

MfCodeGenerator: A Code Generation Tool for NoSQL Data Access with ONM Support

Evandro Miguel Kuszera¹^a, Leticia Mara Peres²^b and Marcos Didonet Del Fabro³^c

¹Federal University of Technology, Paraná, Dois Vizinhos, Brazil

²Federal University of Paraná, Curitiba, Brazil

³Université Paris-Saclay, CEA, List, Palaiseau, France

Keywords: Object-NoSQL Mapper, NoSQL, Code Generation.

Abstract: NoSQL databases are generally employed in scenarios that require horizontal scalability and flexibility in data schema. Applications can access the NoSQL database through native APIs or through ONMs (Object-NoSQL Mappers). The latter provides a uniform data access interface, decoupling the application from the database and reducing vendor lock-in. However, ONM code creation should be performed by developers and can be cumbersome and error prone. In this paper we propose an approach to generate ONM code based on a NoSQL schema that describes the structure of the entities and their relationships. From the NoSQL schema, our tool is used to generate code for three widely used Java-based ONMs. To evaluate the approach we perform experiments to read and write data to and from an existing MongoDB database using the generated code. Through the results obtained, it was possible to verify that the tool is capable of generating code according to the NoSQL schema and the requirements of the target ONM. This not only streamlines developer access to NoSQL data but also facilitates comparative evaluations of different ONMs utilizing the same schema.

1 INTRODUCTION


Relational databases (RDB) are the de facto standard for storing data in most existing applications. However, the relational model based on tables with rows and columns, primary and foreign keys, has its limitations when there is a need to scale the application horizontally (Stonebraker et al., 2007). In addition, there is also the issue of the need to define a schema before storing the data, which impacts its use in applications that handle semi-structured data and require flexible data models.


NoSQL databases (Sadalage and Fowler, 2012) emerged as a solutions for these problems, providing data model, architecture and query languages different from the relational model. From an application standpoint, access to data in the NoSQL database can be performed through native APIs or through middlewares. In the context of relational databases we have ORMs (Object-Relational Mapper) (O’Neil, 2008), that are middlewares which provide mechanisms


to map objects from application to records in the database and vice versa. Regarding NoSQL databases we have ONMs (Object-NoSQL Mappers), which map objects from application to the target NoSQL data model.

ONMs are generally used in the development of new applications, but there are scenarios in which the database already exists or comes from a migration from RDB to NoSQL. In all the cases it is necessary create ONM software artifacts to manipulate the data. This task should be performed by developers and can be cumbersome and error prone, even more so if the existing data is stored in a format other than that supported by ONM. One way to make the developer’s work easier is to generate the ONM code automatically.

There are different works that evaluate and compare the ONMs in terms of features and the introduced overhead to access the data against the native APIs (Störl et al., 2015; Reniers et al., 2017; Rafique et al., 2018), but none of them deal with ONM code generation. In (Chillón et al., 2019), a solution was proposed to read the data from a NoSQL database and extract its schema. Subsequently, the approach automatically

^a <https://orcid.org/0000-0002-4040-0151>

^b <https://orcid.org/0000-0002-8922-6975>

^c <https://orcid.org/0000-0002-8573-6281>

generates ONM code for Morphia ¹ and Mongoose ². However, it does not address limitations of ONMs concerning the extracted NoSQL schema, which may have a hierarchical structure with depth not supported by ONM, or have unsupported relationship types, or even the format of the data stored in NoSQL may not be supported by ONM.

In this paper we present the MfCodeGenerator, an approach for **automatic code generation for ONMs** aimed at document-oriented NoSQL databases. MfCodeGenerator uses a NoSQL schema as input and currently it generates code for three Java-based ONM, namely Impetus Kundera³, Data Nucleus⁴ and Spring Data⁵, but the approach is platform independent and can be extended to others ONMs and languages. Our approach allows customizing and adding validations in the code generation process, allowing to verify if the NoSQL schema is supported by the ONM, warning the developer in case it cannot be defined in the ONM code.

