Revisiting Polyglot Persistence: From Principles to Practice

Omar Lajam, Salahadin Mohammed
Information and Computer Science Department
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia

***Abstract*—To cope with the rapid advancements in information technologies, many database systems have been developed in the last decade to satisfy various data storage requirements, such as NoSQL databases. In many cases, using a single database system cannot be an option because of the limitations posed on the functionalities of the software application. Therefore, applications may use multiple distributed storage databases that complement each other to satisfy the conflicting requirements. Such applications that are called polyglot persistent applications. However, the practical implementation of polyglot persistence and its complexities have not been studied enough. In this paper, the most recent studies related to polyglot persistence are reviewed. Database systems are classified based on their data storage model, and their use cases are discussed. The principles of polyglot persistence and its challenges are expounded. The implementation architectures of polyglot persistence applications are categorized into Application-coordinated Polyglot Persistence, Service-oriented Polyglot Persistence, Polyglot- Persistence-as-a-Service, and Multi-models Databases. An analysis of the issues related to each architecture is presented. In light of the study findings, a practical polyglot persistence implantation strategy is proposed. The outcomes of this work can help design future polyglot persistence applications and influence future research on how to resolve the complexity involved in polyglot persistence solutions.**

***Keywords*—Database system; database architecture; relational database; NoSQL; distributed storage; multi-model database; review; classification**

I. Introduction

Data stores have been an integral component of software applications since the emergence of information technology systems, including shopping, accounting, medicine, and games applications. There is a wide range of database systems that are available for storing data, such as MySQL, MongoDB, and Cassandra, to name a few. These database systems are usually classified based on how they model and store the data. Nevertheless, relational databases are the most popular databases used in the industry [1], and they are the default option for typical applications.

Despite their popularity, relational databases have some limitations, such as expensive queries and vertical scalability, that make the selection of non-relational databases vital. As the application gets popular and the database gets burdened with millions of records, there becomes a need for using more flexible databases, such as NoSQL databases, to support features not supported by relational databases. At the same time, the complete abandonment of relational databases cannot be an option in many cases because they also support important features not supported by other databases, like data consistency and transactions atomicity. To avoid sacrificing any of the different database features, the need for using more than one database system within an application to fulfill the conflicting requirements raises.

When an application uses more than one database system, it is called a polyglot persistence application. Taking this decision of having a polyglot persistence environment is not straightforward because it increases programming complexity and requires developers' knowledge of different database systems. However, successful implementation of polyglot persistence has the great advantage of having the different database systems complement each other and satisfying the conflicting requirements.

In order to ease the development of polyglot persistence applications, several architectures are proposed in the literature on how to design and implement them. Nevertheless, the following problems are identified by this study on those proposals: 1) There is no clear distinction or categorization for the different proposed architectures, which make them irrelevant to each other and difficult to be compared. 2) There is no detailed discussion on the challenges introduced by polyglot persistence architectures, while the focus is mostly on their advantages. 3) Most of the proposed architectures are not abstract in the sense that they cannot be applied on any application domain, and they are mostly targeting specific domains (e.g., e-commerce and healthcare), which makes them, in many cases, lack generalizability.

This review study aims to address these problems and build new knowledge based on the literature findings. The main contributions of this work can be highlighted as follows:

- Review for the most recent studies related to polyglot persistence.
- Classification for database systems with a detailed discussion on their characteristics.
- Description of polyglot persistence and its principles.
- Categorization for the architectures in which polyglot persistence applications can be implemented.
- Analysis of the problems associated with each polyglot persistence architecture.
- A practical strategy for polyglot persistence implementation.

To the best of our knowledge, this is the first work that expounds polyglot persistence comprehensively, from principles to practice. This work opens new opportunities for future research to address polyglot persistence challenges. It guides future polyglot persistence research towards more mindful solutions that consider the theoretical and practical aspects of polyglot persistence. This work can also help practitioners make wiser design decisions when building polyglot persistence applications. In addition, it may serve as a reference for understanding database systems concepts and challenges.

The rest of the paper is organized as follows. Section II discusses the related work. The methodology of this work is explained in Section III. Section IV presents database systems classification. In Section V, an explanation for polyglot persistence principles is given. Section VI describes polyglot persistence architectures and challenges. Section VII outlines the proposed polyglot persistence implementation strategy. And finally, Section VIII concludes the paper.

II. Related Work

Few studies have discussed polyglot persistence concepts, architectures, and challenges. Gessert and Ritter [2] were the first who classified polyglot persistence based on research and industry into three patterns: application-coordinated polyglot persistence, microservices, and polyglot database services. The authors then described them again in [3], where they gave brief details about each pattern without mentioning issues related to each of them.

Another attempt to classify polyglot persistence is given by Khine and Wang [4] based on the polyglot persistence solution orientation. They classify the polyglot persistence solutions into three types: domain-oriented solution, query language-based solution, and other solutions (e.g., frameworks, middleware, and multi-model databases). A main observation on their classification is that it lacks disjointedness and holism.

Wiese [5] discussed polyglot databases architectures and challenges. Three architectures are described: polyglot persistence, lambda architecture, and multi-model databases. However, the lambda architecture is part of polyglot processing, not polyglot databases, as presented in [6].

Jaroslav [7] demonstrated some possible strategies for building an infrastructure that operates on integrated SQL and NoSQL databases. The study provides some approaches to construct such integrated database architectures, mainly by using multi-model databases and multi-level modeling, where interactions occur within and between at least two levels of connected databases.

Clearly, polyglot persistence is not studied enough. In this work, we try to study polyglot persistence principles, architectures, challenges, and implementation, based on the literature findings, as comprehensively as possible.

III. Methodology

The main objectives of this study are to explore how polyglot persistence applications can be architected and to understand polyglot persistence advantages and challenges. The following methodology is used in order to accomplish the study objectives.

A. Literature Review

This work is mainly a review that surveys the literature to gain knowledge on the topic. Four databases were searched to extract the related studies: IEEEXplore, ACM Digital Library, Web of Science, and Google Scholar. These four databases were chosen because they can capture most related studies. The search string used was 'polyglot persistence'. The inclusion criterion was to include any study that proposes a polyglot persistence architecture, model, or framework. The resulted studies were inspected, and 18 relevant studies were included. The selected studies were then downloaded and fully read. Each study was summarized, and all results were aggregated into an Excel file in a tabular format.

