A Framework Model for Supporting Transparent Polyglot Persistence with a Unified API and Extensible for Different Database Types

Fernando de Oliveira Pereira[1,3][a], Eduardo Martins Guerra[2][b] and Reinaldo Roberto Rosa[1][c]
[1]*National Institute for Space Research (INPE), São José dos Campos/SP, Brazil*
[2]*Free University of Bozen-Bolzano - UNIBZ, Bozen-Bolzano, Italy*
[3]*National Centre for Monitoring and Early Warning of Natural Disasters (CEMADEN), São José dos Campos/SP, Brazil*

Keywords: Polyglot Persistence, Information System Integration.

Abstract: This work introduces the Transparent Polyglot Persistence Framework Model (TPPFM) for supporting polyglot persistence through a unified API for extension. The framework employs the Esfinge Query Builder as its basis, restructuring it to provide polyglot functionality in alignment with the proposed framework model. A real database case study is conducted to demonstrate the viability of the proposed framework and its reference implementation. The ease of implementation for the developer and the transparency concerning the utilization of several databases within the same domain model are demonstrated.

1 INTRODUCTION

Today, applications must handle diverse datasets from the same domain (Khine and Wang, 2019; Eisenhuth and Jablonski, 2022). For example, in e-Commerce, multiple structures, including relational databases for financial data and inventory, document-oriented databases for product catalogs, graphs for consumer recommendations, key-value pairs for shopping cart items, and columnar databases for activity logs, can coexist. Different data manipulation paradigms exist for each format.

Each storage format works well in specific scenarios. Some are better for short, frequent reads, while others favor efficient records. Using each type's strengths in their ideal situations makes sense. However, connecting different data types within an application is challenging. Thus, modern applications may require more than classic relational databases for functional and non-functional requirements (Wiese, 2015; de Araújo et al., 2016).

Polyglot persistence is a strategy that utilizes multiple database systems within a single application domain to handle diverse datasets. This approach optimizes the performance of various components of an application by capitalizing on the unique capabilities of their respective database technologies. Using multiple database technologies in a single application requires careful consideration of the most suitable persistence model for each specific purpose. This strategic approach allows firms to effectively utilize the most appropriate technologies for their specific requirements, such as SQL databases for structured data or NoSQL databases for more adaptable or extensive data storage (Schaarschmidt et al., 2015; Villaça et al., 2018).

However, implementing polyglot persistence presents significant questions, particularly in accurately linking data across multiple storage formats within a single application (Wiese, 2015; Lajam and Mohammed, 2022). Polyglot persistence presents challenges in terms of integration and data management. Proficiency in multiple database paradigms and APIs is necessary to access and modify data stored in different formats, raising concerns about interoperability, referential integrity, and security across multiple storage systems (Srivastava and Shekokar, 2016).

These issues have prompted the proposal of various solutions, including unified query languages, middlewares, and multi-model databases (Jiménez-Peris et al., 2016) (Schaarschmidt et al., 2015) (Lu et al., 2018). These solutions are examined in section 2 from several developments. These developments aim to provide the use of several database systems within a single application. However, developers often face architectural inefficiencies and an increased

[a] https://orcid.org/0009-0006-2360-1281
[b] https://orcid.org/0009-0006-2894-9076
[c] https://orcid.org/0000-0002-2962-4322

Pereira, F. O., Guerra, E. M. and Rosa, R. R.
A Framework Model for Supporting Transparent Polyglot Persistence with a Unified API and Extensible for Different Database Types.
DOI: 10.5220/0013369300003929
In *Proceedings of the 27th International Conference on Enterprise Information Systems (ICEIS 2025) - Volume 1*, pages 109-120
ISBN: 978-989-758-749-8; ISSN: 2184-4992

development of maintenance. By implementing more cohesive and clear data access methods, these advancements have the potential to result in software solutions that are more resilient and adaptable, highly suitable for contemporary applications (Villaça et al., 2018).

There is an increasing need for tools and frameworks that improve the robustness of polyglot persistence and reduce developer workload. No existing work offers an extendable approach that employs a unified extension API for several database types, remains transparent to the developer, and leverages the special characteristics of each API without mixing them.

The present research introduces the Transparent Polyglot Persistence Framework Model, abbreviated as TPPFM. This is a conceptual framework that delineates the approach for establishing polyglot persistence via a unified and declarative metadata-driven API. The model's contributions include defining an architecture that facilitates a cohesive extension API and orchestrates polyglot persistence through database-specific libraries, all while remaining transparent to the developer.

A reference implementation of the model was created in a framework called Esfinge Query Builder (Guerra, 2014, p. 12), which provides support for implementing a transparent persistence layer for classes in the same domain model that should be persisted in different types of databases. The framework has an extensible structure that allows the addition of new drivers that can support polyglot persistence for different types of databases. A set of tests demonstrated that the solution worked in four databases of different types, namely PostgreSQL, MongoDB, Neo4J, and Cassandra. As a result, the data could persisted in any combination of them by changing only the metadata configuration. Moreover, the framework was integrated into a virtual lab platform and used to combine data from a relational database of alerts provided by CEMADEN with data in JSON format provided by IBGE stored in a MongoDB. This implementation was used to evaluate the applicability of the solution for a real-practical case. Modularity analysis was performed on the classes in the evaluation to investigate the coupling of the classes with the APIs of the specific databases.

The work is organized as follows: Section 2 shows related works, Section 3 offers a succinct overview of the key concepts pertinent to this study, Section 4 conceptually delineates the PPFM, Section 5 illustrates and explains the model's implementation, Section 6 introduces the case study, Section 7 analyzes the results, and Section 8 provides the conclusions.

2 RELATED WORKS

The studies by (Prasad and Avinash, 2014) and (Kaur and Rani, 2015) delineate systems exhibiting polyglot characteristics, where data manipulation occurs in databases of different types. These works feature a combination of data stored in relational and non-relational databases, approached architecturally such that these data are managed separately in the service layer, meaning the databases are accessed in isolation without any abstraction for their correlation.

