSplus, retrieving raw data directly from these servers and performing processing tasks according to the user queries.

To optimize system responsiveness and reduce redundant computation, the database is leveraged as a caching layer. When a query is executed, its results along with the input parameters are stored in the cache. Subsequent identical queries can then be served directly from the cache, significantly lowering response times and computational resource consumption.

Furthermore, the offloading of data analysis to the Spark cluster enables targeted execution of specific data processing tasks on nodes with specialized hardware or software capabilities. For example, certain Spark Worker nodes may be equipped with GPUs or optimized libraries to accelerate machine learning algorithms or other computationally intensive operations. This flexible assignment enhances the system's adaptability and overall throughput by exploiting heterogeneous computing resources.

Before delving into the implementation details, we will provide an overview of the key technologies involved in the system, including MDSplus Data Management System, Apache Spark, and the Backend technologies.

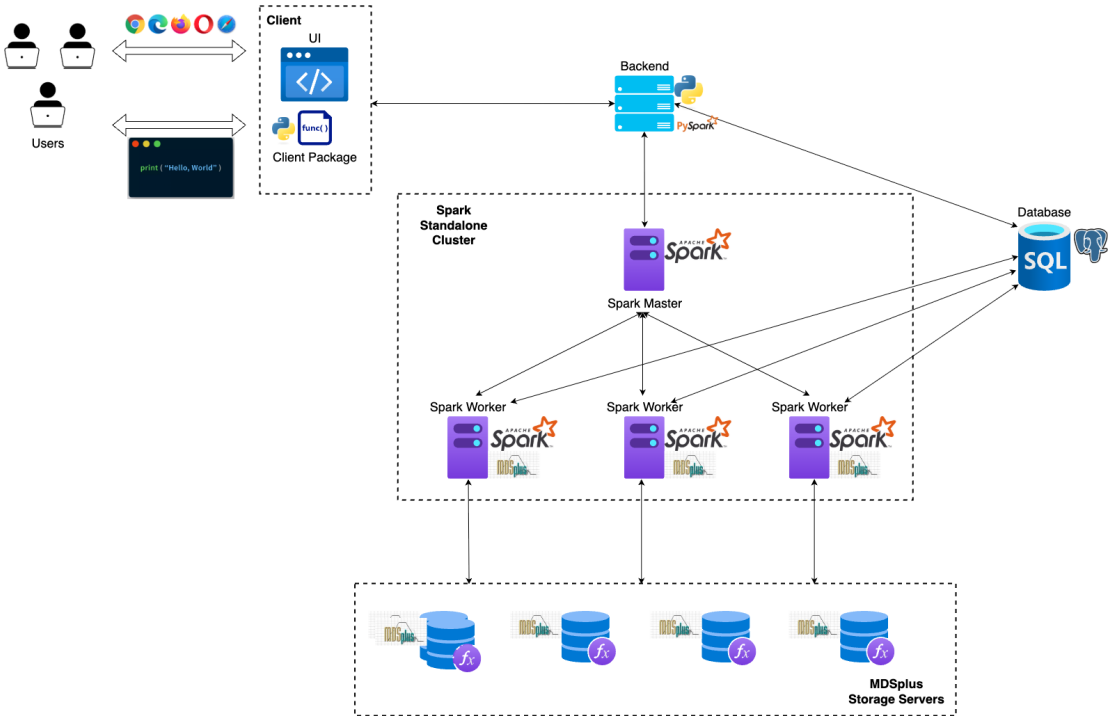


Figure 3.2: Query Engine System Design

3.2.2 MDSplus data management system

MDSplus [22] is a set of software tools for data acquisition and storage and a methodology for management of complex scientific data. MDSplus allows all data from an experiment or simulation code to be stored into a single, self-descriptive, hierarchical structure. The system was designed to enable users to easily construct complete and coherent data sets.
The MDSplus programming interface contains only a few basic commands, simplifying data access even into complex structures. Using the client/server model, data at remote sites can be read or written without file transfers. MDSplus includes x-windows and java tools for viewing data or for modifying or viewing the underlying structures.
Developed jointly by the Massachusetts Institute of Technology, the Fusion Research Group in Padua, Italy (Istituto Gas Ionizzati and Consorzio RFX), and the Los Alamos National Lab, MDSplus is the most widely used system for data management in the magnetic fusion energy program[1].

MDSplus database

The basic data structure of MDSplus is a self-descriptive hierarchy called a *Tree*. The hierarchy consists of large numbers of named *Nodes* which make up the branches (structure) and leaves (data) of each tree. MDSplus *Shots* are trees created from a special type of tree called a *Model*, a template which contains all of the structure and setup data for an experiment or code. *Shots* are copies of the model augmented by the stored data and correspond to particular runs of an experiment or code. The experiment model is filled with experimental data, eventually producing a database containing the complete description of one experiment – referred to as the *Pulse File*.
For a typical experiment, data from various sources are grouped in some logical manner and divided into a number of trees which each form the top level of their respective hierarchies. These trees themselves can be organized into a hierarchy with a root tree and *Subtrees* as in the Figure 3.3[2].

[1] https://www.mdsplus.org/index.php?title=Introduction&open=1015831953252722283 51&page=Introduction

[2] https://www.mdsplus.org/index.php?title=Documentation:Tutorial:CreateTrees&open=188155325341526597252547 19&page=Tutorials%2FTrees+%26+Data

The structure of experiment models and pulse files can be more complicated, and given subtrees can be stored in different triplets of files, possibly on remote machines[3].

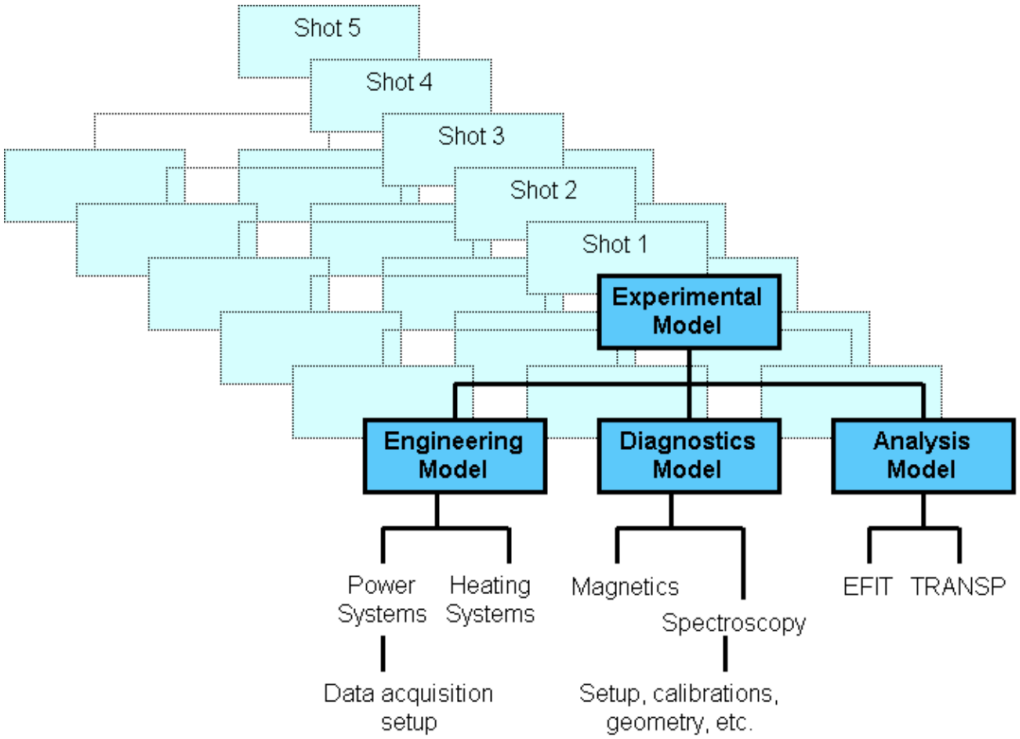


Figure 3.3: MDSplus Data Hierarchy

Remote Data Access

In MDSplus remote data access is natively available. Data communication is based on TCP/IP and uses a mdsip[4] protocol. A mdsip server has to be running at the data server side (i.e. where the pulse files are hosted).
The data client then communicates with the data server to retrieve or store data. There are three possible configurations for the data client:

- Distributed Client

[3] https://www.mdsplus.org/index.php?title=Documentation:Tutorial:QuickOverview&open=1015831942947485127 53&page=Documentation%2FTutorials%2FQuick+Tour

[4] https://www.mdsplus.org/index.php/Documentation:Reference:MDSIP

- Thick Client
- Thin Client

Within the scope of this project, we adopt the Distributed Client approach. In the distributed client configuration most data access operations are carried out at the client site, except for the disk I/O operations which are demanded to the server. In this configuration no changes are required in the code, being data redirection carried out only by re-defining the environment variable `<experiment name>_path`. More precisely, the variable can contain a search list, separated by semicolons (not colons as for Linux search path). Every element of the search list is composed of three elements in the form `<mdsip server IP address>:<port>::<directory>`. The first element is the IP address of the machine hosting the pulse files and running the mdsip server. The second part is optional and specifies the port number of the mdsip server. The third part is the directory containing the pulse files[5].

Object Oriented Interface

MDSplus provided in the past an Application Programming Interface (API) for handling data in pulse files in Fortran and C. While the Fortran API is limited to reading and writing data, the C API is much more complete and allows full data and pulse file structure management (the kernel of MDSplus is based on it). However the C API is very complex, and uses a very large set of data structures, called descriptors, to handle the variety of data types available in MDSplus. For this reason, a new Object Oriented multi language API has been developed in 2008 in order to provide full data and pulse file manipulation using a much simpler interface. Currently three Object Oriented languages are supported in MDSplus: C++, Java and Python[6].
In this work, the Python interface is used for its simplicity and readability, making it well-suited for scripting and data processing tasks. Code 3.1 demonstrates a basic example of how data can be accessed and manipulated using the Python

[5] https://www.mdsplus.org/index.php/Documentation:Tutorial:RemoteAccess

[6] https://www.mdsplus.org/index.php?title=Documentation:Tutorial:MdsObjects&open=381018323181698431624150 89&page=Documentation%2FThe+MDSplus+tutorial%2FThe+Object+Oriented+interface+of+MDSPlus#Introduction

interface. In the example, the content of tree node `NODE1` is copied into node `NODE2` within the experiment model `my_tree`.

```
myTree = Tree('my_tree', -1)
n1 = myTree.getNode('NODE1')
n2 = myTree.getNode('NODE2')
d = n1.getData()
n2.putData(d)
```

Code 3.1: Example of MDSplus Data Access in Python

3.2.3 Apache Spark

Apache Spark[7] is an exciting new technology that is rapidly superseding Hadoop's MapReduce as the preferred big data processing platform. Hadoop is an open source, distributed, Java computation framework consisting of the Hadoop Distributed File System (HDFS) and MapReduce, its execution engine [26]. Spark is similar to Hadoop in that it's a distributed, general-purpose computing platform. But Spark's unique design, which allows for keeping large amounts of data in memory, offers tremendous performance improvements. Spark programs can be 100 times faster than their MapReduce counterparts [27].
By using Spark's elegant API and runtime architecture, you can write distributed programs in a manner similar to writing local ones. Spark's collections abstract away the fact that they're potentially referencing data distributed on a large number of nodes. Spark also allows you to use functional programming methods [14], which are a great match for data-processing tasks.
By supporting Python, Java, Scala, and, most recently, R, Spark is open to a wide range of users: to the science community that traditionally favors Python and R, to the still-widespread Java community, and to people using the increasingly popular Scala, which offers functional programming on the Java Virtual Machine (JVM) [27].
Finally, Spark combines MapReduce-like capabilities for batch programming, real-time data-processing functions, SQL-like handling of structured data, graph algorithms, and machine learning, all in a single framework. This makes it a one-stop shop for most of your big data-crunching needs. It's no wonder, then,

[7] `https://spark.apache.org/`

that Spark is one of the busiest and fastest-growing Apache Software Foundation projects today [27].
Spark consists of several purpose-built components. These are Spark Core, Spark SQL, Spark Streaming, Spark GraphX, and Spark MLlib, as shown in Figure 3.4 [27].