To evaluate the MfCodeGenerator, we conducted experiments to read and write data in MongoDB using the generated code. Based on the results obtained, we verified that the tool is capable of generating code according to the NoSQL schema and requirements of the target ONM. This simplifies the process for developers to access NoSQL data and also facilitates the evaluation of different ONMs based on the same schema. The main contributions of this paper are:

- Our approach automatically generate code from a NoSQL schema, adding necessary annotations to support the ONM;
- Developer can add customizations to MfCodeGenerator, to introduce annotations or specific code into the generation process.
- The MfCodeGenerator validates the schema against the target ONM and warns if certain types of relationships are not supported. This allows the developer to either adjust the input schema accordingly or be informed that modifications to the generated code will be necessary.

The remainder of this paper is structured as follows. Section 2 provides the context and necessary background of the proposed approach. Section 3 presents the MfCodeGenerator, our approach to automatically generated ONM code, describing its execution flow and architecture. Section 4 presents the

experiments to evaluate the approach. Section 5 discusses related work. Section 6 concludes the paper.

2 CONTEXT AND BACKGROUND

The approach proposed in this paper aims to automatically generate code to access data stored in document-oriented NoSQL databases. As there are different ways to model entities as collections of documents, it is interesting to have a way to define a schema and generate the code to create, retrieve, update and delete data from the NoSQL database, with support for different middlewares in a simplified way. The approach proposed in this paper can be applied in different scenarios:

- **Application Development:** Developers provide a data schema, and our approach generates code tailored to the chosen ONM.
- **ONM Evaluation:** Our approach facilitates comparing different ONMs by generating custom code for each from the same input schema.
- **RDB to NoSQL Migration:** We can generate a NoSQL schema based on metadata from the relational database, utilizing it to generates code to read data from the RDB and write into MongoDB.
- **Integration with Migration Tools:** Through adapters, our tool can utilize data schema from other tools to generate code for accessing migrated data.

In a previous study, we introduced a framework for migrating data from RDB to NoSQL (Kusza et al., 2019). Our framework offers a schema that outlines the data structure in the NoSQL database. Subsequently, this schema is employed to generate commands for data migration. In this paper we extend our schema and use it as input to generate ONM code to access the NoSQL database. The following sections present an overview of the schema and the Object-NoSQL mappers supported by the approach.

2.1 NoSQL Schema

We use a set of DAGs (Directed Acyclic Graph) to represent a NoSQL schema. Each DAG in the schema represents a NoSQL entity and it has a tree structure. The root vertex represents the root document and the remaining vertices represent the embedded documents. The edges inside a DAG define the type of embedding relationship (document or array of document). In this paper we extend our schema notation

¹<https://github.com/MorphiaOrg/morphia>

²<https://mongoosejs.com/>

³<https://github.com/Impetus/Kundera>

⁴<https://www.datanucleus.org/products/accessplatform/>

⁵<https://spring.io/projects/spring-data>

to allows reference relationships between DAGs (entities). Figure 1 shows the DAG elements and an example of entity.

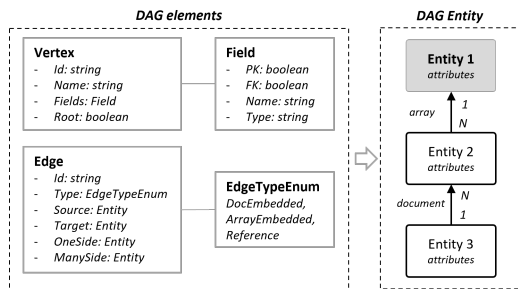


Figure 1: DAG elements (left) and an example of DAG entity (right) composed by three vertices and two edges.

The vertices encapsulate the metadata of the respective document, including name, fields, data types and which fields are identifiers (primary keys) or reference other fields (foreign keys). The edges encapsulate the metadata of the relationship between two vertices (documents), including the primary and foreign keys, which document is on side one or side many of the relationship and the type of relationship (document embedded, array embedded or reference). The edge direction shows how to embed the data to create an entity. Figure 1 (right side) shows an entity composed by three vertices, where the gray color vertex is the root document and the other vertices are the embedding documents.