Another approach is presented in the study by (Schaarschmidt et al., 2015), which delineates the establishment of a mediating component between the service layer and the persistence layer. The term *polyglot persistence mediator* (PPM) is defined in the work in question. It functions as a middleware, serving as a conduit between the two layers and directing data to different query mechanisms. The work discusses polyglot persistence, but does not establish a correlation between different types of databases; instead, it focuses on a decoupled routing mechanism among different types of database based on the SLA rating, one at a time.

The study CloudMdsQL (Kolev et al., 2016) investigates the creation of a universal language for querying different types of databases. The authors propose that the use of a cohesive language serves as a strategy for polyglot persistence. The operational procedure entails employing translation components to transform one query language into another, subsequently routing them to the relevant databases. These studies are in their initial stages and have yielded only a restricted number of prototypes for specific database types. The recent study by (El Ahdab et al., 2024) presents a significant advancement and draws a comparison with CloudMdsQL, maintaining a similar approach utilizing a transformer coupled with multiple APIs.

Apache Drill (Givre and Rogers, 2018) possesses the characteristics of a Database Management System (DBMS), but is recognized as a query layer that allows simultaneous access to multiple database types. It functions as a tool, enabling polyglot queries through the integration of different databases using SQL language. Apache Drill lacks a high-level API or a designated ORM. Its philosophy is to conceptualize every type of storage as a table, regardless of its actual nature, by executing queries on the raw data. Thus, it operates not as an abstract framework but as a practical tool for concrete implementations.

The research carried out by (Lu et al., 2018) delineates a multi-model database known as the *unified database management system* (UDMS), provid-

ing guidance for the development of an integrated system for querying and storing data in a multi-model framework, including relational, document, columnar, graph, and JSON formats, among others, within a singular management system.

Recent studies by (Holubová et al., 2021) and (Van Landuyt et al., 2023) illustrate multimodel database solutions and their operational challenges, together with comparisons with separate database approaches, concluding that the optimal choice largely depends on the type of application.

From the related works above, two horizons become evident. First, when different databases are accessed simultaneously in the same application, but separately, directly from the service layer and later combined. Second, when the data are already combined from different databases in the persistence layer. Three approaches are identified as exemplified in the aforementioned work and according to the review article by (Khine and Wang, 2019). They are as follows: Applied to the Domain; Unified Query Language; Framework/Middleware or Multimodel.

In light of these horizons and approaches, there are no existing works that offer solutions to establish an API characterized by low coupling, uniformity, and transparency for the developer to access and correlate different types of databases. The Transparent Polyglot Persistence Framework Model (TPPFM) is thus proposed.

3 BACKGROUND

The purpose of this section is to provide a concise overview of the major concepts that are relevant to understanding this work. This is an overview of issues that are directly connected to the topic's matter, and these are included in the proposal for the TPPFM.

3.1 Polyglot Persistence Patterns

A pattern is a way of documenting the experience by capturing successful solutions to recurring problems. Currently, standards are carefully documented in many fields of computing and in several areas of academia and industry. For the documentation of a pattern, the context of the problem, the forces acting on the problem solver, the logic, and the context that results when applying the solution must be recorded (Rising, 1998).

Regarding patterns of polyglot persistence, the patterns identified in the work of Pereira et al., 2023 stand out (Pereira et al., 2023b). In the most basic identified pattern named "Independent DAO," the client service accesses several independent persistence modules, and the manipulation for data persistence is carried out at the business layer. In the named pattern "Integrated Polyglot DAO," data manipulation is carried out in an integrated module that combines various APIs. In the pattern named "DAO Compound Mediator", applied in this work, the APIs are independent and all persistence manipulation is abstracted and transparent to the client module. See Figure 1.

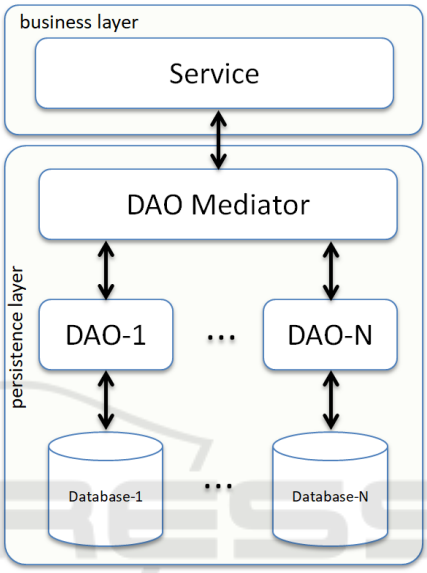

Figure 1: Polyglot persistence pattern. Adapted from (Pereira et al., 2023b).

3.2 Metadata-Based Frameworks

In software engineering, a framework is a fundamental structure consisting of generic code that offers support and standard functionalities to create specific applications. A framework's architecture is often event-driven, allowing developers to insert their code within extension points provided by the framework, often referred to as "hooks" or "customization points". This differs from traditional libraries where the developer calls functions as needed. In a framework, overall control of the flow of execution is often inverted, which is known as the "Inversion of Control" (IoC) principle or "Hollywood Principle" ("don't call us, we'll call you") (Larman, 2012).

Metadata-based frameworks can be defined as frameworks that utilize metadata to affect the software's behavior. This metadata can be described using XML configuration files, annotations, or other kinds of description. The primary concept is that, rather than explicitly customizing or altering the source code, the developer provides information

that the framework utilizes to determine the system's behavior (Lima et al., 2023). Some examples of metadata-based frameworks include SpringData (Java), EsfingeQueryBuilder (Java), and Entity Framework (C#/.NET).

Regarding the resources used in metadata-based frameworks, an example in the Java programming language is the concept of annotation. Annotations are a programming tool that allows you to associate metadata with specific elements of source code, such as classes, methods, or attributes. The compiler can use annotations to incorporate data during the compilation process, or a framework can utilize them during execution (Lima et al., 2023).

Annotations allow for more detailed information about the behavior of source code elements without directly modifying them. Furthermore, they improve the architecture of the code by reducing duplications, centralizing the logic in one place, and automatically implementing it throughout multiple parts (Lima et al., 2023).

3.3 Framework Models

Framework models refer to a conceptual structure that organizes and guides the development of solutions within a given domain. Abstractly, the framework model embodies a theoretical structure that dictates the interaction of software components, the adherence to standards, and the structuring of solutions to enhance efficiency, reuse, and consistency in development. In this sense, the framework model establishes the bridge between theory and practice.