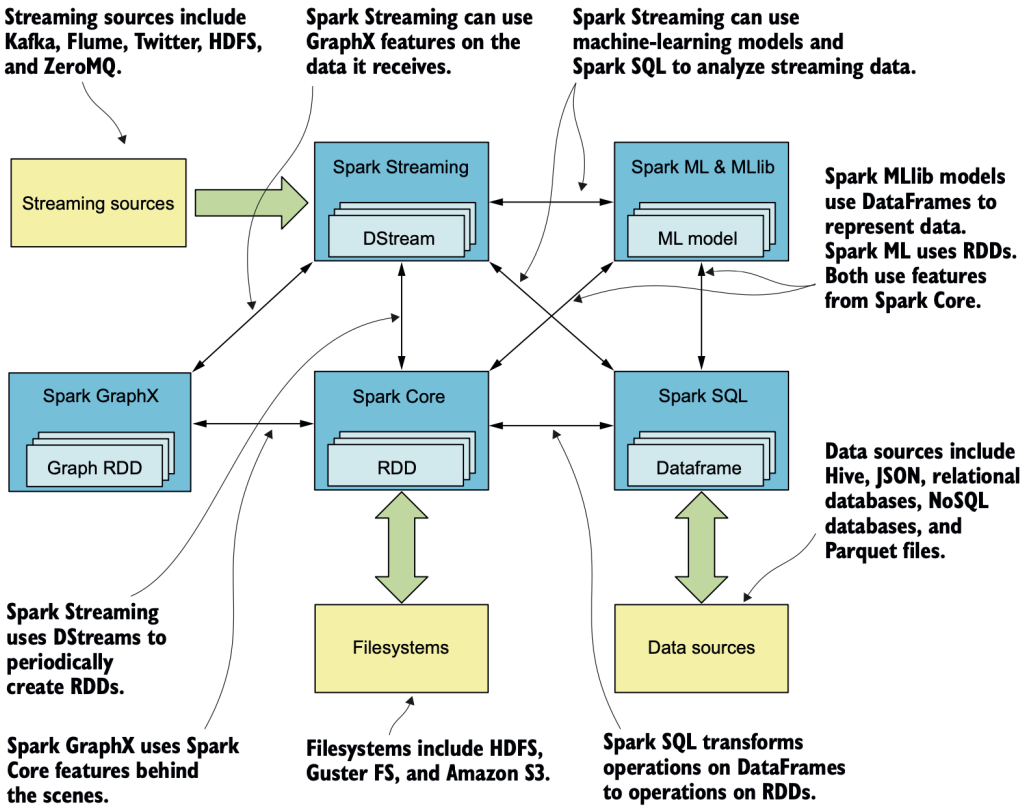


Figure 3.4: Main Spark Components

Within the scope of this project, we will focus only on the Spark Core component. Spark Core contains basic Spark functionalities required for running jobs and needed by other components. The most important of these is the Resilient Distributed Dataset (RDD)[8], which is the main element of the Spark API. It's an abstraction of a distributed collection of items with operations and transformations applicable to the dataset. It's resilient because it's capable of rebuilding datasets in case of node failures [27].

[8] `https://spark.apache.org/docs/3.5.5/api/java/org/apache/spark/rdd/RDD.html`

Spark Fundamentals

The RDD is the fundamental abstraction in Spark. It represents a collection of elements that is [27]:

- **Immutable:** RDDs support a number of transformations that do useful data manipulation, but they always yield a new RDD instance. Once created, RDDs never change; thus the adjective immutable. Mutable state is known to introduce complexity, but besides that, having immutable collections allows Spark to provide important fault-tolerance guarantees in a straightforward manner.
- **Resilient:** RDDs are resilient because of Spark's built-in fault recovery mechanics. Spark is capable of healing RDDs in case of node failure. Whereas other distributed computation frameworks facilitate fault tolerance by replicating data to multiple machines (so it can be restored from healthy replicas once a node fails), RDDs are different: they provide fault tolerance by logging the transformations used to build a dataset (how it came to be) rather than the dataset itself. If a node fails, only a subset of the dataset that resided on the failed node needs to be recomputed.
- **Distributed:** The fact that the collection is distributed on a number of machines (execution contexts, JVMs) is transparent to its users, so working with RDDs isn't much different than working with ordinary local collections like plain old lists, maps, sets, and so on. To summarize, the purpose of RDDs is to facilitate parallel operations on large datasets in a straightforward manner, abstracting away their distributed nature and inherent fault tolerance.

There are two ways to create RDDs: parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat. For this project, we will focus only on Parallelized Collections.

Parallelized Collections

Parallelized collections[9] are created by calling SparkContext's parallelize method

[9] `https://spark.apache.org/docs/latest/rdd-programming-guide.html#parallelized-collections`

on an existing iterable or collection in your driver program. The elements of the collection are copied to form a distributed dataset that can be operated on in parallel. For example, Code 3.2 shows how to create a parallelized collection holding the numbers 1 to 5:

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

Code 3.2: Example of Parallelized Collections

Once created, the distributed dataset (`distData`) can be operated on in parallel. For example, we can call `distData.reduce(lambda a, b: a + b)` to add up the elements of the list. We describe operations on distributed datasets later on.

One important parameter for parallel collections is the number of *partitions* to cut the dataset into. Spark will run one task for each partition of the cluster. Typically you want 2 - 4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to `parallelize` (for example, `sc.parallelize(data, 10)`). Note: some places in the code use the term slices (a synonym for partitions) to maintain backward compatibility.

RDD Operations

RDDs support two types of operations: *transformations*, which create a new dataset from an existing one, and *actions*, which return a value to the driver program after running a computation on the dataset. For example, `map` is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, `reduce` is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program (although there is also a parallel `reduceByKey` that returns a distributed dataset).

All transformations in Spark are *lazy*, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (such as a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently. For example, we can realize that a dataset created through `map` will be used in a `reduce` and return only the result of the `reduce` to

the driver, rather than the larger mapped dataset[10]. Table 3.1 lists some common transformations supported by Spark, while Table 3.2 lists some of the common actions.

Transformation	**Meaning**
`map(func)`	Return a new distributed dataset formed by passing each element of the source through a function `func`.
`filter(func)`	Return a new dataset formed by selecting those elements of the source on which `func` returns true.
`groupByKey([numPartitions])`	When called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Iterable<V>)` pairs.
`reduceByKey(func, [numPartitions])`	When called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function `func`, which must be of type `(V,V) => V`. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
`repartition(numPartitions)`	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

Table 3.1: Common RDD Transformations

Transformations can be classified as having either narrow dependencies or wide dependencies. Any transformation where a single output partition can be computed from a single input partition is a narrow transformation [4]. For example, `map()` and `filter()` represent narrow transformations because they can operate on a single partition and produce the resulting output partition without any exchange of data. However, `groupBy()` or `orderBy()` instruct Spark to perform wide transformations, where data from other partitions is read in, combined, and written to disk (`groupBy()`) will force a shuffle of data from each of the ex-

[10] `https://spark.apache.org/docs/latest/rdd-programming-guide.html`

Action	Meaning
`reduce(func)`	Aggregate the elements of the dataset using a function `func` (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
`collect()`	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
`collectAsMap()`	Similar to `collect()` but return the key-value pairs in this RDD to the master as a dictionary.

Table 3.2: Common RDD Actions

ecutor's partitions across the cluster. In this transformation, `orderBy()` requires output from other partitions to compute the final aggregation).
Figure 3.5 illustrates the two types of dependencies.

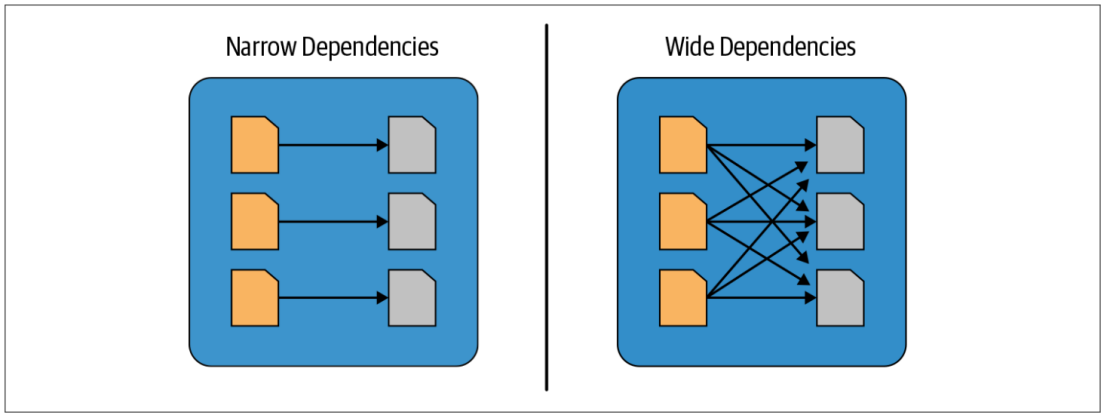


Figure 3.5: Narrow versus Wide transformations

Spark Applications

To understand what's happening under the hood with our sample code, we will need to be familiar with some of the key concepts of a Spark application and how the code is transformed and executed as tasks across the Spark executors. We'll begin with important terms [4]:

- **Application** A user program built on Spark using its APIs. It consists of a driver program and executors on the cluster.

- **Spark Driver** As the part of the Spark application responsible for instantiating a `SparkSession`, the Spark driver has multiple roles: it communicates with the cluster manager; it requests resources (CPU, memory, etc.) from the cluster manager for Spark's executors (JVMs); and it transforms all the Spark operations into DAG computations, schedules them, and distributes their execution as tasks across the Spark executors. Once the resources are allocated, it communicates directly with the executors.

- **`SparkSession`** An object that provides a point of entry to interact with underlying Spark functionality and allows programming Spark with its APIs. In an interactive Spark shell, the Spark driver instantiates a `SparkSession` for you, while in a Spark application, you create a `SparkSession` object yourself. In Spark 2.0, the SparkSession became a unified conduit to all Spark operations and data. Not only did it subsume previous entry points to Spark like the SparkContext, SQLContext, HiveContext, SparkConf, and StreamingContext, but it also made working with Spark simpler and easier.

- **Cluster manager** The cluster manager is responsible for managing and allocating resources for the cluster of nodes on which your Spark application runs. Currently, Spark supports four cluster managers: the built-in standalone cluster manager, Apache Hadoop YARN, Apache Mesos, and Kubernetes.

- **Spark executor** A Spark executor runs on each worker node in the cluster. The executors communicate with the driver program and are responsible for executing tasks on the workers. In most deployments modes, only a single executor runs per node.

- **Job** A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (such as `save()`, `collect()`).

- **Stage** Each job gets divided into smaller sets of tasks called stages that depend on each other.

- **Task** A single unit of work or execution that will be sent to a Spark executor.

Spark Distributed Execution Components and Architecture

At a high level in the Spark architecture, a Spark application consists of a driver program that is responsible for orchestrating parallel operations on the Spark cluster. The driver accesses the distributed components in the cluster, the Spark

executors and cluster manager, through a `SparkSession`[11]. Figure 3.6 shows the Apache Spark components and architecture.

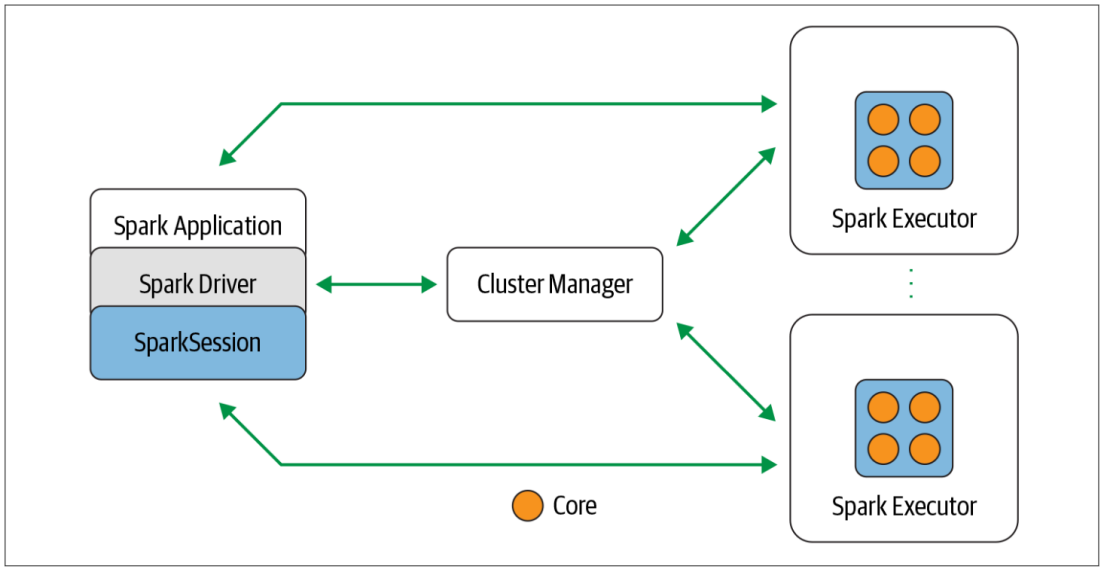


Figure 3.6: Apache Spark components and architecture

Partitioning allows for efficient parallelism. Each executor's core is assigned its own data partition to work on (Figure 3.7) [4].

Spark Jobs

During interactive sessions with Spark shells, the driver converts your Spark application into one or more Spark jobs (Figure 3.8). It then transforms each job into a Directed Acyclic Graph (DAG). This, in essence, is Spark's execution plan, where each node within a DAG could be a single or multiple Spark stages [4].

