Patterns for Polyglot Persistence Layer

FERNANDO PEREIRA, National Institute for Space Research
EDUARDO GUERRA, Free University of Bozen-Bolzano
REINALDO R. ROSA, National Institute for Space Research

There has been an increase in the generation of data due to the fact that apps are now able to take into account or store information in a variety of formats. In this context, modern programs deal with a variety of different types of databases while still operating inside the same domain. The phenomenon known as polyglot persistence describes what happens in this case. It is possible for parts data to be stored in one kind of database more successfully than in another, with dependencies on blocks of data stored in a different format. As a direct consequence of this, the development process becomes more difficult because it is now important to have an understanding of many APIs as well as how to correlate the data. This study makes a proposal to identify polyglot persistence patterns that have been employed in previously published studies. When discovered, these patterns have the potential to make the creation of apps that use polyglot persistence much simpler.

Categories and Subject Descriptors: Software and its engineering [**Software organization and properties**]: Software system structures—*Software architectures*; Software and its engineering [**Software creation and management**]: Designing software—*Software design engineering*; Information systems [**Data management systems**]: Information integration—*Mediators and data integration*

General Terms: Software Architecture

Additional Key Words and Phrases: polyglot persistence, patterns, persistence layer

ACM Reference Format:

Pereira, F. and Guerra, E. and R. R. Rosa 2022. Patterns for Polyglot Persistence Layer. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 29 (October 2022), 8 pages.

1. INTRODUCTION

The idea behind polyglot persistence is to use a number of different database systems inside a single application domain, with each database system catering to a set of requirements in terms of both its functionality and its non-functionality [Sadalage and Fowler 2013] [Schaarschmidt et al. 2015]. This concept is a result of the tendency of many businesses to store their operational data in different groups of technologies, SQL and NoSQL databases [Khine and Wang 2019]. Because of this, it is necessary to choose the most effective persistence model for each individual work that is to be carried out, given that each model can be modified to more efficiently tackle certain difficulties [Villaça et al. 2018]. As a consequence of this, polyglot persistence may be basically stated as the utilization of data storage inside system components that belong to the same application domain [Wiese 2015].

This study identifies patterns in order to record experiences, taking into account the previous researches that has been conducted on polyglot persistence. The patterns are meant to be used in situations in which several databases are being utilized to store instances parts of the same domain model. When working in scenarios that involve polyglot persistence, developers may find that using the identified patterns makes their jobs easier.

Author's address: F. Pereira, Astronautas Ave., 1758, São José dos Campos, São Paulo, Brazil 12227-010; email: fernando.opc@gmail.com; E. Guerra, Piazza Università, 1, 39100 Bolzano BZ, Italy; email: guerraem@gmail.com; R. R. Rosa, Astronautas Ave., 1758, São José dos Campos, São Paulo, Brazil 12227-010; email: rrrosa.inpe@gmail.com

Three patterns were identified. Although each offers a different approach to the problem, the context and the forces remain the same. The patterns that are given are associated with the DAO pattern (*Data Access Object*) [Buschmann et al. 2007]. The DAO pattern allows for the creation of data classes regardless of the type of data source that is being accessed. This includes relational databases, NoSQL databases, text files, and other types of data sources. At DAO pattern, encapsulating data access mechanisms allows them to be modified in a way that is not dependent on the code that makes use of the data. The business logic is not altered in any way as a result of this, and the resulting classes are easier to comprehend [Sun et al. 2010]. The three patterns that were identified are as follows:

(1) **Independent DAO**: access to the various databases is not transparent while using this pattern; rather, it is carried out directly within the program's source code;

(2) **Integrated Polyglot DAO**: this pattern makes it possible for developers to have transparent access to several databases through the use of a single API that incorporates multiple DAOs;

(3) **DAO Compound Mediator**: this pattern makes it possible for developers to have transparent access to several databases by utilizing a mediator API that provides independent access to multiple DAOs.

1.1 Context, Problem and Forces

Context: It may be necessary, when developing an application, to manage correlated data in multiple formats within the same domain. This information can be stored in relational databases and other blocks may perform better if stored in diverse NoSQL databases. Moreover, if the application needs to be distributed and scalable, NoSQL-based solutions are more appropriate [de Queiroz et al. 2013]. In addition, utilizing a variety of NoSQL database types can provide useful features associated with documents, graphs, and time series, among others [Pritchett 2008]. In this scenario, we aim to manipulate the entire domain dataset in an efficient manner. This requires the developer to deal with the persistence of separate data blocks using different paradigms and formats, which can become complex.

