Cause and Solution of Object-Relational Impedance Mismatch

Ken Wang [0009-0002-3064-3421]

BackLogic LLC, Chino Hills, CA 91709, USA

Abstract. Object-relational impedance mismatch is still an unresolved problem. As a common solution to the problem, ORM has many issues and is deemed acceptable on the basis that there is no better alternative. However, it is not an appropriate solution that addresses the cause of object-relational impedance mismatch.

In this paper, a sample of mismatch problems and a list of consolidated problem themes are analyzed. The cause of object-relational impedance mismatch is determined to be a) the data structure mismatch between the object and relational data models; and b) the language impedance between the embedded SQL and host programming language.

The solution to the first issue is to abstract away the object-relational transformation problem from developers. The solution to the second issue is to abstract away the host programming language from SQL. Based on these findings and an analysis of basic data access patterns, a group of three relational database access services: query, command, and repository, is proposed as an appropriate solution to the problem of object-relational impedance mismatch. The feasibility and advantages of the proposed solution are discussed.

Keywords: object-relational impedance mismatch, object-relational persistence, ORM, data access service, query, command, repository.

1 Introduction

Object-relational impedance mismatch occurs when an application uses a relational database for data persistence. It is a term used to refer to the difference between the object schema of the application and the relational schema of the relational database and to the difficulty of moving data between the two. This problem is important in practice not only because it is ubiquitous, but also because it costs significant time and effort to address.

Object-relational impedance mismatch is a wicked and complex problem [1]. First, it is not a single problem but a collection of interrelated problems, or a quagmire of issues as it has been referred to [1][5]. Second, the cause of object-relational impedance mismatch is not well-understood. The solutions are developed only to address the symptoms, and are specific to an object-relational application, Java application for example, and involve comprises either in the design of application or in the design of the database [1].

Object-relational mapping (ORM) is a common solution to the impedance mismatch problem. However, in contrast to the general perception reinforced by ORM technologies such as Hibernate, Entity Framework, SQLAlchemy and Sequelize etc., the problem is not resolved by ORM [1]. There are many criticisms and debates about ORM frameworks [5][6], and the agreed issues [6] at least include:

- They are complex. This complexity implies a grueling learning curve.
- They provide only a leaky abstraction over a relational data store.
- Systems using an ORM often perform badly - due to naive interactions with the underlying database.
- The ORM can handle about 80-90% of the mapping problems, but that last chunk always needs careful work by somebody who really understands how a relational database works.
- People are often forced to compromise their object model to make it more relational in order to please the ORM.

To summarize, ORM is deemed an acceptable solution only on the term that it is helpful in certain percentage of (simple) cases and that there is not yet a better alternative. However, it is not an appropriate solution (a term used by Ireland & Bowers [1]) that addresses the cause of object-relational impedance mismatch. An appropriate solution should not only leave no unaddressed issues, but also be an improvement over ORM, that is:

- Simple in concept. This simplicity would imply an easy learning curve.
- Scalable in both performance and development effort, as the query and object structure become complex.
- flexible, so that developers are no longer forced to comprise their object model.

The purpose of this research is to identify the cause of object-relational impedance mismatch and find a path to an appropriate solution. Thus, a sample of mismatch problems and a list of problem themes compiled by Ireland & Bowers [1] are reexamined. The cause of object-relational impedance mismatch is analyzed and determined to be: a) the data structure mismatch between the object and relational data models; and b) the language impedance between the embedded SQL and host programming language.

The solution to the first issue is to abstract away the object-relational transformation problem from developers. The solution to the second issue is to abstract away the host programming language from SQL. These ideas, along with an analysis of basic data access patterns, lead to the concept of relational database access services, a group of three componentized SQL services: query, command, and repository, as an appropriate solution to the problem of object-relational impedance mismatch.

2 Problems of Object-Relational Impedance Mismatch

Object-relational impedance mismatch encompasses a collection of interrelated problems, each referred to as a mismatch, which may represent either a symptom or cause

of object-relational impedance mismatch. A sample list of these mismatches is sourced from Hibernate [7], as shown in Table 1.