Spark Stages

As part of the DAG nodes, stages are created based on what operations can be performed serially or in parallel (Figure 3.9). Not all Spark operations can happen in a single stage, so they may be divided into multiple stages. Often stages are delineated on the operator's computation boundaries, where they dictate data transfer among Spark executors [4].

[11] `https://www.oreilly.com/library/view/learning-spark-2nd/9781492050032/ch01.html`

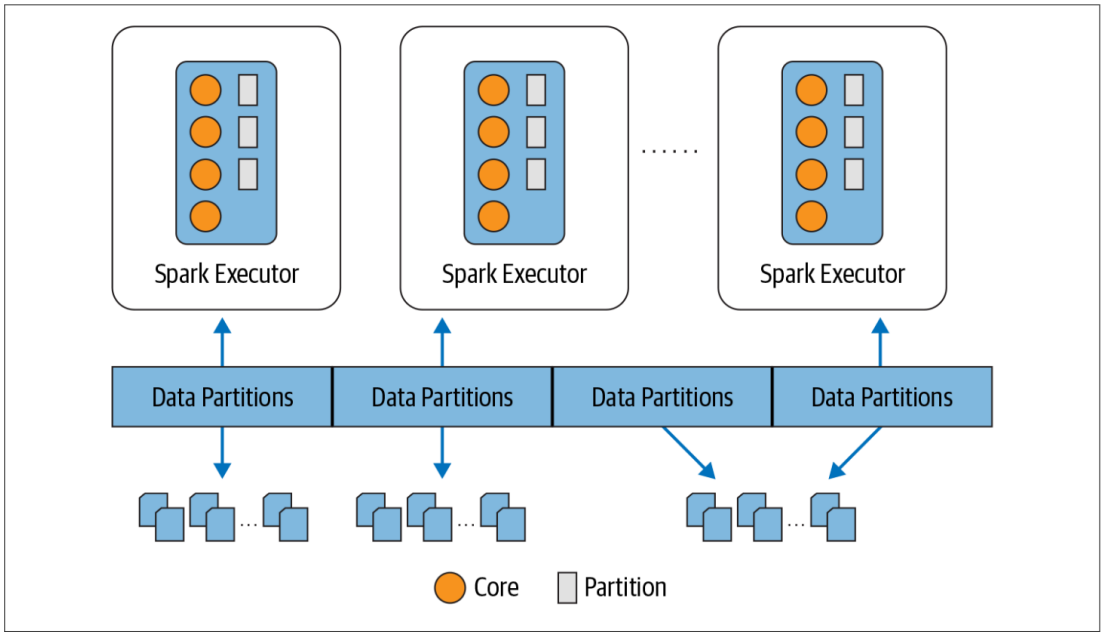


Figure 3.7: Assignment of Spark's partitions and cores

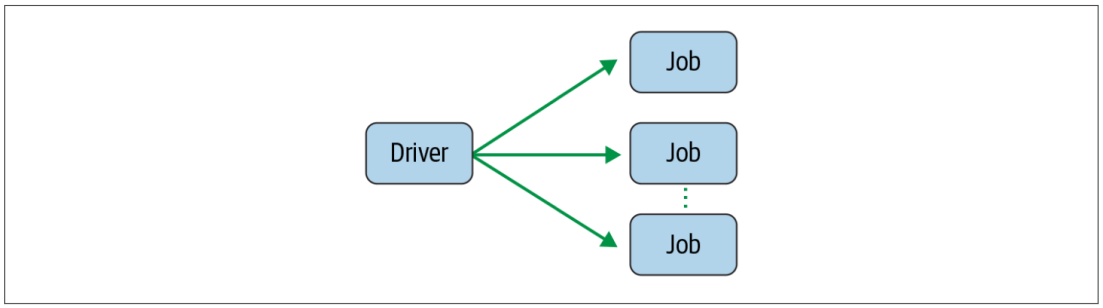


Figure 3.8: Spark driver creates Spark jobs

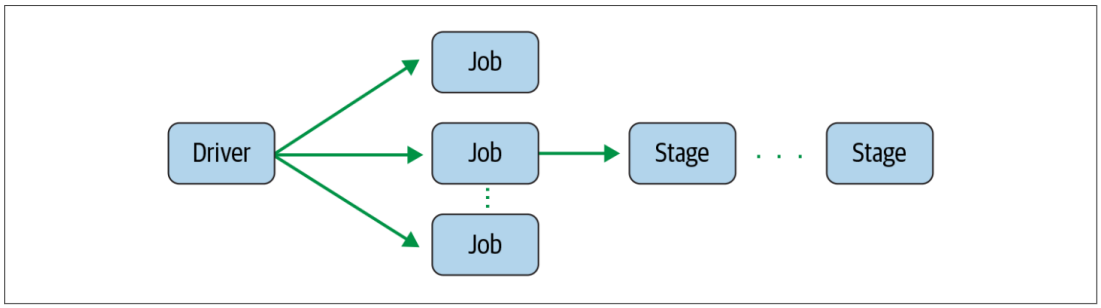


Figure 3.9: Spark job creates Spark stages

Spark Tasks

Each stage is comprised of Spark tasks (a unit of execution), which are then federated across each Spark executor; each task maps to a single core and works on a single partition of data (Figure 3.10). As such, an executor with 16 cores can have 16 or more tasks working on 16 or more partitions in parallel, making the execution of Spark's tasks exceedingly parallel [4].

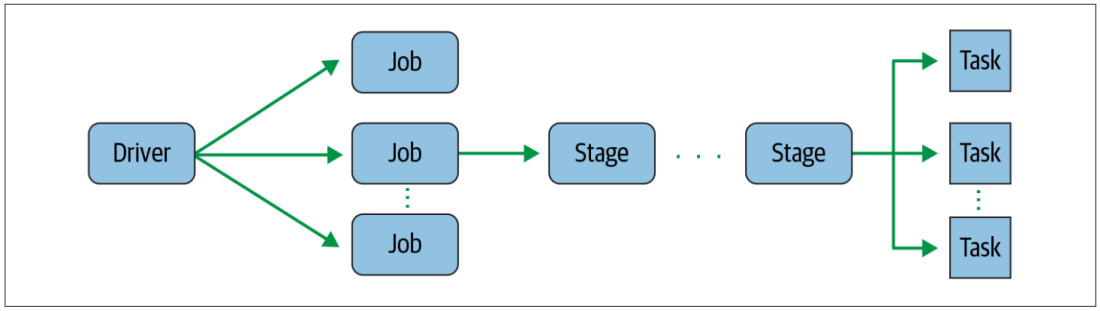


Figure 3.10: Spark stage creates Spark tasks

Spark Cluster

The physical placement of executor and driver processes depends on the cluster type and its configuration. For example, some of these processes could share a single physical machine, or they could all run on different ones. There are two basic ways the driver program can be run [27].

- **Cluster-deploy mode** In this mode, the driver process runs as a separate JVM process in a cluster, and the cluster manages its resources (mostly JVM heap memory) (Figure 3.11).
- **Client-deploy mode** In this mode, the driver is running in the client's JVM process and communicates with the executors managed by the cluster (Figure 3.12).

Spark can run in local mode and in Spark Standalone, YARN, and Mesos clusters[12]. Within the scope of this project, we will examine only the Spark Standalone Cluster.

[12] `https://spark.apache.org/docs/latest/cluster-overview.html`

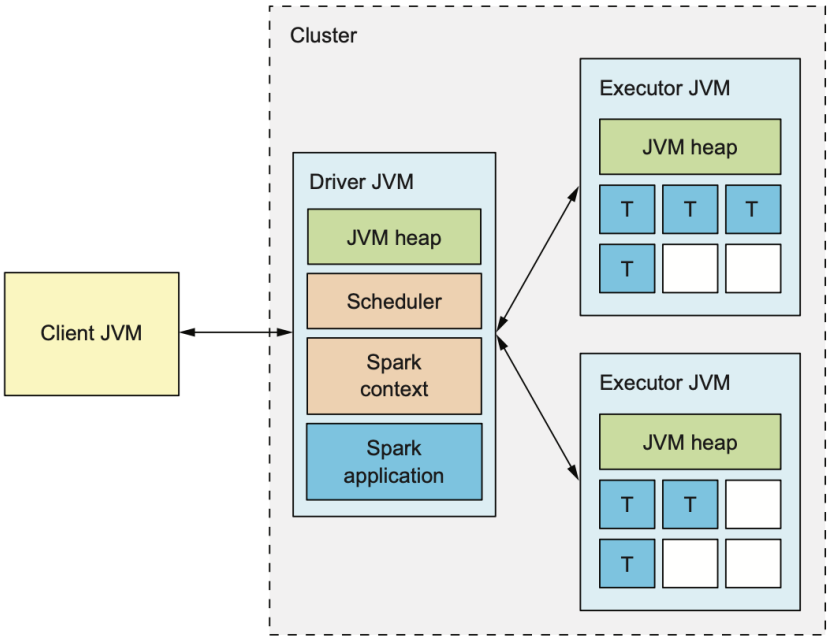


Figure 3.11: Spark runtime components in Cluster-Deploy mode

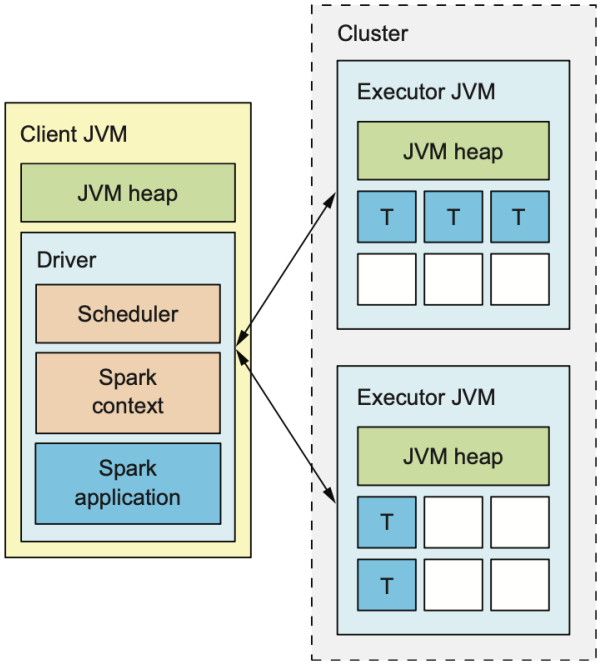


Figure 3.12: Spark runtime components in Client-Deploy mode

Spark Standalone Cluster Components

A standalone cluster comes bundled with Spark. It has a simple architecture and is easy to install and configure. Because it was built and optimized specifically for Spark, it has no extra functionalities with unnecessary generalizations, requirements, and configuration options, each with its own bugs. In short, the Spark standalone cluster is simple and fast [27].

The standalone cluster consists of master and worker (also called slave) processes. A master process acts as the cluster manager. It accepts applications to be run and schedules worker resources (available CPU cores) among them. Worker processes launch application executors (and the driver for applications in cluster-deploy mode) for task execution. To refresh your memory, a driver orchestrates and monitors execution of Spark jobs, and executors execute a job's tasks [27].

Both masters and workers can be run on a single machine, but this isn't how a Spark standalone cluster usually runs. You normally distribute workers across several nodes to avoid reaching the limits of a single machine's resources. Naturally, Spark has to be installed on all nodes in the cluster in order for them to be usable as slaves [27].

Figure 3.13 shows an example Spark standalone cluster running on two nodes with two workers [27].

- **Step 1:** A client process submits an application to the master.
- **Step 2:** The master instructs one of its workers to launch a driver.
- **Step 3:** The worker spawns a driver JVM.
- **Step 4:** The master instructs both workers to launch executors for the application.
- **Step 5:** The workers spawn executor JVMs.
- **Step 6:** The driver and executors communicate independent of the cluster's processes.

