SQL And NoSQL Object Database Mapping To Support CRUD Operation

Billy Montolalu
Informatics Department
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
7025221006@student.its.ac.id

Siti Rochimah
Informatics Department
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
siti@if.its.ac.id

Daniel Siahaan
Informatics Department
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
daniel@if.its.ac.id

2023 International Conference on Information Technology and Computing (ICITCOM) | 979-8-3503-5963-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICITCOM60176.2023.10442056

***Abstract*— SQL and NoSQL databases have different data models and query languages, which can pose challenges when mapping between them for CRUD operations. Utilizing object database mapping can help minimize errors when executing database query commands within program code. Manually mapping object models to database tables remains a labor-intensive and error-prone process. This research introduces an approach for building an object mapping framework on SQL and NoSQL databases. We build a framework by carrying out concept mapping, syntax conversion, and developing object database mapping. Object database mapping is developed using chaining method and active record design pattern. The framework was tested using the hospital information system billing module. The results of this research demonstrate that the constructed framework can execute CRUD query commands on three distinct types of databases: MySQL, MongoDB, and Neo4j.**

Keywords—Object Database Mapping, ORM, NoSql, active record, Universal Schema

I. INTRODUCTION

Object Database Mapping refers to a programming method that maps database tables to classes or objects. These objects, as defined in the program code, represent tables, columns, and operations for creating, reading, modifying, and deleting data (CRUD). Relational (SQL) and non-relational (NoSQL) databases are distinct approaches to storing and managing data.

Currently, SQL database is still widely used in data storage. But with the emergence of modern data applications such as big data, social networks, IoT, and machine learning, NoSQL has emerged to make up for the shortcomings of relational databases. SQL databases cannot meet all data storage needs. Using more than one type of database is a solution to meet all data storage needs [1]. In certain cases, NoSQL databases perform better than relational databases [2].

NoSQL databases present distinct paradigms in terms of features and data structure, encompassing four widely used types: columnar, key-value, document, and graph databases. Object database mapping is formulated based on the design schema. Research [1], [3] introduces a universal schema metamodel designed to represent the data structure across these four NoSQL database types. A universal schema is a concept that can be applied to both relational databases and NoSQL databases. It refers to a unified structure or format for representing data that can be used across different types of databases. The idea behind a universal schema is to provide a consistent and standardized way of organizing and accessing data, regardless of the underlying database technology. In the context of relational databases, a universal schema can be achieved by defining a common set of tables and columns that can accommodate different types of data. The use of a universal schema in both SQL and NoSQL databases offers several advantages. Firstly, it promotes data interoperability and integration, allowing for seamless data exchange between different systems. Secondly, it simplifies the development process by providing a standardized way of working with data, reducing the need for custom mappings and transformations. Lastly, a universal schema can improve data consistency and quality by enforcing common data structures and constraints.

A document database is a database whose data is stored using JSON format. The DBMS (Database Management System) that uses a document database is MongoDB. Data is stored in the form of key-value. Meanwhile, object mapping in a NoSQL document database is called ODM (Object Document Mapper) [4]. Document databases can have more than one data structure. This will cause schema variations within a single database. His variation becomes a challenge in creating schemas in database documents. [5]–[7] created a schema metamodel for document databases.

The mapping stage from basic data into objects includes mapping attributes, relations, and implementing derivatives in classes [8], [9]. The form of implementation in the class influences understanding and performance.

NoSQL databases do not require defining a data schema. But in practice, the data schema in NoSQL databases is implicit in the program code. With the data schema, program code becomes easier to read, easy to maintain, and easy to reuse [10], [11]. Using object database mapping will reduce errors in executing database query commands in program code. Manual mapping from object models to database tables is still a heavy and error-prone task [5].

This paper is divided into five sections: Section 1 provides an overview of the study. Section 2 discuss related work on object database mapping. Section 3 is attentive to propose methods. Section 4 explains the findings of the results and discussion, and section 5 delivers the conclusion and future work.