Table 1. Sample Mismatch Problems [7]

Problem	Concern
Granularity	Sometimes you will have an object model which has more classes than the number of corresponding tables in the database (we says the object model is more granular than the relational model).
Subtypes (inheritance)	Inheritance is a natural paradigm in object-oriented programming languages. However, RDBMSs do not define anything similar on the whole (yes some databases do have subtype support but it is completely non-standardized)
Identity	A RDBMS defines exactly one notion of 'sameness': the primary key. Java, however, defines both object identity a==b and object equality a.equals(b).
Associations	Associations are represented as unidirectional references in Object Oriented languages whereas RDBMSs use the notion of foreign keys. If you need bidirectional relationships in Java, you must define the association twice. Likewise, you cannot determine the multiplicity of a relationship by looking at the object domain model.
Data navigation	The way you access data in Java is fundamentally different than the way you do it in a relational database. In Java, you navigate from one association to an other walking the object network. This is not an efficient way of retrieving data from a relational database. You typically want to minimize the number of SQL queries and thus load several entities via JOINs and select the targeted entities before you start walking the object network.

Copeland & Maier [3], Ambler [4] and Neward [5] have each characterized object-relational impedance mismatch differently, but Ireland & Bowers [1] has consolidated their works, along with contributions from other, into a catalog of six problem themes, as shown in Table 2.

Each problem theme in Table 2 is a collection of mismatches reflecting a particular characterization and representing one aspect of object-relational impedance mismatch that needs to be addressed by a solution.

Notably, many of the mismatch problems listed in Tables 1 and 2, including inheritance, association, data navigation, identity, process model and encapsulation, are either differences between an object-oriented program and a relational database system or concerns specific to the ORM solution. These mismatches are more likely symptoms than causes of object-relational impedance mismatch.

The schema ownership and instance problem themes in Table 2 are two important decisions to be made for a solution. For an appropriate solution, the answer to the schema ownership problem should be that each team is responsible for their own

model, that is, the application team is responsible for the object model and the database team for the data model, to fully decouple the object and data models and allow them to evolve independently.

Table 2. Problem Themes [1]

Problem Theme	Concern
Structure	The structure problem theme is concerned with any difference of data structure between the schema of an object-oriented program and the schema of a relational database, and so adopts a broad interpretation of the notion of structure. The essence of a structure problem is the extent to which an object-oriented data structure can be, and should be, described by a relational data structure. Problems of the structure theme are important because they are concerned with a description of the data processed by an object-relational application.
Instance	The essence of an instance problem theme is, where is the canonical copy of state located? Problems of the instance theme are important because they are concerned with the ownership of and the responsibility for data.
Encapsulation	The principle of encapsulation requires that the state of an object can be determined only by its behavior, so in an object-oriented program the value of an attribute of an object is accessed via a method. Problems of encapsulation are important because, in a database, the value of a column in a row has no such protection. Consequently, once stored in a database, data may be changed without the protection of the semantics encoded in a method.
Identity	The essence of an identity problem is how to identify uniquely a collection of data values between both object-oriented program and a relational database. Such problems of identity are important to ensure the integrity of data between an object-oriented program and a relational database.
Processing Model	The essence of a processing model problem is how to represent in, maintain and retrieve from a database a sufficient set of objects for processing. Such problems are important because they concern issues of software performance
Schema Ownership	The essence of the schema ownership problem is that the team who design and implement an object-oriented program can be different from the team who design and implement a relational database. Such problems are important because they concern the choices made by those responsible, respectively, for the object-oriented program and the relational database.

For an appropriate solution, the answer to the instance problem in Table 2 should be treating the database copy as the canonical copy of state. This straightforward de-

cision not only is sensible, given that modern application is mostly stateless and that state only matters if it is persisted, but also simplifies considerations of the solution.

The structure problem theme in Table 2 concerns the disparity between object and relational data structures. This is a fundamental issue of object-relational impedance mismatch that all solutions must address. This problem theme also concerns the degree of the disparity or how much the object structure can deviate from the relational structure, which determines how much flexibility developers have in their object design in order to meet the application's data needs. The answer to this second aspect should be as much as developers realistically need, for an appropriate solution of object-relational impedance mismatch.