The framework model encapsulates a standard reference architecture that specifies the organization and interaction of several components of the system. It is not a concrete implementation, but a set of principles that guides solution design within the specific domain. Moreover, the framework model needs to impart practical knowledge about a specific domain and codify it in a way that allows its reuse in any context (De Souza et al., 2022).

4 TRANSPARENT POLYGLOT PERSISTENCE FRAMEWORK MODEL (TPPFM)

The concepts of separation of concerns, single responsibility, and dependency inversion provide developers with guidance on structuring modular, scalable, and adaptable code. Taking into account those design principles, this section describes the proposed TPPFM in this work.

4.1 Main Structure

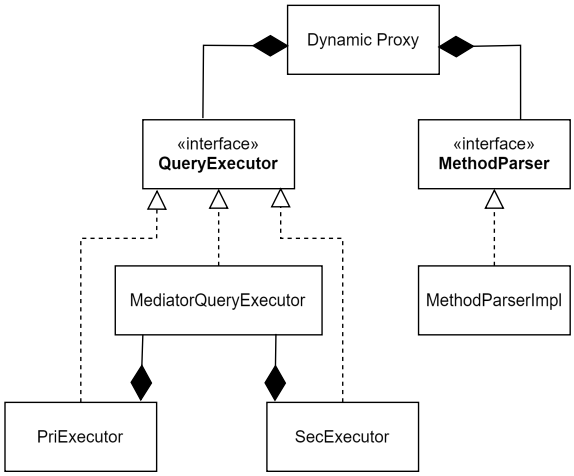

Figure 2: TPPFM - Main strucuture.

The TPPFM has the structure shown in Figure 2 and is inspired by the DAO CompoundMediator polyglot persistence pattern, which can be seen in Figure 1. The service object of the business layer is managed by a dynamic proxy that contains an instance of *QueryExecutor*, tasked with executing queries in the persistence layer, and *MethodParser*, which interprets the method names to convert them into queries. In the polyglot scenario, the persistence layer contains the *MediatorQueryExecutor*, which encapsulates distinct instances of *QueryExecutor* for several types of databases.

The proposed framework model assigns the responsibility of aggregating polyglot queries to *MediatorQueryExecutor*. Starting with a unified language, specifically ORM (object relation mapping), ODM (object document mapping), other Object Mappings, and Internal DSL (Mangal, 2024), while assigning the particularities of each database to designated *QueryExecutor*'s, denoted in the model as *PriExecutor* and *SecExecutor* instances.

4.2 Dynamic Behavior

Figure 3 provides a detailed description of the primary flow described by the TPPFM. A dynamic proxy is used to pass an instance of the service across, which enables the methods of the service to be invoked in a natural manner without requiring any modifications to the business layer. With regard to *QueryExecutor* and *MethodParser*, it is the responsibility of the proxy to generate instances of the various implementations that are dynamically accessible. During the

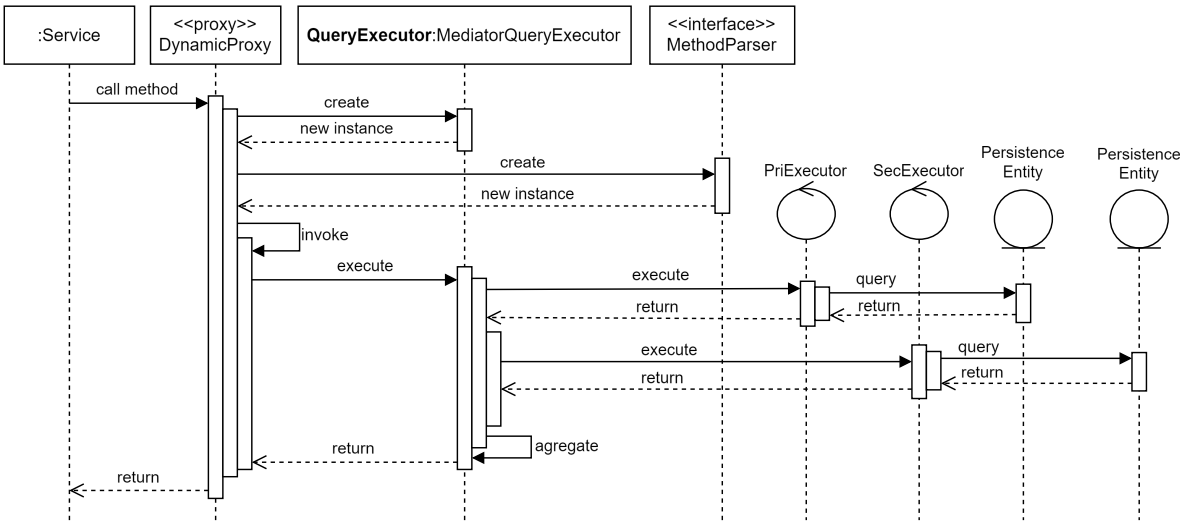

Figure 3: TPPFM - Main flow behavior.

process of invoking a method, the *MediatorQueryExecutor* not only delegates the execution of the method to the available implementations, but also the aggregation. Then it returns the polyglot query result to the requesting service. The aggregation originates from the primary entity, for whom the dynamic proxy was created, and subsequently extends to the secondary entity, which is the one being associated.

5 REFERENCE IMPLEMENTATION

The purpose of the reference implementation is to demonstrate the feasibility of TPPFM. This work utilizes the Esfinge Query Builder as a reference implementation. The Esfinge Query Builder is currently in development, initially described in (Guerra, 2014), and provides a fast and efficient means to create a persistent layer.

5.1 About Esfinge Query Builder

The Esfinge Query Builder is a framework that accelerates the development of persistence layers by utilizing method names to deduce database queries. The primary advantage lies in the simplicity of the interface, as it specifies methods in alignment with the query parameters, enabling automatic query generation at runtime (GUERRA et al., 2017).

