The People's Democratic Republic of Algeria
الجمهورية الجزائرية الديمقراطية الشعبية
Ministry of Higher Education and Scientific Research
وزارة التعليم العالي و البحث العلمي

University Of Abou Bekr Belkaid
Faculty of Sciences
Department of Computer Science

Master's Thesis

For the State Computer Engineering diploma

Option : Software Engineering

SemPart: Yet Another RDF Partitioning Strategy In Action.

Produced by :
KHIAT Sara Nardjes
HAMMOUMI Tarik

Supervised by :
Mr. MATALLAH Houcine (UABT)
Mr. SAIDI Boumediene (LIAS)
Mr. MESMOUDI Amin (LIAS)

Supported on 24/06/2025, Before the jury composed of :

President: Mrs. HALFAOUI Amal (UABT)
Examiner: Mrs. KHITRI Souad (UABT)

Promotion : 2024/2025

Acknowledgements

First and foremost, we express our sincere gratitude to Allah, the Almighty,
for granting us the strength, patience, and perseverance to complete this work.

We extend our heartfelt thanks to our supervisors for their invaluable guidance
throughout this project.

We are especially grateful to **Mr. Houcine MATALLAH**,
whose insightful advice and great patience have been a continuous source of motivation
and inspiration.

Our sincere thanks also go to **Mr. Amin MESMOUDI**,
for his exceptional support, availability, and helpful guidance at each step of this
journey.

We would like to express our deep appreciation to **Mr. Saidi BOUMEDIENE**,
whose unwavering involvement, relevant advice, and generous availability were key to
the success of our work.

We are also thankful to **Mrs. KHITRI SOUAD** for accepting to review our thesis,
and to **Mrs. HALFAOUI AMAL** for kindly agreeing to chair the jury.

We extend our appreciation to all the professors of the Computer Science Department
for their dedication, support, and the knowledge they have shared with us throughout
our academic years.

Our deepest gratitude goes to our **parents**,
for their unconditional love, wise guidance, and continuous moral and financial support,
which have allowed us to pursue our studies and complete this thesis.

Finally, we warmly thank our **friends and colleagues**,
for their constant support, enriching discussions, and encouragement throughout this
academic journey.

Dedication

To my **mother**, gentle and brave,
whose love and support have been my guiding light in every challenge.
and to my **father**, steady and strong,
who has always been my unwavering source of strength.

To my sister and brother, **Amel** and **Mustapha**,
for their caring presence and support throughout this path.

To my supervisors,
Mr. MATALLAH Houcine, **Mr. SAIDI Boumediene**,
and **Mr. MESMOUDI Amin**,
for their valuable guidance, thoughtful supervision, and constant availability,
which greatly contributed to the success of this project.

To all the teachers
who shared their knowledge with passion
and helped me appreciate learning and keep striving for excellence.

To my friends,
especially **SAIDI Sabah** and **BOUBEKEUR Hiba**, faithful companions on this journey,
and to those whom life has distanced,
but who remain deeply present in my thoughts.

Finally, to my teammate **HAMMOUMI Tarik**,
for the good times and teamwork
that made this work memorable.

KHIAT Sara Nardjes

Dedication

All praise is due to **Allah**, the Most Merciful and the Most Compassionate,
whose guidance and blessings have illuminated my path,
and without whom none of this would have been possible.

To my **parents**,
for their unconditional love, sacrifices, and prayers,
and for being my strongest foundation and constant source of strength.

To my supervisors,
Mr. MATALLAH Houcine, **Mr. SAIDI Boumediene**,
and **Mr. MESMOUDI Amin**,
for their insightful feedback, generous time, and commitment to academic excellence,
which helped shape this work in a meaningful way.

To the **University of Tlemcen**
and the **LIAS Laboratory of Poitiers**,
for providing this opportunity, environment, and support
that made this academic and personal growth possible.

To all my teachers,
whose dedication to teaching, clarity, and enthusiasm
ignited in me a deeper appreciation for knowledge and growth.

To my dear friends,
especially **KHALLADI Abdehamid** and the entire group,
for the memories, support, and laughter shared,
and to the Wednesday nights of Factorio, paused for the sake of this theses,
but never forgotten.

To my teammate
KHIAT Sara Nardjes,
for the synergy, trust, and shared determination throughout this experience.
Your collaboration has been both a professional asset and a personal joy.

HAMMOUMI Tarik

Abstract

The Resource Description Framework (RDF) is now an important standard for modeling structured knowledge, but it is still hard to manage large RDF datasets, especially when they are spread out across many computers. The way that current RDF triplestores split up their data into triples can make queries take longer and make communication harder. Also, the fact that database administrators (DBAs) are not involved in the partitioning process makes the system less flexible when workloads or semantic structures change. The PQDAG team at the LIAS lab came up with SemPart, a fragment-based RDF partitioning framework that adds semantic coherence and expert-driven control to get around these problems. The main focus of this work is on how to put it into practice by designing and building the entire SemPart framework and adding it to the distributed PQDAG system. This work makes it possible to do RDF partitioning that is flexible, semantically rich, and focused on performance.

***Keywords*:** RDF, Semantic Web, Data Partitioning, SemPart, PQDAG, Distributed Systems, Database Administrators, Triplestore

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
AST	Abstract Syntax Tree
ANTLR	ANother Tool for Language Recognition
DAG	Directed Acyclic Graph
DBA	Database Administrator
DSL	Domain-Specific Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
LAN	Local Area Network
MMC	Minimum Motif Cut
MPP	Massively Parallel Processing
MPC	Minimal Property Cut
NoSQL	Not Only SQL
NT	N-Triples
OCP	Open/Closed Principle
PQDAG	Parallel Query Data As Graph
QDAG	Querying Data As Graphs
RDPAL	RDF Partitioning And Allocation Language
RDF	Resource Description Framework
RDMA	Remote Direct Memory Access
SPARQL	SPARQL Protocol and RDF Query Language
SRP	Single Responsibility Principle

Table 1: List of commonly used abbreviations

Introduction

0.1 Background

The Resource Description Framework (RDF) has become the foundational standard for representing structured knowledge on the Web, enabling interoperability and semantic integration across heterogeneous data sources. RDF encodes information as triples in the form `<S, P, O>`, where `S` (subject) denotes the entity being described, `P` (predicate) defines the property or relationship, and `O` (object) indicates the value or target of the relationship. This simple yet expressive data model supports the representation of highly interconnected and semantically rich information, making it particularly well-suited for knowledge graphs.

As the volume and complexity of RDF datasets have grown—ranging from academic knowledge bases like DBpedia [1] and Wikidata [2] to domain-specific resources like Bio2RDF [3] and PubChemRDF [4], the need for efficient storage and query processing has intensified. This has led to the emergence of a wide array of RDF systems, including both centralized solutions optimized for single-node performance, and distributed systems designed to scale horizontally across multiple machines [1]. A key differentiator among these systems is their approach to RDF data partitioning, which plays a central role in reducing query latency, balancing workloads, and minimizing inter-node communication in distributed environments.

More precisely, partitioning refers to the process of dividing a large dataset into smaller, more manageable fragments, which can then be distributed across computing nodes. It has become a fundamental technique for enhancing scalability, performance, and manageability in database systems—especially in resource-constrained environments [2]. Over time, data partitioning has undergone significant evolution, reflecting a high degree of maturity backed by decades of research since the early days of relational databases. It has achieved broad adoption, with foundational principles being adapted and extended across generations of data management systems.

Designing effective partitioning schemes often requires balancing competing objectives—such as minimizing data movement, ensuring load balancing, and supporting efficient query execution—making it a non-trivial task. Moreover, growing industry integration has led to the development of dedicated partition manipulation languages, which aim to simplify and standardize partition design, refinement, and reconfiguration in production environments.

A focused analysis of leading RDF stores that rely heavily on partitioning to ensure

[1] `https://www.dbpedia.org/`
[2] `https://www.wikidata.org/`
[3] `https://bio2rdf.org/`
[4] `https://pubchem.ncbi.nlm.nih.gov/docs/rdf`

performance and scalability reveals three core dimensions: (1) query independence vs. dependence, (2) resource utilization, and (3) partitioning granularity. Broadly, partitioning granularity fall into three categories: triple, node and fragment. In this work, we focus solely on partitioning granularity dimension.

In the triple-based category, systems like AdPart [3] and Trinity.RDF [4] adopt simple hash-based strategies to assign each triple to a partition. More advanced systems, such as TriAD [5], apply graph partitioning algorithms like METIS to minimize edge cuts and improve locality. Moving to node-based partitioning, SHAPE [6] considers each node (subject or object) and its neighboring triples as a "triple group", which is then assigned to partitions using heuristic rules. This approach aims to maintain local graph neighborhoods and reduce cross-partition joins. At the highest granularity, fragment-based partitioning introduces semantically meaningful units. Partout [7] adapts the notion of minterm predicates—borrowed from relational databases—to define fragments based on query workload patterns. Similarly, SemStore [8] constructs rooted subgraphs (RSGs) as its core fragmentation units, promoting semantic locality and reuse across queries.

0.2 Problematic

Most existing RDF triplestores rely on triple-based storage and partitioning. While this design simplifies implementation and ensures uniform data distribution, it often lacks the flexibility and scalability required for complex, real-world applications. Specifically, triple-based partitioning tends to scatter semantically related triples across multiple nodes, leading to an increased number of cross-partition joins and higher communication overhead during query processing.

We argue that incorporating mechanisms for logical grouping—where semantically or structurally connected triples are treated as cohesive units—can significantly improve partitioning efficiency. By conceptualizing the RDF graph as a collection of interconnected groups or fragments, rather than isolated triples, it becomes possible to adopt allocation strategies inspired by fragment-based models in relational databases. This perspective enables the system to exploit semantic locality, optimize query execution, and integrate domain knowledge more effectively.

Moreover, current approaches typically lack the flexibility for runtime reconfiguration. To the best of our knowledge, no existing system provides users—particularly those with domain expertise—with tools to dynamically adjust or refine partitions in response to evolving workloads or data changes. This results in rigid frameworks that prioritize internal system convenience over user control and adaptability, thereby limiting their applicability in dynamic or resource-constrained environments.

0.3 Objective

Existing RDF data management approaches often favor simplicity during the preprocessing phase, with data fragmentation and allocation handled automatically by the system. In these systems, the user—typically a database administrator—is expected to remain passive, having little to no control over partitioning decisions. This represents a step backward compared to relational databases and Massively Parallel Processing (MPP) systems such as Spark, where experienced users (e.g., DBAs or data scientists) can leverage domain knowledge to guide the partitioning process through semantically meaningful

abstractions.

To address this limitation, our main objective is to implement a framework called **SemPart**, originally introduced by the PQDAG team at the LIAS laboratory (ISAE-ENSMA, France). The PQDAG team has collaborated extensively with the LRIT group in the development of QDAG [9], and is now focusing on PQDAG, its distributed extension. SemPart supports both initial partitioning and guided repartitioning, empowering users to select specific fragments and apply transformations as needed.

The framework consists of three core phases: (i) Logical fragmentation: This phase uses the concept of characteristic sets to group RDF entities sharing common predicate patterns into logical fragments. For example, a Research Paper fragment may contain entities associated with predicates such as hasAuthor, publishedIn, and cites. Each logical fragment corresponds to a single characteristic set, providing semantic coherence across the dataset. (ii) Physical fragmentation: SemPart offers a set of transformation operators that enable users to split, merge, or replicate fragments. These operations incorporate domain knowledge through both characteristic set structures and graph neighborhood information. (iii) Allocation: The framework provides various allocation strategies that take into account both the structure of the data and the characteristics of the deployment environment. This allows for flexible and efficient distribution of fragments across computational nodes.

To facilitate expert-driven partitioning and transformation, SemPart introduces **RDPAL (RDF Data Partitioning and Allocation Language)**—a language that allows users to express custom strategies for fragmentation and allocation. RDPAL bridges the gap between domain knowledge and system-level execution, enabling fine-grained control over data organization.

0.4 Manuscript organization

The remainder of this manuscript is organized as follows: **Chapter 1** (*State of the Art*) presents a selection of RDF triplestores, classified according to the dimension of partitioning granularity, and discusses the limitations associated with each category. **Chapter 2** (*Current Solutions*) introduces the distributed RDF system PQDAG and outlines the initial theoretical vision of the SemPart framework, designed to address the shortcomings identified in the previous chapter. **Chapter 3** (*Analysis and Design*) explains how the theoretical concept of SemPart was transformed into a practical and functional software solution, detailing the analysis process and design decisions used to structure the system. **Chapter 4** (*Realisation*) discusses the realization of the SemPart framework, the technologies used, and the overall organization of the codebase.

Chapter 1

State of the art

1.1 Introduction

A critical challenge in distributed RDF systems is selecting an optimal strategy for partitioning data across compute nodes. Among the various factors influencing this decision, the granularity of data grouping emerges as a fundamental one. In our work, we focus specifically on this dimension, using fragment-based and triple-based categories to describe the state of the art, as we found it particularly relevant to our study.

The remainder of this section analyzes representative systems in both categories, detailing their partitioning mechanisms and query optimization techniques. We conclude by synthesizing the limitations inherent to current approaches and identifying opportunities for research.

1.2 Triple-based Partitioning

Most RDF database systems adopt the triple as the fundamental unit of data partitioning. The primary objective of this approach is to distribute triples across multiple storage according to a defined allocation strategy. Systems in this category draw significant inspiration from the relational model, which has profoundly influenced their design philosophy. Rather than treating RDF data as a cohesive graph structure with interconnected nodes and edges, these systems decompose it into discrete sets of triples. Below, we discuss the most representative systems in this category:

TriAD [5] uses hash-based partitioning on the subject of each triple and introduces two key optimizations. First, it constructs six SPO permutation indexes (SPO, SOP, OSP, OPS, PSO, POS) over distributed triples, which are stored in a distributed in-memory structure designed for resource efficiency using integer encoding and vector-based representations. Second, it employs a join-ahead pruning technique that leverages an RDF summary graph to eliminate irrelevant intermediate results during query processing. The RDF summary graph is a compressed abstraction of the original RDF graph, in which supernodes represent subgraphs of the original graph. This summary graph is maintained at the master node to provide a global view of data locality.

GStore-D [10] partitions the RDF graph into vertex-disjoint subgraphs using METIS, a widely-used graph partitioning algorithm. Given the RDF graph and the desired number of partitions K, METIS generates distinct partitions while aiming to minimize the number of edges cut between partitions. GStore-D replicates the edges at the boundaries

of the partitions to enable the evaluation of certain queries within a k-hop neighborhood. Its latest version adopts the Minimum Property Cut (MPC) strategy [11] instead of METIS. Unlike METIS, which minimizes edge cuts, MPC's primary goal is to minimize the number of properties crossing between partitions. To achieve this, MPC employs a greedy algorithm that selects internal properties, thereby maximizing the number of internal properties within each partition.