Problem: **How to structure an application with the same domain model whose objects are persisted in different databases?**

Forces:

F1. It is possible that application maintenance will become more difficult if different APIs will need to be accessed simultaneously within the same class in order to access multiple persistence kinds, resulting in strong coupling; — **But** reducing coupling by accessing a single API that takes responsibility for transparent access to all others makes it difficult to update these.

F2. When databases are accessed independently, it is difficult to handle transactions since ensuring consistency is complicated. Considering the circumstance when portions of the same object are stored across multiple databases, accessing various APIs will make it difficult to make the transaction atomic; — **But** accessing many databases at the same time through a single interface can provide significant implementation challenges for the persistence layer.

F3. When database access occurs in an atomically encapsulated pool within the persistence layer, transaction management is transparent to the developer. Isolate APIs usage in the application's persistence layer is structurally more viable to outsource transaction management to this layer; — **But** accessing different databases through different APIs independently can be more practical if we consider the different natures and formats that each one handles.

F4. It may be required for the developer to be familiar with many types of data stores and their APIs. Accessing diverse persistence formats, whether from relational or non-relational databases, may require the programmer to have specialized understanding of each. This is inefficient and inhibits the use of polyglot persistence; — **But**

accessing a single API can generate a lot of complexity for the persistence layer by making data of different formats have to be composed generating a single new format.

F5. In the persistence layer, when APIs from several databases are accessed, business rules do not need to be aware of the data stores. When domain business rules are implemented independent of the characteristics of data repositories, they are transparent to the programmer via a uniform language; — **But** it may be challenging to upgrade APIs in the persistent layer, which may result in refactorings occurring in this layer;

2. BACKGROUND

2.1 Persistence Layer

The layered architecture pattern is by far the most common type of software architecture pattern. This structure divides the components into horizontal levels, each of which serves a specific purpose. The layers are disconnected from one another, thus each of their components is responsible for the logic of its own layer. This guarantees isolation and reduces the amount of work required for refactoring. [Richards 2015].

Transformations, calculations, and aggregations are carried out and returned to the requester on a layer-by-layer basis as the request is passed from one tier to the next. The visibility between the different layers can be adjusted, and further layers can be added or removed, depending on the application domain and the architectural design that is planned. The persistence layer is typically developed using the repository pattern or other that are quite similar to it [Schram 2015].

In the persistence layer, the classes typically encapsulate the required database access logic. Several patterns can be used to accomplish the goals of this layer. These patterns establish abstractions, facilitate maintainability and refactoring, and offer source code decoupling and isolation. This layer's patterns and approaches are described in the following subsections.

2.2 DAO Pattern

Following the DAO pattern (*Data Access Object*), data classes can be created regardless of the data source type accessed, including relational databases, NoSQL databases, and text files. Data access mechanisms are encapsulated and can be modified independently of the code that utilizes them. With this, the business logic is unaffected and the classes are more readable [Sun et al. 2010].

To ensure isolation, it is generally desirable for DAO implementations to restrict data access to this type of class. Moreover, each DAO instance must be uniquely responsible for a domain object and must implement CRUD operations. Internally, transactions, sessions, and connections should not be managed. In addition to accessing the database layer, the DAO classes are responsible for transforming requests into queries and transcribing the results into the desired format.

When applied correctly, the DAO pattern generates data access transparency for the application, which simplifies processes such as migration and database exchange. Moreover, the application now has a centralized data access point, which will facilitate future refactorings.

2.3 Polyglot Persistence

In 2009, researcher Scott Leberknight coined the term polyglot persistence. Primarily as a result of the development of database technologies, the term has become more researched since 2013 [Sadalage and Fowler 2013]. In recent years, the term "polyglot persistence" has generated a great deal of interest among researchers working in the field of software architecture; yet, there are currently no consolidated concrete reference solutions for polyglot structures. A systematic review can be found in the work of Khine and Wang [2019], where models and prototypes have been found in both the state of the art and in the state of the tech.

For the majority of authors, polyglot persistence refers to the practice of employing multiple database systems within a single application domain in order to meet diverse functional and non-functional system requirements

[Schaarschmidt et al. 2015]. It is a viable alternative in order to meet the requirements of modern data management infrastructures instead of defining a single database management system to store all the data [Wiese 2015] [Gamal et al. 2021] [Glake et al. 2022] [Roy-Hubara et al. 2022].

3. INDEPENDENT DAO

Solution: **Make use of a different DAO for each different kind of database that the business layer needs access to. Make calls to these DAOs directly from the source code and perform the transformation and aggregation of retrieved data blocks manually.**

In this solution, the developer opens a connection for each database type before invoking a specific DAO. The query is executed sequentially. For certain fields of an object containing data in multiple databases, data is initially retrieved from a first database. Then, these data are filtered when querying a second database in order to populate the other fields based on the requested query. This procedure is repeated for data blocks located in other databases. See Figure 1.