B. Problem Identification

At this stage, several problems were identified in the reviewed studies. First, many different architectures lack a clear distinction or categorization, making them unrelated to each other and difficult to compare. Second, there is no comprehensive treatment of the issues posed by polyglot persistence architectures, with the emphasis being placed mostly on their benefits. Third, most proposed architectures were specific to a few application domains, e.g., e-commerce and healthcare, which make them difficult to be generalized for other domains. In other words, many proposed architectures were developed with specific database requirements in mind.

These three problems are consistent with what Khine and Wang have found [4], where they stated that it has not yet been determined which architectures are best suited for different kinds of applications and how polyglot persistence can be implemented. Additionally, they observed that the benefits and limitations of polyglot persistence are still open research topics for academics and professionals.

C. Classification and Analysis

After identifying the problems, an intensive investigation was carried on to identify polyglot persistence principles, architectures, challenges, and implementation. Search databases were searched again with the same string. The relevant studies were analyzed, and more information was gathered from the literature to build knowledge that addresses the identified problems.

IV. Databases Classification

There are different options for storing application data, ranging from simply being stored in a file to being stored in a sophisticated data storage system, depending on the degree of complexity of the application requirements. Database Management Systems (DBMSs) are database systems that manage data storage and retrieval. The implementation of a DBMS specifies how the data will be structured and stored into the disk, how the queries will be processed, how access will be granted, and many other functions. What distinguishes one DBMS from another is its functions and features. DBMSs can be categorized into relational (RDBMS), referred to as SQL databases, and non-relational or NoSQL (Not only SQL) databases.

A. Relational Databases

Relational databases use the relational data model to store the data. It is the most popular model for storing structured data. It was first introduced by E. F. Codd in 1970 [8][9]. It is very mature, stable, trusted, and well researched. In this model, the data is organized as tables, and these tables can have relationships with each other. Examples of relational databases include MySQL, PostgreSQL, and Oracle.

Relational databases support transactions that obey ACID properties (Atomicity, Consistency, Isolation, and Durability) [10]. The atomicity property ensures that all instructions of a transaction will be executed at once, i.e., a transaction is an atomic unit of processing. The consistency property guarantees that the database state is always consistent, and the correct execution of a transaction takes the database from one consistent state to another. The isolation property makes each transaction completely isolated from others, and its effect on the database does not become visible until it is committed. This noninterference transaction execution can be achieved using concurrency control techniques [11]. The durability property, also referred to as permanency, ensures that changes made by a successful transaction will not be lost by subsequent unsuccessful transactions [12].

Relational databases have fixed database schema. They enforce data consistency and integrity. They store the data efficiently with minimal redundancy and maximal space utilization [13]. They have powerful query language. And lastly, they have a great community that provides support and help.

Relational databases manifest some drawbacks under some situations, especially when the number of database users dramatically increases and when the data volume becomes too huge. That is mainly due to the nontrivial processing required for user queries and the difficulty of operating on a distributed architecture. With massive amount of data, the relational database requires powerful machine to operate efficiently. The only option to scale the database system up is to upgrade the machine to a more powerful one. In other words, relational databases are only vertically scalable. Because they usually run on one machine, relational databases are prone to the single point of failure threat.

Relational databases are not suitable for unstructured, semi-structured, and graph data. They are not suitable for applications that store schema-less or schema-free data. They incur a high cost for complex query processing due to the table joins and constraints checking involved. They are less suitable for high-velocity ingestion due to the schema constraints validations. The relational database infrastructure (i.e., server machine) cost is expensive due to the powerful processing and storage space resources it needs, especially when the number of simultaneous users and/or the data volume becomes huge [14].

B. NoSQL Databases

The term ”NoSQL” can be interpreted as ”not using SQL query language”, or can be interpreted as ”Not only SQL”, where the latter implies either the support of a database system for a query language that is similar to SQL or implies the co-existing of a non-SQL database with a SQL database in a common polyglot persistence environment. There is no agreed-upon definition of what ”NoSQL” is stand for [15].

The main characteristic of NoSQL databases is their ability to operate in a distributed architecture, running on a cluster of commodity hardware. In NoSQL databases, there is almost no referential integrity constraint among data objects. Therefore, processing data residing in many different machines is feasible, and horizontal scalability is enabled by simply adding new processing and storage resources without replacing old ones. In addition, distributed storage architecture enables the migration of processes to data and data to processes, which facilitates big data analysis tasks.

An important problem with SQL databases solved by NoSQL databases is the data structure *impedance mismatch* [16]. The in-memory data can be kept in complex structure (e.g., nested lists), while with SQL databases, the data is always in a simple tabular format. This difference between the two stores (in-memory and database) causes the impedance mismatch and requires translation work upon data writing and reading to and from the database. For object-oriented programming language, it would be more favorable to replicate the data objects stored in memory directly into the database. For SQL databases, this problem is mitigated by the Object-Relational Mappers (ORMs) [17], where they take the responsibility to map data objects to their corresponding underlying database structure. With most NoSQL databases, the in-memory data structure can be stored as-is into the database, and this feature reduces programming overhead and enhances the performance.

NoSQL databases are schema-less, and they do not enforce data integrity constraints. That makes NoSQL databases more efficient because constraints checking and integrity validation upon data insertion are eliminated [18]. The distribution architecture of NoSQL databases makes them fault-tolerant because they are not prone to the single point of failure threat. Data replication across distributed storage nodes in NoSQL databases is easy because there is no obligation to the ACID properties. Alternatively, NoSQL databases adhere to the BASE properties (Basic Availability, Soft state, Eventual consistency) [12]. The basic availability property ensures that every request will get a response. However, consistency among responses is not guaranteed, and multiple users requesting the same data object can get different versions. The soft state property allows the database system to remain inconsistent after query execution. The eventual consistency property promises to propagate the changes to storage nodes until eventually the entire distributed database system becomes on a global consistent state [19].