The framework operates independently of persistent technology and currently includes extensions for JDBC, JPA, Cassandra, MongoDB, and Neo4J. In addition to fundamental searches, it facilitates the execution of CRUD capabilities, including the saving, deletion, and listing of records. The framework utilizes internal DSL (domain-specific language) (Mangal, 2024), which enables the use of ”domain terms” in methods to express queries, thereby improving expressiveness and readability.

This set of features aligns with the TPPFM by establishing a basis for implementation. The Esfinge Query Builder is capable of operating with a single database at a time. The proposed work improves the Esfinge Query Builder and redesigns it to operate concurrently with several databases in a polyglot approach.

5.2 TPPFM Implementation - General View

The Esfinge Query Builder was restructured to meet the TPPFM, taking advantage of its characteristics already adherent to the model, being readjusted to operate with multiple databases in order to retrieve information from the same domain that is in databases of different types.

Prior to this restructure, Esfinge Query Builder permitted the utilization of only a single database at once, rendering the simultaneous usage of multiple database types unfeasible. The rewriting of the framework to incorporate TPPFM enabled its ability to interface with several diverse databases concurrently through a standardized API among entities within the same domain.

For illustration, consider a basic user registration process in which users are stored in a PostgreSQL database. Additionally, note that there exists a MongoDB database that stores user profiles, which are semi-structured data retaining access permissions and

specific configurations for application functionalities accessible to the user.

To transparently obtain this information from a unified API, we can utilize the framework demonstrated in the code in Listing 1, Listing 2, Listing 3, and Listing 4.

The *User* class in Listing 1 is a mapped entity following the JPA specification using Hibernate ORM (Tudose et al., 2023). It contains some specific TPPFM annotations, which are *@PersistenceType*, *@PolyglotOneToOne*, and *@PolyglotJoin*. The class *Profile* in the Listing 2 was mapped by Morphia ODM (Scherzinger and Sidortschuck, 2020), and contains the annotation *@PersistenceType*. The interface *ExampleDAO* in the Listing 3 contains the annotation *@TargetEntity*. Finally, the *Main* class in Listing 4 that invokes and obtains a dynamic proxy for an instance of *ExampleDAO* does not contain specific annotations. All the specific annotations used here and the others in the framework are detailed in section 5.3.

```
1  //omitted code
2  import javax.persistence.Entity;
3  import javax.persistence.Id;
4  import javax.persistence.Transient;
5  @Entity
6  @PersistenceType(value = "JPA1", secondary = "MONGODB"
       ↪)
7  public class User {
8    @Id
9    private Integer id;
10   private String login;
11   private String password;
12   private ObjectId profileId;
13   @Transient
14   @PolyglotOneToOne(referencedEntity = Profile.class)
15   @PolyglotJoin(name = "profileId",
         ↪referencedAttributeName = "id")
16   private Profile profile;
17   //getters and setters
18 }
```

Listing 1: User class.

```
1  //omitted code
2  import dev.morphia.annotations.Entity;
3  import dev.morphia.annotations.Id;
4  @Entity
5  @PersistenceType("MONGODB")
6  public class Profile {
7    @Id
8    private ObjectId id;
9    private String configs;
10   private String permissions;
11   //getters and setters
12 }
```

Listing 2: Profile class.

```
1  //omitted code
2  @TargetEntity(User.class)
3  public interface ExampleDAO extends Repository<User> {
4    List<User> getUserByLogin(String login);
5    //omitted code
6  }
```

Listing 3: ExampleDAO interface.

```
1  //omitted code
2  public class Main {
3    public static void main(String[] args) {
4      var example = QueryBuilder.create(ExampleDAO.class
         ↪);
5      var user = example.getUserByName("John");
6      System.out.println(user);
7    }
8  }
```

Listing 4: Main class - business layer.

In this example case, when executing *example.getUserByLogin("john")*, the *user* object will contain data from both the PostgreSQL database and the MongoDB database filtered by the user's login (Listing 4, lines 5-6). The access is transparent to the business layer, and the delegation of queries and aggregation is performed in the persistence layer without strong coupling.

5.3 TPPFM Implementation - Internal Structure

The Esfinge Query Builder framework is divided into modules and has been updated to support the polyglot approach of TPPFM. It contains extension points that utilize the Java Service Provider Interface (SPI) functionality, allowing the discovery and use of service implementations at runtime. The main module is called *Esfinge Query Builder Core*. This module contains the main machinery of the framework. Other modules extend and use the main module to invoke concrete implementations for each type of database.

The evolution, to the TPPFM, demanded the creation of annotations, as cited in subsection 5.2. Next, an explanation about the usefulness of all the notes available for the polyglot context.

- **@TargetEntity** - defines the main entity for the interface for which the dynamic proxy is created;
- **@PersistenceType** - defines the type of persistence assigned to each entity being manipulated;
- **@QueryExecutorType** - extension annotation that defines with which type of persistence a specific implementation of *QueryExecutor* is operating;
- **@PolyglotJoin** - specifies a polyglot relationship within the primary entity, indicating the field name that references the secondary entity, or otherwise;
- **@PolyglotOneToOne** - denotes the one-to-one relationship type, signifying that for each instance of the primary entity, there exists a singular corresponding instance of the secondary entity. It can be delineated from left to right or from right to left; that is, there may be a primary field that links an identifier to the secondary, or vice versa;

- **@PolglotOneToMany** - establishes the one-to-many relationship, indicating that for each instance of the main entity, there are several corresponding instances of the secondary object;

The *Esfinge Query Builder Core*, which maps to the TPPFM, creates the dynamic proxy from the *QueryBuilder* class and provides some hotspots, such as the *QueryExecutor* and *MethodParser* interfaces. The *QueryBuilder* class sets up a list of available implementations for each interface and creates an instance of *MediatorQueryExecutor*, which, in turn, contains 2 attributes that define the *PriExecutor* (implementation of persistence for the primary entity - e.g.: JPA) and *SecExecutor* (implementation of persistence for the secondary, e.g.: MongoDB).

The identification of the primary entity is determined by the attribute of named "value" obtained from the *@TargetEntity* annotation of the class for which the dynamic proxy is generated. We retrieve the persistence type from *@PersistenceType*. So, the operations are first done on the primary entity, then on the secondary one, and finally on both entities simultaneously, based on the defined polyglot relationship, as shown in the model's Figure 3.