Trinity.RDF [4] employs hash-based triple partitioning to partition the initial RDF graph. The partitioned triples within each partition are stored using bidirectional adjacency lists (capturing both inward and outward edges of nodes). Trinity.RDF is built on top of the distributed in-memory database Trinity. Since Trinity.RDF stores the entire RDF dataset in main memory, the initial arbitrary hash-based partitioning strategy is justified, albeit at the cost of substantial memory overhead.

S2RDF [12] extends vertical partitioning—which involves creating a separate subject-object table for each predicate in the RDF dataset—with a novel technique called **ExtVP**. ExtVP precomputes join reductions between predicate-based tables during preprocessing (e.g., subject-subject joins between table A corresponding to predicate $p1$ and table B corresponding to predicate $p2$). All possible joins are computed upfront. Built on top of Spark, S2RDF materializes the resulting tables in HDFS, and queries are executed using Spark SQL. While ExtVP effectively reduces join operations during query execution, it substantially increases storage requirements—by approximately an order of magnitude—compared to standard vertical partitioning, due to the exhaustive computation and storage of all predicate-table joins.

In triple-based partitioning schemes, partitioning is performed at the physical level, which prevents the database administrator from leveraging domain-specific knowledge for customization. Most approaches in this category rely on join-based evaluation for SPARQL queries, which can generate large intermediate results, leading to expensive shuffle operations and reduced performance. We also believe that grouping triples into more cohesive and semantic groups will significantly improve performance.

1.3 Fragment-based Partitioning

In fragment-based partitioning, larger subgraphs—referred to as fragments—are formed by grouping sets of interrelated triples based on either logical structures (e.g., characteristic sets of different entities in the RDF graph) or physical structures (e.g., rooted subgraphs). RDF database systems in this category typically require significant preprocessing time to construct the fragments, although various techniques have been proposed to accelerate this step. In the following, we describe the most representative systems that fall under this category.

SemStore [8] leverages the physical structure of the RDF graph by defining fragments based on Rooted Subgraphs (RSGs)—tree-like data structures that start from a root node and expand outward by exploring neighboring nodes until reaching the leaves; multiple such subgraphs can exist in an RDF graph. Each RSG corresponds to a distinct fragment. These fragments are then allocated to computing nodes using the K-Means clustering algorithm. SemStore is built on top of the centralized RDF store TripleBit. On each node, data fragments are stored and locally indexed using TripleBit.

SPT+VP [13] combines two partitioning strategies. First, it employs a modified

property table approach by constructing a single, large property table—called the Subset Property Table (SPT)—which includes a subject column and one column for each property in the RDF graph. This table is then split into smaller tables, which serve as the partitioned fragments. In addition to these fragments, traditional vertical partitions are also generated. SPT+VP is built on top of Spark, translating SPARQL queries into Spark SQL statements specifically optimized for the underlying partitioning schema.

Partout [7] is a workload-driven approach that leverages the query workload to extract a set of simple predicates. A simple predicate is a boolean condition on the predicates of the RDF data. These are then used to generate minterms, a concept originally introduced in the relational model. Minterms represent the minimal combinations of simple predicates that define the different fragments. Partout iterates over these minterms to create one fragment per minterm, where each fragment contains all triples that satisfy the corresponding conditions. Based on the query workload, Partout computes a benefit score for assigning each fragment to a specific machine and performs the allocation accordingly. It is built on top of the well-known RDF system RDF-3X.

Stylus [14] defines a high-level schema for RDF data, similar to RDFs, and partitions the data into fragments based on this schema. Unlike static schemas, Stylus dynamically constructs its schema using insights from query workloads. Entities belonging to the same semantic category (as defined by the schema) are grouped into fragments, which are stored as materialized views for efficient querying.Stylus is built on top of Trinity.

Leon [15] partitions triples based on the characteristic sets of their subjects, grouping all entities with the same characteristic set into a single fragment. Leon employs a multi-query optimization technique based on the query workload: given a workload, the method searches for an effective way to evaluate and share the results for common subqueries. Leon is an in-memory RDF system.

Fragment-based approaches are rigid and ignore the semantics of graph entities. Even when these systems use a workload-driven approach to define fragments, their structure is initially fixed and never updated. Similar to triple-based partitioning, these systems exclude the database administrator from the loop, preventing customization of fragmentation and repartitioning.

1.4 Conclusion

The design of a new graph partitioning framework is necessary to overcome the limitations of existing systems. This framework should keep the administrator in the loop by providing a set of tools that enable personalized partitioning and repartitioning. It should also strike a balance between scalability and performance when integrated into an RDF system. In the following chapter, we discuss the SemPart framework, which addresses these limitations.

Chapter 2

Current Solutions

2.1 Introduction

In the previous section, we categorized the different RDF systems based on their partitioning strategies and highlighted the limitations of each category. In this section, we introduce PQDAG, a distributed RDF system that offers potential compromises to address these limitations and can be extended to support solutions for existing systems. We also present our initial vision of the framework proposed by the PQDAG team: the SemPart framework. We note that at the beginning of our work, SemPart was purely theoretical, and we carried out all the necessary implementation.

2.2 PQDAG

PQDAG is a distributed RDF system built on top of QDAG [9]. Unlike most RDF systems that rely on join-based query evaluation, PQDAG adopts a logical graph exploration strategy, made possible by its fragmentation-based partitioning approach. In this work, we deliberately do not address the query optimization or execution layers, as they fall outside the scope of our study. Instead, we focus exclusively on the partitioning module, which is central to and highly influential on our contribution.

2.2.1 Initial Partitioning Approach

We present the partitioning approach that existed at the time we began our work. This approach is based on fragmentation, which consists of grouping sets of RDF triples to form multiple subgraphs. In PQDAG, this grouping relies on the concept of characteristic sets [16]. In all the following examples, we refer to the RDF graph shown in Figure 2.1.

Definition 1 (**Characteristic set**). *A characteristic set is defined as the set of all predicates associated with a given subject. Formally, for a subject s in an RDF graph G, its characteristic set is defined as:*

$$SC(s) := \{\, p \mid \exists\, o : (s, p, o) \in G\}.$$

Similarly, for objects, the characteristic set of an object o is the set of all predicates that have o as their object in an RDF triple.

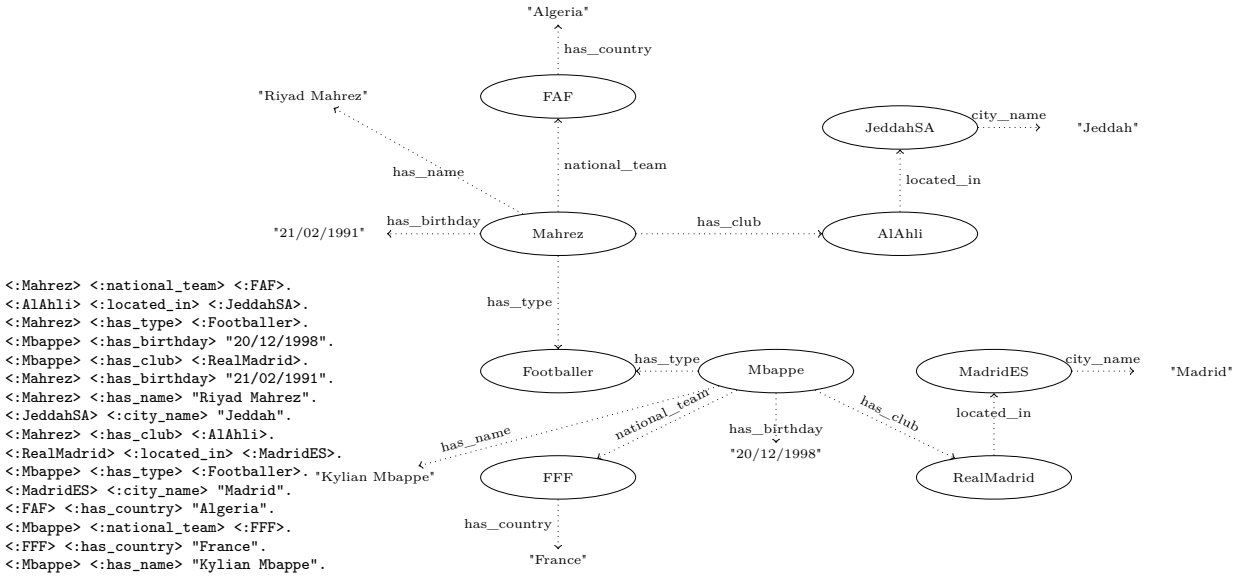

Figure 2.1: An RDF graph $\mathcal{G}$ and its corresponding triple representation

To enumerate the set of all characteristic sets in the RDF graph G, we consider two logical structures: forward graph stars, which represent outgoing edges from a subject, and backward graph stars, which represent incoming edges to an object.

Definition 2 (**Forward graph star**)**.** A forward graph star *centered at a subject s in an RDF graph G, denoted by $\overrightarrow{\mathcal{G}s}$, is defined as the subset of triples in G that share the same subject s. Formally,*

$$\overrightarrow{\mathcal{G}s} = \{\, t \in G \mid f_s(t) = s\},$$

Example 1. *Consider our "Footballers-Clubs" graph. A forward graph star $\overrightarrow{\texttt{Mahrez}}$ is the set of all triples in G whose subject is* `Mahrez`*:*

$$\begin{aligned}\overrightarrow{\texttt{Mahrez}} = \{&\langle \texttt{Mahrez}, \texttt{has\_type}, \texttt{Footballer}\rangle, \langle \texttt{Mahrez}, \texttt{has\_name}, \texttt{Riyad Mahrez}\rangle,\\ &\langle \texttt{Mahrez}, \texttt{has\_birthday}, \texttt{21/02/1991}\rangle, \langle \texttt{Mahrez}, \texttt{national\_team}, FAF\rangle,\\ &\langle \texttt{Mahrez}, \texttt{has\_club}, \texttt{AlAhli}\rangle\}.\end{aligned}$$

Definition 3 (**Backward graph star**)**.** A backward graph star *centered at an object o in an RDF graph G, denoted by $\overleftarrow{\mathcal{G}s}$, is defined as the subset of triples in G that share the same object o. Formally,*

$$\overleftarrow{\mathcal{G}s} = \{\, t \in G \mid f_o(t) = o\},$$

Example 2. *Continuing with our "Footballers-Clubs" graph, consider a backward graph star $\overleftarrow{\texttt{MadridES}}$ of head* `MadridES`*. It contains all triples whose object is* `MadridES`*, for instance:*

$$\overleftarrow{\texttt{MadridES}} = \{\langle \texttt{RealMadrid}, \texttt{located\_in}, \texttt{MadridES}\rangle\}.$$

Based on the notions of forward and backward graph stars, we define the set of all characteristic sets of an RDF graph G as follows.

Definition 4 (Set of all characteristic sets)**.** *Let $P(\overrightarrow{\mathcal{G}s})$ denote the set of all predicates labeling the edges in the forward graph star $\overrightarrow{\mathcal{G}s}$ of the RDF graph G, and let $P(\overleftarrow{\mathcal{G}s})$ denote*

the set of all predicates labeling the edges in the backward graph star $\overleftarrow{\mathcal{G}s}$ *of* G*. Then the* set of all characteristic sets *of* G*, denoted* $\mathcal{SC}(G)$*, is given by*

$$\mathcal{SC}(G) \;=\; \Big\{P\big(\overrightarrow{\mathcal{G}s}\big) \;\Big|\; \overrightarrow{\mathcal{G}s} \subseteq G\Big\} \;\cup\; \Big\{P\big(\overleftarrow{\mathcal{G}s}\big) \;\Big|\; \overleftarrow{\mathcal{G}s} \subseteq G\Big\}.$$

Example 3. *Examples of characteristic sets include a forward characteristic set describing Footballer,* $\{$`has_club`, `has_birthday`, `has_name`, `national_team`, `has_type`$\}$*, and a backward characteristic set describing cities based on their incoming edges,* $\{$`located_in`$\}$*.*

Hence, each distinct characteristic set in $\mathcal{SC}(G)$ defines either a forward or a backward graph fragment, depending on whether the associated edges are outgoing or incoming. The formal definition of these fragments is provided below.

Definition 5. *(Forward graph fragment) For a given characteristic set* $C \in \mathcal{SC}(G)$ *that represents a set of* outgoing *predicates (from a subject), the* forward graph fragment $\overrightarrow{\mathcal{G}f}(C)$ *is defined as the set of all forward graph stars whose associated predicate set is exactly* C*:*

$$\overrightarrow{\mathcal{G}f(C)} \;=\; \Big\{\overrightarrow{\mathcal{G}s} \subseteq G \;\Big|\; P(\overrightarrow{\mathcal{G}s}) = C\Big\}.$$

Example 4. *All footballer (forward graph stars) whose characteristic set is* `{has_club, has_birthday, has_name, national_team, has_type}`*, namely* $\overrightarrow{\texttt{Mahrez}}$*,* *and* $\overrightarrow{\texttt{Mbappe}}$*, belong to the same forward graph fragment.*

Definition 6. *(Backward graph fragment) For a given characteristic set* $C \in \mathcal{SC}(G)$ *that corresponds to* incoming *predicates (to an object), the* backward graph fragment $\overleftarrow{\mathcal{G}f}(C)$ *is defined as the set of all backward graph stars whose associated predicate set is exactly* C*:*

$$\overleftarrow{\mathcal{G}f(C)} \;=\; \Big\{\overleftarrow{\mathcal{G}s} \subseteq G \;\Big|\; P(\overleftarrow{\mathcal{G}s}) = C\Big\}.$$

Example 5. *All city entities (backward graph stars) whose characteristic set is* `{located_in}`*,* *namely* $\overleftarrow{\texttt{MadridES}}$ *and* $\overleftarrow{\texttt{JeddahSA}}$*, belong to the same backward graph fragment.*

We also define the notion of neighborhood between fragments, which is crucial in our context: when the structure of a fragment is modified, its neighboring fragments must also be updated accordingly.