From the perspective of migrating from RDB to NoSQL, developers can create a NoSQL schema based on the metadata of the RDB. Figure 2 illustrates the relational schema of a database storing data about a DVD store. From this relational database we define the schema with three DAGs of Figure 3. For the sake of simplicity, only the fields of type document are showed, but the entities also have fields of simple types that are suppressed. This schema can be leveraged to generate code to aid in data migration or to access migrated data from another tool. In this step, data access middlewares can be used.

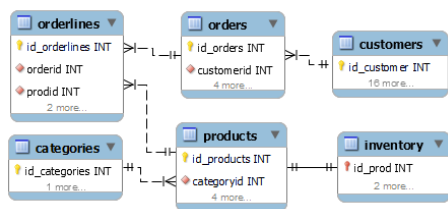


Figure 2: Relational schema of the database used to illustrate the proposed approach.

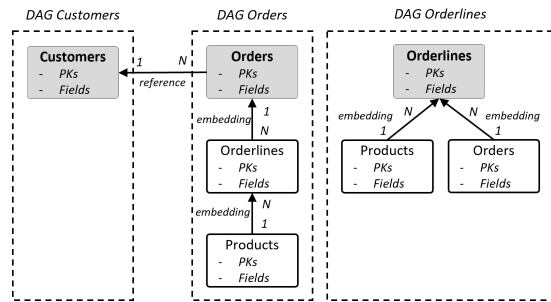


Figure 3: NoSQL schema represented by a set of DAGs.

2.2 Object-NoSQL Mappers (ONMs)

ONMs offer advantages to applications, including decoupling from the database, enhancing portability, and simplifying NoSQL access, thereby boosting productivity. However, despite their benefits, encoding with ONMs remains a laborious and error-prone task.

In this work we focus on Impetus Kundera, DataNucleus and Spring Data. All of them provide access to relational and non-relational databases, and support for CRUD operations (*Create, Retrieve, Update, and Delete*). They also support MongoDB, the most widely used document-oriented NoSQL. DataNucleus implements JPA (Java Persistence API) and JDO (Java Data Object) interfaces to access the database, meanwhile Kundera implements only JPA interface. Both implements JPQL (Java Persistence Query Language) to query the database. Spring Data provides custom interface to access the database, by object-mapping abstractions based on repository concept and annotations. Although the three ONM above are based on annotations to configure data access, they differ in terms of configuration, format and number of annotations available.

3 MfCodeGenerator

In this section we present our approach, named MfCodeGenerator⁶. Figure 4 shows the execution flow of the approach.

MfCodeGenerator takes as input a DAG schema and an ONM config, and generates as result ONM code to access data in document databases. The main components of it are MfSchemaGenerator, MfCustomization and CodeGenerator. In the first phase (1), MfSchemaGenerator traverses the DAG Schema and converts it into a MfSchema, which is an abstraction to represent the DAG entities and their relationships

⁶<https://github.com/evandrokuszera/metamorfose-code-generator>

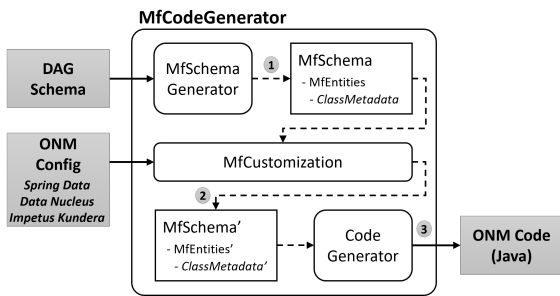


Figure 4: Execution flow of the MfCodeGenerator.

using a Java class notation. At this point, we have a MfSchema that is agnostic of any ONM. In the second phase (2) the generated MfSchema is passed to MfCustomization, in which it is enriched with new annotations, fields and imports, according to target ONM config. In the last phase (3), the CodeGenerator saves the enriched MfSchema as Java classes on disk for developers to use in Java projects for NoSQL database data operations. The target NoSQL database is MongoDB, but it can be extended to support new ONMs and databases.