The modules are configured with SPI and can be seen from Listing 5 with the *module-info* file of the *Esfinge Query Builder JPA*, and Listing 6 with the *module-info.java* file of the *Esfinge Query Builder MongoDB*. In this way, the main module obtains specific implementations for the interfaces at run-time according to the dependencies injected into the client project. As seen in the *module-info.java* files, all contain implementations available for the special *Repository* interface, which has also become polyglot by providing polyglot operations for CRUD.

```
module querybuilder.jpaone {
requires transitive querybuilder.core;
requires java.persistence;
exports ef.qb.jpa1;
opens ef.qb.jpa1;
uses ef.qb.jpa1.EntityManagerProvider;
provides ef.qb.core.Repository with
    ef.qb.jpa1.JPARepository;
provides ef.qb.core.executor.QueryExecutor with
    ef.qb.jpa1.JPAQueryExecutor;
}
```

Listing 5: Esfinge Query Builder JPA - module-info.

```
module querybuilder.mongodb {
requires transitive querybuilder.core;
requires morphia;
//omitted code
exports ef.qb.mongodb;
opens ef.qb.mongodb;
uses ef.qb.mongodb.DatastoreProvider;
provides ef.qb.core.Repository with
    ef.qb.mongodb.MongoDBRepository;
provides ef.qb.core.executor.QueryExecutor with
    ef.qb.mongodb.MongoDBQueryExecutor;
//omitted code
}
```

Listing 6: Esfinge Query Builder MongoDB - module-info.

This approach allows for the integration of additional database implementations as dependencies with little coupling, improving maintainability, and aligning with the pattern depicted in Figure 1 in *DAO Compound Mediator*.

5.4 TPPFM Tests

The examples presented in section 5.2 and the module-info files in section 5.3 have been taken from MongoDB and JPA implementations. It is crucial to emphasize that TPPFM underwent implementation testing with various cases, including PostgreSQL and MongoDB, PostgreSQL and Cassandra, as well as MongoDB and Cassandra. These tests demonstrate that the proposed mechanism functions effectively in various database types and can seamlessly correlate them, thereby improving the capabilities of *Esfinge Query Builder CORE*. The tests are accessible at the following link: https://github.com/EsfingeFramework/querybuilder/tree/master/PolyglotDemo/PolyglotDemoDataGenerator.

6 CASE STUDY

We developed a case study to implement the TPPFM in a polyglot setting using real data. To achieve this, it was crucial to evaluate the utilization of annotations, illustrate the relationship between classes and utilized frameworks through the Design Structure Matrix (DSM) (Eppinger, 2012), and confirm the feasibility of the TPPFM.

6.1 Method

The selection of case studies involved the identification of databases of different types that contain data within the same application domain. The authors used two databases provided by CEMADEN (National Centre for Monitoring and Early Warning of Natural Disasters). It is a Brazilian government organization that monitors in real time and distributes alerts about the probability of natural disasters, such as landslides, flash floods, and floods.

To validate the proposed framework, we selected databases from CEMADEN for their ability to demonstrate a heterogeneous environment comprising both relational and NoSQL databases. This choice was made because we need to recreate a real-life situation where multiple persistence models exist at the same time. This will allow us to test how well the framework works with others and in real-life situations.

The authors developed the case study's source code in Java and integrated it into the CEMADEN software ecosystem, allowing access to real-time-generated data. We added Esfinge Virtual Lab (explained in section 6.2) to the CEMADEN software ecosystem and uploaded JAR modules to make it easier for the integration of study materials and produce insightful results for CEMADEN scientists using databases.

6.2 Esfinge Virtual Lab

The Esfinge Virtual Lab tool has been used for the front-end of the case study. It is open-source and uses substantially the Esfinge query builder. It has been updated to take into account TPPFM, enabling a polyglot approach for the present work.

The Esfinge Virtual Lab is detailed in a comprehensive way in the publication by (Pereira et al., 2023a) and can be found at https://github.com/EsfingeFramework/virtual-lab/releases. It is a virtual laboratory featuring a metadata-driven API for the development of dynamic software components. The platform, developed in Java, seeks to streamline system prototype, enabling developers to focus on business concepts rather than information retrieval or interface programming. The Esfinge Virtual Lab employs declarative programming techniques to facilitate the fast creation of data visualizations, including tables, charts, and maps, with minimal coding requirements. The components, encapsulated as JAR files, can be dynamically loaded and retrieved, facilitating code reuse. For the case study presented here, it is important to understand the following annotations that were used.

- **@ServiceDAO** - defines the access configurations for the database of the main entities and provides data access services;
- **@PolyglotConfig** - defines the access configurations for the secondary entities' database (developed in the Esfing VirtualLab framework due to the case study);
- **@ServiceClass** - defines a control class that can contain available service methods;
- **@ServiceMethod** - defines that a method will be made available as a service that can be invoked;
- **@Inject** - allows injecting a service or data class into another service class. Example: injected data access class;
- **@BarChartReturn** - returns the result of the service method as a bar chart;
- **@TableReturn** - returns the result of the service method as a data table.

6.3 Context

The case study was formulated with regard to the realm of natural disasters. Two databases were provided at CEMADEN. A relational database utilizing PostgreSQL and a NoSQL database utilizing MongoDB.

The relational database stored alert data. An alert includes a creation date, a termination date, details on the probable natural disaster, the severity rating, and the city location. In the case of CEMADEN, notifications are sent by cities, which include a unique identification code established by IBGE (Brazilian Institute of Geography and Statistics).

The NoSql database comprises data from the BATER (Statistical Territorial Base of Risk Areas), in verbatim transcription, Statistical Territorial Base of Risk Areas (Instituto Brasileiro de Geografia e Estatística (IBGE), 2018). The BATER technique is comprehensive and encompasses numerous specific topics relevant to censitary statistics and natural disasters. For the purposes of this case study, it is adequate to recognize that a BATER delineates a geographical area within a city, intersecting risk area delimitation data with census sectors to estimate, with varying degrees of precision, the probable number of individuals exposed to natural disasters.