As with the other cluster types, you can run Spark programs on a standalone cluster by submitting them with the spark-submit command, running them in a Spark shell, or instantiating and configuring a `SparkContext` object in your own application. In all three cases, you need to specify a master connection

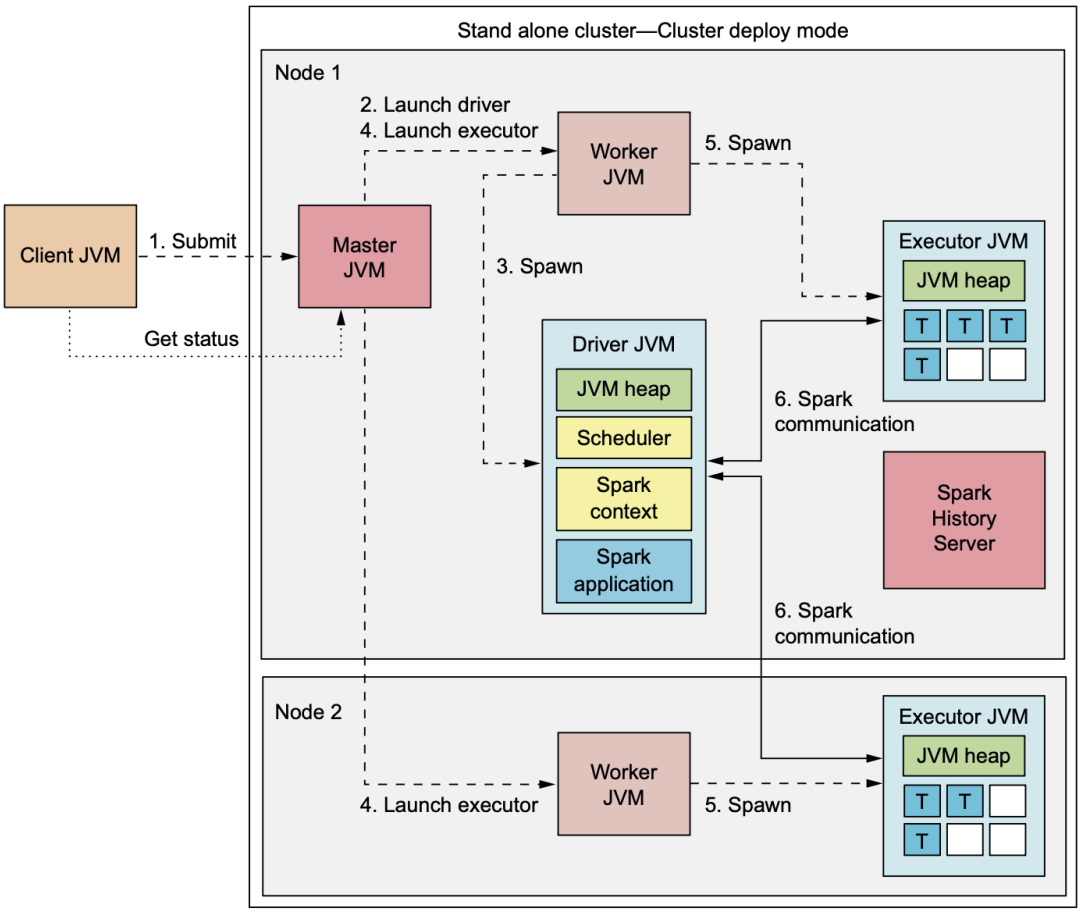


Figure 3.13: A Spark Standalone Cluster with an application in cluster-deploy mode

URL with the hostname and port of the master process. If you're embedding `SparkContext` in your application and you're not using the `spark-submit` script to connect to the standalone cluster, there's currently no way to specify deploy mode. It will default to client-deploy mode, and the driver will run in your application [27].

3.2.4 Back-end Technologies

The Backend in the Query Engine system is responsible for handling client requests. It acts as the intermediary between the clients and the core system, providing endpoints for creating, reading, updating, and deleting queries, as well as managing their execution. By leveraging Python and its ecosystem including Flask for web server, SQLAlchemy for database interaction, and PyS-

park for performing Apache Spark operations, the Backend is developed to be modular, maintainable, and capable of efficiently handling the client requests.

Python

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.
Python was selected as the back-end service language for this project due to several key advantages:

- Python is a popular choice for web development due to its readability, extensive libraries and frameworks, and a large community that provides support and resources.
- An MDSplus package is available for the Python language which defines many classes for accessing MDSplus data repositories (trees/pulse files) and manipulating the many advanced data types provided by MDSplus.
- Python is a strong choice for working with Apache Spark, particularly through its PySpark API. PySpark offers a good balance of ease of use and powerful data processing capabilities.

Flask

Flask[13] is a lightweight Web Server Gateway Interface (WSGI) [9] web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications.
Flask is a popular choice for web application development in Python due to its flexibility, ease of use, and ability to scale as needed

- **Lightweight and Scalable** Flask's minimal core means it's easy to learn and use for simple projects, and its modular design allows for easy expansion as the application grows.

[13] `https://flask.palletsprojects.com/en/stable/`

- **Flexibility** Unlike more opinionated frameworks, Flask doesn't force a specific architecture or set of libraries, giving developers more control and the freedom to choose the tools that best suit their needs.
- **Beginner-Friendly** The simple syntax and minimal setup make Flask a good choice for those new to web development, while the flexibility allows experienced developers to customize the framework to their preferences.
- **Integration** Flask easily integrates with various extensions, making it easy to add features like databases, user authentication, and more.
- **Community and Documentation** Flask has a strong and active community that provides support and resources, as well as extensive documentation.

SQLAlchemy

SQLAlchemy[14] is the Python SQL toolkit and Object Relational Mapping [13] that gives application developers the full power and flexibility of SQL.
It provides a full suite of well known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple and Pythonic domain language.
SQL databases behave less like object collections the more size and performance start to matter; object collections behave less like tables and rows the more abstraction starts to matter. SQLAlchemy aims to accommodate both of these principles.
SQLAlchemy considers the database to be a relational algebra engine, not just a collection of tables. Rows can be selected from not only tables but also joins and other select statements; any of these units can be composed into a larger structure. SQLAlchemy's expression language builds on this concept from its core.
SQLAlchemy is most famous for its object-relational mapper, an optional component that provides the data mapper pattern, where classes can be mapped to the database in open ended, multiple ways – allowing the object model and database schema to develop in a cleanly decoupled way from the beginning[21].

[14] `https://www.sqlalchemy.org/`

PySpark

PySpark[15] is the Python API for Apache Spark. It enables you to perform real-time, large-scale data processing in a distributed environment using Python. It also provides a PySpark shell for interactively analyzing your data.
PySpark combines Python's learnability and ease of use with the power of Apache Spark to enable processing and analysis of data at any size for everyone familiar with Python.
PySpark supports all of Spark's features such as Spark SQL, DataFrames, Structured Streaming, Machine Learning (MLlib) and Spark Core (Figure 3.14).

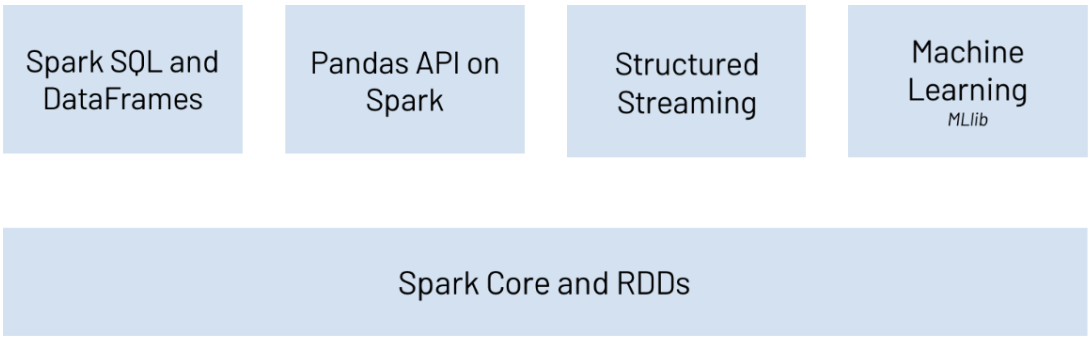


Figure 3.14: PySpark features

Apache Spark is written in Scala programming language. PySpark has been released in order to support the collaboration of Apache Spark and Python, it actually is a Python API for Spark. In addition, PySpark, helps you interface with Resilient Distributed Datasets in Apache Spark and Python programming language. This has been achieved by taking advantage of the Py4j library.
Py4J[16] is a popular library which is integrated within PySpark and allows python to dynamically interface with JVM objects. PySpark features quite a few libraries for writing efficient programs. Furthermore, there are various external libraries that are also compatible[17].

[15] https://spark.apache.org/docs/latest/api/python/index.html
[16] https://www.py4j.org/
[17] https://www.databricks.com/glossary/pyspark

3.3 Development Details

This section provides the necessary details on how each component is implemented and integrated to form the Query Engine system.

3.3.1 Data Model

Figure 3.15 presents the Entity Relationship (ER) Diagram of the Query Engine Data Model. It consists of two tables: `queries` and `execution_units`. The `queries` table represents the individual queries, including their unique name (`queryName` field), their dependencies (`dependencies` field), a descriptive text (`queryDescription` field), and the execution logic function to be executed for each shot (`executionUnitFunction` field). The `execution_units` table caches specific executions of these queries, storing the resulting data for each shot. A foreign key constraint ensures referential integrity between these two tables, linking `execution_units.queryName` to `queries.queryName`.

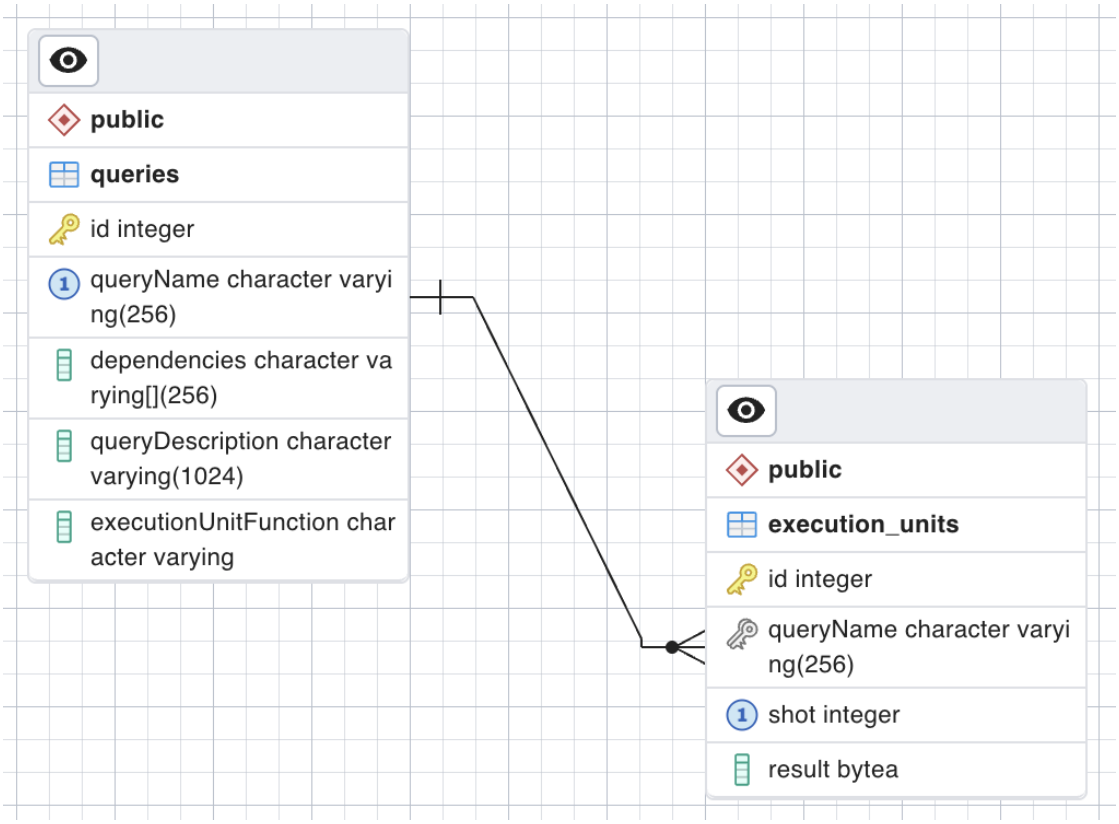


Figure 3.15: Entity Relationship Diagram of Query Engine Data Model

Using SQLAlchemy, the corresponding data models are declared as shown in Code 3.3 and Code 3.4.

```python
class Query(db.Model):
    __tablename__ = 'queries'
    id = db.Column(db.Integer, primary_key=True)
    queryName = db.Column(db.String(256), unique=True, nullable=False)
    dependencies = db.Column(ARRAY(db.String(256)), nullable=False)
    queryDescription = db.Column(db.String(1024))
    executionUnitFunction = db.Column(db.String)

    def to_dict(self):
        return {
            "id": self.id,
            "queryName": self.queryName,
            "dependencies": ", ".join(self.dependencies),
            "queryDescription": self.queryDescription,
            "executionUnitFunction": self.executionUnitFunction
        }
```

Code 3.3: Query Model Declaration with SQLAlchemy

```python
class ExecutionUnit(db.Model):
    __tablename__ = 'execution_units'
    id = db.Column(db.Integer, primary_key=True)
    queryName = db.Column(db.String(256), db.ForeignKey('queries.queryName', ondelete="CASCADE"), nullable=False)
    shot = db.Column(db.Integer, nullable=False)
    result = db.Column(db.LargeBinary, nullable=False)
    __table_args__ = (UniqueConstraint('queryName', 'shot', name='unique_queryName_shot'),)

    def to_dict(self):
        return {
            "id": self.id,
            "queryName": self.queryName,
            "shot": self.shot,
            "result": self.result
        }
```

Code 3.4: ExecutionUnit Model Declaration with SQLAlchemy

3.3.2 APIs Implementation

For the Create - Read - Update - Delete (CRUD) operations on queries, we adopt the RESTful API request-response pattern, consistent with our earlier approach. Similarly, the API for query execution is implemented by defining appropriate endpoints and suitable HTTP methods.
We will now examine the implementation details, focusing on routing the requests and handling request-response flows with Flask.