II. RELATED WORKS

The data schema represents the entities in the database. Therefore, the schema can be used as a reference in mapping objects. NoSQL databases have a wide variety of data. These data variations need to be mapped into a universal schema form by distinguishing entity types and relationship types, representing aggregation and reference relationships, and including the notion of structural variability. One approach is the use of unified metamodel, which provide a way to create

schema from various data models [1]. However, this research only focuses on schema development, not developing object mapping

There are currently many frameworks for mapping objects. Research [12] compared 341 object mapping frameworks. The results of the research show that 54 frameworks can support more than one NoSQL database but do not fully support the advantages of NoSQL databases.

One approach is the use of object-to-NoSQL database mappers (ONDM) frameworks, which provide a systematic way to abstract from various data models and non-standardized APIs [13]. These frameworks aim to bridge the gap between object-oriented programming and NoSQL databases by providing mapping mechanisms that allow developers to interact with the database using object-oriented paradigms.

In creating object mapping, special coding techniques are required to build class entities, as was done by [14] who built an ORM based on table data gateway design pattern. In the Table data gateway design pattern, every table in the database is fed into a single object. The general object mapping framework follows the architectural patterns of the active record pattern and the repository pattern.

Overall, mapping between SQL and NoSQL databases for CRUD operations can be facilitated using entity-to-object mappers, mapping roles, and integration approaches. These approaches provide mechanisms to bridge the gap between different data models and query languages, enabling users to perform CRUD operations seamlessly.

III. RESEARCH METHOD

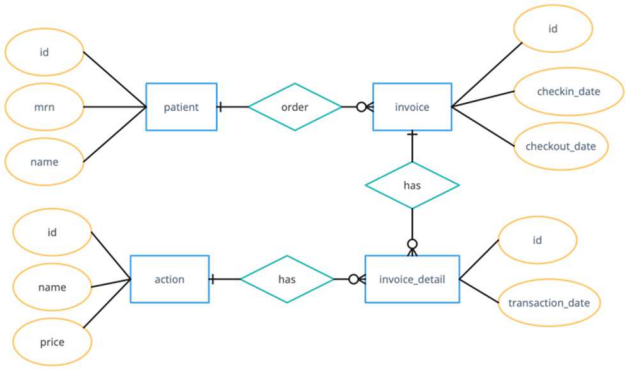

Fig. 1. Entity Relationship Diagram hospital information system - billing module.

A. Mapping Concept

In this research, concept mapping was carried out on the three types of databases. This concept mapping is based on the perspective of a SQL database. To make it easier to explain the mapping concept, we use a case study of Hospital Information System database design (billing module). The characteristic of this information system is that it only contains CRUD operations. Fig. 1 explains the database design in Entity Relationship Diagram (ERD).

TABLE I. DATABASE MAPPING CONCEPT

MySQL	MongoDB	Neo4j
table	collection	node
column	document	property
relation	Document references	relation

We apply the rules in mapping database concepts. Mapping rules can be seen in TABLE I. As an initial condition for the data schema, the ERD design in Fig. 1 is implemented in the MySQL database, as shown in TABLE II, TABLE III, TABLE IV and TABLE V.

TABLE II. PATIENT TABLE IN MYSQL

id	mrn	name
1	00221001	Billy

TABLE III. INVOICE TABLE IN MYSQL

id	patient_id	checkin_date	checkout_date
1	1	2023-10-17	null

TABLE IV. ACTION TABLE IN MYSQL

id	name	price
1	Anesthesiologist examination	20000

TABLE V. INVOICE DETAIL TABLE IN MYSQL

id	action_id	invoice_id	transaction_date
1	1	1	2023-10-17

Concept mapping from MySQL to MongoDB, tables are mapped to collections, columns to documents, and relations to document references. The results of the concept mapping to MongoDB can be seen in TABLE VI. In this concept mapping, we ignore the concept of embedded documents in MongoDB. This was done because of consistency problems.