The BATER data do not cover all Brazilian cities or all risk regions, comprising 183 variables about residents, predominantly quantitative, categorized by gender, age, and other factors. It encompasses 135 attributes of households, including property ownership status and urban services obtained, among others. Furthermore, not all variables possess values for every existing BATER. This illustrates a versatile structure of the data and the utilization of a non-relational database.

The alert data are operational, which means that they are integral to the daily operations of CEMADEN. The BATER data are finely organized datasets that offer significant insights about Brazilian cities within the researched context, utilized by CEMADEN predominantly for research endeavors and, to a lesser degree, for operational applications. However, there is no mechanism for correlated data within the same domain in a manner that is feasible for developers and capable of generating configurable and dynamic services that address both the research and operational requirements of end users.

Consequently, the suggested case study seeks to fulfill this identified requirement. In this context, an

Alert is the primary entity, and an *Alert* possesses a list of *Bater*'s. The variable investigated was the total number of individuals present in a *Bater*. Consequently, it may be feasible to determine the number of individuals possibly at risk within the city by aggregating the population figures from all the *Bater*'s in that area.

6.4 Implementations

The implementation of the case study required the creation of four Java classes, as illustrated in the diagram in Figure 4. The class *AlertService* utilizes a dynamic proxy to inject an instance of *AlertDAO*. *AlertDAO* includes service methods that access data purely through method declarations, using a declarative API. In *AlertDAO*, the annotation *@TargetEntity* designates *Alert* as the primary entity. *Alert* is annotated with *@PersistenceType*, indicating the persistence type for the primary and secondary entities. In the scenario of the case, *Bater* is utilized as the secondary entity, associated with *Alert* through *@PolyglotOneToMany*. The *ibgecode* attribute in *Alert* contains the same value as the *citycode* in *Bater*, allowing for the association to be made.

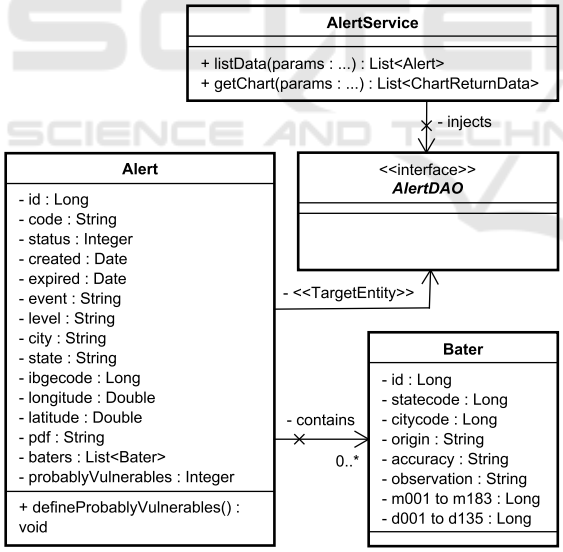

Figure 4: Class diagram from case study.

6.5 Results

6.5.1 Feasibility Results

The feasibility results are illustrated in Figure 5. The focus is on the visualization in tabular and bar chart formats corresponding to the two service methods provided in *AlertService*, namely *listData* and *getChart*. A list of alerts is presented that indicates the probable number of vulnerable individuals in Rio Grande do Sul, Brazil, during the period from April 1, 2024, to July 1, 2024. A bar graph illustrates the progression of alerts in relation to the probable number of vulnerable individuals, taking into account the alert creation date and its expiration.

This case study specifically examined the period during which numerous alerts spread to the population, which ultimately resulted in significant catastrophes in the Rio Grande do Sul region due to excessive rainfall over several days.

6.5.2 Implementation Analysis

Concerning the association of classes with frameworks, Table 1 delineates the number of annotations assigned to each class, with EQB representing Esfinge Query Builder, JPA denoting Java Persistence API, MOR indicating Morphia ODM, EVL referring to Esfinge Virtual Lab, and POL for classes using polyglot annotations. This shows high modularity and low coupling of the solution.

Table 1: Number of annotations used from each framework.

	JPA	MOR	EQB	POL	EVL
AlertService	0	0	0	0	6
AlertDAO	0	0	2	1	3
Alert	5	0	0	3	0
Bater	0	3	0	1	0

From the TPPFM perspective, observe that the *AlertService* class, which represents the business layer, lacks any polyglot annotations, allowing the developer to concentrate solely on the business logic. In the persistence layer, the majority of polyglot annotations are found in the *Alert* class, which represent the primary entity, denoting the persistence type (*@PersistenceType*) and the association annotations (*@PolyglotOneToMany* and *@PolyglotJoin*). In the *AlertaDAO* class, it is necessary to specify only the target primary entity (*@TargetEntity*). In the *Bater* class, only persistence type requires declaration (*@PersistenceType*).

Figure 6 shows the DSM corresponding to the four classes created. We constructed the DSM using the Esfinge Virtual Lab packages, Esfinge Query Builder CORE packages, Esfinge Query Builder JPA packages, and Esfinge Query Builder MongoDB packages. From the perspective of coupling, focus on the colorful rectangles highlighted in the figure. The purple area delineates the dependencies for Esfinge Virtual Lab, specifically, the service specification and the DAO annotations for the front-end. However, the primary goal of this work is to focus on the elements in-

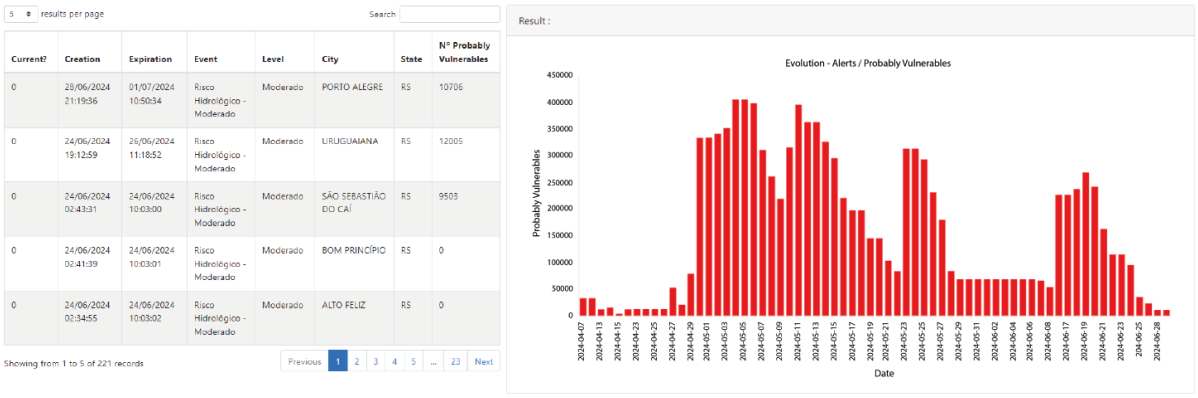

Figure 5: Table with a list of alerts and probable number of vulnerable individuals (*left). Bar chart showing the evolution of alerts versus probable number of vulnerable individuals (*right).