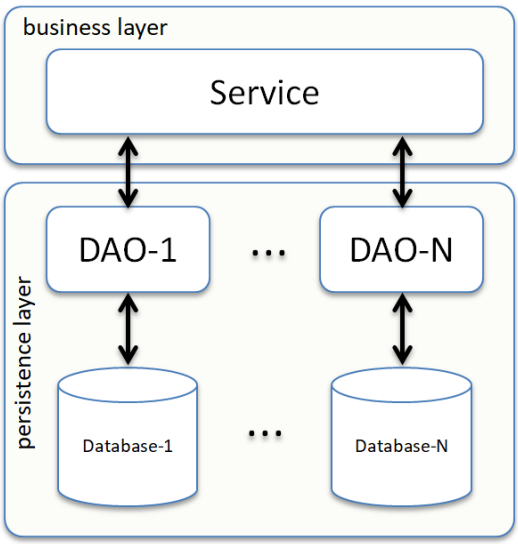


Fig. 1. Independent DAO Pattern.

It is important to keep in mind that the obtained data will first be delivered to the business and service layer before being filtered. The developer of this solution needs to be knowledgeable about all possible types of storage and be able to independently manage data correlation transactions. This means that any changes made to a block of data that is being persisted in one database but could potentially affect the consistency of data in another database must be handled manually by the developer.

Consequences:

(+) Maintenance. Does not mix APIs in persistence layer;

(-) Transaction. As APIs are invoked independently in the business layer, transactions are independent, making it difficult to maintain consistency;

(-) Transparency. Data integration is done at the business layer level, losing transparency for the developer;

(+) Cohesion. Decoupled maintenance of APIs can be performed independently;

(-) Upgrade. Business rules need to manage multiple data repositories;

Known Uses: The Spring DATA framework [Spring 2022] enables developers to work in polyglot persistence scenarios compliant with the Independent DAO specification. It contains distinct APIs for each type of storage, allowing developers to use them independently while correlating data at the business layer level. This pattern is also identified in Gamal et al. [2021], where independently correlating data accessed via different APIs in the health domain.

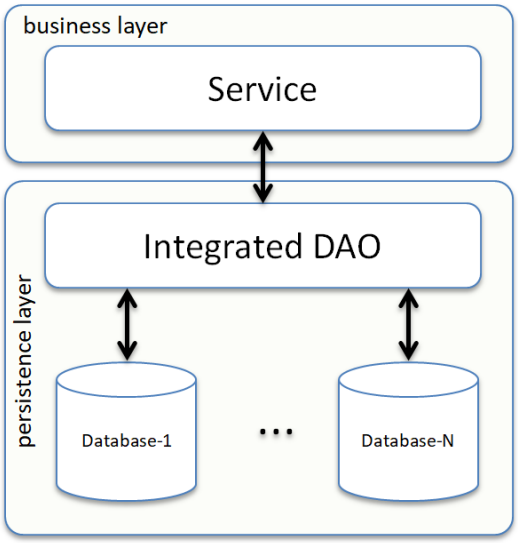
Related Patterns: The DAO pattern and the ORM technique (*Object Relational Mapper*) [Fowler 2012] are related to the Independent DAO pattern and can be implemented concomitantly as part of the solution schema.

4. INTEGRATED POLYGLOT DAO

Solution: **Use one integrated DAO for all of the different kinds of databases that are used in the persistence layer. After associating the data, transfer it to the business layer.**

This approach is based on the developer calling just one built-in DAO. The service and business layers are unaware of the different types of databases or even that they exist. The request for an object that contains data from many databases is sent to a single interface in the persistence layer. The result is obtained by combining various APIs in order to manage blocks of data that are stored in different databases. See Figure 2.

Fig. 2. Integrated Polyglot DAO Pattern.



In this approach, the DAO API is responsible for managing transactions within a single method. As a result, any alterations to the data contained within one database that have the potential to result in inconsistencies within another database are already taken care of automatically, ensuring that the entire process is atomic. Implementations are constructed ad hoc, independently of the storage logic, by joining APIs and carrying out the necessary filters and transformations in order to produce the return.