Definition 7. *(Neighborhood Relation). Fragments* $\mathcal{G}_{f_i}$ *and* $\mathcal{G}_{f_j}$ *are neighbors via predicate* p *if there exists an edge labeled* p *that connects a vertex in* $\mathcal{G}_{f_i}$ *to a vertex in* $\mathcal{G}_{f_j}$ $(\mathcal{G}_{f_i} \sim_p \mathcal{G}_{f_j})$*. In this formulation, we assume that both fragments are forward graph fragments, and the connection is therefore of type* object-subject*. Formally*

$$\exists \mathcal{G}_{s_i} \in \mathcal{G}_{f_i},\, \mathcal{G}_{s_j} \in \mathcal{G}_{f_j} : (s,p,o) \in \mathcal{G}_{s_i} \;\wedge\; o \in \mathit{Subjects}(\mathcal{G}_{f_j})$$

where $\mathit{Subjects}(\mathcal{G}_f)$ *denotes the set of subjects in fragment* $\mathcal{G}_f$*.*

At this stage, the partitioning framework in PQDAG was highly rigid: fragment creation was strictly tied to characteristic sets, with each set defining a distinct fragment. Once the fragments were generated, no further modifications could be made, and no repartitioning could be triggered. This rigidity highlighted the need for a more flexible partitioning strategy—still grounded in the concept of characteristic sets, but one that could support guided dynamic repartitioning.

To achieve this flexibility, the PQDAG team introduced the database administrator as an active participant in the partitioning process. By providing a rich and expressive language, the administrator can tailor the partitioning strategy based on their expertise and knowledge of the RDF data.

In the following section, we present the theoretical foundations of the SemPart framework and outline its key components.

2.3 The SemPart Framework

In this section, we introduce the SemPart partitioning framework, which supports both initial partitioning and guided dynamic repartitioning. Figure 2.2 illustrates the overall architecture of the SemPart framework.

SemPart operates in three main phases: (i) It begins with logical fragmentation, where logical fragments are defined based on the notion of characteristic sets. (ii) These logical fragments are then transformed into physical fragments through a suite of transformation operators such as filtering, splitting, and integration (physical fragmentation). (iii) The resulting physical fragments are then allocated to different sites using a chosen allocation strategy.

Additionally, the database administrator (DBA) can initiate dynamic repartitioning at any point using the same set of transformation operators. All partitioning operations—initial or dynamic—are orchestrated through RDPAL (RDF Data Partitioning and Allocation Language), a dedicated language designed for flexible and customizable partitioning workflows.

2.3.1 Logical Fragmentation

The process begins by generating logical fragments of the RDF dataset from characteristic sets, following precisely the procedure detailed in the preceding section. This step concludes with all logical fragments created.

2.3.2 Physical Fragmentation

In this section, we offer a unified perspective on how logical fragments are transformed into physical fragments within SemPart. The process relies on two families of transformation operators:

1. **Core transformation operators** These operators select, split, and merge fragments, and can also act on neighboring fragments. Each invocation produces virtual fragments: transient fragments that are not yet materialized on disk.

2. **Control operators** These govern the life cycle of virtual fragments, providing explicit actions to persist them to storage, merge them with existing fragments, or discard them altogether.

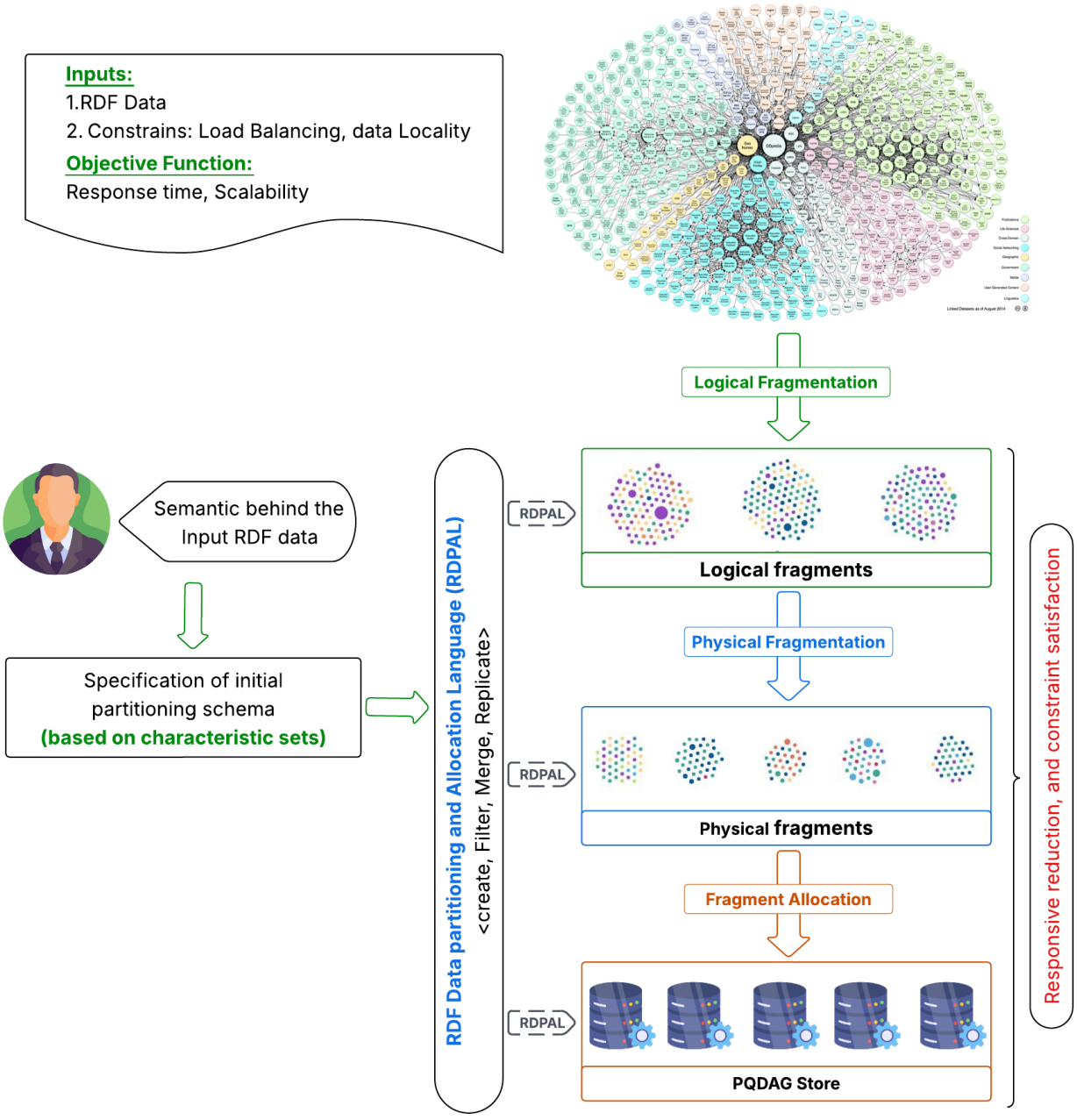

Figure 2.2: Overall Architecture of our System

All operators are exposed through the PQDAG RDF Data Partitioning and Allocation Language (RDPAL). With RDPAL, a database administrator can tailor the partitioning workflow by chaining operator calls on logical and virtual fragments, and then materializing the final results as physical fragments. SemPart follows a lazy-evaluation strategy: physical fragments are materialized only when needed. The sequence of operator calls is internally captured as a directed acyclic graph (DAG), a mechanism described in detail

in the following section. Table 2.1 provides a comprehensive summary of all operators and their corresponding descriptions.

Table 2.1: Unified summary of operators and their purpose.

Operator	Description
Filter $\nu_\Gamma(\{\mathcal{G}_{f_1}, \ldots, \mathcal{G}_{f_k}\}) \rightarrow \{\mathcal{G}_{f_i}, \ldots, \mathcal{G}_{f_j}\}$	Filters graph-stars in each fragment that satisfy the condition Γ.
Selection $\sigma_\Gamma(\{\mathcal{G}_{f_1}, \ldots, \mathcal{G}_{f_k}\}) \rightarrow \{\mathcal{G}_{f_i}, \ldots, \mathcal{G}_{f_j}\}$	Selects whole fragments that match Γ.
Neighbor selection $\nu_{\text{path}}(\{\mathcal{G}_{f_1}, \ldots, \mathcal{G}_{f_k}\}) \rightarrow \{\mathcal{G}^{\texttt{nbr}}_{f_1}, \ldots, \mathcal{G}^{\texttt{nbr}}_{f_k}\}$	Follows a predicate path from each fragment and produces fragments for the reached entities.
Simple split ($\texttt{split}_N$)	Splits fragments when they exceed a size or count threshold.
Conditional split $\text{split}_\Gamma(\{\mathcal{G}_{f_1}, \ldots, \mathcal{G}_{f_m}\}) \rightarrow \{\mathcal{G}^{\text{T}}_{f_1}, \mathcal{G}^{\text{F}}_{f_1}, \ldots\}$	Partitions every fragment into *true* / *false* subsets according to Γ.
Group-by split $\text{groupby}_p(\{\mathcal{G}_{f_1}, \ldots, \mathcal{G}_{f_m}\}) \rightarrow \{\mathcal{G}^o_{f_i}\}_{i,o}$	Forms subsets by grouping graph-stars according to the distinct values of predicate p.
Derived split $\text{derivedSplit}(\{\mathcal{G}_{f_1}, \ldots, \mathcal{G}_{f_m}\}) \rightarrow \{\mathcal{G}'_{f_1}, \ldots, \mathcal{G}'_{f_n}\}$	Applies strategy S to source fragments, then propagates the split to their neighbors along *path*. Another important parameter in this case is the list of fragments, from which we select the neighbors. Additionally, the conflict resolution strategy is also a key parameter, which will be discussed in the next chapter.
Union ($\cup$)	Combines graph-stars from two fragments.
Intersection ($\cap$)	Keeps only graph-stars present in both fragments.
Difference ($\setminus$)	Removes graph-stars from the first fragment that also occur in the second.
Materialization ($\mathcal{M}$)	Persists virtual fragments to physical storage for later reuse.
Integration (μ)	Merges virtual fragments into an existing collection.
Remove ($\mathcal{R}$)	Marks fragments inactive or deletes them from the working set.

DAG-Based Execution of Transformation Operators

To reduce the overhead associated with materializing temporary fragments, SemPart adopts a lazy evaluation strategy, postponing physical storage until it is explicitly required. Transformation workflows are captured as a Directed Acyclic Graph (DAG), which allows for systematic tracking of dependencies and facilitates optimization opportunities. The structure of the DAG is defined as follows:

- **Initial Nodes**: Represent the original logical fragments.

- **Intermediate Nodes**: Correspond to virtual fragments produced by transformation operators. These fragments remain virtual unless explicitly materialized.
- **Arcs**: Directed edges labeled with the applied transformation operators, indicating the flow of operations.
- **Materialized Nodes**: Denote fragments that have been persisted to physical storage through the invocation of the $\mathcal{M}$(Materialization) operator.

When materialization is triggered, the DAG is traversed from the initial nodes to the target materialized node, following all dependency paths required to compute the final fragment.

Example In the following, we show a use case of our framework to partition the initial RDF graph in Figure 2.1, and apply some transformations to it. We consider only the forward direction, so the set of initial logical fragments is $\{\mathcal{G}_{f_{\text{footballers}}}, \mathcal{G}_{f_{\text{teams}}}, \mathcal{G}_{f_{\text{clubs}}}, \mathcal{G}_{f_{\text{cities}}}\}$.

Example 6. *Consider an example where the DBA needs to retrieve complete footballer informations (team, club and club location) grouped by their clubs. The following transformations over the RDF graph in Figure 2.1 can be applied:*

1. ***Initial Filtering**: We begin with four forward fragments, $\{\mathcal{G}_{f_{footballers}}, \mathcal{G}_{f_{teams}}, \mathcal{G}_{f_{clubs}}, \mathcal{G}_{f_{cities}}\}$. Among these, we choose the fragment whose characteristic set is associated with footballers (e.g., predicates such as* `has_club` *and* `national_team`*). As a result, we obtain the single fragment $\{\mathcal{G}_{f_{footballers}}\}$.*
2. ***Derived Split on Footballers**: We apply the* Derived Split *operator twice to the footballers fragment using a baseline* `groupby` *strategy on the* `has_club` *predicate. This is done in order to propagate the split to the neighbors of the footballers through two paths:* `has_club` $\rightarrow$ `located_in` *and* `national_team`*. As a result, we obtain the new fragments in $\mathcal{G}_2$ and $\mathcal{G}_3$, respectively.*
3. ***Integration**: We first integrate $\mathcal{G}_3$, and before integrating $\mathcal{G}_2$ we remove the duplicates $\mathcal{G}^{AlAhli}_{f_{footballers}}$ and $\mathcal{G}^{RealMadrid}_{f_{footballers}}$ from $\mathcal{G}_2$ because they already exist in $\mathcal{G}_3$. As a result, $\mathcal{G}_5$ contains the initial fragments, along with the new fragments generated from the two derived split operations. Then, we remove from $\mathcal{G}_5$ the initial fragments that were originally in $\mathcal{G}_0$ to avoid duplicates.*
4. ***Materialization**: The split fragments in G_6 are finally materialized ($\mathcal{M}$) into physical fragments.*

Figure 2.3 shows the DAG for this example. These transformations produce independent subgraphs, each containing footballers, their teams, their associated clubs, and corresponding cities. These could be distributed across different sites, enhancing load balancing and data locality for footballers-centric queries.