TABLE VI. DATA SCHEMA IN MONGODB

```
Patient collection
{
  _id: 1,
  mrn: "00221001",
  name: "Billy"
}
Invoice Collection
{
  _id: 1,
  checkin_date: "2023-10-17",
  checkout_date: null
}
Action Collection
{
  _id: 1,
  name: " Anesthesiologist examination",
  price: 20000
}

Invoice_Detail Collection
{
  _id: 1,
  action_id: 1,
  invoice_id: 1,
  transaction_date: "2023-10-17"
```

}

Neo4j databases are often schemaless, allowing for the existence of data with unstructured schemas. For example, nodes and connections may share a common label while possessing different sets of properties. In our approach, we disregard this variation. Within our method, a table denotes a collection of nodes that share a common label. Similarly, a relationship type represents a grouping of relationships that share a common label or a combination of labels. Each relationship type is linked to origin and destination entity types. The outcomes of concept mapping to Neo4j databases are visualized in Fig. 2.

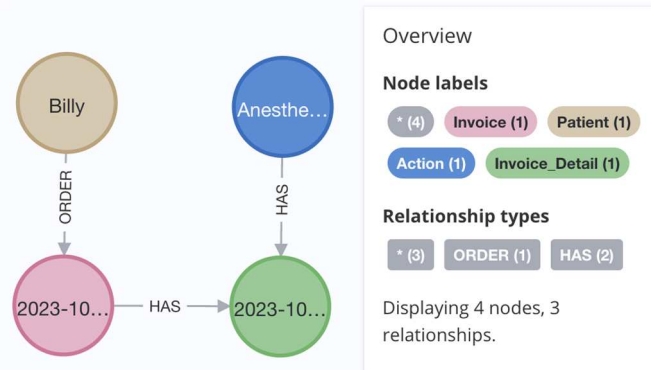

Fig. 2. Data Schema In Neo4j.

B. *Query Syntax Conversion*

To manipulate data in the database, it is necessary to do a query command. SQL and NoSQL commands have different syntax depending on each type of DBMS [6]. Data manipulation in the database is carried out in the CRUD process. Query commands on MySQL and Neo4j are based on text. While the query command in MongoDB is in the form of an object. The query syntax conversion can be seen in TABLE VII.

TABLE VII. QUERY SYNTAX CONVERSION

Scenario	Query Statement
Q1	Get Data MySQL: select * from <table> MongoDB: db.collection.find() Neo4j: MATCH (alias:Node Type) RETURN alias
Q2	Get data with condition MySQL: select * from <table> where column_name = value MongoDB: db.collection.find({column_name: value}) Neo4j: MATCH (alias:Node Type) WHERE alias.column_name = value RETURN alias
Q3	Get data with specific column MySQL: select colum_name from <table> MongoDB: db.collection.find({column_name: 1}) Neo4j: MATCH (alias:Node Type) RETURN alias.column_name
Q4	Get data with join MySQL: Select * from table1 join table2 on table1 = table2 MongoDB: db.collection1.aggregate([$lookup:{from: collection2, localField: '', foreignField: '', as: ''}]) Neo4j: MATCH (alias:Nodetype1)--(alias:Nodetype2) RETURN alias.*
Q5	Update data MySQL: update from <table> where MongoDB: db.collection.update({condition: value}) Neo4j: MATCH (alias:NodeType {condition}) SET alias.colum_name = value RETURN alias.*
Q6	Delete Data MySQL: Delete from <table> where MongoDB: db.collection.remove({condition: value}) Neo4j: MATCH (n:Person {condition: value}) DELETE n
Q7	Insert Data MySQL: insert into <table> (..colum_name) values () MongoDB: db.collection.insert({column: value}) Neo4j: CREATE (n:Person {column: value})

Based on the query syntax conversion, we define the general form of each command. This general form is in the form of an abstract syntax tree (AST) which is used as a reference for building query commands. AST in SQL queries can be seen in Fig. 3.