Consequences:

(-) Maintenance. Mixing APIs within the persistence layer can produce couplings;

(+) Transaction. Transactions are managed together favoring consistency;

(+) Transparency. Integration at the persistence layer level with developer transparency;

(-) Cohesion. Joint maintenance is dependent on multiple APIs, causing an API update of a database type to affect the functionality of a persistence layer method that uses it;

(+) Upgrade. Transparency of the existence of multiple data repositories for business rules;

Known Uses: The Apache Drill framework [Hausenblas and Nadeau 2013] is one example where this pattern can be identified. In it, multiple APIs are combined and consolidated using the SQL language in order to gain access to various databases via raw files. Access to data is transparent to the service/business layer, and transactions are processed concomitantly; however, the variety of APIs makes maintenance difficult.

This pattern is also observed in the CloudMdsQL project [Jiménez-Peris et al. 2016], where, despite the fact that native queries can be interpreted as functions within the unified language, there is in fact no separation of APIs and the entire set is viewed as a single component.

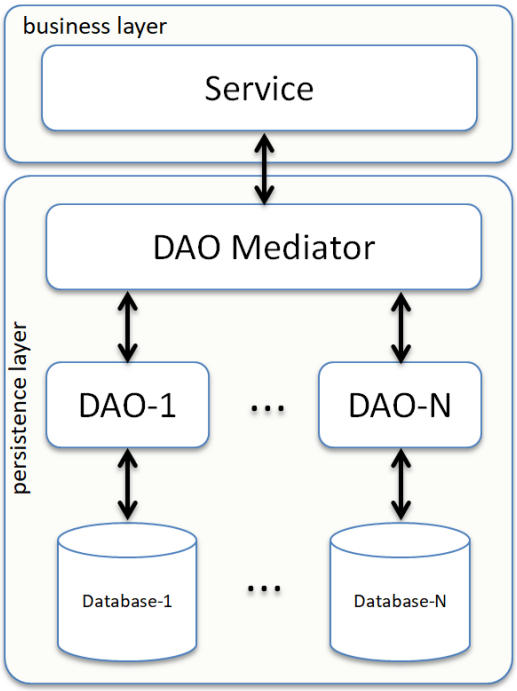
Related Patterns: This pattern is compatible with the DAO pattern. However, mixed APIs make it difficult to use the ORM technique.

5. DAO COMPOUND MEDIATOR

Solution: **Use a DAO mediator in the persistence layer. This mediator calls separate DAOs for each kind of database, aggregates the results, and then returns them to the service or business layer.**

In this approach, a mediator DAO is invoked by the developer. The service/business layer is transparent to database types and their existence. A request for an object containing data in multiple databases is sent to the DAO mediator in the persistence layer. According to the storage type of each data block, the DAO mediator interprets the request and splits the operation using separate APIs for each data block. See Figure 3.

Fig. 3. DAO Compound Mediator Pattern Scheme.



Transactions are managed independently during data correlation. The mediator DAO is responsible for ensuring storage consistency and making the atomic transaction and data repositories transparent from the perspective of the service/business layer.

Consequences:

(-) Maintenance. Mixing APIs in persistence layer;

(+) Transaction. Transactions are managed independently during data correlation. The mediator DAO is responsible for ensuring storage consistency and making the atomic transaction and data repositories transparent from the perspective of the service/business layer;

(+) Transparency. Persistence layer level integration;

(+) Cohesion. Decoupled maintenance of APIs can be performed independently;

(+) Upgrade. Transparency of the existence of multiple repositories for business rules;

Know Uses: This pattern is recognized by the PPM mediator [Schaarschmidt et al. 2015]. It is also identified similarly in the work WA-RAF [Zambom Santana and dos Santos Mello 2019], which proposes scalable middleware for multiple NoSQL databases.

Related Patterns: The DAO pattern and the ORM technique are related to the DAO Composite Mediator pattern and can be implemented concomitantly as part of the solution strategy.

6. CHOOSING THE PATTERNS

Each of the identified patterns generates a solution that meets contradictory forces. It is necessary for the developer to make a trade-off analysis to verify the best approach to be adopted. The table I assists in selecting the best appropriate pattern based on the indicated consequences.

Table I. Patterns x consequencies of it adotion.

Pattern	Maintenance	Transaction	Transparency	Cohesion	Upgrade
Independent DAO	+	-	-	+	-
Integrated Polyglot DAO	-	+	+	-	+
DAO Compound Mediator	-	+	+	+	+

The consequences that are met by the pattern are marked with positive or negative impact.

7. FINAL CONSIDERATIONS

The results of this research identified three patterns that have implications for the field of polyglot persistence. This identification could be of assistance to developers in better structuring their applications when the polyglot persistence context is present. Not only was polyglot persistence a primary consideration during the design of the patterns, but so was the possibility that objects belonging to the same domain might have portions of their data stored in more than one database.