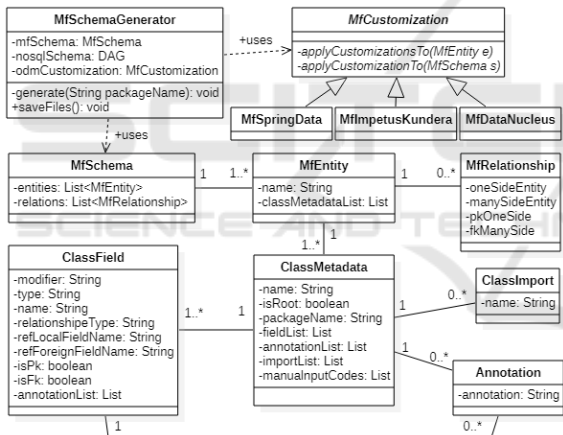


Figure 5: Class diagram of MfCodeGenerator.

Figure 5 depicts the class diagram of MfCodeGenerator, where MfSchema serves as the foundation. It comprises one or more MfEntity objects, that can be interconnected via MfRelationships. MfEntity structures can be simple, composed of a single ClassMetadata object, or hierarchical, comprising multiple ClassMetadata objects. ClassMetadata encapsulates entity metadata akin to Java class structure, encompassing fields, imports, and annotations represented by ClassField, ClassImport, and Annotation classes. Furthermore, ClassMetadata retains metadata regarding identifier fields, relationship fields, and relationship direction.

MfCustomization is an abstract class that provides an extension point for adding customizations to Mf-

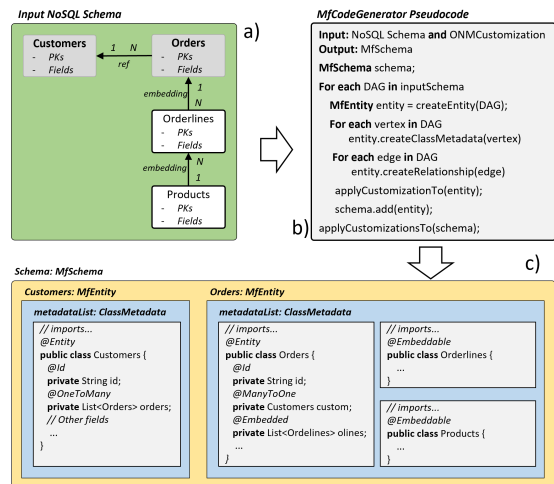


Figure 6: Example of ONM code generated from a NoSQL schema with two entities (Customers and Orders).

Schema and its respective MfEntity objects, according to target ONM.

3.1 Code Generation Process

To illustrate the process to generate ONM code let us considering the scenario of Figure 6.

Figure 6a shows a NoSQL schema that is composed of entities *Customers* and *Orders*. This schema must be previously created by the developer according to the application’s requirements (for example, its access pattern).

Figure 6b shows the pseudocode of MfCodeGenerator. Each schema DAG of input schema is traversed to generate MfEntity objects from vertices and edges. The information encapsulated in the DAG is used to create the ClassMetadata objects, denoting the structure of the entity. After that, these objects are customized according to the target ONM (applyCustomizationTo) and added to the list of MfSchema entities. Finally, MfSchema is also customized, allowing to establish relationships between the entities created in the previous step.

Figure 6c shows an in-memory representation of MfSchema, with the generated ONM code. At that point, the developer can either inspect the schema or save the code to disk.