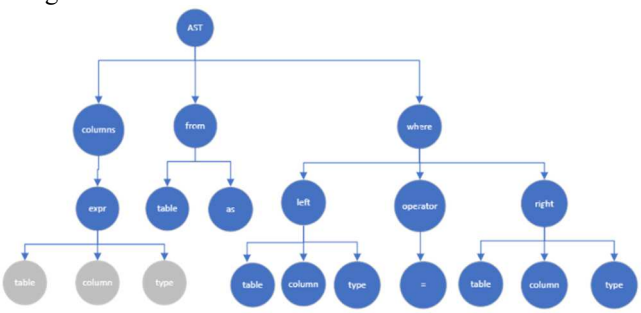

Fig. 3. SQL Abstract Syntax Tree

C. *Object Database Mapping*

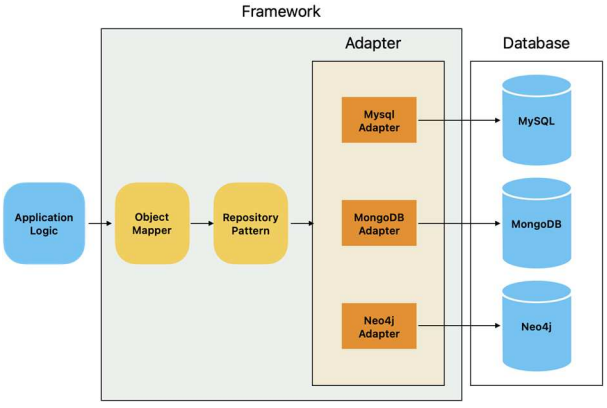

Fig. 4. Object Database Mapping Framework Architecture

Fig. 4 illustrates the proposed architecture for the object database mapping framework. The adapter is part of the framework whose job is to communicate with the DBMS directly. The adapter is responsible for converting the syntax of insert, select, update, and delete queries in the DBMS. The adapter receives query commands from the pattern repository. Then, the adapter performs syntax matching according to the database used.

Repository pattern is an approach to separate business logic from query logic. This separation is done because the framework uses three different databases but generally has the same command.

Object mapper is a class that will be used to hold data. This data represents the tables, columns, and relationships that exist in the database. Object mapper is built by mapping the attributes that exist in the three databases used. The implementation of the object mapper and repository pattern can be seen in the following class diagram, Fig. 5.

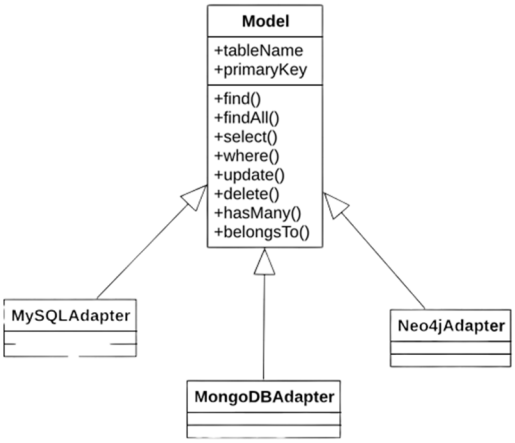

Fig. 5. Class Diagram Object Database Mapping

The object model is developed using active record design pattern. An active record object represents a single row in a database table, making it a natural and intuitive way to work with database records in object-oriented programming languages. It is a straightforward interface primarily comprising a set of methods designed to facilitate the CRUD operations of specific database entities. Each method translates input parameters into queries and subsequently executes these statements through a database connection established by a lower-level component known as the database adapter. There is an example about active record object implementation in Source Code 1.

```
const patient = new Patient()
patient.name = "Billy"
patient.mrn = "100003"
patient.save()
```