There are several challenges identified for implementing each pattern, with both positive and negative consequences. The forces involved are contradictory, and it is up to the developer to define the best approach according to the application. In future work, it is intended to address good code implementation practices for the use of the pattern described in this paper.

In conclusion it is important to mention that the approaches defined in the identified patterns has the potential to be applied in several application areas such as space sciences, e.g. Rosa [2020] and Lara et al. [2008], environmental sciences, e.g. Velho et al. [2001] and also in the areas of health sciences, e.g. Agrawal [2020].

REFERENCES

Anurag Agrawal. 2020. Bridging digital health divides. *Science* 369, 6507 (2020), 1050–1052.

Frank Buschmann, Kevlin Henney, and Douglas Schmidt. 2007. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, 4th Volume*. John Wiley & Sons.

Gilberto Ribeiro de Queiroz, Antônio Miguel Vieira Monteiro, and Gilberto Câmara. 2013. Bancos de dados geográficos e sistemas NoSQL: onde estamos e para onde vamos. *Revista Brasileira de Cartografia* 65, 3 (2013), 20.

Martin Fowler. 2012. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley.

Aya Gamal, Sherif Barakat, and Amira Rezk. 2021. Standardized electronic health record data modeling and persistence: A comparative review. *Journal of biomedical informatics* 114 (2021), 103670.

Daniel Glake, Felix Kiehn, Mareike Schmidt, Fabian Panse, and Norbert Ritter. 2022. Towards Polyglot Data Stores–Overview and Open Research Questions. *arXiv preprint arXiv:2204.05779* (2022).

Michael Hausenblas and Jacques Nadeau. 2013. Apache drill: interactive ad-hoc analysis at scale. *Big data* 1, 2 (2013), 100–104.

Ricardo Jiménez-Peris, Marta Patino-Martinez, Iván Brondino, and Valerio Vianello. 2016. Transactional Processing for Polyglot Persistence. In *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. IEEE, 150–152.

Pwint Phyu Khine and Zhaoshun Wang. 2019. A review of polyglot persistence in the big data world. *Information* 10, 4 (2019), 141.

Alejandro Lara, Andrea Borgazzi, Odim Jr Mendes, Reinaldo R Rosa, and Margarete Oliveira Domingues. 2008. Short-period fluctuations in coronal mass ejection activity during solar cycle 23. *Solar Physics* 248, 1 (2008), 155–166.

Dan Pritchett. 2008. BASE: An Acid Alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability. *Queue* 6, 3 (2008), 48–55.

Mark Richards. 2015. *Software architecture patterns*. Vol. 4. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA.

Reinaldo R Rosa. 2020. Data Science Strategies for Multimessenger Astronomy. *Anais da Academia Brasileira de Ciências* 93 (2020).

Noa Roy-Hubara, Peretz Shoval, and Arnon Sturm. 2022. Selecting databases for Polyglot Persistence applications. *Data & Knowledge Engineering* 137 (2022), 101950.

Pramod J Sadalage and Martin Fowler. 2013. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education.

Michael Schaarschmidt, Felix Gessert, and Norbert Ritter. 2015. Towards automated polyglot persistence. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015).

Aaron Schram. 2015. *Software architectures and patterns for persistence in heterogeneous data-intensive systems*. Ph.D. Dissertation. University of Colorado at Boulder.

Spring. 2022. Spring Data. `https://spring.io/projects/spring-data`. (2022). (Acessado em 04/19/2022).

Xia Sun, Yiming Zhao, Chang Yao, and Runping Yang. 2010. Research and implementation of an efficient data persistence model. In *2010 2nd International Conference on Future Computer and Communication*, Vol. 1. IEEE, V1–361.

Haroldo F Campos Velho, Reinaldo R Rosa, Fernando M Ramos, Roger A Pielke, Gervásio A Degrazia, Camilo Rodrigues Neto, and Ademilson Zanandrea. 2001. Multifractal model for eddy diffusivity and counter-gradient term in atmospheric turbulence. *Physica A: Statistical Mechanics and its Applications* 295, 1-2 (2001), 219–223.

Luís HN Villaça, Leonardo G Azevedo, and Fernanda Baião. 2018. Query strategies on polyglot persistence in microservices. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 1725–1732.

Lena Wiese. 2015. Polyglot Database Architectures= Polyglot Challenges.. In *LWA*. 422–426.

Luiz Henrique Zambom Santana and Ronaldo dos Santos Mello. 2019. A Middleware for Polyglot Persistence of RDF Data into NoSQL Databases. In *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*. 237–244. `DOI:http://dx.doi.org/10.1109/IRI.2019.00046`

Received July 2022; revised October 2022; accepted December 2022