The code snippet in the Figure 7 shows how to call the MfCodeGenerator in a Java project to create Impetus Kundera ONM code inside a package named *model*. As output, two packages are generated, named *”model.customer”* and *”model.orders”*, with their respective classes to denote the entities *Customers* and *Orders*.

```

public class Generator {
    public static void main(String[] args)
        throws FileNotFoundException {
        // Loading the NoSQL Schema from disk
        NoSQLSchema schema = loadNosqlSchema("schema.json");
        // Schema generator for Kundera ONM and MongoDB
        MfSchemaGenerator schemaCodeGenerator = new
            MfDagSchemaGenerator(schema, new
                MfKunderaMongoCustomization());
        // Generating the schema using the package name model
        schemaCodeGenerator.generate("model");
        // Saving the generated Java classes into the project
        schemaCodeGenerator.saveFiles();
    }
}

```

Figure 7: Code snipped to execute the MfCodeGenerator in a Java project.

3.2 ONM Config

In our approach an ONM config is denoted by a subclass of MfCustomization, in which the developer can provide implementations to enriched the generated code with new annotations, imports and fields.

Figure 8 shows a snippet of ONM customization code implemented for Impetus Kundera. This code is called by MfSchemaGenerator to apply customizations to all ClassMetadata objects of the MfEntities created in the previous stage. In the first method of the sample code, all root classes of schema are annotated with @Entity and all nested classes with @Embeddable. In the second method, classes are annotated with @OneToMany and @ManyToOne to establish relationships between entities. Besides Impetus Kundera, we provide a default implementation for Spring Data and Data Nucleus (see repository), but the developer can provide new implementations.

```

public class MfKunderaMongoCustomization
    extends MfClassCustomization {
    // Applying customizations for each entity
    public void applyCustomizationsTo(MfEntity en){
        en.getRootClass().addImport("javax.persistence.Entity");
        en.getRootClass().addImport("javax.persistence.Id");
        en.getRootClass().addAnnotation("@Entity");
        en.getRootClass().getId().addAnnotation("@Id");
        en.getRootClass().addField("private", "int", "cod");
        // ...
        for (ClassMetadata nested : entity.getNestedClasses()) {
            nested.addImport("javax.persistence.Embeddable");
            nested.addAnnotation("@Embeddable");
            // ...
        }
    }
    // Applying customizations for schema
    public void applyCustomizationsTo(MfSchema schema){
        for (Relationship ref : schema.relationships) {
            ref.oneEntity.addImport("javax.persistence.OneToMany");
            ref.manyEntity.addImport("javax.persistence.ManyToOne");
            ref.oneEntity.addAnnotation("@OneToMany(mappedBy = ...)");
            ref.manyEntity.addAnnotation("@ManyToOne");
            // ...
        }
    }
}

```

Figure 8: Code snipped with example of customizations for the ONM Impetus Kundera.

4 EXPERIMENTS

To evaluate the MfCodeGenerator let us consider the scenario in which there is a Java application that needs to persist data in MongoDB database. We choose MongoDB because it is the most popular document oriented NoSQL database⁷. Rather than creating the code to access the data manually, we will define a NoSQL schema and provide it as input to the MfCodeGenerator. Three ONMs are used in the experiments: Spring Data, Data Nucleus and Impetus Kundera.

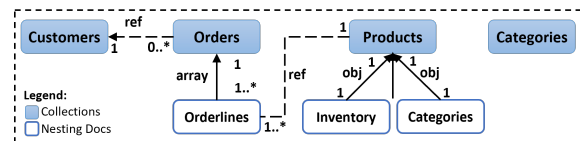


Figure 9: NoSQL schema that represents how the data are stored in the MongoDB.

Figure 9 illustrates the NoSQL schema employed in the experiments. We derived this schema from the relational schema depicted in Figure 2. For the sake of simplicity, only the names of collections and embedded documents are presented. The schema comprises four collections: *Customers*, *Orders*, *Products*, and *Categories*. There are two types of relationships between entities: references and nesting, e.g. *Customers* and *Orders* have a reference relationship whereas *Orders* and *Orderlines* have a nesting relationship. *Categories* appears twice in the schema, as a collection and as an embedded object in *Products* collection.

We choose this NoSQL schema because it presents different types of relationships (references and nesting) and data redundancy (*Categories*), which are interesting aspects to evaluate the MfCodeGenerator. However, due to flexibility of NoSQL databases, it is possible to structure the data in different ways.