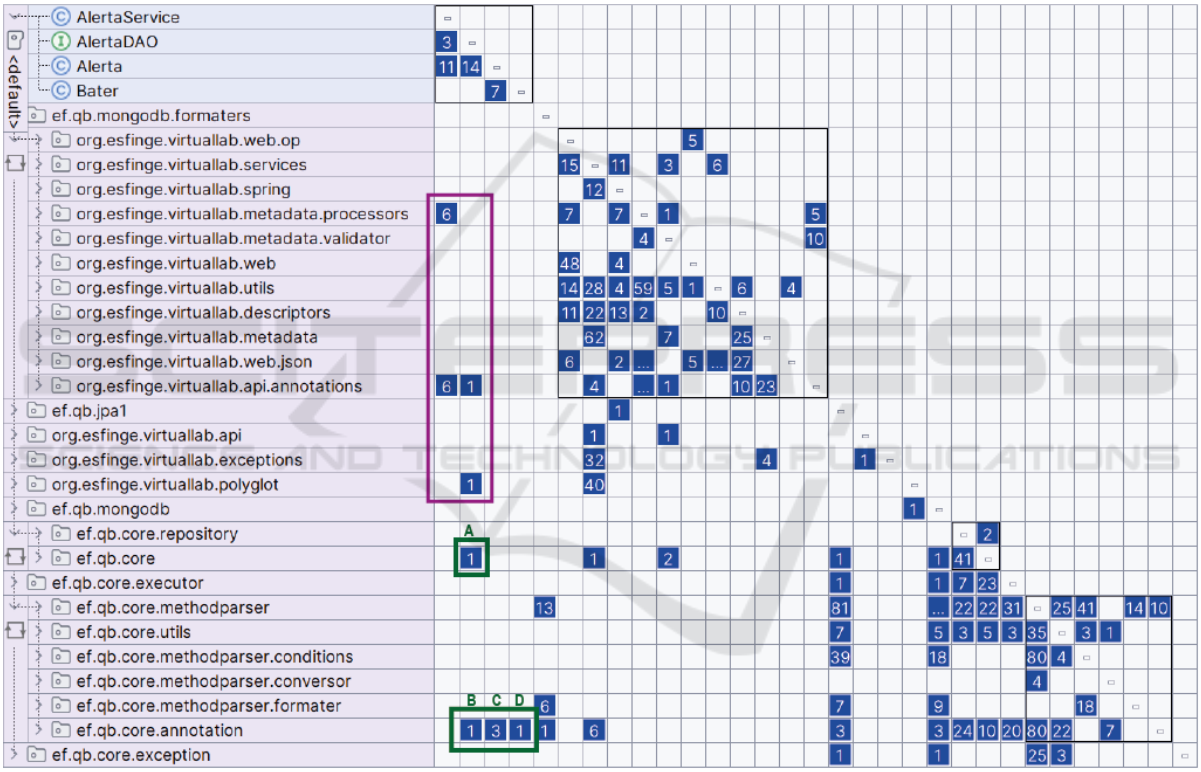

Figure 6: Case Study - Dependency Structure Matrix.

dicated by the green rectangles labeled *A* to *D*. These are, in reality, the dependencies of the TPPFM implementation. In *A*, there is a dependency on *AlertDAO* for the *Repository* interface. From *B* to *D*, we have only annotations. In *B*, we have @*TargetEntity* for *AlertDAO*. In *C*, utilize @*PersistenceType*, @*PolyglotOneToMany*, and @*PolyglotJoin* for *Alert*. In *D* @PersistenceType for *Bater*. The framework exhibits high modularity, since applications rely just on metadata rather than concrete implementations.

7 DISCUSSIONS

7.1 Case Study Conclusions

The case study effectively illustrated the operation of the reference implementation for TPPFM. The implementation used real data. Some difficulties merit consideration. The CEMADEN relational data model is extensively normalized and includes multiple tables, significantly increasing the difficulty of mapping. For

the case study, the authors opted to develop an *Alert View*, which allowed the study to be carried out without compromising the coherence of the data.

The Esfinge VirtualLab facilitated the fast generation of visualizations for front-end development. It internally incorporates the Esfinge Query Builder, and the utilization of the *@ServiceDAO* and *@PolyglotConfig* annotations facilitates fast setting of database access.

In general, the case study may be developed with minimal source code, which is advantageous for the framework given its significant internal complexity. The advantages of this design can enhance the developer experience due to minimal integration between frameworks, a unified API, enhanced maintainability, and autonomy.

7.2 Limitations

The case study has limitations, including the testing of only two types of databases and the absence of performance analysis. A study involving two distinct database types is adequate to validate the framework's operation, as the particular implementations do not directly influence the *Esfinge Query Builder CORE*; instead, they serve as extensions. However, the framework was designed to allow polyglot operation with a maximum of two, independent of their type, simultaneously. Nevertheless, instances involving three or more databases within the same topic are infrequent in the literature, prompting the authors to use this approach.

8 CONCLUSIONS

The research work delineated the definition of the TPPFM, establishing a conceptual framework as a foundation for the transparent implementation of polyglot persistence for developers. Using the Esfinge Query Builder as a reference framework, it was feasible to evolve it for polyglot functionality alongside its existing capabilities that correspond to the requested work.