API routers Using Flask's `Blueprint`[18] feature, we declare API routers for CRUD operations on queries, as shown in Code 3.5. For brevity, the detailed implementation logic of these endpoints is not included here, but follows standard practices in Flask development.

```
query_blueprint = Blueprint('query', __name__)

@query_blueprint.route('/query-engine/queries', methods=['GET'])
def getAllQueries():
    # Implementation

@query_blueprint.route('/query-engine/queries', methods=['POST'])
def createQuery():
    # Implementation

@query_blueprint.route('/query-engine/queries/<int:id>', methods=['
    PUT'])
def updateQuery(id):
    # Implementation

@query_blueprint.route('/query-engine/queries/<int:id>', methods=['
    DELETE'])
def deleteQuery(id):
    # Implementation
```

Code 3.5: CRUD APIs routers implementation in Flask

Similarly, Code 3.6 demonstrates how we implement API routers for query execution. For brevity, the detailed implementation logic of these endpoints is not included here, but follows standard practices in Flask development.

[18] https://flask.palletsprojects.com/en/stable/blueprints/

```python
execute_query_blueprint = Blueprint('execute_query', __name__)

@execute_query_blueprint.route('/query-engine/executeQuery/execute',
    methods=['POST'])
def execute():
    # Implementation
```

Code 3.6: Query Execution routers implementation in Flask

Handling request-response Code 3.7 and Code 3.8 demonstrate the implementation of creating a new query in detail. The route handler receives the request payload in JSON format, extracts the relevant fields, and delegates the creation process to the `QueryService` class. The service layer encapsulates the database logic, ensuring that the new query is added to the database and handling transaction management. Note that this is just one example of an API endpoint among many others implemented in the system.

```python
@query_blueprint.route('/queries', methods=['POST'])
def createQuery():
    try:
        # Get JSON data from the request
        data = request.get_json()
        queryName = data["queryName"]
        dependencies = [item.strip() for item in data["dependencies"
    ].split(',')]
        queryDescription = data["queryDescription"]
        executionUnitFunction = data["executionUnitFunction"]

        query = QueryService.createQuery(queryName, dependencies,
    queryDescription, executionUnitFunction)
        return jsonify(query.to_dict()), 201
    except Exception as e:
        return jsonify({"error": e}), 500
```

Code 3.7: Router implementation for creating a query

The implementation for executing a query will be discussed in detail in the following subsection.

3.3.3 Query Execution Workflow

Code 3.9 illustrates the implementation for handling the execution of a query.

```python
class QueryService:
    @staticmethod
    def createQuery(queryName, dependencies, queryDescription, executionUnitFunction):
        try:
            query = Query(queryName=queryName, dependencies=dependencies, queryDescription=queryDescription, executionUnitFunction=executionUnitFunction)
            db.session.add(query)
            db.session.commit()
            return query
        except Exception as e:
            db.session.rollback()
            raise ValueError(f"An unexpected error occurred: {str(e)}")
```

Code 3.8: Service layer implementation for creating a query

```python
@execute_query_blueprint.route('/api/executeQuery/execute', methods=['POST'])
def execute():
    queryName = request.args.get('queryName', type=str)
    query = QueryService.getQueryByName(queryName)
    if query is None:
        abort(404)

    queryInput = getQueryInput(request)

    results = QueryExecutor.execute(sparkContext=sparkContext, query=query, queryInput=queryInput)

    js = request.args.get('js', default='0')
    if js == '0':
        # For Python Client
        pickled_result = pickle.dumps(results)
        return Response(pickled_result, content_type='application/octet-stream')

    # For Web-based Client
    return jsonify({key: str(value) for key, value in results.items()})
```

Code 3.9: Router implementation for query execution

The workflow begins by retrieving the query parameters from the HTTP request, which are used to fetch the corresponding query and to construct the `queryInput`

object which is essentially a list of shots.
Next, the core logic responsible for executing the query is encapsulated in the following line:

```
    results = QueryExecutor.execute(sparkContext=sparkContext, query=query, queryInput=queryInput)
```

This statement triggers the entire query execution workflow, leveraging Apache Spark's distributed processing capabilities to efficiently process the input data. It relies on a properly initialized `SparkSession` to provide the `SparkContext` needed for distributed execution. For instance, the `SparkSession` can be created as shown below:

```
    # Initialize SparkSession
    spark = SparkSession.builder.appName("Query-Engine") \
    .master('spark://localhost:7077') \
    .config('spark.submit.pyFiles', 'app.zip') \
    .getOrCreate() \
```

This configuration enables Spark to distribute the application code across the cluster and efficiently execute queries in a distributed, parallelized manner. To gain a deeper understanding, we will now explore the implementation details behind this call, presented in Code 3.10.

```
class QueryExecutor:
    @staticmethod
    def execute(sparkContext: SparkContext = None, query: Query = None, queryInput: QueryInput = None, cache: bool = True) -> dict:
        # Retrieve the list of shots relevant to this query
        shotList = queryInput.getShotList()

        # If no shots are provided, return an empty result
        if (shotList is None) or (len(shotList) == 0):
            return {}

        # If no Spark context is provided, abort execution and return None
        if sparkContext is None:
            results = None
            return results

        # Recursively execute any dependency queries first
```

```
        dependencyQueries = QueryService.getDependencyQueries(queryName=query.queryName)
        for dependencyQuery in dependencyQueries:
            QueryExecutor.execute(sparkContext=sparkContext, query=dependencyQuery, queryInput=queryInput, cache=cache)

        # Parallelize the shots for distributed execution
        shotsRDD = sparkContext.parallelize(shotList)

        # For each shot, execute the query's unit function with caching support
        resultsRDD = shotsRDD.map(lambda shot: (shot, UnitFunctionExecutor.executePerShotWithCache(query, shot, cache)))

        # Collect results into a dictionary mapping shot to result
        results = resultsRDD.collectAsMap()
        return results

class UnitFunctionExecutor:
    @staticmethod
    def executePerShotWithCache(query: Query, shot: int, cache = False):
        from app import create_app
        from app.utils.execution_unit_cache import ExecutionUnitCache
        from app.config import ConfigSpark

        app = create_app(None, ConfigSpark)

        # Use Flask app context to manage any application-specific resources
        result = None
        with app.app_context():
            # Attempt to retrieve cached result if caching is enabled
            if cache and ExecutionUnitCache.hasCached(query.queryName, shot):
                result = ExecutionUnitCache.getCachedResult(query.queryName, shot)
            else:
                # Execute the unit function if no cache exists
                result = UnitFunctionExecutor.executePerShot(query, shot)

            # Cache the result for future use if caching is enabled
```

```
            if cache:
                ExecutionUnitCache.cache(query.queryName, shot,
    result)

        return result

    @staticmethod
    def executePerShot(query: Query, shot: int):
        import re

        # Extract the function name from the provided function source
     code
        executionName = re.search(r'\bdef (\w+)\s*\(', query.
    executionUnitFunction).group(1)

        # Execute the function code within a local namespace
        localContext = {}
        exec(query.executionUnitFunction, {}, localContext)

        # Invoke the extracted function with the current shot
        result = localContext[executionName](shot)
        return result
```

Code 3.10: Query Execution Implementation Details

Explanation:

- `QueryExecutor.execute(...)`: This static method orchestrates the entire query execution process. It begins by obtaining the list of relevant shots from the `queryInput`. If no shots exist or if the Spark context is unavailable, it exits early. Otherwise, it recursively resolves any dependencies the current query might have by executing them first.

- The shots are then parallelized into an RDD, allowing Spark to distribute computation across multiple nodes or cores.

- For each shot, the method `executePerShotWithCache` is called. This method manages caching and ensures that redundant computation is avoided whenever possible.

- `UnitFunctionExecutor.executePerShotWithCache(...)`: This method wraps the execution of the per-shot execution unit function with caching logic. It

uses Flask's application context to handle any app-specific resources or configurations, checks for cached results, executes the function if needed, and caches the output.

- `UnitFunctionExecutor.executePerShot(...)`: This method extracts and executes the unit function from a string containing the function's source code. It uses Python's `exec()` to compile and run the function in an isolated local context, then calls the function with the current shot as input.

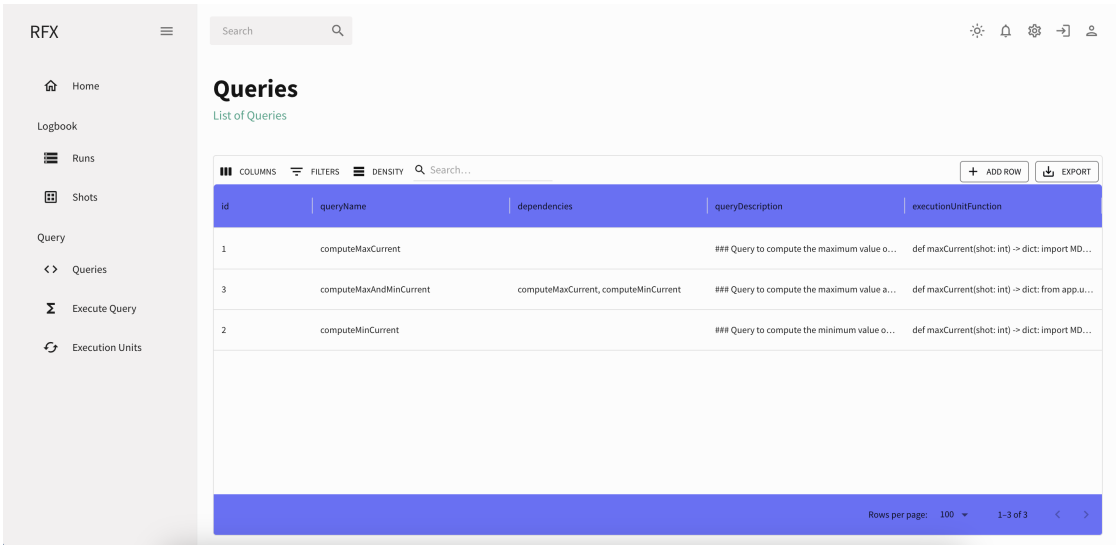
This design provides flexibility by allowing query execution logic to be defined at runtime while leveraging Spark for scalability and caching to optimize repeated executions.
Finally, once the query has been executed and results are available, Code 3.9 demonstrates how these results are returned to the client via an HTTP response. The response handling supports two modes:

- For Web-based Client, the results are serialized to JSON format, making them easily consumable for visualization or further processing.
- For Python clients using the dedicated client package, the results are returned as a binary payload by pickling[19] the data. This enables efficient data transfer and seamless deserialization on the client side.

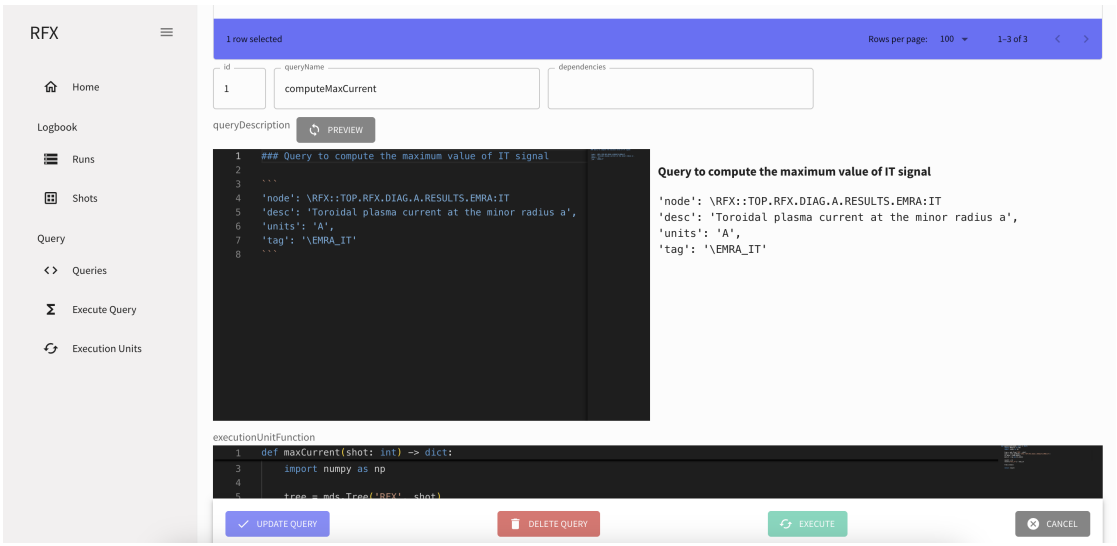
3.3.4 User Interface

Figure 3.16 shows the User Interface for Query Manipulation, while Figure 3.17 demonstrates the User Interface for Query Execution.
Building upon the components introduced earlier, we also integrate `monaco-editor/react`[20] for the code editor, `react-markdown`[21] for rendering Markdown content, and `react-json-view`[22] for displaying JSON objects. Additionally, we continue to use the Axios library for RESTful API interactions, as demonstrated in Code 2.7.