The analysis leading to Table 1 and Table 2 focuses exclusively on the difference between an object-oriented program and a relational system. However, the term object-relational impedance mismatch also refers to the difficulty of moving data between the two. From this perspective, the transformation between object and relational data structures is one issue; while working with SQL embedded in a programming language is another challenge. Therefore, an additional problem theme, SQL handling, might need to be added to Table 2.

3 Cause of Object-Relational Impedance Mismatch

Looking at the problems and problem themes in Table 1 and Table 2, it is not easy to understand the causes of object-relational impedance mismatch except for one, the disparity between the object and relational data structures, which complicates transformation between object and relational data structures. This disparity is the concern of the structure problem theme in Table 2 and also the reason behind the processing model problems in Table 2. We can refer to this issue as data structure mismatch.

The cause for all other mismatches is harder to qualify. However, we know that these mismatches occur only when technologies based on the concepts of both “object” and “relation” are combined into an object-relational application [1]. This combination takes place when SQL is embedded into the host programming language, either explicitly as native SQL or implicitly as ORM mapping. Therefore, the second cause of object-relational impedance mismatch is due to embedding of SQL in a programming language. We can refer to this second issue as language impedance, as it represents the friction between the embedded SQL and host language or the resistance of the host language to embedded SQL.

Thus, the cause of object-relational impedance mismatch comprises two orthogonal issues:

- the data structure mismatch between the object and relational data structures, and
- the language impedance between the embedded SQL and host language.

3.1 Data Structure Mismatch

The problem of data structure mismatch is an inherent problem of object-relational persistence. To read data into an object, a relational data set needs to be somehow transformed into the object structure. To write data back to database, the object needs to be transformed into one or more relational data sets. Transforming object into data sets is tedious but not difficult. However, transforming a data set into an object is hard or impossible at times, even for a skilled developer, when the object is complex. In such case, the so-called N+1 query technique is the only option, which not only is laborious to code but also has performance impact on the system.

The solution to this problem is to abstract away the object-relational transformation problem from the developer and have it handled by the library or framework. This is in fact what ORM frameworks do, except they work well only for relatively simple objects. When the object becomes complex, they opt to the N+1 query technique, which is one of reasons why they often perform poorly.

A appropriate solution should include an object-relational transformation engine that can efficiently transform a data set into a complex object structure and vice versa.

3.2 Language Impedance

The intensity of language impedance varies with the host language. As expected, it is strongest with an object-oriented language like Java, as evidenced by a long list of mismatches; moderate with a functional programming language, as some of the problems such as encapsulation and inheritance are no longer concerns; and minimum with stored procedure, which embeds SQL natively.

Since the language impedance stems from embedding of SQL in a host programming language, the solution to this problem is naturally to separate SQL from the host programming language, in other words, to abstract away the host language from SQL.

The implication of this approach is that we must find a way to build the data access layer with SQL only. For this, we need to analyze the basic data access patterns. It also implies that the data access layer must be physically separated from the application and be run as data access services on a data access platform.

4 Data Access Patterns

In modern applications, data access logics are exclusively developed with the data access object (DAO) and repository design patterns. Therefore, to find out the basic data access pattern, it is sufficient just to study these two design patterns.

There is a misconception about DAO that DAO is only for row-level CRUD operations. In reality, DAO is about separation of data access interface and implementation, to encapsulate data access details from application [9]. It should be seen as a general-purpose design pattern for all data access needs. Instead, the repository design pattern can be seen as a special case of the general DAO pattern, for CRUD operations of aggregate objects.

If we examine the data access methods of a DAO, we can see that they are either a query method to read data from database or a command method that writes data back to database or otherwise manipulates the data in the database. Combining the DAO and repository design patterns together, we can easily identify three basic data access patterns, as:

- query, for retrieving data from database
- command, for write or change data in database, and
- repository, for CRUD operation of aggregate objects.

In practice, the repository access pattern is mostly for read and write of domain objects. The query pattern is often used for retrieving ad hoc data transfer objects (DTOs); while the command pattern is often for ad hoc data change. A data access solution needs and only needs to support these three data access patterns.