NoSQL databases are considered ”non-relational” databases because their models are divergent from the traditional relational data model and implemented differently. NoSQL data models are categorized into Key-value, Document, Column Family, and Graph data models [20]. The next discussion for each model is mostly inspired from [15] [20], [21], and [22].

1) Key-Value Model: This is the simplest data model, where the data object is stored as a pair consisting of a key and a value. The key is a unique alpha-numeric identifier for the value. The value can be a string or complex lists and sets, with no constrains on its content structure. The structure of this

model is very similar to hash tables and dictionaries. In this model, the data can only be searched by key, i.e., the value is not searchable. Examples of key-value databases include Redis and Memcached.

The simplicity of this model makes it scalable and suitable for application that requires fast access to self-contained schema-less data. Examples of these data are user profiles, web sessions, shopping carts, and products information. On the other hand, the key-value model is not suitable if relationships exist between data objects, data is queried by its value, values are updated frequently, or for operating on multi-key transactions.

2) Document Model: This model can be considered an expansion to the key-value model, where the value contains semi-structured data and can be fully searched and indexed. Each data object is stored in a document that contains one or more keys. Groups of logically related documents are called collections, which are equivalent to tables in relational databases. To get the flexibility in accessing the data by its value, the database may store metadata that describes the allowable value structure and types. The database can retrieve part of the document based on the user query. The document can be formatted in a standard data exchange format such as XML, YAML, JSON, or BSON (Binary JSON). Examples of document databases include MongoDB and Couchbase.

The design of this model is inspired by a business software called Lotus Notes [23], a document database that enables sharing data across a local network [24]. Document databases use cases include storing and managing large-size collections of text files, such as literal documents, email messages, and XML files. Also, aggregated data objects such as products information or user profiles which are accessed at once together, are another use case for document databases. In general, document databases are best used for searchable data that has no fixed schema and which may add many nulls in an equivalent relational database. Document databases are not the best option for complex application queries or for transactions that require accessing multiple documents at once.

3) Column Family Model: Column family (or wide-column) model stores data objects in key-value pairs, where the value points to a second-level of key-value pairs. These second-level keys are called columns, and a subset of them forms a column family. The values can be accessed by any key in the first or second level.

One of the first column databases is BigTable [25], where it was designed to handle big data on a petabyte-scale. Another example of a column-family database is Cassandra [26], with a slightly different design philosophy that supports nested columns.

Column family databases may be the best choice with structured data when the distributed architecture is used, with data batch processing on a large scale, or real-time distributed big-data analysis tools such as MapReduce [27].

4) Graph Model: The graph model stores the data object as a graph consisting of nodes and connection edges. The nodes represent data objects while the edges represent relationships between them. Relationships are associated with properties, and two nodes can have one or more relationships. This model is schema-less, and nodes and edges can be inserted with any content. Examples of graph databases include Neo4j and JanusGraph.

This model is the only NoSQL model that supports relationships and ACID transactions. In fact, this model is closer to the relational model but categorized as NoSQL because of its dissimilarity with the relational model in how the data is structured and queried [28]. A key difference between the graph model and the relational model is in the query cost, where the navigation along the graph network to explore information is cheaper with graph database due to the absence of the expensive join operations. Another obvious difference is that the SQL query language is not supported in graph databases [28].

The graph representation of the data helps extract information that is hard to get with other models. The data of real-world problems that have interconnected entities, such as social networking, maps, products recommendations, pattern detection, network topologies, or any problem that can be represented as a graph, is a good candidate to be stored in a graph database. Nevertheless, graph databases are not good in horizontal scalability and big data processing.

V. Polyglot Persistence Principles

Due to the availability of many heterogeneous database systems, the decision of which database system should be used for a given application can be embarrassing. The concepts of polyglot persistence can be utilized in such cases. This section explains the meaning of polyglot persistence and spot the situations in which it is really needed.

A. What is Polyglot Persistence?

The term polyglot persistence was first coined in an online blog by Scott Leberknight in 2008 [29], and it then became famous after the book [15]. Leberknight explained the meaning of polyglot persistence by "*like polyglot programming, is all about choosing the right persistence option for the task at hand*". The term 'polyglot' implies the ability to talk to more than one database system. Polyglot persistence can be defined as *a situation in which different parts of data are stored in the most persistent database system that satisfies the storage requirements*.

Traditionally, relational databases were the default acceptable persistent option for data storage. However, the appearance of non-relational databases has changed the norm since there can be non-relational databases that are more persistent in some cases. The determination of the most persistent database system is totally dependent on the application's storage requirements.

A common example to illustrate the meaning of polyglot persistence is with an e-commerce application. In a typical e-commerce application, queries about clients' shopping data can be easily answered using a key-value NoSQL database. However, if the interest is on what the client's friends have purchased, the problem becomes entirely different. To answer this question, a graph database should be used [15]. Fig. 1 shows an example of a possible implementation for polyglot persistence in an e-commerce application.

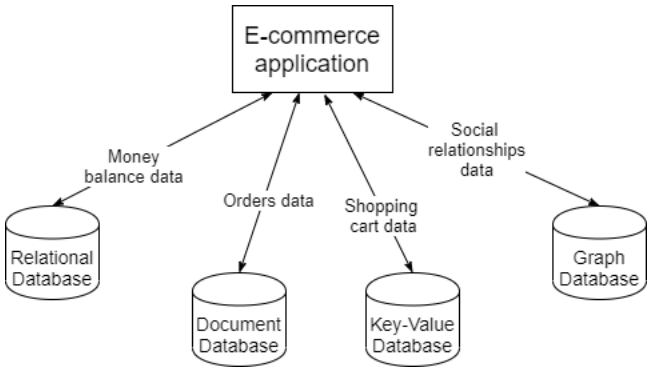

Fig. 1. Example of implementation for Polyglot Persistence in E-Commerce Application

Large and well-known applications are found to be employing polyglot persistence. Examples of these applications are Google, Facebook, Amazon, and Twitter. The practical aspects of polyglot persistence are not new for the industry, but they are not studied enough by the research community. There is a shortage of addressing the problems associated with polyglot persistence in the literature, especially with designing, implementing, and maintaining polyglot persistence architectures [3].