[19] `https://docs.python.org/3/library/pickle.html`
[20] `https://www.npmjs.com/package/@monaco-editor/react`
[21] `https://www.npmjs.com/package/react-markdown/v/8.0.6`
[22] `https://www.npmjs.com/package/@uiw/react-json-view`

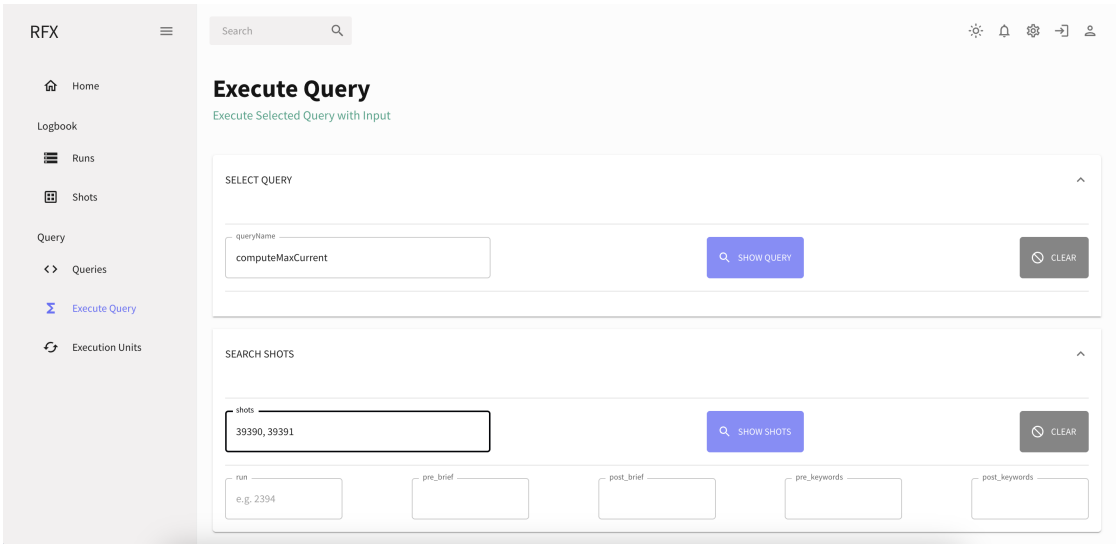


(a)

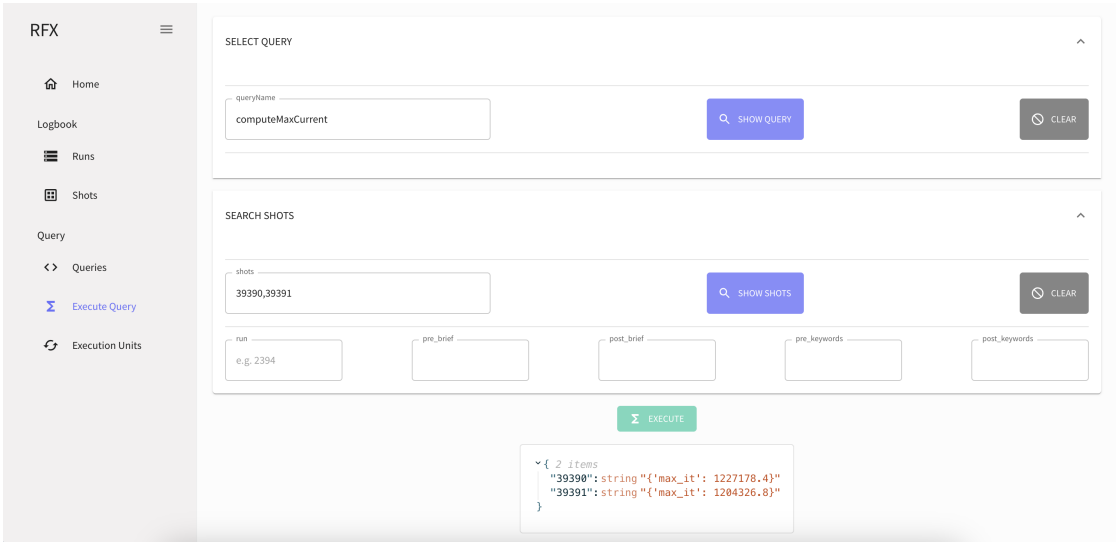


(b)

Figure 3.16: User Interface for Query Manipulation



(a)



(b)

Figure 3.17: User Interface for Query Execution

3.3.5 Client Package

As mentioned previously, we implemented the Client Package to enable interaction with the Query Engine system in Python. Code 3.11 demonstrates an example of creating a query and executing it to obtain the results.

```python
from query_client import QueryDecorator, QueryClient

@QueryDecorator
def maxCurrent(shot: int) -> dict:
    import MDSplus as mds
    import numpy as np

    tree = mds.Tree('RFX', shot)
    node = tree.getNode('\\RFX::TOP.RFX.DIAG.A.RESULTS.EMRA:IT')
    it_data = node.data()
    max_it = np.max(it_data)

    result = {}
    result["max_it"] = max_it

    tree.close()
    return result

if __name__ == "__main__":
    SERVER = "http://localhost:5001"

    # Query Creation
    response = QueryClient.createQuery(server=SERVER,
    queryName="computeMaxCurrent",
    dependencies=[], queryDescription="",
    executionUnitFunction=maxCurrent)

    # Query Execution
    result = QueryClient.execute(SERVER, "computeMaxCurrent", [39384,
     39385, 39386, 39387, 39388, 39389, 39390, 39391])
```

Code 3.11: Usage of Client Package of Query Engine

In detail, the Client Package provides two main components: `QueryDecorator` and `QueryClient`. The `QueryDecorator` allows users to wrap the Python function intended for the execution unit function of the query. The `QueryClient` offers methods to create queries on the server and to execute these queries with specified input parameters.

In the example above, the `maxCurrent` function calculates the maximum current (`max_it`) from a specified node associated with a shot in the MDSplus data tree. This function is wrapped with the `QueryDecorator` and sent to the Query Engine server using `QueryClient.createQuery`, along with additional query information such as the name, description, and dependencies. Finally, the `QueryClient.execute` method runs the query for a list of shot numbers, returning the computed maximum currents. This approach abstracts the complexities of remote execution and data retrieval, providing a streamlined interface for interacting with the Query Engine in Python-based data workflows.

Following the detailed description of the Query Engine implementation in this chapter, the next chapter turns to evaluating its practical performance and effectiveness. It presents the experimental setup, evaluation methodology, and performance results to assess the system's capabilities across real-world scenarios. This evaluation aims to validate the system's functionality, efficiency, and scalability under varying conditions.

4

Results and System Evaluation

This chapter provides the results and evaluation of the developed Query Engine system, focusing on its ability to handle a range of query scenarios. The primary objective is to assess the system's functionality, efficiency, and scalability across different configurations. We begin by describing the system setup, including the deployment environment and hardware specifications. Subsequently, we present a series of performance experiments that investigate the effects of parameters such as the total number of Spark Worker cores and the benefits of caching mechanism. The following sections present a detailed account of the system configuration, the evaluation methodology, and the experimental results.

4.1 System Setup

To facilitate a controlled and replicable evaluation environment, the backend system was deployed using Docker Compose[1], enabling containerized orchestration of all system components [1]. The system architecture comprises multiple services including one Backend service, one PostgreSQL server, one Spark Cluster composed of one Master Node and one Worker Node.
The deployment environment used was as follows:

- **Host Machine** macOS Sequoia Version 15.4.1
- **Docker**

[1] `https://docs.docker.com/compose/`

```
% docker version

Client:
    Version:           28.1.1
    API version:       1.49
    Go version:        go1.23.8
    Git commit:        4eba377
    Built:             Fri Apr 18 09:49:45 2025
    OS/Arch:           darwin/arm64
    Context:           desktop-linux

Server: Docker Desktop 4.41.1 (191279)
    Engine:
     Version:          28.1.1
     API version:      1.49 (minimum version 1.24)
     Go version:       go1.23.8
     Git commit:       01f442b
     Built:            Fri Apr 18 09:52:08 2025
     OS/Arch:          linux/arm64
     Experimental:     false
    containerd:
     Version:          1.7.27
     GitCommit:        05044ec0a9a75232cad458027ca83437aae3f4da
    runc:
     Version:          1.2.5
     GitCommit:        v1.2.5-0-g59923ef
    docker-init:
     Version:          0.19.0
     GitCommit:        de40ad0
```

- **Hardware Specs**

```
    Chip     Apple M1
    Memory   16GB
    Disk     Macintosh HD

    kern.sched_rt_avoid_cpu0: 0
    kern.cpu_checkin_interval: 5000
    hw.ncpu: 8
    hw.activecpu: 8
    hw.perflevel0.physicalcpu: 4
    hw.perflevel0.physicalcpu_max: 4
    hw.perflevel0.logicalcpu: 4
```

```
        hw.perflevel0.logicalcpu_max: 4
        hw.perflevel0.cpusperl2: 4
        hw.perflevel1.physicalcpu: 4
        hw.perflevel1.physicalcpu_max: 4
        hw.perflevel1.logicalcpu: 4
        hw.perflevel1.logicalcpu_max: 4
        hw.perflevel1.cpusperl2: 4
        hw.physicalcpu: 8
        hw.physicalcpu_max: 8
        hw.logicalcpu: 8
        hw.logicalcpu_max: 8
        hw.cputype: 16777228
        hw.cpusubtype: 2
        hw.cpu64bit_capable: 1
        hw.cpufamily: 458787763
        hw.cpusubfamily: 2
        machdep.cpu.cores_per_package: 8
        machdep.cpu.core_count: 8
        machdep.cpu.logical_per_package: 8
        machdep.cpu.thread_count: 8
        machdep.cpu.brand_string: Apple M1

```

An important parameter in this evaluation is the `SPARK_WORKER_CORES`[2] setting of the Spark Worker, which determines the total number of cores allocated for query processing. In the next section, we provide a detailed discussion of this parameter and its impact on system performance.

4.2 Results and Evaluation

This section presents the results of performance evaluations conducted on the developed Query Engine system. The evaluations aim to measure the system's behavior under different configurations, focusing on the impact of the number of Spark Worker cores and the effectiveness of caching mechanisms. The evaluations include both independent query executions and scenarios involving query dependencies.

We start by considering the query `computeMaxCurrent`, whose execution unit

[2] `https://spark.apache.org/docs/latest/spark-standalone.html`

function is implemented as Code 4.1:

```
@QueryDecorator
def computeMaxCurrent(shot: int) -> dict:
    import MDSplus as mds
    import numpy as np

    tree = mds.Tree('RFX', shot)
    node = tree.getNode('\\RFX::TOP.RFX.DIAG.A.RESULTS.EMRA:IT')
    it_data = node.data()
    max_it = np.max(it_data)
    result = {}
    result["max_it"] = max_it

    tree.close()
    return result
```

Code 4.1: Execution unit function of `computeMaxCurrent`

To analyze the system's performance, we fix the input shots to the following list: `[39384, 39385, 39386, 39387, 39388, 39389, 39390, 39391]`. We then measure the processing time of this query as we vary the `SPARK_WORKER_CORES` parameter of the single Spark Worker node. The processing time is measured using the Client Cackage, as demonstrated in Code 4.2. The resulting data, as depicted in Figure 4.1, demonstrates how the processing time decreases as the number of worker cores increases, highlighting the scalability of the system.

```
from query_client import QueryDecorator, QueryClient
import timeit
import math

if __name__ == "__main__":
    SERVER = "http://localhost:5001"
    # Query Execution
    start_ts = timeit.default_timer()
    result = QueryClient.execute(SERVER, "computeMaxCurrent", [39384,
    39385, 39386, 39387, 39388, 39389, 39390, 39391])
    elapsed_time = timeit.default_timer() - start_ts
    print("Running time = {} ms".format(math.ceil(elapsed_time *
    1000)))
```

Code 4.2: Script for measuring query processing time

Next, we assess the effect of result caching mechanism. By repeating the same experiment after the query results have been cached, we observe a significant reduction in processing time, as shown in Figure 4.2.