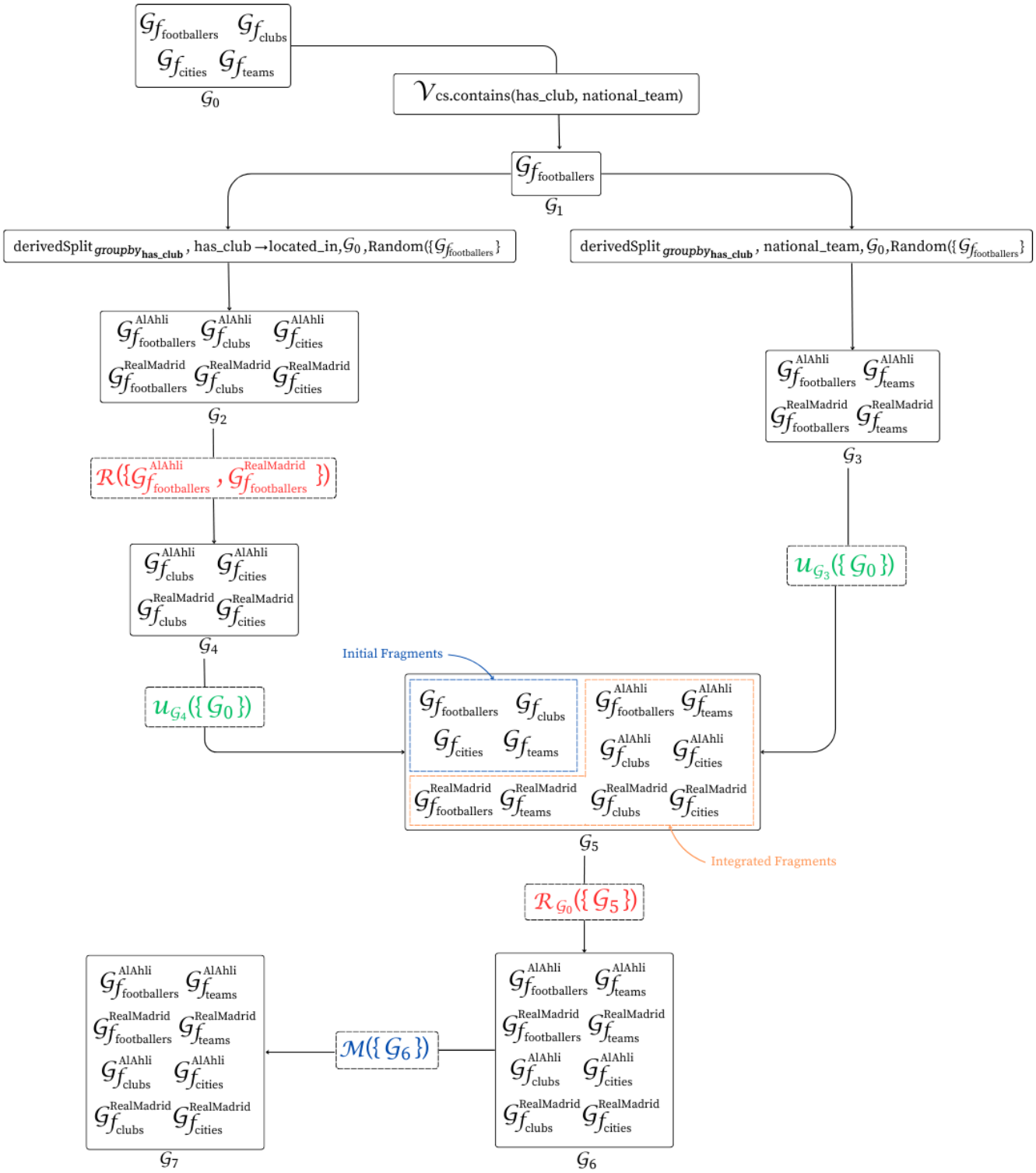

Figure 2.3: DAG representation of the transformation process for Example 6

2.3.3 Allocation

As described in the previous step, the result of physical fragmentation is a set of materialized fragments (also referred to as physical fragments). The allocation operator takes these fragments as input and distributes them across multiple sites based on a strategy selected by the database administrator (DBA).

The allocation task can be formalized as a graph partitioning problem, which is known to be NP-Hard [17]. Figure 2.4 illustrates an example of a fragment graph, where nodes represent materialized fragments, and edges represent shared predicates between fragments. Edge weights indicate the number of predicate connections between the corresponding fragments. Node weights indicates the number of triples in each fragment.

As a result, classical solutions to the graph partitioning problem are directly applicable to the allocation phase. SemPart provides the DBA with several allocation strategies, including the following:

- **Naïve Strategy**: A simple round-robin approach that assigns fragments to machines without considering the structure of the fragment graph.
- **Heuristic-Based Strategies**: These strategies take the graph structure into account and are suitable for large-scale scenarios:
  - `METIS`: A widely used graph partitioning tool that aims to minimize the number of edge cuts.
  - `MPC (Minimum Property Cut)`: A strategy that minimizes cuts across shared properties (predicates) between fragments.

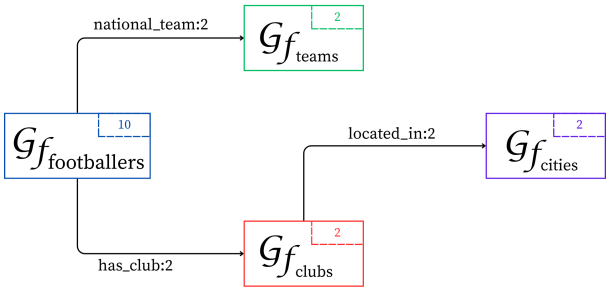

Figure 2.4: An example of forward fragments graph and their connections.

2.3.4 Repartitionning

After the initial partitioning, repartitioning becomes a crucial and mandatory step, guided by the database administrator (DBA). During this phase, the DBA can select and transform existing fragments before reallocating them using one of the available strategies. In addition to the allocation operator, the language also provides a dedicated redistribute operator, which allows the DBA to move fragments from one site to another, enabling fine-grained control over fragment placement during guided dynamic repartitioning.

Throughout all stages of the SemPart framework, the RDPAL (RDF Data Partitioning and Allocation Language) plays a central role, providing unified access to all functionalities involved in the partitioning and allocation process. The language is driven by the database administrator (DBA), who uses it to guide and customize each step. At the beginning of our work, these aspects were purely theoretical; our primary objective was to fully implement RDPAL, equipping it with all the necessary features to support each phase of the framework.

2.4 Conclusion

In this section, we presented the PQDAG team's vision of the SemPart framework, outlining each of its key components. This framework is built around the RDPAL language, whose full implementation—with all associated features—constitutes the core focus of our work. We have carefully studied and analyzed every aspect of the framework to ensure a comprehensive understanding. In the following section, we present our design of the proposed solution for implementing RDPAL, along with all relevant details.

Chapter 3

Analysis and Design

3.1 Introduction

As previously discussed, the SemPart framework was originally introduced as a theoretical concept. This chapter explains how we transformed the ideas of the PQDAG team into a concrete and functional software solution. We present the analysis and design of our solution, which is intended to offer a structured and efficient response to the problem under study. This phase of the system development process is essential since it helps define the structure of the system and make the requirements more clear. This chapter also presents detailed interaction flows between the various components of the SemPart implementation, using sequence diagrams to offer a comprehensive view of their dynamic behavior.

3.2 Requirement Analysis

This section outlines the key requirements that guided the design and development of the SemPart framework. These requirements explain what SemPart is expected to accomplish and how well it should perform.

3.2.1 Functional Requirements

The implementation of the SemPart framework is guided by three main functional requirements, which cover all its underlying theoretical aspects. These requirements are presented in order of importance, starting from the most critical to the least:

1. **Physical Transformation Operators:** Physical transformations are a central aspect of SemPart, as they encapsulate the DBAs expertise. Without them, domain knowledge cannot be effectively incorporated into the resulting RDF partitions. For this reason, they are considered the most important and critical requirement, forming the starting point of our design and implementation process.

2. **RDF Data Partitioning and Allocation Language:** The next key requirement is RDPAL, as it is the only component that enables the DBA to interact with all stages of SemPart, including the physical transformations introduced in the first requirement.

3. **Graphical User Platform:** To simplify the DBA's work, a graphical user interface should be provided, enabling them to write RDPAL code and execute it directly on the RDF data.

3.2.2 Non-Functional Requirements

The following points outline the non-functional requirements that our implementation of SemPart must fulfill to ensure a powerful and scalable framework.

- **Performance:** Performance stands out as the most critical non-functional requirement. All operations in SemPart, especially physical transformations should be implemented in an optimized manner, enabling the framework to minimize execution time and handle large RDF data fragments effectively.
- **Ergonomics:** The graphical user interface should be designed to be clear, intuitive, and ergonomically suited for efficient use by the DBA.

3.3 Detailed Design

This section provides a conceptual overview of our implementation of SemPart. Each functional requirement is discussed individually, accompanied by a class diagram illustrating its architecture. We explain our design decisions and justify the selection of specific classes and relationships.

3.3.1 Physical Transformation Operators

Two key aspects define this requirement: (1) all physical transformation operators are classified into two categories, core operators and control operators; and (2) a DAG (Directed Acyclic Graph) structure is used to organize these operators. Our design is fundamentally based on these two principles. Figure 3.1 presents the class diagram that encapsulates both aspects.

The RDF data is represented by the class `RDFGraph`, which contains a list of `Fragment` objects. These fragments come from the logical fragmentation phase and consist of RDF triples. Each triple represented by the `Triple` class includes a subject, a predicate, and an object.

Once the logical fragments are available, the transformation process begins. The class `FragmentationEngine` is responsible for executing a transformation workflow, which is defined as a Directed Acyclic Graph (DAG). The `DAG` is made up of several `DAGNode` elements. Each node contains an operator that applies a transformation to one or more fragments. These nodes are linked together so that the operations are executed step by step, in a specific order.

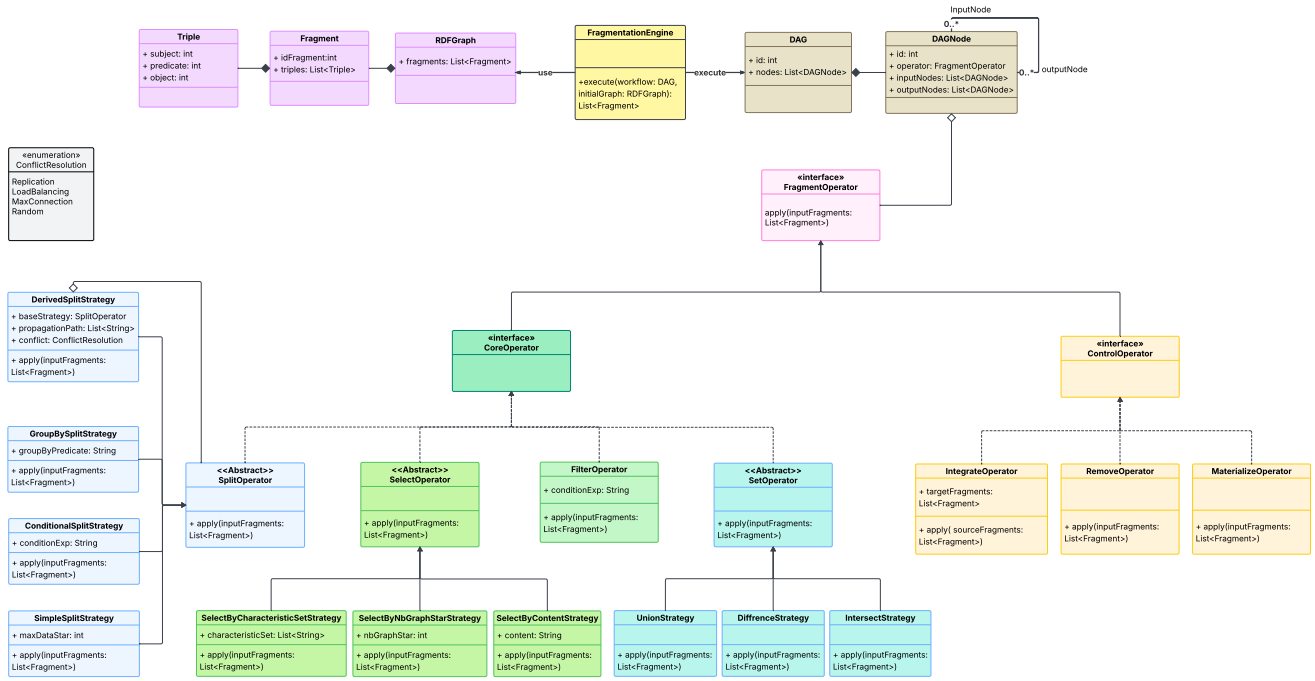

Figure 3.1: Class Diagram for Physical Transformation Management

All transformation operations implement the **`FragmentOperator`** interface, which defines the general behavior expected from any transformation step. Among them, core transformation types such as selection, splitting, and set operations are grouped under the **`CoreOperator`** interface, these operators produce virtual fragments, which are temporary results not yet stored in the system. On the other hand, **`ControlOperators`** are responsible for managing these virtual fragments, this includes actions such as materializing, integrating, or removing fragments.
The operators are organized into distinct categories, each represented by an abstract class, **`SplitOperator`**, **`SelectOperator`**, and **`SetOperator`**, with each class handling a specific type of transformation. All the operators listed in Table 2.1 are provided in the class diagram. As previously described in the chapter "Current Solutions", the derived split operator includes a parameter called conflict strategy, which is used to handle propagation conflicts when splitting data across neighboring fragments. To support this mechanism, we define a ConflictResolution enumeration that specifies the available strategies: **(1) Replication:** Copies the graph star to all resulting sub-fragments. **(2) LoadBalancing:** Assigns the graph star to the subfragment with the least current load. **(3) MaxConnection:** Prioritizes subfragments that have the strongest edge weights with the original fragment. **(4) Random:** Performs an arbitrary assignment of the graph star.

This conception was chosen not only to satisfy the functional requirements of the system, but also to keep the solution solid, easy to maintain, and flexible for future changes. To support this, we applied several key principles of object-oriented design, specifically the **SOLID principles**, as introduced by Robert C. Martin [18]:

- **Single Responsibility Principle (SRP):** A class should have just one reason to change. For instance, FragmentationEngine executes workflows, while each operator class handles one transformation type.

- **Open/Closed Principle (OCP):** Classes should be open for extension but closed for modification. For example, it is easy to add new strategies, just create a class that implements the existing interfaces, without modifying any existing code.

In terms of design patterns, we applied two where they bring real value, following the structure described by Alexander Shvets in *Dive Into Design Patterns* [19]:

- **The Strategy pattern** is used in the selection, splitting and sets operations. Each strategy is encapsulated in its own class, making it easy to switch or add new ones without touching the main logic.
- **The Composite pattern** is used in the DerivedSplitStrategy class. This class includes another splitting strategy and treats it as part of its own logic. Because all split strategies follow the same interface, the system can use both simple and combined strategies in the same way.

Overall, this design is modular, flexible, and easy to extend. Each component has a clear role, and new features can be added without destabilizing the whole architecture. This makes it the most appropriate structure for implementing physical transformations within the SemPart framework.

3.3.2 RDF Data Partitioning And Allocation Language

RDPAL is the main high-level component of SemPart, serving as the bridge between the DBA and the physical partitions. We implemented RDPAL with inspiration from the Spark framework, particularly in the way it enables method chaining—translated in our case as operator chaining. Figure 3.2 presents the class diagram illustrating the key components of RDPAL.