5 Cause of Object-Relational Impedance Mismatch

Based on the analysis in the previous sections, we propose a group of three relational data access services, namely

- query service, for ad hoc data retrieval;
- command service, for ad hoc data writing and change; and
- repository services, for read and write of domain aggregates

as an appropriate solution to the problem of object-relational impedance mismatch.

These componentized services are pure relational, that is, they are composed of only SQL and JSON files and to be developed with only SQL and JSON, as described in the following.

5.1 Query Service

Query service is for retrieving data from a data source. It takes an input object of query parameters and returns an object or an array of objects as output.

Fig.1 illustrates the structure of the query service *getCustomersByCity*:

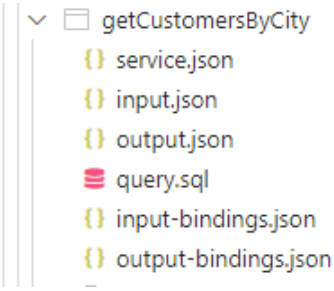

Fig. 1. Structure of query service

Table 3 lists the JSON and SQL files composing the query service. The *service.json* file is not a component of the service, but contains the meta data of the service.

Table 3. List of query files

File	Description
input.json	json file for service input
output.json	json file for service output
query.sql	sql file for service query
input-bindings.json	json file for input bindings that map query parameters to input data fields
output-bindings.json	json file for output bindings that map output data fields to query columns

The contents of the files are given below for illustration:

Input:

```
{
  "city": "Los Angeles"
}
```

Output:

```
[{
    "customerNumber": 1,
    "customerName": "Land of Toys Inc.",
    "address": {
        "addr": "1023 Grand Ave.",
        "city": "Los Angeles",
        "state": "CA"
    }
}]
```

Query:

```
select customerNumber, customerName, address, city, state
  from customers
 where city = :city
```

Input-Bindings:

```
[
  {"field": city", "column": ".city"}
]
```

Output-Bindings:

```
[
   {"field": "..customerNumber", "column": "customerNumber"},
   {"field": "..customerName", "column": "customerName"},
```

```
    {"field": "..address.addr", "column": "address"},
    {"field": "..address.city", "column": "city"}
    {"field": "..address.state", "column": "state"}
]
```

Given a city name, this exemplary query service returns a list of nested customer objects. The input component specifies the input parameters of the query service; the output component specifies the structure and data fields of the output object or array; while the data access logic is implemented with the single parameterized SELECT statement in the query file.

The query service is to be executed by a service engine, which will

- Bind the input parameters to the query using the input-bindings;
- Execute the query to get a relational data set; and
- Transform the data set into the output object or array using the output-bindings.

The input, output and query are the essential elements of the query service. The input-bindings and output-bindings can be generated from them at development time.

Assuming the service engine is capable of transforming a relational data set to an unrestrictedly complex object or array, then we have obtained a way to freely retrieve objects of any shape from a relational database, with just SQL.

This query service is potentially very efficient, as the N+1 query technique is completely ruled out and the SQL is in the control of the developer. This query service is also potentially very flexible, as the same data model can be linked to any and many object models, with different SQLs.

5.2 Command Service

The command service is for writing and changing data in data source. The basic command service takes an input object of SQL parameters, update the data in the data source, but returns nothing. The advanced version appends a query to the command.

Fig.2 illustrates the structure of the basic command service *cloneProductLine*.

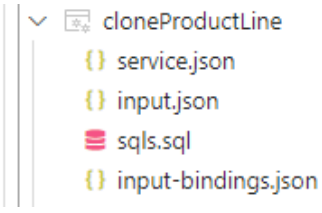

Fig. 2. Structure of command service

Table 4 lists the JSON and SQL files composing the command service:

Table 4. List of command files

File	Description
input.json	json file for service input
sqls.json	sql file for service DML statements
input-bindings.json	json file for input bindings that map sql parameters to input data fields

The contents of the files are given below for illustration:

Input:

```
{
   "sourceProductLineName": "Classic Cars",
   "targetProductLineName": "Cloned Classic Cars",
   "productCodePrefix": "N-"
}
```

Sqls:

```
Insert into productlines (
    productLine, textDescription
)
select :targetProductLineName, textDescription
  from productlines
 where productLine = :sourceProductLineName
;

Insert into products (
     productCode, productName, productLine,
     productScale, productVendor, productDescription,
     quantityInStock, buyPrice, MSRP
)
select concat(:productCodePrefix, productCode),
       productName, :targetProductLineName,
       productScale, productVendor, productDescription,
       0, buyPrice, MSRP
  from products
 where productLine = :sourceProductLineName
```

Input-Bindings:

```
[
  {"parameter": "targetProductLineName",
         "field": ".targetProductLineName"},
  {"parameter": "productCodePrefix",
       "field": ".productCodePrefix"},
```

```
  {"parameter": "sourceProductLineName",
         "field": ".sourceProductLineName"}
]
```

Given the source product line name, this exemplary command service clones the source product line for a target product line of given name and product code prefix. The input component specifies the command parameters. The data access logic is implemented with a sequence of parameterized DML statements in the sqls file.

The command service is to be executed by a service engine, which will

- bind the input parameters to the DML statements in the sqls file using the input-bindings; and
- execute the DML statements in the order that they appear in the file.

The input and sqls are the essential elements of the query service. The input-bindings can be generated from them at development time. The command service is a very efficient way to make ad hoc data changes inside the database.

5.3 Repository Service

Repository service is for CRUD operations of aggregate object. It includes both read operations for querying objects and write operations for creating, updating, and deleting the object.

Fig.3 illustrates the structure of the repository service *Order*.

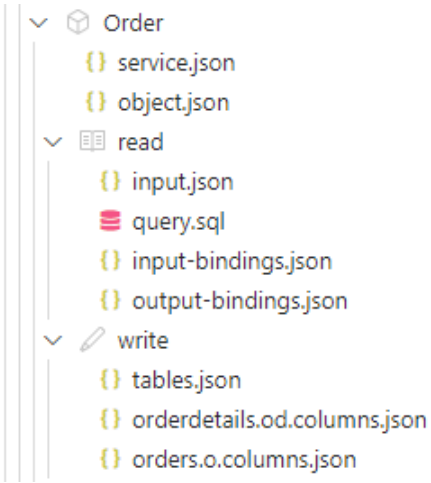

Fig. 3. Structure of repository service

Table 5 lists the JSON and SQL files composing the repository service.

Table 5. List of repository files

Component	File	Description
object	object.json	json file for service object
read	input.json	json file for read input
	query.sql	sql file for read query
	input-bindings.json	json file for input bindings that map query parameters to input data fields
	output-bindings.json	json file for output bindings that map object data fields to query columns
write	tables.json	json file for table bindings that map object structures to tables
	orders.columns.json	json file for column bindings that maps object data fields to table columns.
	orderdetails.columns.json	Same as above. One file per table.

The contents of the files are given below for illustration:

Object:

```
{
    "orderNumber": 1,
    "orderDate": "2020-12-10T00:00:00.000",
    "customerNumber": 103,
    "customerName": "John",
    "requiredDate": "2020-12-10T00:00:00.000",
    "shippedDate": "2020-12-10T00:00:00.000",
    "status": "shipped",
    "comments": "shipped on time",
    "total": "345.60",
    "lines": [{
        "orderLineNumber": 10,
        "productCode": "S12_1099",
        "productName": "1968 Ford Mustang",
        "qty": 2,
        "price": 35.45,
        "subtotal": 70.90
    }]
}
```

Read - Input:

```
{
    "orderNumber": 1,
    "customerNumber": 103,
    "startDate": "2020-12-01T00:00:00.000",
    "endDate": "2020-12-31T00:00:00.000"
}
```