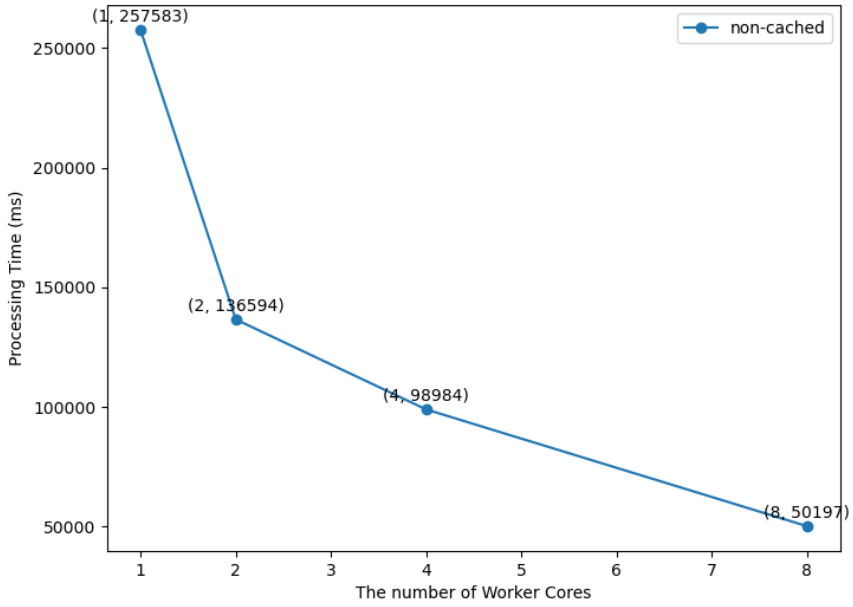


Figure 4.1: Relationship between the number of Worker Cores and Processing Time

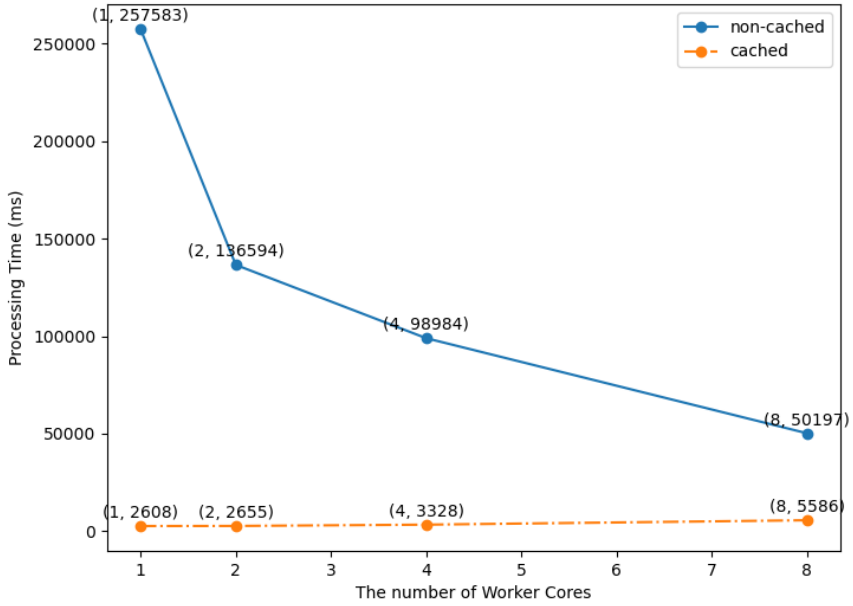


Figure 4.2: Relationship between the number of Worker Cores and Processing Time With Caching

We then explore a scenario involving query dependencies. Specifically, we evaluate the `computeMaxAndMinCurrent` query, which depends on the results of two sub-queries: `computeMaxCurrent` and `computeMinCurrent`. The execution unit function of `computeMinCurrent` is defined in Code 4.3. This code also shows how the execution unit function of `computeMaxAndMinCurrent` aggregates the cached results from its two dependency queries. The workflow of this dependency-based scenario is illustrated in Figure 4.3.
To evaluate the performance of this scenario, we implemented the script shown in Code 4.4. This script measures the processing times of three queries executed sequentially: `computeMaxCurrent`, `computeMinCurrent`, and `computeMaxAndMinCurrent`. The results, presented in Figure 4.4, illustrate how caching mechanism significantly reduces the processing time of the dependency-based query, thereby demonstrating the system's efficiency in managing dependencies and leveraging intermediate cached results.

```python
@QueryDecorator
def computeMinCurrent(shot: int) -> dict:
    import MDSplus as mds
    import numpy as np

    tree = mds.Tree('RFX', shot)
    node = tree.getNode('\RFX::TOP.RFX.DIAG.A.RESULTS.EMRA:IT')
    it_data = node.data()
    min_it = np.min(it_data)

    result = {}
    result["min_it"] = min_it

    tree.close()

    return result

@QueryDecorator
def computeMaxAndMinCurrent(shot: int) -> dict:
    from app.utils.execution_unit_cache import ExecutionUnitCache

    result = {}
    result.update(ExecutionUnitCache.getCachedResult("
    computeMaxCurrent", shot))
    result.update(ExecutionUnitCache.getCachedResult("
    computeMinCurrent", shot))

    return result
```

Code 4.3: `computeMinCurrent` and `computeMaxAndMinCurrent` functions

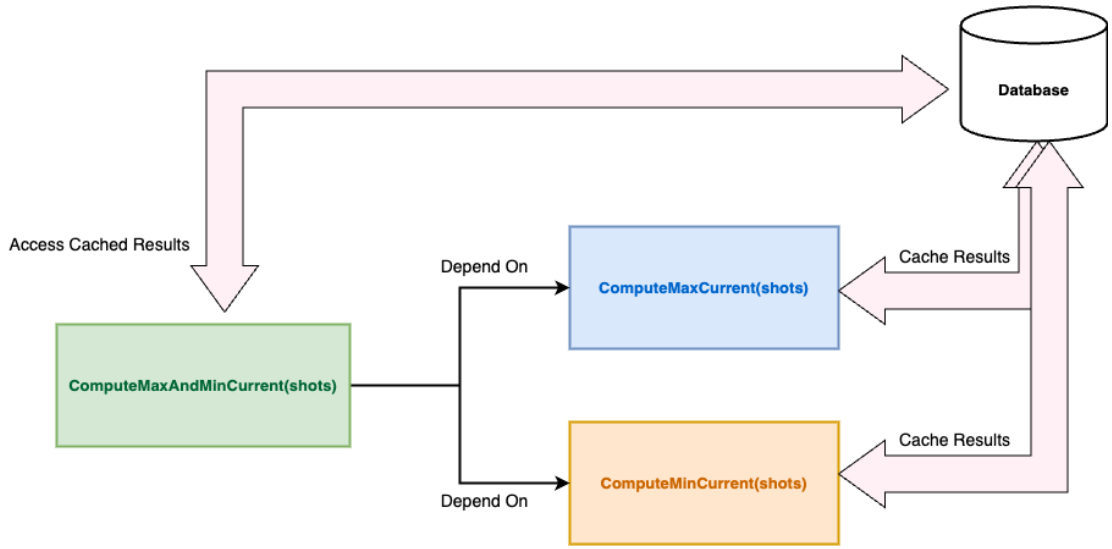


Figure 4.3: Query Dependency Workflow of `computeMaxAndMinCurrent`

```
if __name__ == "__main__":
    SERVER = "http://localhost:5001"
    # Query Dependency
    start_ts = timeit.default_timer()
    result = QueryClient.execute(SERVER, "computeMaxCurrent", [39384,
    39385, 39386, 39387, 39388, 39389, 39390, 39391])
    elapsed_time = timeit.default_timer() - start_ts
    print("Running time of computeMaxCurrent  = {} ms".format(math.
   ceil(elapsed_time * 1000)))

    start_ts = timeit.default_timer()
    result = QueryClient.execute(SERVER, "computeMinCurrent", [39384,
    39385, 39386, 39387, 39388, 39389, 39390, 39391])
    elapsed_time = timeit.default_timer() - start_ts
    print("Running time of computeMinCurrent = {} ms".format(math.
   ceil(elapsed_time * 1000)))

    start_ts = timeit.default_timer()
    result = QueryClient.execute(SERVER, "computeMaxAndMinCurrent",
   [39384, 39385, 39386, 39387, 39388, 39389, 39390, 39391])
    elapsed_time = timeit.default_timer() - start_ts
    print("Running time of computeMaxAndMinCurrent = {} ms".format(
   math.ceil(elapsed_time * 1000)))
```

Code 4.4: Script for evaluating query dependency

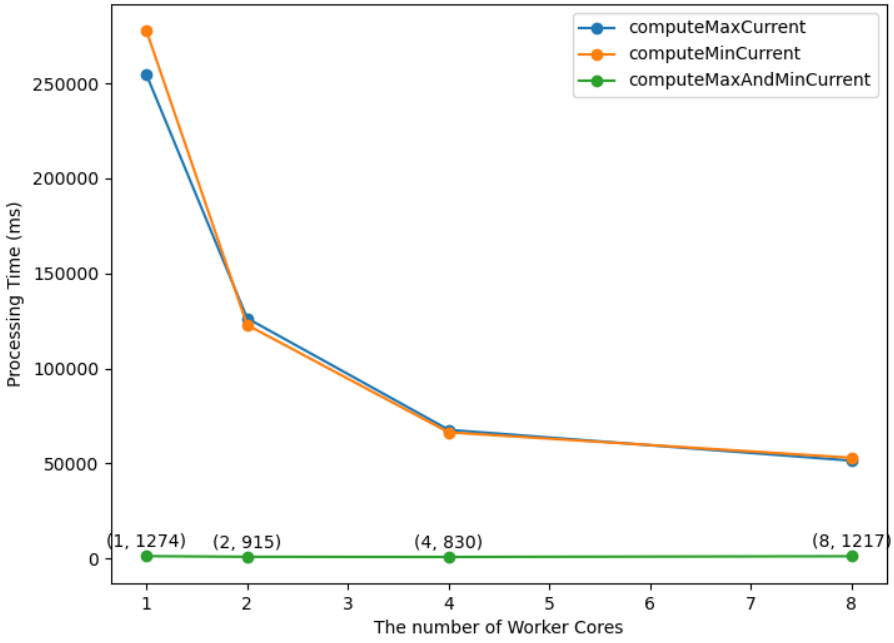


Figure 4.4: Processing time of the dependency-based query

We further investigate a more complex scenario involving multi-level query dependencies. Specifically, we define a chain of three queries: `getSpectrum`, `getMode7Ratio`, and `getMode7DominantRatio`. These queries are hierarchically dependent, where `getMode7DominantRatio` depends on the output of `getMode7Ratio`, which in turn relies on the results produced by `getSpectrum`. The structure of these dependencies is illustrated in Figure 4.5.

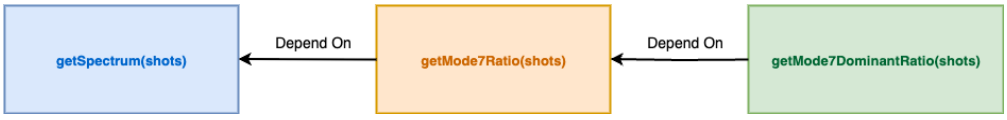


Figure 4.5: Hierarchical dependency chain of `getSpectrum`, `getMode7Ratio`, and `getMode7DominantRatio` functions

The `getSpectrum` function extracts the radial component of the magnetic field's Fourier modes at the sensor radius for the poloidal number $m = 1$, toroidal numbers $n = -6, \ldots, -15$. Using these spectral components, the `getMode7Ratio` function calculates, at each time point, the ratio of the amplitude of the $m = 1, n = -7$ mode to the sum of amplitudes of the remaining modes $m = 1, n = -8, \ldots, -15$. Finally, the function `getMode7DominantRatio` calculates the frac-

tion of time points where the amplitude of the $m = 1, n = -7$ mode exceeds the combined amplitude of the $m = 1, n = -8, \ldots, -15$ modes, reflecting its dominance over the others.

For conciseness, the implementation details of the `getSpectrum` function are omitted. As an illustrative example, Figure 4.6a displays the signal amplitudes extracted by this function for shot 30873.

Code 4.5 presents the implementation of the `getMode7Ratio` function, while Figure 4.6b shows the resulting ratios calculated from the spectral data obtained by `getSpectrum` for the same shot (30873).

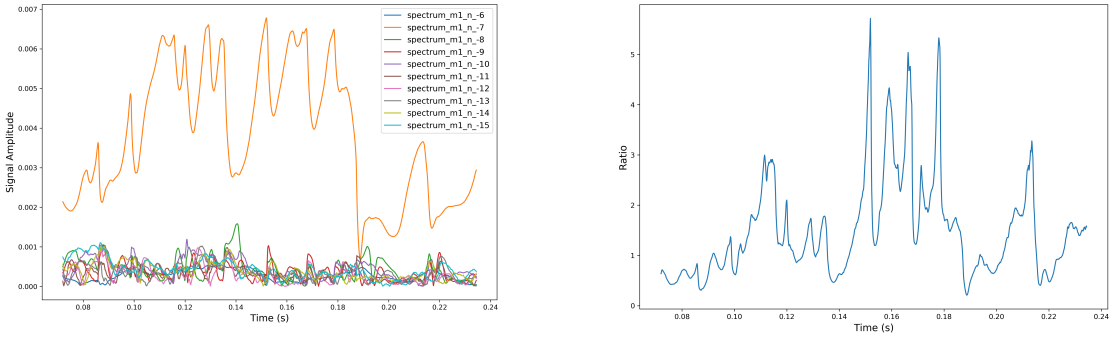
The implementation of the `getMode7DominantRatio` function is provided in Code 4.6.