B. Why is Polyglot Persistence?

As shown in Section IV, each of SQL and NoSQL databases have their own characteristics and advantages. Polyglot persistence is employed to get the optimal benefits of each different database system. A usual concern is with database scalability, schema flexibility, data consistency, system availability, and performance [3].

In general, polyglot persistence is adopted to resolve conflicting requirements [3] [5]. These conflicts exist due to the limitations of the database systems, where no one database system can satisfy all requirements. With polyglot persistence, all requirements can be satisfied by using as many databases as needed. If there was a one-size-fits-all solution, as promised by NewSQL [30], then polyglot persistence can be overlooked.

Next, examples of three types of conflicting requirements: functional, non-functional, and data requirements, are discussed.

1) Conflicting Functional Requirements: The boundaries of database system functions can be determined by the available commands supported by its query language. Examples of these commands in SQL are SELECT and INSERT. The database functional requirements of an application can be determined by listing the commands it needs to perform on its data storage. A conflict in the functional requirements occurs when no single database system supports the entire set of query commands required by an application.

For example, consider a key-value database that only supports GET and PUT commands used by a simple web blog application. If the application added new features that require more complex queries, such as user authentication and online course registration features, then either the database system will be changed, or a new database system will be added beside the existing one. The latter polyglot persistence solution eliminates the need for a complex data migration process from the legacy database into the new one [31]. Note that the conflict in functional requirements can also be related to the database system security commands, such as the commands related to user authentication and access control [32].

2) Conflicting Non-Functional Requirements: The CAP theorem states that three demanding non-functional requirements cannot be satisfied at the same time when designing an application on a distributed architecture: Availability, Consistency, and Partition tolerance [33] [34] [35]. Therefore, there must be a trade-off for these requirements when selecting the database system for a given application. To resolve the conflict, different parts of application data can be stored in different databases.

Many non-functional requirements are subject to the ability of the database system to operate in a distributed architecture on commodity hardware. This ability reduces hardware resource costs, increases fault tolerance and availability, raises processing power, enables data replication, and eases big data analysis [36]. Since NoSQL databases can be deployed in a distributed architecture, they can be used to satisfy the mentioned requirements. On the other hand, other non-functional requirements cannot be accomplished unless the database system runs on a single server. Examples of these requirements are data consistency and integrity.

3) Conflicting Data Requirements: The data requirements for an application can also encourage the decision of using more than one database system. For example, a customer profile data (e.g., name, age, job, etc.) may not be designed in a fixed schema since many details can be null-able, and they may be frequently changed along the lifetime of the application development cycle. In addition, it might be impossible sometimes to design a fixed schema because the shape of the data is unknown in advance, as with the case when the data is inserted based on the user preferences. An example of such data object is salary, which contains many data items like basic salary, insurance allowance, transportation allowance, etc. In these cases, the usage of a schema-less non-relational database is recommended.

On the other hand, there are cases where using a relational database is the only possible option. An example of such a case is with money balance data that is used, for example, to purchase products or services within a web application. In this case, ACID transactions must be used to ensure data integrity and avoid race conditions [37].

Another example to illustrate the conflicting data requirements is the speed of data write or read operations. Some data have higher priority for reading speed over writing speed, such as product information, while other data might have higher priority for writing speed over reading speed, such as viewers counter for a product. To satisfy these conflicting requirements, different database systems might be selected for each part of the data.

VI. Polyglot Persistence Architectures

To implement polyglot persistence in an application, one can consider more than one architecture. According

to 18 reviewed studies that implement polyglot persistence, these architectures can be categorized into four categories: Application-coordinated Polyglot Persistence, Service-oriented Polyglot Persistence, Polyglot-Persistence-as-a-Service, and Multi-models Databases. Table I shows the four categories, the number of studies implemented each of them, and their references. A description for each of the categories is given in this section.

TABLE I. CATEGORIES OF POLYGLOT PERSISTENCE ARCHITECTURES

Category	Count	Reference
Application-coordinated Polyglot Persistence	6	[38], [21], [39], [40], [41], [42]
Service-oriented Polyglot Persistence	9	[22], [43], [44], [45], [46], [47], [48], [49], [50]
Polyglot-Persistence-as-a-Service	2	[51], [52]
Multi-models Databases	1	[53]

A. Application-Coordinated Polyglot Persistence

With this architecture, the application itself coordinates the polyglot persistence. This coordination requires the application to control the mapping of the data to databases, i.e., to have explicit knowledge about where each part of the data is stored. Typically, if the application is not small, it would be divided into modules [54] (aka packages or components). Each module is responsible for part of the application and has its own logic and functions. If the data managed by a module is specific to it (not shared by any other module), managing polyglot persistence would be simple because each module can have a different exclusive database system. However, usually, data application is shared by more than one module. In this case, many challenges to support polyglot persistence arise. Also, in some cases, a single module may have conflicting requirements that have to be satisfied using more than one database system.

To distinguish between these different cases and ease the discussion of the challenges of each of them, we classify the relationships between modules and databases within an application as follows:

- **One-to-One**: A module has a connection with one exclusive database.
- **One-to-Many**: A module has connections with more than one exclusive databases.
- **Many-to-One**: More than one module have connections with one mutual database.
- **Many-to-Many**: More than one module have connections with more than one mutual databases.

An application can have a combination of these relationships. The four relationships are shown in Figure 2, where 'M' stands for module and 'D' for database. The assumption is that different databases in the figure are of different storage models. Next, each of these relationships is discussed.