Read - Query:

```
select o.orderNumber, o.orderDate,
       o.customerNumber, _c.customerName,
       o.requiredDate, o.shippedDate, o.status, o.comments,
       od.orderLineNumber, od.productCode, _p.productName,
       od.quantityOrdered as qty, od.priceEach,
       od.quantityOrdered, od.priceEach as subtotal,
       (select sum(quantityOrdered*priceEach) from orderdetails
         where orderNumber = o.orderNumber) as total
  from orders o
  join customers _c on o.customerNumber = _c.customerNumber
  left join orderdetails od on od.orderNumber = o.orderNumber
  left join  products _p on _p.productCode = od.productCode
 where ( 0 = 1
         or :orderNumber is not null
         or :customerNumber is not null
         or :startDate is not null
         or :endDate is not null
       )
   and o.orderNumber = :orderNumber
   and o.customerNumber = :customerNumber
   and o.orderDate between :startDate and :endDate
```

Read - Input-Bindings:

```
[
   {"parameter": "orderNumber", "field": ".orderNumber"},
   {"parameter": "customer Number", "field":".customerNumber"},
   {"parameter": "endDate", "field": ".endDate"},
   {"parameter": "startDate", "field": ".startDate"}
]
```

Read - Output-Bindings:

```
[
   {"field": "..customerNumber", "column": "customerNumber"},
   {"field": "..customerName", "column": "customerName"},
   {"field": "..address.addr", "column": "address"},
   {"field": "..address.city", "column": "city"}
   {"field": "..address.state", "column": "state"}
]
```

Write - Tables:

```
[
    {
        "name": "orders",
        "alias": "o",
        "object": ".",
        "rootTable": true
    },
    {
        "name": "orderdetails",
        "alias": "od",
        "object": ".lines..",
        "rootTable": false
    }
]
```

Write – Columns - orders:

```
[
    {
        "position": 1,
        "column": "orderNumber",
        "field": ".orderNumber",
        "key": true,
        "autoGenerate": false,
        "inputField": ".orderNumber",
        "insertValue": null,
        "updateValue": null,
        "version": false,
        "softDelete": false
    },
    {
        "position": 2,
        "column": "orderDate",
        "field": ".orderDate",
        "key": false,
        "autoGenerate": false,
        "inputField": null,
        "insertValue": null,
        "updateValue": null,
        "version": false,
        "softDelete": false
    },

    ...
]
```

Write – Columns - orderdetails:

```
[
    {
        "position": 1,
        "column": "orderNumber",
        "field": ".orderNumber",
        "key": true,
        "autoGenerate": false,
        "inputField": null,
        "insertValue": null,
        "updateValue": null,
        "version": false,
        "softDelete": false
    },
    {
        "position": 2,
        "column": "productCode",
        "field": ".lines..productCode",
        "key": true,
        "autoGenerate": false,
        "inputField": null,
        "insertValue": null,
        "updateValue": null,
        "version": false,
        "softDelete": false
     },

    ...
]
```

This exemplary repository service is for an ORDER aggregate. The object component specifies the structure and data fields of the ORDER object. The read component specifies a dynamic query service for the ORDER objects. This exemplary repository service supports read by order number, read by customer number, and read by order date range, as specified in the read input and query files.

The write component is composed of a table-bindings sub-component and a number of column-bindings components, one for each table to write. The table-bindings sub-component specify what data to write into what tables. The column-bindings sub-component specifies how to create, update and delete a record in a table. This exemplary repository service writes the root ORDER object into the root table "orders" and the "lines" array inside the root object into the "orderdetails" table, as specified in the table-bindings" table.

The repository service is to be executed by a service engine. For read operations, it executes the query and transforms the data set into the aggregate objects. For write operations, it creates the insert, update and delete statements using the table-bindings and column-bindings and can then execute them in a batch.

The object and read components are the essential elements of the repository service. The write component can be generated from them, with help of table meta data, at development time.

Unlike ORM which uses a two-way mapping, the repository service uses two one-way mappings. The read mapping maps the object to a SQL for maximum flexibility. The write mapping maps the object to the tables, once for all insert, update and delete operations.

6 Feasibility of Proposed Solution

The query, command and repository services described in the previous section relies on a service engine to execute them. Therefore, the feasibility of the proposed solution is determined by

1. The feasibility of developing such service engine.
2. The capability of the developed service engine to transform a relational data set into a complex object structure.