Both `getMode7Ratio` and `getMode7DominantRatio` functions initially retrieve cached results from their respective dependencies before performing their calculations, demonstrating a hierarchical approach that leverages intermediate cached outputs for efficiency and modularity.

```python
def getMode7Ratio(shot: int) -> dict:
    from app.utils.execution_unit_cache import ExecutionUnitCache
    import numpy as np

    cached_result = {}
    cached_result.update(ExecutionUnitCache.getCachedResult("
    getSpectrum", shot))

    n7 = cached_result["spectrum_m1_n_-7"]
    all_others = np.sum([cached_result[f"spectrum_m1_n_-{i}"] for i
    in list(range(8, 16))], axis=0)

    result = {}
    result["ratio"] = n7 / all_others

    return result
```

Code 4.5: `getMode7Ratio` function

To evaluate the performance of this multi-level dependency structure, we fixed the `SPARK_WORKER_CORES` parameter to one to isolate the effect of increasing the number of processed shots. The number of shots was incrementally increased from one shot (30873), to two (30873, 30874), three (30873, 30874, 30876), and finally four shots (30873, 30874, 30876, 30878). For each configuration, we recorded the processing times of the three queries – `getSpectrum`, `getMode7Ratio`, and `getMode7DominantRatio` – executed sequentially.



(a) (b)

Figure 4.6: (a) Signal amplitudes of radial magnetic field Fourier modes extracted by `getSpectrum` for shot 30873. (b) Corresponding ratios computed by `getMode7Ratio`, comparing the dominant $m = 1, n = -7$ mode to the sum of modes $m = 1, n = -8, \ldots, -15$.

```python
def getMode7DominantRatio(shot: int) -> dict:
    import numpy as np
    from app.utils.execution_unit_cache import ExecutionUnitCache

    ratio = {}
    ratio.update(ExecutionUnitCache.getCachedResult("getMode7Ratio",
    shot))

    result = {}
    result['dominant_ratio'] = 100 * np.sum(ratio > 1) / len(ratio)

    return result
```

Code 4.6: `getMode7DominantRatio` function

The resulting performance measurements, illustrated in Figure 4.7, demonstrate the system's efficiency in managing multi-level dependencies. Notably, the use of intermediate cached results significantly reduces the processing time of the dependent queries, even as the number of shots increases. This result reinforces the scalability and effectiveness of the caching mechanism in optimizing complex query workflows.

The evaluation results presented in this chapter demonstrate the Query Engine system's capability to efficiently handle both independent and dependency-based queries across a range of configurations. The system exhibits clear scalability with respect to the number of allocated Spark Worker cores and achieves substantial performance gains through its caching mechanism. These findings validate the effectiveness of the system's design and highlight its suitability

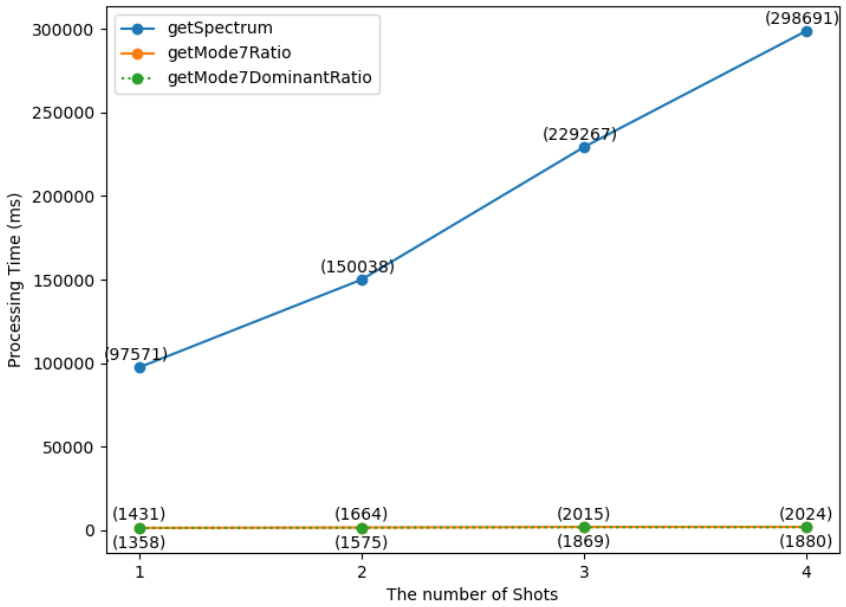


Figure 4.7: Relationship between Processing Time of dependency-based queries and the number of Shots

for high-throughput, reusable data processing in experimental research environments. Building upon these results, the next chapter summarizes the key contributions of this work and discusses potential directions for future development and enhancement of the system.

5

Conclusions and Future Works

5.1 Conclusions

This thesis has presented the design, implementation, and evaluation of a unified platform for experimental data access and documentation in the context of the RFX-mod2 experiment, which aims to study the physics of fusion plasmas and magnetic confinement in Reversed Field Pinch (RFP) configuration.
The evolution from RFX-mod to RFX-mod2 has brought substantial upgrades to the experimental infrastructure. These include modified mechanical components, an updated electrical configuration, and a more advanced CODAS (Control, Data Access, and Communication Systems). In addition to improving operational performance and machine reliability, these enhancements significantly expand the range of available diagnostics. This increased data richness provides deeper insight into plasma phenomena but also introduces the complex challenge of efficiently managing, accessing, and interpreting large volumes of data.
To address these challenges, this work has developed a unified platform that integrates two core components:

- **Experiment Logbook Management:** A web-based application designed to manage semantic information, enabling researchers to efficiently document, retrieve, and contextualize experiments. By incorporating detailed logbook entries, this system enriches raw diagnostic data with experimental configurations, objectives, and observations, supporting reproducibility, collaboration,

and cross-experiment analysis.

- **Scientific Data Access:** The Query Engine system was developed as a framework for efficient, scalable data access and analysis of experimental data. This component allows researchers to construct, manage, and reuse data processing logic across multiple shots, streamlining access to a large volume of diagnostic data and supporting advanced data analytics workflows.

The results of performance evaluations highlight the Query Engine's efficiency and scalability across different deployment configurations, demonstrating its potential to handle the increasing data demands of RFX-mod2 experiments effectively. By integrating these two components, the unified platform provides a more accessible and integrated framework for managing and analyzing experimental data, ultimately improving the efficiency and reliability of the entire research workflow.

5.2 Future Works

While the unified platform has addressed the key challenges identified in this thesis, several avenues for future work remain:

- **Organizing the Experiment Hierarchy:** Establishing a more effective structure for the experiment hierarchy by organizing entries such as experiments, campaigns, runs, and shots will be essential. This could be reflected in a RESTful API structure to provide consistent and predictable access patterns across the platform.

- **RESTful Query Management:** Exposing queries via REST endpoints with clear, unique names would simplify integration with external systems, particularly for retrieving summaries. However, this requires careful consideration of naming conventions to avoid confusion and ensure consistency.

- **Collaborative Logbook Input:** Enabling collaborative editing for logbook entries, including real-time multi-user input, will enrich experiment documentation and support dynamic workflows.

- **Flexible Structures of Logbook Entries:** Since experimental requirements can evolve rapidly, future work should include support for defining flexible

structures and fields within logbook entries to effectively accommodate these changes.

- **User-Specific Views:** Supporting role-based front-end views for each user will make the system more intuitive and tailored to individual workflows.
- **Content Identifiers for Queries:** Introducing Content Identifiers (CIDs) [12] for queries can reduce duplication, improve traceability, and support robust verification mechanisms – particularly important for tracking changes to MDSplus data.

In summary, the unified platform developed in this thesis represents a significant step toward addressing the challenges of managing and analyzing the growing volumes of data produced by plasma physics experiments. By integrating semantic documentation with efficient data access, it provides a foundation for more advanced data analytics and collaborative research, paving the way for future improvements and scientific discoveries in the RFX-mod2 experiment and beyond.

References

[1] Sathyajith Bhat and Sathyajith Bhat. "Understanding docker compose". In: *Practical Docker with Python: Build, Release, and Distribute Your Python App with Docker* (2022), pp. 165–198.

[2] M. Cecconello. *Experimental studies of confinement in the EXTRAP T2 and T2R reversed field pinches*. Alfvén Laboratory, Fusion Plasma Physics, Royal Institute of Technology, Stockholm, 2003.

[3] Ed. D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Internet Engineering Task Force, Oct. 2012. URL: `https://datatracker.ietf.org/doc/html/rfc6749`.

[4] Jules S. Damji et al. *Learning Spark: Lightning-Fast Data Analytics*. O'Reilly Media, 2020.

[5] DF Escande et al. "Single helicity: a new paradigm for the reversed field pinch". In: *Plasma Physics and Controlled Fusion* 42.12B (2000), B243.

[6] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". Accessed: 2025-06-12. Ph.D. Dissertation. University of California, Irvine, 2000. URL: `https://ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

[7] David Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[8] Jeffrey P. Freidberg. *Plasma Physics and Fusion Energy*. Cambridge University Press, 2007.

[9] James Gardner. "The web server gateway interface (wsgi)". In: *The Definitive Guide to Pylons* (2009), pp. 369–388.

[10] Google. *Google's Material Design*. Accessed: 2025-06-12. 2024. URL: `https://m3.material.io/get-started`.

[11] Martin Greenwald. "Density limits in toroidal plasmas". In: *Plasma Physics and Controlled Fusion* 44.8 (July 2002), R27. DOI: 10.1088/0741-3335/44/8/201. URL: https://dx.doi.org/10.1088/0741-3335/44/8/201.

[12] IPFS Project. *Content Addressing*. https://docs.ipfs.tech/concepts/content-addressing/. Accessed: 2025-06-14.

[13] Christopher Ireland et al. "Understanding object-relational mapping: A framework based approach". In: *International lournal On Advances in Software* 2.2 (2009).

[14] Md Rezaul Karim and Sridhar Alla. *Scala and Spark for Big Data Analytics: Explore the concepts of functional programming, data streaming, and machine learning*. Packt Publishing Ltd, 2017.

[15] Rita Lorenzini et al. "Self-organized helical equilibria as a new paradigm for ohmically heated fusion plasmas". In: *Nature Physics* 5.8 (2009), pp. 570–574.

[16] L. Marrelli et al. "The reversed field pinch". In: *Nuclear Fusion* 61 (2021).

[17] Ed N. Sakimura, J. Bradley, and N. Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636. Internet Engineering Task Force, Sept. 2015. URL: https://www.rfc-editor.org/rfc/rfc7636.html.

[18] Rob Pike. *Go at Google: Language Design in the Service of Software Engineering*. https://go.dev/talks/2012/splash.article. Accessed: 2025-06-12.

[19] M.E. Puiatti et al. "Overview of the RFX-mod contribution to the international Fusion Science Program". In: *Nuclear Fusion* 55.10 (July 2015), p. 104012. DOI: 10.1088/0029-5515/55/10/104012. URL: https://dx.doi.org/10.1088/0029-5515/55/10/104012.

[20] G. Rostagni et al. "Topical Issue on the RFX-experiment, Padova". In: *Fusion Engineering and Design* 25 (1995), pp. 301–504.

[21] SQLAlchemy. *SQLAlchemy Philosophy*. Accessed: 2025-06-14. URL: https://www.sqlalchemy.org/philosophy.html.

[22] JA Stillerman et al. "MDSplus data acquisition system". In: *Review of Scientific Instruments* 68.1 (1997), pp. 939–942.

[23] The PostgreSQL Global Development Group. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/.

[24] J. Wesson and D.J. Campbell. *Tokamaks*. International series of monographs on physics. Clarendon Press, 2004. ISBN: 9780198509226. URL: https://books.google.it/books?id=iPlAwZI6HIYC.

[25] John Wesson and D. J. Campbell. *Tokamaks*. Oxford University Press, 2011.

[26] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[27] Petar Zečević and Marko Bonaći. *Spark in Action*. Manning, 2016.

Acknowledgments

I would like to express my deepest gratitude to Prof. Rigoni Garola Andrea and Ing. Gianluca Moro for their invaluable guidance and support throughout the course of this thesis. Their mentorship and the favorable conditions they created have been essential to the successful completion of this work.
I extend my deepest thanks to my grandparents and parents for their unconditional love, sacrifices, and lifelong dedication to my growth and well-being. Their support has been the foundation of everything I have achieved.
I sincerely thank Huyen Linh for her unwavering support, deep understanding, and constant care throughout my studies, and for patiently standing by me during this entire journey abroad.
I am also truly grateful to my friends at the University of Padua - Matteo, Maria, Roger, Francesco, Giulio, Gabriel, Samuel, Filippo, Lorenzo, Vaidas, and Zoren - for their companionship and support during my time in Padova. Sharing this experience with you made it all the more meaningful.
I would also like to thank the Veneto Region and the University of Padua for awarding me Regional Scholarships, which greatly improved my living and studying conditions and enabled me to fully dedicate myself to my academic pursuits.
To all of you, a heartfelt “Grazie Mille!”