4.1 ONM Code Generation

To carry out the evaluation we create a Java project for each ONM with dependencies for MongoDB and MfCodeGenerator. Then, we execute the MfCodeGenerator, providing as input parameters the NoSQL schema, target ONM, and the base package name for saving the generated classes within the project.

Figure 10 shows the list of generated Java classes for the Data Nucleus. The number and name of generated classes are the same for the other ONMs, since the input NoSQL schema used is the same. There are four code packages in which each package

⁷<https://db-engines.com/en/ranking>

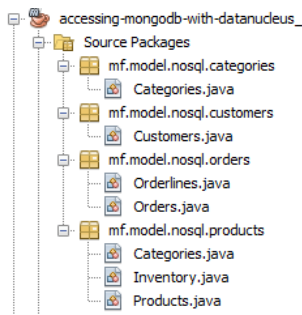


Figure 10: Generated ONM code for data access.

stores one entity of the schema. An entity is composed by one root class, that represent the root document of the collection and it could have other classes that represent nested documents. For example, package *mf.model.nosql.categories* stores a single class to represent the document structure of *Categories* collection and package *mf.model.nosql.products* stores three classes to represent the document structure of collection *Products*, where *Products* is the root class and *Categories* and *Inventory* classes are the nested documents.

MfCodeGenerator generates code that conforms to the NoSQL schema, adding attributes to establish relationships between classes. Figures 11, 12 and 13 present the code generated by MfCodeGenerator, highlighting the ONM annotations added in the classes. For the sake of simplicity, we only show identifier fields and relationship fields, suppressing other fields and getter and setter methods. The *Categories* entity is not presented because the code is similar to *Customers* and it does not bring new aspects to highlight.

```
// SPRING DATA
@Entity(collection="Customers")
public class Customers {
    @Id
    private String _id;
    private Integer id_customer;
    @ReadOnlyProperty
    @DocumentReference(lookup=
        "'customerid':?#{self.id_customer}")
    private List<Orders> orders;
}

//DATA NUCLEUS and IMPETUS KUNDERA
@Entity(name = "Customers")
public class Customers {
    @Id
    private Integer id_customer;
    @OneToMany(mappedBy = "customerid")
    private List<Orders> orders;
}
```

Figure 11: Customers code generated for all ONMs.

4.2 ONM Code Customization

It is worth noting the use of *@AttributeOverride* annotation in *Products* class for Data Nucleus (Figure 12). The annotation defines the attributes name of the

```
// DATA NUCLEUS
@Entity(name="Products")
public class Products {
    @Id
    private Integer id_prod;
    @Embedded
    @AttributeOverride(name=
        "prod_id",column=
        "prod_id")
    @Column(name="prod_id")
    @AttributeOverride(name=
        "quan_stock",column=
        "quan_stock")
    @AttributeOverride(name=
        "sales",column=
        "sales")
    @Column(name="sales")
    private Inventory inventory;
    @Embedded
    @AttributeOverride(name=
        "id_category",column=
        "id_category")
    @Column(name="id_category")
    @AttributeOverride(name=
        "categoryname",column=
        "categoryname")
    @Column(name="catgoryname")
    private Categories cats;
    @OneToMany(mappedBy=
        "prod_id")
    private List<Orderlines>
        olines;
}

// IMPETUS KUNDERA
@Entity(name="Products")
public class Products {
    @Id
    private Integer id_prod;
    @Embedded
    private Inventory inventory;
    @Embedded
    private Categories cats;
    private List<Orderlines>
        olines;
}
```