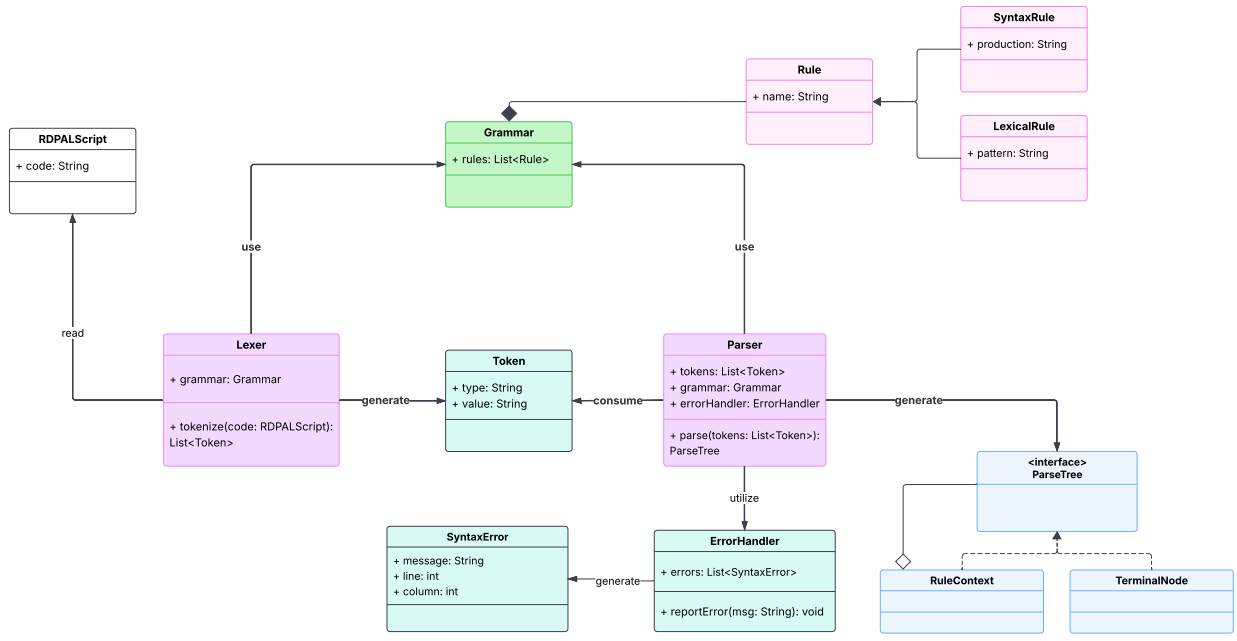

Figure 3.2: Class Diagram for Dedicated Language

The `RDPALScript` class acts as the starting point, representing the code written by the administrator. This code goes through two main steps: lexical analysis and syntactic analysis.

The `Lexer` component performs lexical analysis by dividing the script into a series of `Token` objects. The smallest fundamental components of the language, such as identifiers, symbols, or keywords, are represented by these tokens. The `Grammar` component contains a predefined set of lexical rules that are used by the `Lexer`. These rules are based on a general `Rule` class, extended specifically by `LexicalRule` to define the patterns needed to recognize tokens.

Once the tokens are generated, they are sent to the `Parser`, which uses the grammar's syntax rules to determine their order. These rules, provided by the `SyntaxRule` class, guide the construction of a parse tree that reflects the hierarchical structure of the input. The parser builds this tree using two main types of nodes: `RuleContext` for grouped expressions, and `TerminalNode` for individual tokens. Both node types implement the `ParseTree` interface, ensuring uniform manipulation of the tree structure during analysis.

An `ErrorHandler` component is also integrated into the parser to guarantee clarity and robustness. This component collects and reports any syntactic errors that are encountered during parsing. Each issue is stored in a `SyntaxError` object, which includes useful details like the error message, the line number, and the exact column. This helps administrators quickly fix mistakes in their scripts.

In the design of RDPAL, we applied the **Single Responsibility Principle (SRP)**, one of the core **SOLID** principles introduced by Robert C. Martin [18]. This principle ensures that each class is focused on a single responsibility. For example, the Lexer handles only lexical analysis, the Parser builds the parse tree from tokens, and the ErrorHandler is dedicated to managing error reporting independently. This clear separation of concerns improves code readability, simplifies debugging, and facilitates future maintenance and evolution of the system.

We also used the **Composite** design pattern to model the structure of the parse tree, following the structure described by Alexander Shvets in **Dive Into Design Patterns** [19]. The `RuleContext` class can contain other nodes inside it, such as `TerminalNode` or even other `RuleContext` instances. This conception allows all elements of the tree to be processed using the same interface, which simplifies tree traversal.

Altogether, this class structure ensures a solid foundation for implementing and evolving the RDPAL while keeping the architecture clean, modular, and adaptable to future needs.

3.3.3 Graphical User Platform

In a complex management system like the one we are designing, the presence of a graphical interface is essential in order to ensure smooth and efficient interaction between the DBA and the system's various features. This requirement therefore focuses on the implementation of a user-friendly web platform designed to centralize all administrative actions within a unified, intuitive, and accessible interface.

This platform serves as an intermediary between the administrator and the internal components of the system. It encapsulates technical operations within accessible features, streamlining the management of projects, the configuration of machines, dataset handling, and execution monitoring.

The main focus is on the interaction aspect: enabling the administrator to perform

tasks through a clear interface that hides technical details while ensuring good usability throughout the system.

To better understand the structure supporting this platform, the following class diagram (Figure 3.3) illustrates its main components and their relationships.

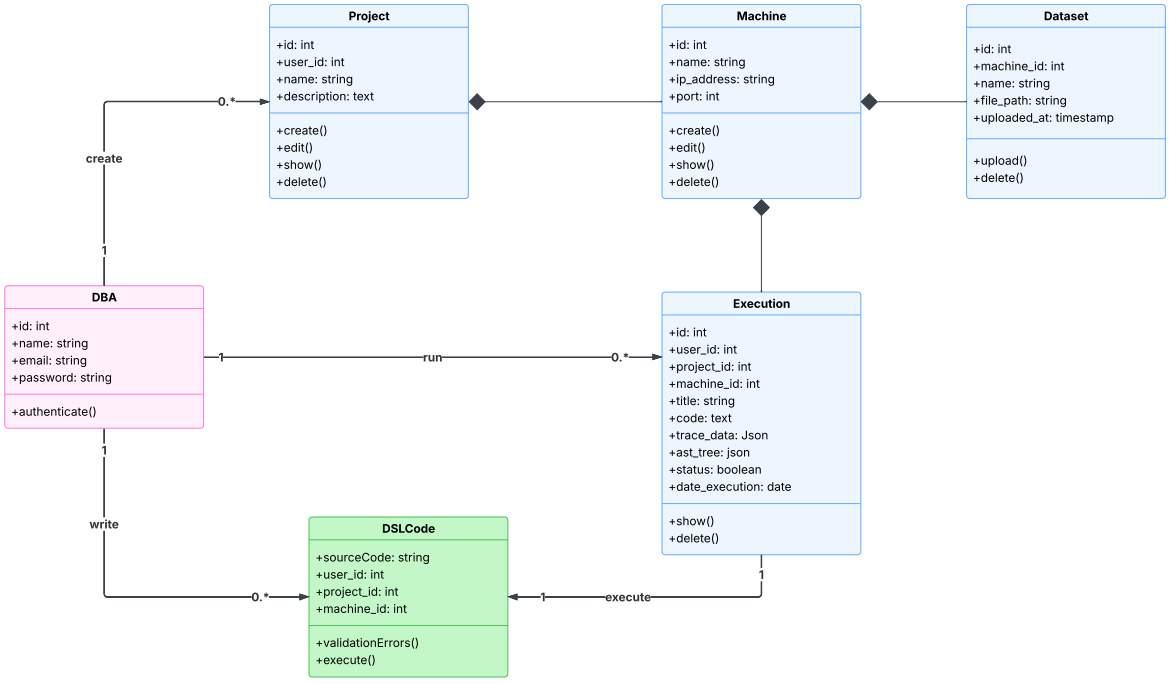

Figure 3.3: Class Diagram for Graphical User Platform

The **DBA** class, represents the administrator who interacts with the platform through a graphical interface. This user initiates and supervises the system's operations without directly handling technical configurations.

To organize the administrator's work, the **Project** class acts as a central workspace. It brings together all the necessary elements for a fragmentation task, including the machines that will perform the operations, the datasets to be processed, and the executions that record the results.

To support execution, the **Machine** class models the computing resources involved in processing. Each machine is linked to one or more projects and stores the datasets needed for execution. It also serves as the environment where the RDPAL script is executed. This structure lets the system distribute tasks across different machines.

The **Dataset** class represents the graph data used in fragmentation. These datasets are essential for executing any fragmentation process. To apply fragmentation rules, the administrator writes scripts captured in the **RDPALScript** class. These scripts define how the data should be partitioned, using a domain-specific language (DSL).

Once a fragmentation task is launched, the **Execution** class captures all relevant details of the process. It links the script used, the dataset targeted, the machine that performed the operation, and the resulting outputs. This record ensures traceability and lets the administrator review past operations as needed.

The structure of this class diagram was carefully chosen to ensure a clear and efficient organization of the platform's components. Each class has a specific and limited responsibility, which makes the system easier to understand, maintain, and extend. This

clarity in design is essential for a platform that will be actively used by administrators to manage complex tasks.

One of the strongest arguments for this conception is that it follows a modular approach. This idea is inspired by the **Unix philosophy** [20]: "do one thing and do it well". Each part of the system is responsible for a specific task. For example, the `RDPALScript` class is responsible for script definitions, while the Machine class represents the computing environment, and the Execution class records each run. This separation of concerns avoids complexity and improves reusability.

This approach also makes the system more reliable. Because each part has a clear role, problems are easier to locate and fix. The diagram reflects a modular conception, where changes in one part do not risk breaking others. This is especially important in a graphical platform where multiple operations are connected but must remain manageable.

3.4 System Interaction

To fully understand how the system operates as a whole, it is essential to observe the interaction between its main components. This section presents sequence diagrams that illustrate the flow from user input in the web interface to the execution of physical transformations. These interactions demonstrate how high-level transformation rules can easily be turned into concrete operations, ensuring a transparent, structured, and automated workflow.

3.4.1 From DBA Domain Knowledge to DAG

In this step, the DBA defines domain knowledge by expressing various transformations on the RDF fragments of the current partitions using RDPAL code. When the DBA runs the RDPAL script, the system interprets it and translates the high-level instructions into a DAG that represents the transformation workflow to be executed.

The sequence diagram in Figure 3.4 illustrates the process starting from the RDPAL code and leading to the generation of the DAG.

When the DBA triggers the execution of the RDPAL code, the system first verifies that all prerequisites are satisfied, including the existence of an active project, a configured machine, and the availability of the referenced dataset. If any of these elements are missing, the interface prompts the user to complete the setup. This validation loop continues until all conditions are satisfied. Once all requirements are met, the dataset is sent to the external server, and the RDPAL script is submitted for interpretation. The server parses and validates the rules to ensure syntactic and semantic correctness. If the code contains errors, they are reported back to the user. Otherwise, the server generates a DAG representing the transformation workflow. This DAG is then returned to the interface for visualization and stored for future execution.

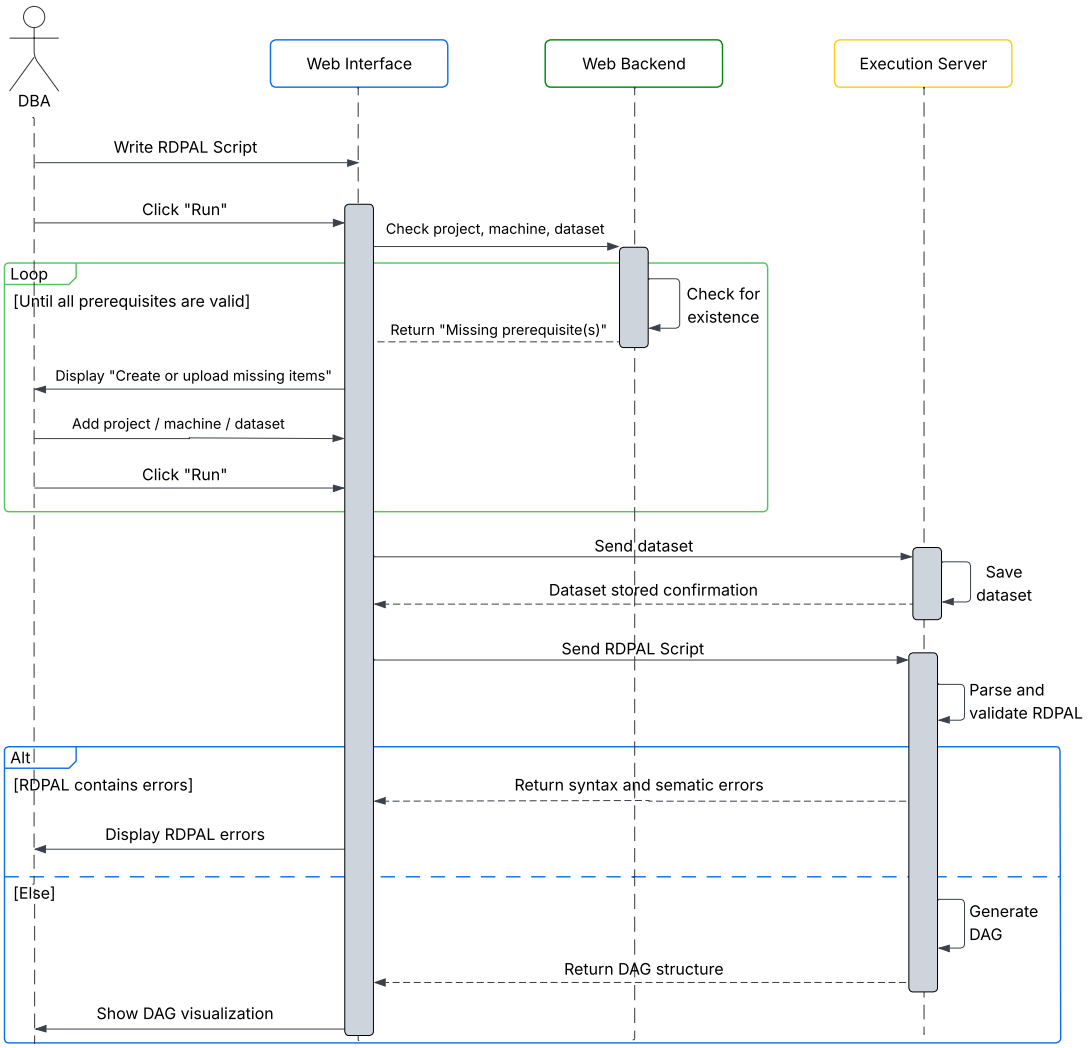

Figure 3.4: Run Code Interaction

3.4.2 DAG Execution

In this step, once the DAG is generated and reviewed, the DBA proceeds to submit it for execution. This launches the transformation engine, which goes through each node in the DAG one by one and executes the physical operations on the RDF fragments.

The sequence diagram 3.5 illustrates how the system orchestrates the execution flow, from the initial DAG submission to the completion of all transformation steps.