1) One-to-One Relationship: This is the simplest relationship. The module controls everything related to its data in one sole database. One query language can be used to manipulate the data. Thus, programmers need to learn only one query language. The application development will also

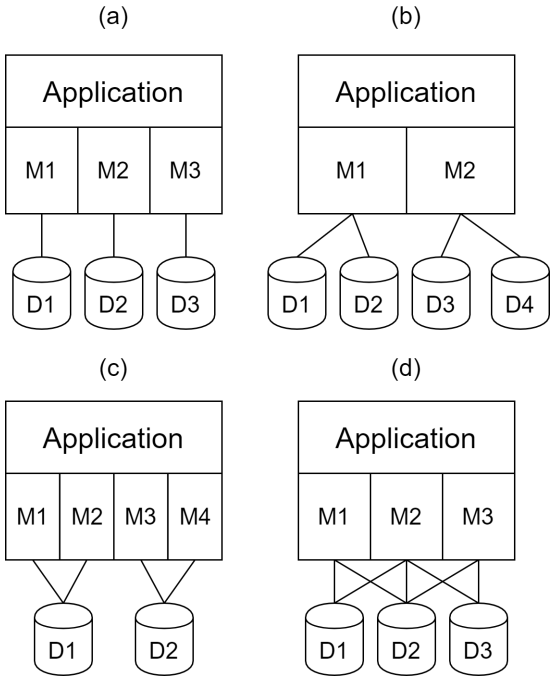

Fig. 2. Polyglot Relationships: (a) One-to-One, (b) One-to-Many, (c) Many-to-One, and (d) Many-to-Many

be easier because different programmers can work separately on different modules with knowledge about only one database system. A failure of one database will affect only part of the application and will not be propagated. Database configurations and security are taken care of by one module. A main concern here is with the design decision that will decide which database system is the best to satisfy the requirements for a given module. However, the approach proposed by [55] can ease the problem, where the functional, non-functional, and data requirements can be analyzed systemically to determine the most persistent database system.

2) One-to-Many Relationship: In this relationship, one module controls data stored by more than one database system, and that can cause several problems.

First, there will be a need to use more than one query language, one for each database system, which requires wider knowledge and longer training for developers. In addition, that may make the programming task more confusing. Possible mitigation to this problem can be by using a uniform query language for heterogeneous database systems [56], by a query mediator [57], or by translating SQL queries into NoSQL queries [58].

Second, cross-database consistency of dependent data objects stored in different databases needs to be managed by the module because there are no global referential integrity constraints enforced on the different databases. Consider Fig. 2(b). If dependency exists between a data object X on D1 and another data object Y on D2, then module M1 should maintain consistency across databases D1 and D2 by reflecting changes of X on Y and vice versa.

Third, running a query across different databases is not straightforward since data need to be processed and integrated

at the module, not at the database system, which may increase the query cost and require complicated data processing logic [59]. For example, a read query for a data object that is scattered on multiple databases may require several different sub-queries, one for each database, and the results of these sub-queries should be integrated and structured by the module. Data integration from different sources is studied in [60].

Forth, data redundancy might be a problem when different parts of the same data object are stored at different databases because common parts of the data object can be unnecessarily duplicated.

Fifth, in case of a failure of one database, this failure might be logically cascaded to other databases that are used by the same module in case a dependency exists between data objects.

3) Many-to-One Relationship: In this relationship, two or more modules use one mutual database. A main issue with this relationship is that the database configurations and security depend on more than one module, which may violate the Least Common Mechanism security design principle [61]. More on database security can be found in [62].

If all modules of the application are using the same database, then the polyglot persistence concepts are not applied. If this is not the case, then one can think of the modules that share the same database as one logical module, and the relationship becomes as if it were a one-to-one relationship. To simplify the control of a single database, a data manager (e.g., ORM) can be used as an intermediate layer between the modules and the database. It should control the database queries and configurations and mitigate security threats.

4) Many-to-Many Relationship: This relationship is the most complex, where each module uses at least two mutual databases. From the modules side, one can think of this relationship as a one-to-many. On the other hand, from the database side, one can think of this relationship as a many-to-one. Therefore, the same discussion of the two previous relationships can be said again here. However, the consistency problem is expended here because there will be a need to maintain cross-modules consistency for the data objects that are dependent on each other and stored in different databases, and are used by different modules. For example, in Fig. 2 (d), if a data object X on D1 is dependent on another data object Y on D2, and another data object Z on D3 is dependent on the same data object Y on D2, then we need to ensure consistency across modules M1 and M3 because they both share a common data object Y at D2.

A summary of the four relationships and their issues is given in Table II, where 'm' in the table header stands for 'many'. Note that the discussion here was at the module level, but it can be generalized to a larger programming unit, such as an entire application or even a set of applications, or smaller programming units, such as classes or methods.

TABLE II. ISSUES OF POLYGLOT PERSISTENCE RELATIONSHIPS

Issue	1-1	1-m	m-1	m-m
More than one query language might be needed	No	Yes	No	Yes
The module(s) need(s) to control cross-database consistency	No	Yes	No	Yes
Data integration might be required at the application side	No	Yes	No	Yes
The application should control cross-modules consistency	No	No	No	Yes
Data might be unnecessarily redundant	No	Yes	No	Yes
Database failure will be logically cascaded	No	Yes	No	Yes
Database configurations & security are dependent on more than one module	No	No	Yes	Yes

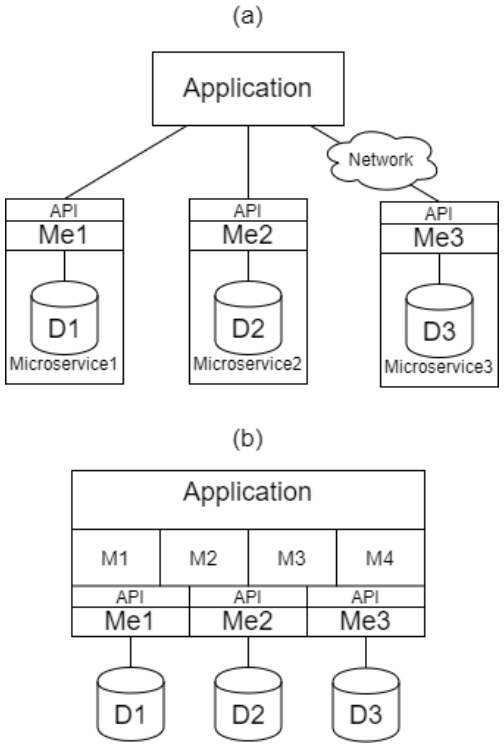

Fig. 3. Service-Oriented Architecture with (a) Microservices and (b) Modular Mediators

B. Service-Oriented Polyglot Persistence

If the database is being used by more than one module or application, then the database can be *decoupled* from the application to reduce the complexity. In this case, only one *mediator* will be able to access and control the database. This mediator is an independent module or a small application that offers an API for external users. The degree of decoupling this mediator can have different levels. In one extreme, the mediator will completely be independent from the application and can be deployed in a different machine, and it can even be programmed with a different programming language. The API in this case will be network calls (e.g., REST API [63]). In this extreme, the mediator is part of what is called a *microservice* [64]. In the other extreme, the mediator is part of the application, and it offers an API as public function calls to other modules, or even other applications.