Figure 12: Products, Categories and Inventory code generated for all ONMs. Getters and setters are omitted.

```
// DATA NUCLEUS
@Entity(name = "Orders")
public class Orders {
    @Id
    private Integer id_order;
    @Embedded
    private List<Orderlines>
        olines;
    @ManyToOne
    private Customers custom;
}

@Embeddable
public class Orderlines {
    private Integer olineid;
    private Integer orderid;
    private Integer prod_id;
    private Integer quantity;
    private Date olinedate;
    private Products products;
}

// SPRING DATA
@Document(collection=
    "Orders")
public class Orders {
    @Id
    private String _id;
    private Integer id_order;
    private List<Orderlines>
        olines;
    @DocumentReference(lookup=
        "'id_customer':?#{customerid}")
    private Customers custom;
}

public class Orderlines {
    private Integer olineid;
    private Integer orderid;
    private Integer prod_id;
    private Integer quantity;
    private Date olinedate;
    @DocumentReference(lookup=
        "'id_prod':?#{prod_id}")
    private Products products;
}

// IMPETUS KUNDERA
@Entity(name="Orders")
public class Orders {
    @Id
    private Integer id_order;
    @ElementCollection
    private List<Orderlines>
        olines;
    @ManyToOne
    @JoinColumn(name=
        "customerid")
    private Customers custom;
}
```

Figure 13: Orders and Orderlines code generated for all ONMs.

Inventory and *Categories* classes, otherwise the ONM uses a different strategy for naming the data fields in MongoDB, preventing the loading of data from the database. Adding annotations like `@AttributeOverride` can be configured in MfCodeGenerator for a specific ONM.

The embedded relationships were all generated by MfCodeGenerator according to the schema. However, it was not possible to generate all reference-type relationships. Table 1 shows the list of reference-type relationships supported by each ONM. We use a notation *entity-direction-entity* to represent the relationships.

Table 1: List of reference-type relationships defined in the NoSQL schema supported by ONM and MfCodeGenerator.

Relationship	Spring	Impetus	Nucleus
<i>Customers</i> → <i>Orders</i>	Yes	Yes	Yes
<i>Orders</i> → <i>Customers</i>	Yes	Yes	Yes
<i>Orders.Orderlines</i> → <i>Product</i>	Yes	Not	Yes
<i>Product</i> → <i>Orderlines</i>	Not	Not	Not

The relationship between *Products* and *Orderlines* is generated only in the *Orders.Orderlines* → *Products* direction for Spring Data and Data Nucleus. In the opposite direction, the relationship *Products* → *Orders.Orderlines* does not work for any ONM. For Impetus Kundera it is possible to establish the reference-type relationships only between classes that are annotated with `@Entity` (see ONM documentation). The reason for this is that *Orderlines* is not a root entity, and it is not possible to load the document from the database directly.

It is important to note that the generated code conforms to the NoSQL schema and annotations are added to the code only if they are supported by the target ONM.

4.3 Evaluation of ONM Code

To evaluate the generated code we seek to answer two questions: (Q1) Is the generated code able to create, retrieve, update and delete documents in MongoDB? (Q2) Do the generated code and ONM support the relationships from the NoSQL schema?

4.3.1 Answer to Q1

To answer the first question, we perform CRUD operations using the generated code for all the ONMs. The main objective is check if the generated code can write entities into MongoDB and, then read the data into memory. For Data Nucleus, Impetus Kundera and Spring Data it was possible to create an entity, read, update and persist it again in MongoDB. This test did not include related data from other entities.

4.3.2 Answer to Q2

For second question we read one document of *Categories*, *Customers*, *Orders* and *Products* collections and check if after reading the root entity the related data could also be loaded. For all the ONMs the embedded entities could be automatically loaded from MongoDB without problem and the reference-type relationships all the related root entities could load data automatically. However, loading related non-root entities works only for Spring Data and Data Nucleus in the direction embedded-entity to root-entity, as showed in Table 1.

We repeated the above two tests with a larger number of documents (*Customers*: 10k, *Orders*: 20k, *Orderlines*: 40k, *Products*: 10k and *Categories*: 20). The results were the same, making it possible to write data to MongoDB and then, read the data correctly.

4.4 Discussion of the Results

MfCodeGenerator automates the generation of code for NoSQL database access following the input NoSQL schema, however, it is necessary to consider the ONM limitations, such as support for certain types of relationships and the supported depth of nesting.