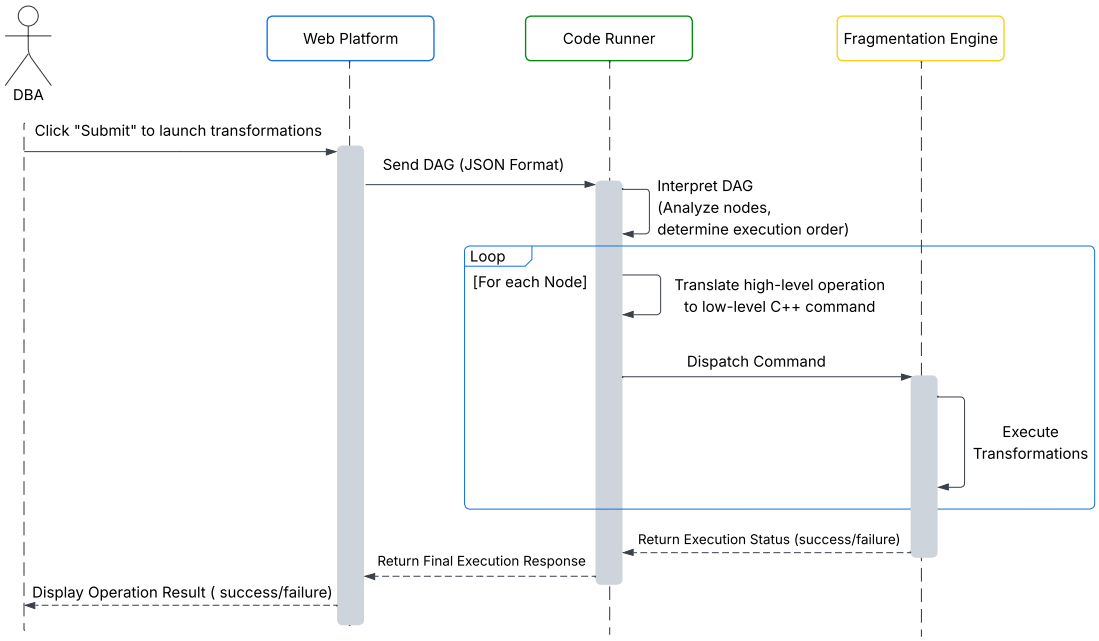

Figure 3.5: DAG Submission Interaction

This diagram (Figure 3.5) illustrates the complete execution workflow that begins once the DAG is validated and the DBA clicks Submit. It details the interaction between the DBA, the web application, the external Code Runner, and the Fragmentation Engine.

Once the DAG is approved, the DBA just clicks the Submit button on the interface. The platform then sends the DAG, formatted as JSON, to the external Code Runner.

Upon receiving the DAG, the Code Runner starts by analyzing it to understand the sequence of operations to perform. It goes through each node step by step, following dependencies to make sure everything runs in the right sequence. Each high-level transformation is then turned into a low-level command that the Fragmentation Engine can understand and execute. The Fragmentation Engine then applies the requested transformation and sends the result of the operation (success or failure) to the Code Runner.

At the end of the execution, the Code Runner sends a summary response to the web platform. The interface informs the administrator whether the process completed successfully or encountered any issues. This sequence ensures that the transformations are executed in a structured and automated manner, following the logic defined in the DAG.

3.5 Conclusion

This chapter explained how the system is designed and how its main parts work together to achieve the desired functionality. The focus was on defining clear roles for each part, ensuring a smooth flow from DBA domain knowledge expression to transformation execution. The diagrams helped clarify the structure and interactions within the system, laying a solid foundation for the next stages of development.

Chapter 4

Realisation

4.1 Introduction

This chapter begins with an overview of the project management approach and sprint planning adopted during the development process. It then provides a comprehensive view of the implementation, detailing the rationale behind key design choices. The tools and technologies used are also presented, along with explanations of how each component operates. Finally, this chapter demonstrates how the theoretical requirements of the SemPart framework were successfully addressed.

4.2 Project Management

Effective project management was essential to the success of our work, especially given the technical complexity and the need for close coordination among team members. In this section, we present how we structured and organized our project using the Scrum methodology. We detail the planning of each sprint, how we adapted Scrum principles to our specific needs, and the collaborative tools that supported our workflow throughout the development process. This approach helped us stay aligned, focused, and responsive from start to finish.

4.2.1 Life Cycle and Planning

Life Cycle: Scrum

For our project, we adopted **Scrum** [21], one of the *Agile* project management methods. It aims to improve the productivity of agile teams, even when working remotely. It also enabled continuous optimization of the product through regular feedback from end-users. This method enabled us to organize our work into sprints, monitor progress and adapt our solution according to the feedback we received.

The Main Scrum Principles

Scrum is based on three core principles that guided our team collaboration:

- **Transparency**: Everyone on the team always knew what was happening and had access to important information and documents.

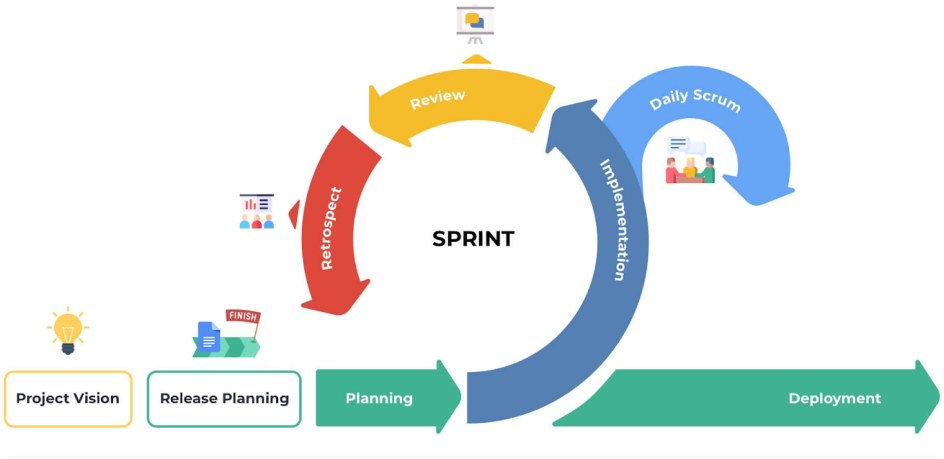

Figure 4.1: Scrum Process Schema

- **Inspection**: We checked our progress often to catch problems early.
- **Adaptation**: When needed, we quickly changed our plans based on what we learned and the feedback we received.

Scrum Framework

We followed the standard Scrum workflow, which includes the following activities:

- **Sprint Planning**: At the start of each sprint, we defined goals and allocated tasks based on our roadmap.
- **Check-in Meetings**: We held midweek progress reviews between the two of us to assess what had been done and what remained.
- **Sprint Review**: During the end-of-week team meetings, we demonstrated the progress and collected feedback from the rest of the team and supervisor.
- **Sprint Retrospective**: We took time to reflect on the sprint's execution and noted areas of improvement for the next iteration.

Planning

Sprint 1

- **Date:** 31/01/2025 to 15/02/2025
- **Content:** Read assigned papers, Learn C++ code base
- **Description:**
  We started this period with a meeting going through RDF (Resource Description Framework) and SPARQL by reading assigned research papers to grasp these new concepts of triples and fragmentation.
  After we got access to the existing code base and tried to understand the overall process by executing the code and trying different data sets to examine the results.

Sprint 2

- **Date:** 16/02/2025 to 08/03/2025
- **Content:** C++ : simple split implementation, DSL: choose the best tools for the current use case
- **Description:**
  Since observing the results is not enough, we started the work on the code base by adding a new functionality later named simple split, which takes a fragment or more and a max data star per fragment, then for each fragment that exceeds that cap, it will be divided into smaller fragments. This task was broken down into smaller steps as a pseudo-algorithm and worked in parallel on the independent parts.
  Then we started to search for domain-specific languages' best practices, examples, and tools to choose from.

Sprint 3

- **Date:** 09/03/2025 to 05/04/2025
- **Content:** DSL: implementation and the variants, Web: Diagrams
- **Description:**
  After some iterations, we created a SemPart language using Antlr4 to have complete control of the parser and lexer instead of using a host language. The language had an SQL-like syntax which we later changed to a Java/Scala-like syntax to create a familiar yet powerful syntax that can be scaled later on if needed.
  With this syntax, it was essential to have a way to execute the syntax that resulted in a web application that gives the database administrator the needed interface.

Sprint 4

- **Date:** 06/04/2025 to 03/05/2025
- **Content:** DAG, C++: implement functions
- **Description:**
  Once the research team completed the list of C++ functionalities, we started to implement each, following the Unix philosophy [20] to be easier to test and replace. Then created a C++ portable version (and a bash script to automatically update it) to be at the same ANTLR repository without exposing the C++ code. In the parser, we extracted every keyword needed to be present in the DAG so the DAGRunner can pass them to the C++ extracted binary.

Sprint 5

- **Date:** 04/05/2025 to 25/05/2025
- **Content:** Web: implement web, DAGRunner

- **Description:**
  Having clear diagrams and a working parser made the web platform faster for development, with some clear separation between the back and front end. Using Laravel and Inertia + React.js, we created a solid first proof of concept that worked promptly.
  Since the communication between our web application and Kotlin service is now working correctly, the last step is to implement DAGRunner, the program that triggers different operations depending on the DAG and the C++ binary.

4.2.2 Collaborative Tools

To effectively manage our project and foster seamless collaboration, we leveraged a suite of specialized tools. These tools were instrumental in facilitating communication, code management, design, and overall project organization throughout the development lifecycle.

Lucidchart
Used for creating and sharing various diagrams, including architecture diagrams, flowcharts, and potentially class/sequence diagrams during the analysis and design phases. Its collaborative features allowed real-time feedback and iterative refinement of visual models.

Overleaf
Utilized for collaborative writing and typesetting of the project documentation, including this thesis. Its real-time synchronization and LaTeX compilation capabilities significantly streamlined the document creation process among team members.

GitHub
Served as our central version control system for all source code. It enabled efficient collaboration through features like branching, merging, pull requests, and issue tracking, ensuring code integrity and facilitating parallel development.

Docker
Employed for containerization of our development and deployment environments. This ensured consistency across different team members' machines and simplified the deployment process by packaging the application and its dependencies into isolated containers.

Visual Studio Code
The primary Integrated Development Environment (IDE) used by the team. Its extensive extensions, debugging capabilities, and support for multiple programming languages (C++, Kotlin, JavaScript) provided a productive coding environment.

WebEx
Used for virtual meetings, including daily stand-ups, sprint planning, and sprint reviews. It provided reliable video conferencing and screen sharing functionalities, crucial for remote team communication.

Discord
Leveraged for informal and quick team communication, discussions, and real-time support. Its chat and voice channels fostered a dynamic and responsive environment for daily interactions.

Excalidraw
Used for quick, collaborative sketching and brainstorming sessions. Its intuitive interface allowed us to rapidly visualize ideas, flows, and simple diagrams during meetings or informal discussions, enhancing real-time understanding and problem-solving.

Jira
Utilized as our primary project management and issue tracking tool. Jira enabled us to organize our product backlog, manage sprints, assign tasks, track progress, and facilitate communication around user stories and bugs, crucial for our Scrum methodology.

4.3 Technologies and Languages

The implementation of our system is part of a modern technical environment and requires the use of specific technologies. This section details the tools and frameworks used for the implementation of each component of the functional requirements.

4.3.1 Physical Transformation Operators

All physical transformations are implemented in C++ due to its low-level execution, resulting in high performance. To support lookup operations, we used a key-value store, **RocksDB** [22], an embedded database known for its high write performance and durability. It provides a C++ API and is maintained by the Facebook Database Engineering Team. Since our running process required many commands, we used **makefiles** to have short commands for business logic execution, build, test, and database visualization. To make the delivery of a stable version easier, we used a bash script and an existing testing library written in Java.

For DAG management, we use **JSON** as the data format due to its readability, flexibility, and compatibility with most software environments. **Kotlin** is used as the primary language to parse the DAG objects and generate the corresponding make commands.

4.3.2 RDF Data Partitioning AND Allocation Language

The main technology used in this part is the fourth and latest version of **ANTLR** (ANother Tool for Language Recognition), a powerful parser generator designed for reading, processing, executing, or translating structured text or binary files. ANTLR is widely used to build programming languages, tools, and frameworks. Based on a given grammar, it generates a parser capable of constructing and traversing parse trees. To tailor its behavior to our needs, we extended ANTLR4 using custom Java classes.

4.3.3 Graphical User Platform

Since the ultimate goal of the web platform was to visualize parts of our work, we chose to use **Laravel**, a PHP back-end framework known for its comprehensive feature set and support for fast, reliable development. A **MySQL** database was used to store each execution. To connect the back end with the front end without requiring a traditional RESTful API, we used **Inertia.js**, which seamlessly bridges Laravel with a **React.js** front end. We also adopted **TypeScript** to ensure type safety and more predictable outcomes, and **Shadcn UI** as a design system to maintain consistent styling across components.

For visualization, we used **React Flow** to create interactive DAG nodes, and **Monaco Editor** as a web-based text editor for writing and editing code directly within the interface.

4.4 Implementation Architecture

Figure 4.2 illustrates the implementation architecture across three separate repositories, along with the components contained in each. The HTTP layer, implemented in both the web and Kotlin repositories, handles the sending of requests and the reception of responses. The "Export" arrow from the C++ repository to "C++ Exported" represents the process of building and packaging the source code into a distributable version.

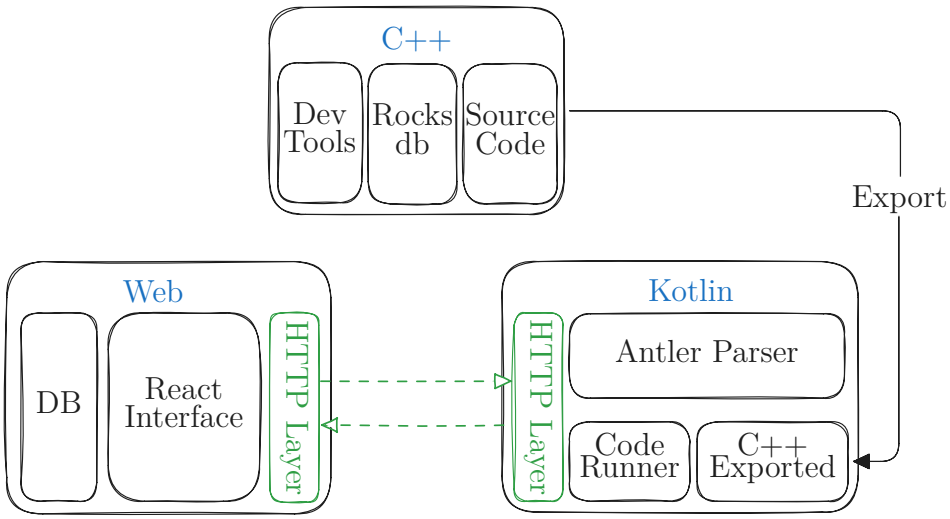

Figure 4.2: System Architecture

4.4.1 Physical Transformation Operators

In Figure 4.2, the blocks responsible for the "Physical Transformation Operators" are the C++ package, the code runner and "C++ Exported" in the Kotlin package. In the following, we describe the three components of the C++ package.