Source Code 1. Active Record Object

IV. Results and Discussion

The framework that has been built is tested by defining model objects and performing query commands. The object model is defined based on ERD, Fig. 1. Source Code 2, Source Code 3, Source Code 4, and Source Code 5 are the final results of mapping database objects. Each table in the database is defined as a class. This class is a derived class from the model class. In the model class there are functions to create save(), read find(), findAll(), update update(), and delete delete(). This model class also has functions namely hasMany() and belongsTo() for managing one to many relationships.

```
export class Patient extends Model {
  protected tableName: string = "patient";
  invoice = (): HasMany => {
    return this.hasMany(new Invoice(), 'patient_id','id');
  }
}
```

Source Code 2. Patient Model

```
export class Action extends Model {
  protected tableName: string = "patient";
  invoice_detail = (): HasMany => {
    return this.hasMany(new InvoiceDetail(), 'action_id','id');
  }
}
```

Source Code 3. Action Model

```
export class Invoice extends Model {
  protected tableName: string = "invoice";
  patient = (): BelongsTo => {
    return this.belongsTo(new Patient(), "id", "patient_id");
  };
  invoice_detail = (): HasMany => {
    return this.hasMany(new InvoiceDetail(), "", "")
  }
}
```

Source Code 4. Invoice Model

```
export class InvoiceDetail extends Model {
  protected tableName: string = "invoice_detail";
  action = (): BelongsTo => {
    return this.belongsTo(new Action(), "id", "action_id");
  };
  invoice_detail = (): HasMany => {
    return this.hasMany(new Invoice(), "patient_id", "id");
  };
}
```

Source Code 5. Invoice Detail Model

The framework is structured using method chaining to build nested select and where commands. Source Code 6 is an application of method chaining in building SQL *select* and *where* commands.

```
export class Model {
selectPart: string;
wherePart: string;

select(column: string) {
  this.selectPart = this.selectPart + "," + column;
  return this;
```

```
 }

where(column: string, operator: string, conditionValue: string) {
 this.wherePart = ` and ${column} ${operator} ${conditionValue}`;
  return this;
 }

whereOr(column: string, operator: string, conditionValue: string) {
  this.wherePart = ` or ${column} ${operator} ${conditionValue}`;
  return this;
 }
}
```

Source Code 6. Chaining method implementation

Based on CRUD scenario shown in TABLE VII, this section verifies the correctness of our framework. Testing is carried out by calling the function as in TABLE VIII. The correctness can be guaranteed by the fact that we can get the same result from MySQL, Neo4j and MongoDB databases by using the same query statement. As for Q1 scenario, get data query statement in patient table, there are 66331 records with attribute “name” got from MySQL, while there are the same result got from MongoDB and Neo4j. The experiment for the rest query statement also show the same consistency.

TABLE VIII. Object Database Mapping On CRUD Operations

Scenario	Query Statement
Q1	Get Data Patient().findAll()
Q2	Get data with condition Patient().where('mrn', '00221001').get()
Q3	Get data with specific column Patient().select('name').get()
Q4	Get data with join Patient().find(1).invoice()
Q5	Update data Patient().where('mrn', '00221001').update({'name', '00221001'})
Q6	Delete Data Patient().where('mrn', '00221001').delete()
Q7	Insert Data Patient({column:value}).save()

The framework successfully executes the entire test scenario; however, it currently operates within the scope of simple query scenarios. It's important to note that the proposed framework encounters limitations when handling more complex query commands, such as nested queries, unions, aggregations, and bidirectional relationships. These commands remain unsupported within the framework's current capabilities.

V. Conclusion And Future Work

In this paper, we built an object database mapper to support CRUD operations in SQL and NoSQL databases. A framework is constructed through the implementation of concept mapping, syntax conversion, and the creation of object database mapping components. The proposed framework can execute CRUD query commands on three different types of databases. The framework is built based on a SQL database point of view so that in NoSQL databases, there are features that are missing. For example, nested query, union, aggregation, and bidirectional relationship. There are still many opportunities that can be developed based on other points of view.