A case study was created using real databases that illustrated the operation of the framework. The incorporation of Esfinge VirtualLab facilitated the fast acquisition of data visualization that connected information across PostgreSQL and MongoDB databases within a unified domain model, enabling seamless polyglot operations through various ORM mappings. The framework features a cohesive declarative API that abstracts the utilization of diverse APIs from several databases.

The results indicated that the framework promotes development with minimal coding and substantial modularity. Future endeavors will focus on creating performance assessments and, crucially, executing experiments to evaluate the developer's experience, yielding suggestions for enhancing the framework and broadening its capabilities.

REFERENCES

de Araújo, A. M. C., Times, V. C., and da Silva, M. U. (2016). Polyehr: A framework for polyglot persistence of the electronic health record. In *Proceedings on the International Conference on Internet Computing (ICOMP)*, page 71. The Steering Committee of The World Congress in Computer Science, Computer ....

De Souza, W. S., Pereira, F. O., Albuquerque, V. G., Melegati, J., and Guerra, E. (2022). A framework model to support a/b tests at the class and component level. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 860–865, Los Alamitos, CA, USA. IEEE.

Eisenhuth, P. and Jablonski, S. (2022). Knowledge-based recommendation for polyglot persistence. In *CDMS@ VLDB*.

El Ahdab, L., Megdiche, I., Péninou, A., and Teste, O. (2024). Unified models and framework for querying distributed data across polystores. In *International Conference on Research Challenges in Information Science*, pages 3–18. Springer.

Eppinger, S. (2012). *Design Structure Matrix Methods and Applications*. MIT Press.

Givre, C. and Rogers, P. (2018). *Learning Apache Drill: Query and Analyze Distributed Data Sources with SQL*. " O'Reilly Media, Inc.".

Guerra, E. (2014). Designing a framework with test-driven development: A journey. *IEEE software*, 31(1):9–14.

GUERRA, E. M., BATISTA, J. A., and NASCIMENTO, L. W. T. (2017). Esfinge query builder - framework de acesso a dados para diferentes paradigmas de banco. In *CBSoft VIII Congresso de Software Brasileiro*, pages 65–72, Porto Alegre, RS, Brazil. Sociedade Brasileira de Computação (SBC).

Holubová, I., Contos, P., and Svoboda, M. (2021). Multimodel data modeling and representation: State of the art and research challenges. In *Proceedings of the 25th International Database Engineering & Applications Symposium*, pages 242–251.

Instituto Brasileiro de Geografia e Estatística (IBGE) (2018). *População em Áreas de Risco no Brasil*. IBGE, Rio de Janeiro.

Jiménez-Peris, R., Patino-Martinez, M., Brondino, I., and Vianello, V. (2016). Transactional processing for polyglot persistence. In *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 150–152, Piscataway, NJ, USA. IEEE, IEEE.

Kaur, K. and Rani, R. (2015). Managing data in healthcare information systems: many models, one solution. *Computer*, 48(3):52–59.

Khine, P. P. and Wang, Z. (2019). A review of polyglot persistence in the big data world. *Information*, 10(4):141.

Kolev, B., Valduriez, P., Bondiombouy, C., Jiménez-Peris, R., Pau, R., and Pereira, J. (2016). Cloudmdsql: querying heterogeneous cloud data stores with a common language. *Distributed and parallel databases*, 34(4):463–503.

Lajam, O. and Mohammed, S. (2022). Revisiting polyglot persistence: From principles to practice. *International Journal of Advanced Computer Science and Applications*, 13(5).

Larman, C. (2012). *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. Pearson Education India, Delhi, India.

Lima, P., Pereira, N. S., Gomes, E., Guerra, E., and Meirelles, P. (2023). Annotation visualizer: A software visualization tool for code annotations. *Software Impacts*, 16:100491.

Lu, J., Liu, Z. H., Xu, P., and Zhang, C. (2018). Udbms: road to unification for multi-model data management. In *International Conference on Conceptual Modeling*, pages 285–294, Cham, Switzerland. Springer, Springer.

Mangal, H. (2024). Cpsl: A domain-specific language for modelling the behaviour of cyber-physical systems. B.S. thesis, University of Twente.

Pereira, F., França, D., Paschoal, V., Nardes, M., Rosa, R. R., and Guerra, E. (2023a). Esfinge virtual lab—a virtual laboratory platform with a metadata-based api and based on dynamic component. *IEEE Access*, 11:143167–143181.

Pereira, F., Guerra, E., and Rosa, R. R. (2023b). Patterns for polyglot persistence layer. In *Proceedings of the 29th Conference on Pattern Languages of Programs*, PLoP '22, USA. The Hillside Group.

Prasad, S. and Avinash, S. (2014). Application of polyglot persistence to enhance performance of the energy data management systems. In *2014 International Conference on Advances in Electronics Computers and Communications*, pages 1–6. IEEE.

Rising, L. (1998). *The patterns handbook: Techniques, strategies, and applications*, volume 13. Cambridge University Press.

Schaarschmidt, M., Gessert, F., and Ritter, N. (2015). Towards automated polyglot persistence. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, 241:73–82.

Scherzinger, S. and Sidortschuck, S. (2020). An empirical study on the design and evolution of nosql database schemas. In *Conceptual Modeling: 39th International Conference, ER 2020, Vienna, Austria, November 3–6, 2020, Proceedings 39*, pages 441–455, Vienna, Austria. Springer, SpringLink.

Srivastava, K. and Shekokar, N. (2016). A polyglot persistence approach for e-commerce business model. In *2016 International Conference on Information Science (ICIS)*, pages 7–11. IEEE.

Tudose, C., Bauer, C., and King, G. (2023). *Java persistence with spring data and hibernate*. Simon and Schuster, New York, NY, USA.

Van Landuyt, D., Benaouda, J., Reniers, V., Rafique, A., and Joosen, W. (2023). A comparative performance evaluation of multi-model nosql databases and polyglot persistence. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pages 286–293.

Villaça, L. H., Azevedo, L. G., and Baião, F. (2018). Query strategies on polyglot persistence in microservices. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1725–1732, Pau, France. ACM.

Wiese, L. (2015). Polyglot database architectures= polyglot challenges. In *LWA*, pages 422–426, Göttingen, Germany. University of Göttingen.