- **Source code**: This component contains the code responsible for implementing the physical operators. The operators are structured in a modular fashion, making them easily interchangeable to facilitate development, debugging, testing, and maintenance. The main program uses a switch statement to map the first argument

to the appropriate operator, while the remaining arguments serve as dependencies for that operator. To simplify execution workflows, we use a `Makefile` to organize all necessary commands into a more readable and maintainable format. As a reference, and using the same example from Figure 2.1—which selects all players who play for both a club and a national team—the corresponding `make` command is as mentioned in Listing 4.1.

Listing 4.1: Make command example

```
make selectFragments
    CONDITION="characteristic_set([has_club,national_team])"
```

- **Rocks db**: This component is responsible for handling various operations on the RocksDB database, such as creating, dropping, or querying a database. For each required lookup operation, a dedicated database is created over the RDF data. Some of these databases include subject or object identifier stores, which assign a unique numeric ID to each distinct subject or object. This is essential for SemPart, where fragments are managed in terms of numeric identifiers rather than raw RDF terms.
- **Dev Tools**: The Dev Tools component includes a set of utilities that greatly facilitate the development process. Two main tools are discussed below:
  - **Java Test Library**: Validating generated files is a crucial step toward delivering a stable release. This testing library played a key role during implementation. By running the command `make test`, the program iterates through all the files generated by the C++ components and verifies the correctness of each one.
  - **DB Visualizer**: Since no existing tool offers native visualization support for RocksDB databases, we developed a custom utility for debugging purposes. The tool can be executed with the command `make visualizeDB Database=X Key=Y`, where `Database` is a required argument referring to the database path, and `Key` is an optional argument specifying a particular key to retrieve its value. If no key is provided, the tool generates a text file containing all keys and their corresponding values in the specified database.

In the following, we describe the process of exporting the C++ code along with the Runner component.

a) C++ Code Exportation Process

We separated the C++ code base and exported only a stable version intended for use by the distance service, also known as the Kotlin Docker container.

To ensure a seamless export process, we use a Bash script that creates or updates (if it already exists) a directory named `portable-package/`. This script is triggered by running the command `make export`. It performs the following actions:

1. **Rebuild Application**: By cleaning the existing binary and rebuilding, we ensure that the exported version will ship the latest source code.

2. **Binary Deployment**: Copies the new binary into a new directory named `portable-package/`.
3. **Dependency Management**: Identifies all library dependencies of the compiled binary using the `ldd` utility and adds them to `portable-package/lib/`.
4. **Include Needed Files**: Adds the Makefile for command management and a README file as documentation.

b) Code Runner

The code runner follows the following steps to execute the DAG:

1. **Receive the request:** The server receives a POST request to the endpoint `/api/v1/coderunner`, and checks if the body is valid.
2. **Process each step in order:** For each DAG node:
   - Extract node ID, original statement, and dependencies.
   - Verify that the dependencies are satisfied.
   - Process the statement based on type (newGraph, getFragments, etc.) into commands.
3. **Handle errors (if occurred)**: Catch and log any exceptions that occur during execution, and return detailed error information if the execution fails.
4. **Send response**: Return response to client with HTTP 200 OK status.

For example, Listing 4.2 shows a node in a DAG object. This node is responsible for the command created in Listing 4.1 and for adding parameters related to the directory name (the same as the variable name). It also uses the dependency to get the name of the base directory (where the virtual fragments are stored).

Listing 4.2: DAG Node example in JSON Format

```
{
  "id": "2",
  "originalStatement":
      "List<Fragment>carsFragment=fragments.select(fragment->fragment
      .getCharacteristicSet().contains([\"has_club\",\"national_team\"]))",
  "typeName": "List<Fragment>",
  "returnValue": "fragments",
  "params": {
      "field": "characteristic_set",
      "operator": "contains",
      "value": "has_club, national_team"
  },
  "operation": "select",
  "contextInfo": {
      "variableName": "carsFragment",
      "nodeType": "DeclarationStatement"
  },
  "dependsOn": [
      "1"
  ]
},
```

4.4.2 RDF Data Partitioning and Allocation Language

The block responsible for handling RDPAL is the ANTLR parser within the Kotlin package (see Figure 4.2). We implemented RDPAL using **ANTLR4**, which enabled us to define the grammar and syntax of the language in a `.g4` file, automatically generate the lexer and parser in the host language Kotlin, and traverse the resulting **Abstract Syntax Tree (AST)** using a custom **visitor** that translates each instruction into a corresponding node within the execution DAG.

The following section outlines the syntax and semantics of RDPAL instructions and explains how the visitor processes them into executable DAG nodes.

a) Syntax and Semantics of Key Operations

- **Logical Fragmentation**
  In the first step, the RDF graph is loaded, and logical fragments based on graph star directions are generated.

  ```
  // Loading an RDF file and constructing logical fragments
  Graph graph = new Graph("footballers.nt");
  List<Fragment> fragments = graph.getFragments(Direction.FORWARD);
  ```

  The method `graph.getFragments(Direction)` can be invoked with `Direction.FORWARD`, `Direction.BACKWARD`, or `Direction.BOTH` to specify the direction of graph stars to consider.

- **Physical Fragmentation**
  Once logical fragments are established, RDPAL provides operators to refine and

transform them, leading to the creation of optimized physical fragments.
In the following, we provide the syntax for some operators.

Selection Operator

The selection operator can be used at various levels of granularity.

- **Selection by Fragment Properties:**
  Allows retaining fragments based on the properties of their characteristic sets.

```
// Retain fragments with a characteristic set including the
    predicates "has_club" and "national_team"
List<Fragment> selectedFragments = fragments.select(fragment ->
    fragment.getCharacteristicSet().contains(["has_club",
    "national_team"])
);
```

- **Selection by Number of Graph Stars:**
  Allows retaining fragments based on the count of graph stars they contain.

```
// Retain fragments that contain exactly 2 DataStars
List<Fragment> selectedFragments = fragments.select(fragment ->
    fragment.getDataStar().size().equals(2));
```

- **Selection by Graph Star Content:**
  Allows retaining fragments that contain at least one graph star satisfying a condition on its content.

```
// Retain fragments containing at least one city with the name
    "Madrid"
List<Fragment> selectedFragments = fragments.select(fragment ->
    fragment.getGraphStars().filter(graphStar ->
        graphStar.getValueOf("city_name").equals("Madrid")
    ).notEmpty()
);
```

Filter Operator

The `filter` operator produces new fragments by retaining only the graph stars that satisfy a specified condition.

```
// Retain fragments where has_birthday is greater than 1995.
List<Fragment> filteredFragments = fragments.filter(
    fragment -> fragment.getGraphStars().filter(graphStar ->
        graphStar.getValueOf("has_birthday").greaterThan(1995)
    )
);
```

Split Operator

The `split` operator allows horizontally dividing one or more existing fragments into multiple smaller sub-fragments, according to a predefined splitting strategy.

```
// General syntax: Apply a splitting strategy to a fragment
splittedFragment = fragment.split(StrategyOfSplit);
```

RDPAL supports the following splitting strategies:

1. **`SimpleSplit(N)`:**
   Divides fragments such that each sub-fragment does not exceed a maximum number `N` of graph stars.

   ```
   // Each sub-fragment should not exceed 100 graph stars
   StrategyOfSplit SimpleStrategy = new SimpleSplit(100);
   ```

2. **`GroupBySplit("predicate")`:**
   Groups graph stars into sub-fragments based on the distinct values a specified predicate.

   ```
   // Produce one fragment per footballer type
   StrategyOfSplit GroupByStrategy = new GroupBySplit("has_type");
   ```

3. **`ConditionalSplit("condition")`:**
   Partitions a fragment into two sub-fragments (positive/negative) based on a boolean condition involving predicates.

   ```
   // Defines a conditional split strategy based on 'AlAhli' club AND
       birth year > 1990.
   StrategyOfSplit ConditionalStrategy = new
       ConditionalSplit("has_club = 'AlAhli' && has_birthday > 1990");
   ```

4. **`DerivedSplit(baselineStrategy, ["path"],conflictResolution)`:**
   Applies a baseline splitting strategy (e.g., Simple, GroupBy, or Conditional) to the current fragment, then extends this logic to connected fragments through specified neighboring paths. In case of conflicts during propagation, a conflict resolution strategy (e.g., random) is applied.

   ```
   // Derived split with GroupBy by constructor, path propagation, and
       random conflict resolution
   StrategyOfSplit DerivedStrategy = new DerivedSplit(new
       GroupBySplit("has_club"), ["has_club->located_in"], random);
   ```

Execution Control Operator

The set operator manages the life-cycle of virtual fragments, allowing their persistence, merging, or removal from the working set.

– Integration:
Merges one or more new fragments into an existing fragment set

```
// Integrates Fragments1 into the 'fragments' list.
List<Fragment> integratedFragments= fragments.integrate(Fragments1);
```

– Remove:
Removes one or more fragments from a given fragment set

```
// Removes fragments present in fragments from integratedFragments.
List<Fragment> newFragments = integratedFragments.remove(fragments);
```

b) Visitor

The visitor transforms the syntax analysis tree (parse tree) generated by ANTLR into an abstract syntax tree (AST) that represents the semantic structure of the program. Then it traverses the parse tree node by node. For each type of node, the corresponding visit method is called. These methods create Abstract Syntax Tree (AST) objects that represent the semantic structure of the RDPAL program. At the root of the AST is the **Program** object, which contains all declarations and expressions. A **Declaration** node represents variable declarations, while an **Expression** node captures various types of expressions, including literals, function calls, and more. AST nodes are organized hierarchically. For example, a declaration contains an expression that may itself contain other expressions.

4.4.3 Graphical User Platform

The package responsible for the web interface is "Graphical User Platform," along with its components (see Figure 4.2). We chose to use **Laravel Starter Kit** [23] specifically the React option to have clean code, well-designed dashboard, user authentication with components that share consistent styling and are ready for use. The main use cases offered by the web interface are grouped into five key functionalities. First, users can connect to a machine by specifying an IP address and port number. Once connected, they can send datasets to the selected machine for processing. The interface also includes a DAG visualizer, which allows users to inspect the raw JSON representation of the DAG and view it in a graphical format. Through the code runner, users can execute RDPAL code directly on a connected machine. Finally, the interface provides access to a history of executions, enabling users to review their previous runs.

a) Database Structure

The data associated with the web platform is organized into several tables, as illustrated in Figure 4.3. The Users table stores user information and ensures that each user's data and relationships are isolated and accessible only by them. The Projects table allows users to group related processes under a common theme or objective, and each project can include multiple machines. The Machines table tracks the services running Kotlin-based Docker containers; two instances with different ports are treated as distinct machines. Finally, the Execution Histories table stores detailed records of each execution, ensuring traceability and enabling users to review past activities.

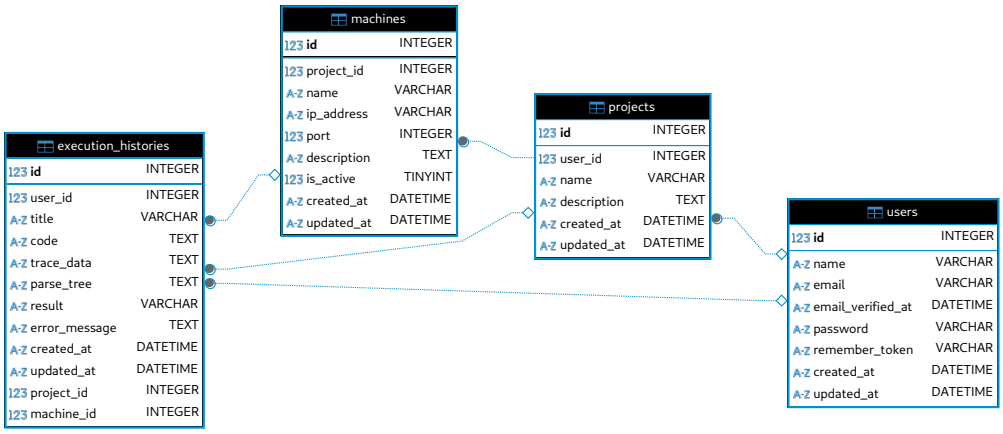

Figure 4.3: Web Platform Database

b) Error Handling

In the case of an error, the framework handles two specific use cases:

- **Code Syntax**: When the code is sent for conversion into a DAG and the parser encounters a syntax error, the built-in error handler by default returns the line and the column where the error occurs, then returns all details to the front-end. On the React side, when the return is a syntax error, a pop-up appears showing the first 5 error details and a count of how many errors are left to provide a clean user experience and comply with common industry standards. This method is also used by debug tools, since the first error often triggers a cascade of subsequent errors. Fixing it usually resolves many of the others automatically.

- **Missing Dataset**: In case the administrator uses a dataset that does not exist on the machine, the code will not execute as expected. For that, we added a check before submitting the DAG to the code runner, where we send a request to the machine checking for the existence of the dataset first. If missing, the administrator is required to update a dataset with the same name or edit the script to mention an existing dataset.

c) Hosting Options

Since this platform can connect to remote machines, we have two hosting options:

- **Public Hosting**: As shown in Figure 4.4, this setup is useful when machines are on different networks or in remote locations that require an Internet connection to communicate. For more security, we can add region blocking or a logs page with the IP address locations for a history of who accessed it and from where to ensure traceability.

- **LAN Hosting**: As shown in Figure 4.5, this setup is useful when machines are on the same network and at the same location. The internet connection is not required. It is more secure since this approach is only accessible from the local network, although user logs are still important, especially in large teams.