Without going into the detail, we like to report that:

1. A reference implementation of such a service engine has been built, tested with a good number of example query, command, and repository services, and made available [10][11][12][13].
2. A complexity test [11][14] has proved that this service engine is capable of returning an object as complex as a nested structure that is 5-level deep and 5-level wide, as shown below:

```
[{
    "id11": 1,
    "a2": [{
        "id22": 1,
        "b2": [{
            "id23": 1
         }]
    }],
    "a3": [{
        "id32": 1,
        "b2": [{
            "id33": 1
         }]
    }]
}]
```

Therefore, the proposed solution is not only feasible but also realized.

7 Advantages of Proposed Solution

The proposed solution has the following advantages over ORM:

- It is conceptually simple. This simplicity implies an easy learning curve.
- It has solved the data structure mismatch problem, implying developers can design their object model anyway they see fit for the application.
- It has turned the data access development into SQL development, implying developers can now work on their SQLs in a SQL-friendly environment.
- It is a scalable solution with respect to both performance and development effort, as the query and object gets complex.
- It is a complete solution covering all persistence patterns, as opposed to ORM suitable only for the repository pattern.
- Last but least, it is quick to develop.

The proposed solution does require a 4-tier application architecture due to the physical separation of data access logic from the application logic. However, this distributed architecture also makes the proposed solution a cross-platform solution, to serve the Java, C#, Node.js, and Python applications alike.

This 4-tier architecture is not uncommon in the industry. In fact, it is supported by frameworks like Spring REST Data, which exposes repository resources as services for client applications to consume, except that it is still based on ORM technology.

8 Conclusions

Sample mismatch problems and consolidated problem themes are analyzed. Cause of object-relational impedance mismatch is identified as the data structure mismatch between relational and object data structures and the language impedance between the embedded SQL and host programming language. To resolve these issue, a group of three relational data access services: query, command and repository, are proposed as an appropriate and feasible solution to the problem of object-relational impedance mismatch. This proposed solution has a number of important advantages over ORM technologies.

References

1. Ireland C., Bowers D.: Exposing the Myth: Object-Relational Impedance Mismatch is a Wicked Problem, https://oro.open.ac.uk/43318/1/download.php_articleid%3Ddbkda_2015_2_10_50020 (2015), last accessed 2024/04/18.
2. Ireland, J.: Object-relational impedance mismatch: a framework based approach, PhD thesis The Open University (2011).
3. G. Copeland, and D. Maier, “Making Smalltalk a database system,” ACM SIGMOD Record, vol. 14, no. 2, June 1984, pp. 316-325.

4. W. Ambler, Agile Database Techniques - Effective Strategies for the Agile Software Developer: Wiley, 2003.
5. Neward, T.: The Vietnam of Computer Science, https://www.semanticscholar.org/paper/The-Vietnam-of-Computer-Science-Neward/331e490c55ee72d6011bbceb323c03f0572a5235 (2006), last accessed 2024/04/18
6. Fowler, M.: ORM Hate, https:// www.martinfowler.com/bliki/OrmHate.html, last accessed 2024/04/18.
7. Hibernate, https://hibernate.org/orm/what-is-an-orm/#the-object-relational-impedance-mismatch, last accessed 2024/04/18.
8. Bauer, C., King G.: Java Persistence with Hibernate. Manning Publications Co., 209 Bruce Park Ave. Greenwich, CT 06830 (2007).
9. Oracle Data Access Object, https://www.oracle.com/java/technologies/data-access-object.html, last accessed 2024/04/18.
10. Wang K., Solving the relational data access problem with data access services, WorldCIST24 proceeding (2024), in process.
11. GitHub BackLogic Project, https://github.com/bklogic/backlogic-project, last accessed 2024/04/18.
12. GitHub Service Builder, https://github.com/bklogic/ServiceBuilder, last accessed 2024/04/18.
13. GitHub Data Access Service Example, https://github.com/bklogic/data-access-service-example, last accessed 2024/04/18.
14. GitHub Complexity Test, https://github.com/bklogic/complexity-test, last accessed 2024/04/18.