Regardless of the detailed structure of this architecture, it can be seen as service-oriented architecture [65], where the database is wrapped with a software controller (which is called inhere a mediator). Illustrative examples of this architecture are shown in Figure 3, where 'M' stands for module, 'Me' for mediator, and 'D' for database. In Fig. 3(a), the network cloud implies the possibility for the entire microservice to be remotely accessed.

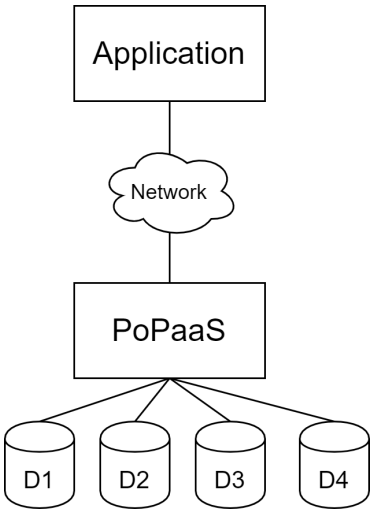

Fig. 4. An Illustrative Example for PoPaaS Architecture

A main advantage of this architecture is that different databases are accessed using similar APIs. The application is completely unaware about the underlying query languages of the databases. However, this architecture puts the load of managing consistency and cross-queries on the application modules. In case the relationships between the application and the databases are many-to-one or many-to-many, the same issues discussed in Section VI-A can be considered here.

Several strategies are proposed in the literature to manage polyglot persistence queries with microservices, and they are presented in [48].

C. Polyglot-Persistence-as-a-Service

In this architecture, the burden of managing polyglot persistence is conveyed to a completely different location, the cloud. The application here only provides the data requirements, and then these requirements should be satisfied by the Polyglot-Persistence-as-a-Service (PoPaaS) provider. The provider should automatically specify the appropriate database system for each segment of the data, based on its requirements, and then provide an API for data access. Such an API design strategy is proposed by [51].

The problem with this architecture is how the client can formulate the requirements in a standard format? Another problem is the selection of the appropriate database system for the given requirements, which should be automated based on quantifiable metrics [3]. A possible solution is to use an automated rule-based data model selection technique, as proposed by [52]. In reality, we do not know an example of such a service. An illustrative example of this architecture is shown in Fig. 4.

D. Multi-Model Databases

Instead of dealing with the complexity of managing multiple databases, one possible solution is to use a database that supports multiple data models, which is called a multi-model database. In this case, the application will manage a single database that fulfills its requirements. Multi-model database engines can be designed to manage a combined data model that has the features of several data models. OrientDB and ArangoDB are two examples of such databases [66]. They support document, graph, and key-value data models in one database instance. OrientDB has a query language that is very similar to SQL, while ArangoDB has a new language called ArangoDB Query Language (AQL), which is similar to an extent to SQL.

A new paradigm to support more than one model is with the Flexible Schema Data Management (FSDM) [67]. This paradigm integrates the JSON data model into SQL databases. The stored JSON data is storable, indexable, and queryable, without the need for upfront schema definition. This support will reduce the use cases where polyglot persistence is needed because there will be no need to use NoSQL databases to store schema-less data. PostgreSQL and Oracle databases are examples of such database systems that support this feature [68].

Despite the support for multiple data models in those databases, they do not eliminate the demand for polyglot persistence because there are still non-functional requirements that are not satisfied, such as scalability and performance [69]. This emphasizes the fact that satisfying all polyglot persistence requirements in one database system is an engineering challenge [3], and apparently, having a one-size-fits-all solution is not yet feasible.

VII. Polyglot Persistence Implementation Strategy

In this section, we propose an implementation strategy that can aid the development of polyglot persistence applications.

A. Step 1: Database Requirements

The first thing to start with is the requirements. The correctness of the requirements should be ensured because the following steps will be dependent on them. Database requirements must be consistent with the application requirements, and they should include functional, non-functional, and data requirements. Database functional requirements should include the queries that will be used to manage application data. Examples of non-functional requirements include consistency, integrity, and availability. One of the most important parts of data requirements is the conceptual data model, which should be created at this step using Unified Modeling Language (UML) [70] or Entity-Relationship (ER) [71] diagrams, for example.

B. Step 2: Database Selection

Based on the gathered requirements, the database system should be selected. At this step, conflicts between database requirements should be identified and resolved by using as many database systems as needed. If there was no conflict, the implementation will normally proceed using one database system without considering polyglot persistence. Otherwise, different database systems should be selected carefully, considering the most persistent database system for each part of application data. A clear mapping between each part of the data and the selected database systems should be preserved. Note that steps 1 and 2 can be accomplished using the systematic approach proposed in [55].

C. Step 3: Database-Specific Data Models

Conceptual data models for each database system should be derived from the general conceptual data model identified in step 1, considering the process proposed by [72]. Since different database systems use different storage models, each of them may require a specific data model to determine database schema. At this step, using a database-independent schema declaration language can be helpful [73].

D. Step 4: Polyglot Persistence Architecture

In this step, the architecture in which the application will be implemented should be designed. The architecture determines how the application will be talking to the different selected databases. The architecture design should consider implementation cost, time, and resources. The discussion on polyglot persistence architectures given in Section VI can guide this step.

E. Step 5: Issues Identification

After selecting the architecture, polyglot persistence issues associated with the selected architecture should be identified. After identifying the architecture issues, a clear plan on how they will be resolved should be made. This plan should include the technical details on how each issue will be addressed. Again, the issues and their resolutions can be inspired by the discussion in Section VI.