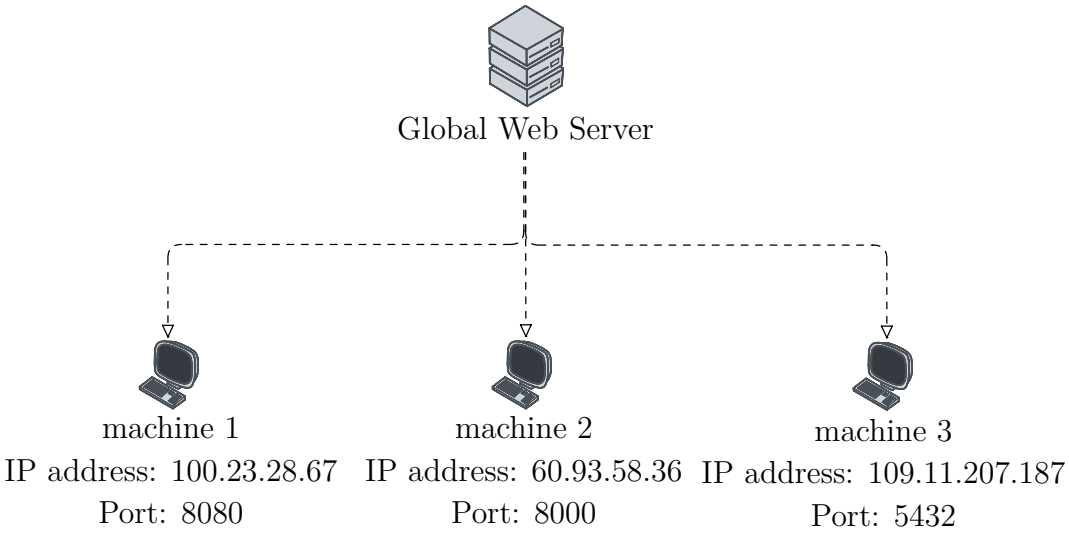

Figure 4.4: Public Hosting

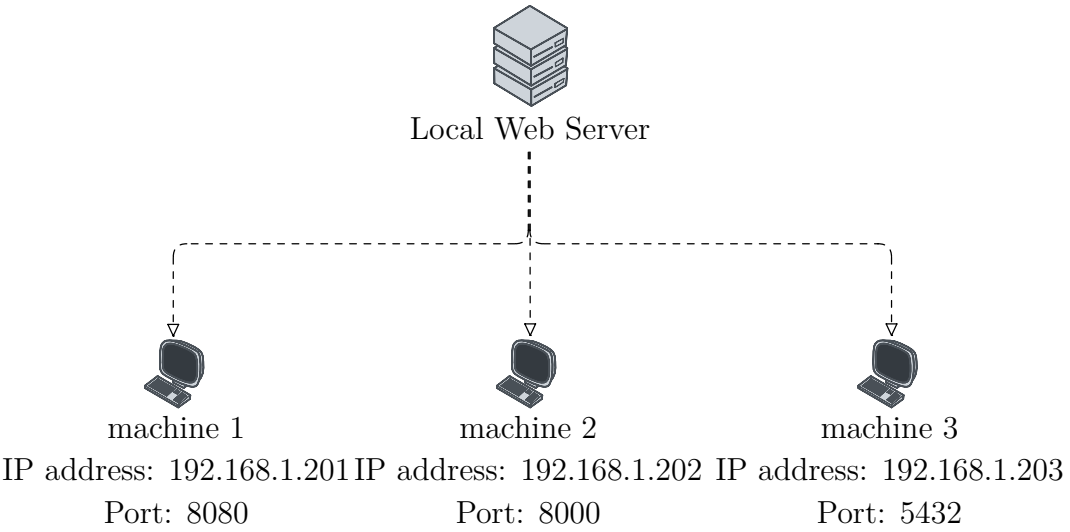

Figure 4.5: Local Area Network (LAN) Hosting

d) Overview of the SemPartWeb Workflow

Figure 4.6 provides a comprehensive overview of the workflow within SemPartWeb, illustrating the key interactions between system components and the process steps, from rule definition to execution history management.

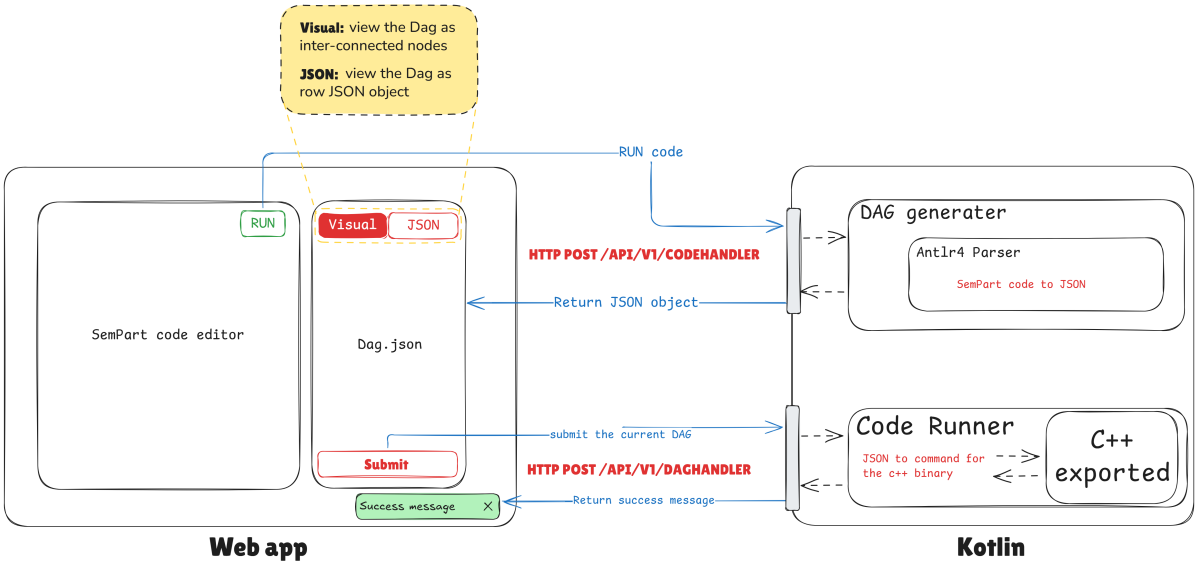

Figure 4.6: Overall SemPartWeb Workflow

4.5 Conclusion

The implementation phase successfully transformed the SemPart framework from concept to reality, resulting in a coherent, modular, and fully functional system. By effectively integrating C++, Kotlin, React, and Laravel, we built a user-friendly environment that empowers RDF database administrators to define, visualize, and execute semantic partitioning strategies with ease. This achievement marks a key step toward more dynamic, customizable, and domain-aware management of RDF data in distributed systems.

Conclusion and Perspectives

5.6 Conclusion

The growth of RDF graphs has led to the emergence of distributed RDF data management systems, which operate across multiple machines. A crucial aspect of these systems is the partitioning of the initial RDF data. The RDF research community has long focused on this topic, proposing a wide range of strategies.

In our state-of-the-art analysis, we first identified a key dimension: the granularity of partitioning. We focused on two main approaches—those based on individual triples and those based on fragments. Triple-based approaches adopt a low-level physical view of RDF graphs, often separating entities that naturally belong together, rather than grouping them into cohesive and semantically meaningful units. Fragment-based approaches also rely heavily on physical structures to define fragments, without incorporating semantic coherence.

In both cases—triples and fragments—there is a notable absence of integration with a database administrator (DBA), whose expertise could be instrumental in defining partitioning and repartitioning rules to enhance partition quality.

To address these limitations, the PQDAG team proposes SemPart, a complete framework designed to support and guide the partitioning process. Our main objective was to implement SemPart and all of its components while ensuring modularity and performance in the provided code. This work involved three main contributions: (1) the implementation of all physical transformations, (2) the development of the RDPAL language, and (3) the creation of a web interface designed for DBAs. We applied all the software engineering principles we acquired to deliver a robust and flexible solution.

5.7 Perspectives

One primary perspective of this research is the exploration of adapting the SemPart framework and its RDPAL language to accommodate alternative RDF system partitioning strategies. Currently, SemPart is specifically tailored to systems that utilize characteristic sets as their data granularity. Expanding the framework to support systems with different granularities presents an exciting challenge and opens up the potential for broader applicability across diverse RDF systems.

Another promising avenue for future research is already in progress. A PhD student from the PQDAG team is actively investigating how Large Language Models (LLMs) can be leveraged to optimize the Directed Acyclic Graph (DAG) generated from RDPAL code. This collaboration aims to harness the capabilities of LLMs to enhance the efficiency and effectiveness of SemPart, pushing the boundaries of its performance and adaptability.

Bibliography

[1] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. A survey of rdf stores & sparql engines for querying knowledge graphs. *The VLDB Journal*, pages 1--26, 2022.

[2] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020.

[3] Razen Al-Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB Journal*, 25(3):355--380, 2016.

[4] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4):265--276, 2013.

[5] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. Triad: a distributed shared-nothing RDF engine based on asynchronous message passing. In *ACM SIGMOD*, pages 289--300, 2014.

[6] Kisung Lee and Ling Liu. Scaling queries over big rdf graphs with semantic hash partitioning. *Proceedings of the VLDB Endowment*, 6(14):1894--1905, 2013.

[7] Luis Galárraga, Katja Hose, and Ralf Schenkel. Partout: a distributed engine for efficient RDF processing. In *WWW Conference*, pages 267--268, 2014.

[8] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Hai Jin, and Ling Liu. Semstore: A semantic-preserving distributed rdf triple store. In *ACM CIKM*, pages 509--518, 2014.

[9] Abdallah Khelil, Amin Mesmoudi, Jorge Galicia, and Mohamed Senouci. Should we be afraid of querying billions of triples in a graph-based centralized system? In *9th International Conference on Model and Data Engineering (MEDI)*, pages 251--266, 2019.

[10] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. Processing SPARQL queries over distributed RDF graphs. *VLDB Journal*, 25(2):243--268, 2016.

[11] Peng Peng, M Tamer Özsu, Lei Zou, Cen Yan, and Chengjun Liu. Mpc: minimum property-cut rdf graph partitioning. In *IEEE ICDE*, pages 192--204, 2022.

[12] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF querying with SPARQL on spark. *PVLDB*, 9(10):804--815, 2016.

[13] Mahmudul Hassan and Srividya K Bansal. Data partitioning scheme for efficient distributed rdf querying using apache spark. In *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*, pages 24--31, 2019.

[14] Liang He, Bin Shao, Yatao Li, Huanhuan Xia, Yanghua Xiao, Enhong Chen, and Liang Jeff Chen. Stylus: a strongly-typed store for serving massive rdf data. *Proceedings of the VLDB Endowment*, 11(2):203--216, 2017.

[15] Xintong Guo, Hong Gao, and Zhaonian Zou. Leon: A distributed rdf engine for multi-query processing. In *DASFAA*, pages 742--759. Springer, 2019.

[16] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, pages 984--994, 2011.

[17] Juris Hartmanis. Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review*, 24(1):90, 1982.

[18] Robert C. Martin. Design principles and design patterns, 2000. Accessed April 2025.

[19] Alexander Shvets. *Dive Into Design Patterns*. Refactoring.Guru, 2019. Accessed April 2025.

[20] Eric S. Raymond. *The Art of Unix Programming*. Addison-Wesley Professional, 2004. Published 2003-09-23.

[21] Ken Schwaber. *Agile Project Management with Scrum*. Developer Best Practices. Microsoft Press, 2004.

[22] RocksDB. Rocksdb documentation. `https://rocksdb.org/docs/getting-started.html`, 2025. Accessed: 2025-06-13.

[23] Laravel Team. Laravel documentation. `https://laravel.com/docs/12.x/starter-kits`, 2025. Accessed: 2025-06-16.

Abstract

The Resource Description Framework (RDF) is now an important standard for modeling structured knowledge, but it is still hard to manage large RDF datasets, especially when they are spread out across many computers. The way that current RDF triplestores split up their data into triples can make queries take longer and make communication harder. Also, the fact that database administrators (DBAs) are not involved in the partitioning process makes the system less flexible when workloads or semantic structures change. The PQDAG team at the LIAS lab came up with SemPart, a fragment-based RDF partitioning framework that adds semantic coherence and expert-driven control to get around these problems. The main focus of this work is on how to put it into practice by designing and building the entire SemPart framework and adding it to the distributed PQDAG system. This work makes it possible to do RDF partitioning that is flexible, semantically rich, and focused on performance.

***Keywords*:** RDF, Semantic Web, Data Partitioning, SemPart, PQDAG, Distributed Systems, Database Administrators, Triplestore

Résumé:

Le RDF (Resource Description Framework) est désormais une norme importante pour la modélisation des connaissances structurées, mais il reste difficile de gérer de grands ensembles de données RDF, en particulier lorsqu'ils sont répartis sur plusieurs ordinateurs. La manière dont les bases de données actuelles triples RDF divisent leurs données en triplets peut ralentir les requêtes et compliquer la communication. De plus, le fait que les administrateurs de bases de données (DBA) ne soient pas impliqués dans le processus de partitionnement rend le système moins flexible lorsque les charges de travail ou les structures sémantiques évoluent. L'équipe PQDAG du laboratoire LIAS a mis au point SemPart, un cadre de partitionnement RDF basé sur des fragments, qui apporte une cohérence sémantique et un contrôle par des experts afin de contourner ces problèmes. Ce travail se concentre sur la mise en oeuvre de SemPart, en concevant et en construisant l'ensemble du cadre, puis en l'ajoutant au système PQDAG distribué. Il permet d'effectuer un partitionnement RDF flexible, riche sur le plan sémantique et axé sur les performances.

***Mots-clès*:** RDF, Web sémantique, Partitionnement des données, SemPart, PQDAG, Systèmes distribués, Administrateurs de bases de données, Triplestore

ملخص:

أصبح RDF (Framework Description Resource) الآن معيارًا مهمًا لنمذجة المعرفة المنظمة، ولكن لا يزال من الصعب إدارة مجموعات بيانات RDF الكبيرة، خاصةً عندما تكون موزعة على العديد من أجهزة الكمبيوتر. الطريقة التي تقسم بها مخازن ثلاثيات RDF الحالية بياناتها إلى ثلاثيات يمكن أن تجعل الاستعلامات تستغرق وقتًا أطول وتجعل الاتصال أكثر صعوبة. كما أن عدم مشاركة مسؤولي قواعد البيانات (DBAs) في عملية التقسيم يجعل النظام أقل مرونة عند تغير أحمال العمل أو الهياكل الدلالية. ابتكر فريق PQDAG في مختبر LIAS إطار عمل ،SemPart وهو إطار عمل لتقسيم RDF قائم على الأجزاء يضيف التماسك الدلالي والتحكم المدفوع بالخبرة للتغلب على هذه المشكلات. ينصب التركيز الرئيسي لهذا العمل على كيفية وضعه موضع التنفيذ من خلال تصميم وبناء إطار عمل SemPart بالكامل وإضافته إلى نظام PQDAG الموزع. يتيح هذا العمل إجراء تقسيم RDF بطريقة مرنة وغنية بيانياً ومركزًا على الأداء.

الكلمات المفتاحية: ،RDF الدلالي، Web، تقسيم البيانات، ،SemPart ،PQDAG الأنظمة الموزعة، مخازن الثلاثيات