Acknowledgement

This research was funded by the Informatics Engineering Department of Sepuluh Nopember Institute of Technology through the Department’s Research Funding scheme with Work Agreement/Contract Number 1386/PKS/ITS/2023.

References

[1] C. J. F. Candel, D. Sevilla Ruiz, and J. J. García-Molina, “A unified metamodel for NoSQL and relational databases,” *Inf Syst*, vol. 104, Feb. 2022, doi: 10.1016/j.is.2021.101898.

[2] B. Jose and S. Abraham, “Performance analysis of NoSQL and relational databases with MongoDB and MySQL,” *Mater Today Proc*, vol. 24, pp. 2036–2043, 2020, doi: 10.1016/j.matpr.2020.03.634.

[3] A. Gueidi, H. Gharsellaoui, and S. Ben Ahmed, “Towards unified modeling for NoSQL solution based on mapping approach,” in *Procedia Computer Science*, Elsevier B.V., 2021, pp. 3637–3646. doi: 10.1016/j.procs.2021.09.137.

[4] D. S. Ruiz, S. F. Morales, and J. García-Molina, “An MDE approach to generate schemas for object-document mappers,” in *MODELSWARD 2017 - Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, SciTePress, 2017, pp. 220–228. doi: 10.5220/0006279102200228.

[5] L. Y. Ismailova and S. V. Kosikov, “Metamodel of Transformations of Concepts to Support the Object-Relational Mapping,” in *Procedia Computer Science*, Elsevier B.V., 2018, pp. 260–265. doi: 10.1016/j.procs.2018.11.055.

[6] S. Bjeladinovic, Z. Marjanovic, and S. Babarogic, “A proposal of architecture for integration and uniform use of hybrid SQL/NoSQL database components,” *Journal of Systems and Software*, vol. 168, Oct. 2020, doi: 10.1016/j.jss.2020.110633.

[7] Z. Huang, Z. Shao, G. Fan, H. Yu, K. Yang, and Z. Zhou, “HBSNIFF: A static analysis tool for Java Hibernate object-relational mapping code smell detection,” *Sci Comput Program*, vol. 217, May 2022, doi: 10.1016/j.scico.2022.102778.

[8] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley, 2003.

[9] S. Philippi, “Model driven generation and testing of object-relational mappings,” *Journal of Systems and Software*, vol. 77, no. 2, pp. 193–207, Aug. 2005, doi: 10.1016/j.jss.2004.07.252.

[10] E. M. Kuszera, L. M. Peres, and M. Didonet Del Fabro, “Exploring data structure alternatives in the RDB to NoSQL document store conversion process,” *Inf Syst*, vol. 105, Mar. 2022, doi: 10.1016/j.is.2021.101941.

[11] A. H. Chillon, D. S. Ruiz, J. G. Molina, and S. F. Morales, “A Model-Driven Approach to Generate Schemas for Object-Document Mappers,” *IEEE Access*, vol. 7, pp. 59126–59142, 2019, doi: 10.1109/ACCESS.2019.2915201.

[12] V. Reniers, D. Van Landuyt, A. Rafique, and W. Joosen, “Object to NoSQL Database Mappers (ONDM): A systematic survey and comparison of frameworks,” *Information Systems*, vol. 85. Elsevier Ltd, pp. 1–20, Nov. 01, 2019. doi: 10.1016/j.is.2019.05.001.

[13] V. Reniers, A. Rafique, D. Van Landuyt, and W. Joosen, “Object-NoSQL Database Mappers: a benchmark study on the performance overhead,” *Journal of Internet Services and Applications*, vol. 8, no. 1, Dec. 2017, doi: 10.1186/s13174-016-0052-x.

[14] V. Kaczmarczyk, Z. Bradáč, J. Arm, O. Baštán, and Z. Kaczmarczyková, “A Simple and effective ADO.NET-based ORM layer,” *IFAC-PapersOnLine*, vol. 52, no. 27, pp. 228–234, 2019, doi: 10.1016/j.ifacol.2019.12.761.