F. Step 6: Application Development

This is the last step in which the actual implementation for the application and its infrastructure should start based on the results of the previous steps.

VIII. Conclusion

Large applications may require more than one database system to satisfy their requirements. This environment that operates on multiple databases is called a polyglot persistence environment. The polyglot persistence environment is fraught with many challenges and problems. This paper presented classification of database systems with details about their features. Polyglot persistence principles, its possible architectures, and the issues related to each architecture are identified. A polyglot persistence implementation strategy is proposed in light of the study outcomes.

The authors believe that this work has clarified most of the concepts related to polyglot persistence. This work can be a helpful reference for solving polyglot persistence problems.

Future research can further propose solutions to issues introduced by polyglot persistence. A protocol for inter-database negotiations might be devised to support interoperability between different database systems as a means to address issues related to polyglot persistence. Furthermore, future research can study how database functional, non-functional, and data requirements can be reported standardly. In addition, it can study the possibility of having an approach for representing and analyzing database requirements that can automatically determine the needed database systems.

Acknowledgment

The authors would like to thank King Fahd University of Petroleum and Minerals for the support and facilities provided to perform this research.

References

[1] (2022) Db-engines ranking. Https://db-engines.com/en/ranking (accessed: 2022-03-02).

[2] F. Gessert and N. Ritter, "Scalable data management: Nosql data stores in research and practice," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 1420–1423.

[3] F. Gessert, W. Wingerath, and N. Ritter, "Polyglot persistence in data management," in *Fast and Scalable Cloud Data Management*. Springer, 2020, pp. 149–174.

[4] P. P. Khine and Z. Wang, "A review of polyglot persistence in the big data world," *Information*, vol. 10, no. 4, p. 141, 2019.

[5] L. Wiese, "Polyglot database architectures= polyglot challenges." in *LWA*, 2015, pp. 422–426.

[6] M. K. Bavirisetty. (2015) Polyglot processing - an introduction 1.0. Https://www.slideshare.net/MohanBavirisetty/polyglot-processing-an-introduction-10 (accessed: 2022-03-02).

[7] J. Pokorný, "Integration of relational and nosql databases," *Vietnam Journal of Computer Science*, vol. 6, no. 04, pp. 389–405, 2019.

[8] E. Codd, "A relational model of data for large relational databases," *Communications of the ACM*, vol. 13, no. 6, pp. 77–87, 1970.

[9] E. F. Codd, *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[10] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM computing surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983.

[11] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley Reading, 1987, vol. 370.

[12] N. Banothu, S. Bhukya, and K. V. Sharma, "Big-data: Acid versus base for database transactions," in *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*. IEEE, 2016, pp. 3704–3709.

[13] J. L. Harrington, *Relational database design and implementation*. Morgan Kaufmann, 2016.

[14] C. A. Győrödi, D. V. Dumşe-Burescu, D. R. Zmaranda, R. Ş. Győrödi, G. A. Gabor, and G. D. Pecherle, "Performance analysis of nosql and relational databases with couchdb and mysql for application's data storage," *Applied Sciences*, vol. 10, no. 23, p. 8524, 2020.

[15] P. J. Sadalage and M. Fowler, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2013.

[16] S. Ambler, *Agile database techniques: Effective strategies for the agile software developer*. John Wiley & Sons, 2012.

[17] J. M. Barnes, "Object-relational mapping as a persistence mechanism for object-oriented applications," 2007.

[18] B. Jose and S. Abraham, "Performance analysis of nosql and relational databases with mongodb and mysql," *Materials today: PROCEEDINGS*, vol. 24, pp. 2036–2043, 2020.

[19] M. Diogo, B. Cabral, and J. Bernardino, "Consistency models of nosql databases," *Future Internet*, vol. 11, no. 2, p. 43, 2019.

[20] A. Moniruzzaman and S. A. Hossain, "Nosql database: New era of databases for big data analytics-classification, characteristics and comparison," *International Journal of Database Theory and Application*, vol. 6, no. 4, 2013.

[21] K. Srivastava and N. Shekokar, "A polyglot persistence approach for e-commerce business model," in *2016 International Conference on Information Science (ICIS)*. IEEE, 2016, pp. 7–11.

[22] C. Shah, K. Srivastava, and N. Shekokar, "A novel polyglot data mapper for an e-commerce business model," in *2016 IEEE Conference on e-Learning, e-Management and e-Services (IC3e)*. IEEE, 2016, pp. 40–45.

[23] K. Moore, "The lotus notes storage system," *ACM SIGMOD Record*, vol. 24, no. 2, pp. 427–428, 1995.

[24] L. Kawell Jr, S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif, "Replicated document management in a group communication system," in *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, 1988, p. 395.

[25] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[26] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[27] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[28] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *Proceedings of the 48th annual Southeast regional conference*, 2010, pp. 1–6.

[29] S. Leberknight. (2008) Polyglot persistence. Http://www.sleberknight.com/blog/sleberkn/entry/polyglot_persistence (accessed: 2022-03-02).

[30] A. Pavlo and M. Aslett, "What's really new with newsql?" *ACM Sigmod Record*, vol. 45, no. 2, pp. 45–55, 2016.

[31] S. Velimeneti, "Data migration from legacy systems to modern database," 2016.

[32] G. D. Samaraweera and J. M. Chang, "Security and privacy implications on database systems in big data era: a survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 1, pp. 239–258, 2019.

[33] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[34] E. Brewer, "A certain freedom: thoughts on the cap theorem," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, 2010, pp. 335–335.

[35] S. Gilbert and N. Lynch, "Perspectives on the cap theorem," *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[36] S. Venkatraman, K. Fahd, S. Kaspi, and R. Venkatraman, "Sql versus nosql movement with big data analytics," *Int. J. Inform. Technol. Comput. Sci*, vol. 8, pp. 59–66, 2016.

[37] R. Paleari, D. Marrone, D. Bruschi, and M. Monga, "On race vulnerabilities in web applications," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 126–142.

[38] C. Zdepski, T. A. Bini, S. N. Matos, and S. Hammoudi, "An approach for modeling polyglot persistence," in *ICEIS (1)*, 2018, pp. 120–126.

[39] S. Prasad and M. N. Sha, "Nextgen data persistence pattern in healthcare: polyglot persistence," in *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. IEEE, 2013, pp. 1–8.

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

[41] S. Prasad and S. Avinash, "Application of polyglot persistence to enhance performance of the energy data management systems," in *2014 International Conference on Advances in Electronics Computers and Communications*. IEEE, 2014, pp. 1–6.

[42] S. Nadkarni, A. Kadakia, and K. Shrivastava, "Providing scalability to data layer using a novel polyglot persistence approach," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, 2018, pp. 1–5.

[43] K. Trivedi, S. Shah, and K. Srivastava, "An efficient e-commerce design by implementing a novel data mapper for polyglot persistence," in *Advanced Computing Technologies and Applications*. Springer, 2020, pp. 149–156.

[44] L. H. Z. Santana and R. dos Santos Mello, "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)*. IEEE, 2019, pp. 237–244.

[45] K. Kaur and R. Rani, "A smart polyglot solution for big data in healthcare," *IT Professional*, vol. 17, no. 6, pp. 48–55, 2015.

[46] ——, "Managing data in healthcare information systems: many models, one solution," *Computer*, vol. 48, no. 3, pp. 52–59, 2015.

[47] R. Jiménez-Peris, M. Patiño-Martinez, I. Brondino, and V. Vianello, "Transactional processing for polyglot persistence," in *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. IEEE, 2016, pp. 150–152.

[48] L. H. Villaça, L. G. Azevedo, and F. Baião, "Query strategies on polyglot persistence in microservices," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 1725–1732.

[49] H. Singhal, A. Saxena, N. Mittal, C. Dabas, and P. Kaur, "Polyglot persistence for microservices-based applications," *International Journal of Information Technologies and Systems Approach (IJITSA)*, vol. 14, no. 1, pp. 17–32, 2021.

[50] L. G. Azevedo, R. d. S. Ferreira, V. T. d. Silva, M. de Bayser, E. F. d. S. Soares, and R. M. Thiago, "Geological data access on a polyglot database using a service architecture," in *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*, 2019, pp. 103–112.

[51] F. Gessert, S. Friedrich, W. Wingerath, M. Schaarschmidt, and N. Ritter, "Towards a scalable and unified rest api for cloud data stores," in *GI-Jahrestagung*, 2014, pp. 723–734.

[52] M. Schaarschmidt, F. Gessert, and N. Ritter, "Towards automated polyglot persistence," *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, 2015.

[53] I. Košmerl, K. Rabuzin, and M. Šestak, "Multi-model databases-introducing polyglot persistence in the big data world," in *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, 2020, pp. 1724–1729.

[54] J. B. Dennis, "Modularity," in *Software Engineering*. Springer, 1975, pp. 128–182.

[55] N. Roy-Hubara, P. Shoval, and A. Sturm, "Selecting databases for polyglot persistence applications," *Data & Knowledge Engineering*, vol. 137, p. 101950, 2022.

[56] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, and J. Pereira, "Cloudmdsql: querying heterogeneous cloud data stores with a common language," *Distributed and parallel databases*, vol. 34, no. 4, pp. 463–503, 2016.

[57] R. Sellami and B. Defude, "Complex queries optimization and evaluation over relational and nosql data stores in cloud environments," *IEEE transactions on big data*, vol. 4, no. 2, pp. 217–230, 2017.

[58] J. Rith, P. S. Lehmayr, and K. Meyer-Wegener, "Speaking in tongues: Sql access to nosql systems," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 855–857.

[59] G. C. Deka, "Nosql polyglot persistence," in *Advances in Computers*. Elsevier, 2018, vol. 109, pp. 357–390.

[60] M. Lenzerini, "Data integration: A theoretical perspective," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2002, pp. 233–246.

[61] T. V. Benzel, C. E. Irvine, T. E. Levin, G. Bhaskara, T. D. Nguyen, and P. C. Clark, "Design principles for security," NAVAL POSTGRADUATE SCHOOL MONTEREY CA DEPT OF COMPUTER SCIENCE, Tech. Rep., 2005.

[62] N. Ron Ben, *Implementing Database Security and Auditing*. Elsevier, 2005.

[63] V. Surwase, "Rest api modeling languages-a developer's perspective," *Int. J. Sci. Technol. Eng*, vol. 2, no. 10, pp. 634–637, 2016.

[64] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[65] R. Dörbecker and T. Böhmann, "The concept and effects of service modularity–a literature review," in *2013 46th Hawaii International Conference on System Sciences*. IEEE, 2013, pp. 1357–1366.

[66] E. Płuciennik and K. Zgorzałek, "The multi-model databases - a review," in *International Conference: Beyond Databases, Architectures and Structures*. Springer, 2017, pp. 141–152.

[67] Z. H. Liu, B. Hammerschmidt, D. McMahon, Y. Liu, and H. J. Chang, "Closing the functional and performance gap between sql and nosql," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 227–238.

[68] D. Petković, "Json integration in relational database systems," *Int J Comput Appl*, vol. 168, no. 5, pp. 14–19, 2017.

[69] F. R. Oliveira and L. del Val Cura, "Performance evaluation of nosql multi-model data stores in polyglot persistence applications," in *Proceedings of the 20th International Database Engineering & Applications Symposium*, 2016, pp. 230–235.

[70] E. Naiburg, E. J. Naiburg, and R. A. Maksimchuck, *UML for database design*. Addison-Wesley Professional, 2001.

[71] T. J. Teorey, *Database modeling and design: The entity-relationship approach*. Morgan Kaufmann Publishers Inc., 1990.

[72] M. Kolonko and S. Müllenbach, "Polyglot persistence in conceptual modeling for information analysis," in *2020 10th International Conference on Advanced Computer Information Technologies (ACIT)*. IEEE, 2020, pp. 590–594.

[73] A. H. Chillón, D. S. Ruiz, and J. G. Molina, "Athena: A database-independent schema definition language," in *International Conference on Conceptual Modeling*. Springer, 2021, pp. 33–42.