MfCodeGenerator offers a validation method to check if the NoSQL schema aligns with the capabilities of the target ONM, notifying the developer of any unsupported relationships. For example, when using Impetus Kundera, MfCodeGenerator flags relationships between entities lacking the `@Entity` annotation. Subsequently, developers can choose to adjust the schema or generate the code and implement required modifications accordingly.

In future work, we aim to broaden the tool's evaluation scope by incorporating complex schemas and diverse NoSQL databases and ONMs.

5 RELATED WORK

There are works that evaluate and compare the ONMs in terms of features (CRUD operations and query support) and the introduced overhead to access the data against the native APIs (Störl et al., 2015; Reniers et al., 2017). In (Reniers et al., 2019) the authors presented a survey with an comprehensive comparison of eleven ONMs, in terms of program language, database, query interface and mapping strategies support. On the contrary, in this work we propose an approach for automatic generation of ONM code, that can be used to assist in the evaluation of ONMs.

The work presented in (Chillón et al., 2019) introduces a solution for generating ONM code from a NoSQL schema extracted from an existing database. While the solution supports Morphia and Mongoose, it does not address the limitations of ONMs, such as differences in document embedding levels and allowed annotations for relationship types like @OneToOne and @OneToMany. Additionally, variations in persisting data formats to represent reference relationships across ONMs are not discussed, necessitating consideration when reading data from an existing database.

Our solution is similar to (Chillón et al., 2019), in which we can generate ONM code from our NoSQL schema definition. However, MfCodeGenerator focus in Java-based ONM and support Impetus Kundera, Data Nucleus and Spring Data. Furthermore, our approach allows customizing and adding validations in the code generation process, alerting developers if it cannot be defined in the generated code.

MongoDB introduced the Relational Migrator tool⁸, enabling the creation of a NoSQL schema from an existing RDB and subsequent data migration. Additionally, the tool generates ONM code for Java (Spring Data). However, manual modifications are still required for the generated code before integration into the application, and reference-type relationships between classes are not generated. In contrast, our approach supports a broader range of ONMs and can be integrated with MongoDB tool to generate code for other ONMs.

6 CONCLUSION

This paper introduced the MfCodeGenerator, an approach to automatically generate ONM code from an input NoSQL schema and a set of customization rules for the target ONM. As a result, a set of Java classes is generated, enriched with the code to allow reading and writing objects in the NoSQL database.

MfCodeGenerator generates code for Spring Data, Data Nucleus and Impetus Kundera ONMs, but it is worth noting that the approach is platform independent and can be extended to other ONMs and languages. Experimental results demonstrate that the tool generates the Java classes correctly, with imports, annotations, fields, methods and relationships, adhering to the specified NoSQL schema. The generated code was evaluated by performing CRUD operations on MongoDB.

⁸<https://www.mongodb.com/products/relational-migrator>

Not all features specified in the NoSQL schema could be translated directly into the generated code, mainly due to constraints imposed by the target ONM, such as limitations on supported relationship types and data storage formats required by MongoDB. To address this, MfCodeGenerator issues warnings to developers, indicating whether the selected ONM fully supports the provided NoSQL schema.

As future works it is intended to add support for new ONMs and NoSQL databases and improve the mechanism for defining rules for customizing the generated code, which today is embedded in the MfCodeGenerator source code. Furthermore, we aim to expand the evaluation of the tool to encompass more complex scenarios.

REFERENCES

- Chillón, A. H., Ruiz, D. S., Molina, J. G., and Morales, S. F. (2019). A model-driven approach to generate schemas for object-document mappers. *IEEE Access*, 7:59126–59142.
- Kuszera, E. M., Peres, L. M., and Fabro, M. D. D. (2019). Toward rdb to nosql: Transforming data with metamorfose framework. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, page 456–463, New York, NY, USA.
- O’Neil, E. J. (2008). Object/relational mapping 2008: Hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD*, page 1351–1356, New York, NY, USA.
- Rafique, A., Landuyt, D. V., Lagaisse, B., and Joosen, W. (2018). On the performance impact of data access middleware for nosql data stores A study of the trade-off between performance and migration cost. *IEEE Trans. Cloud Comput.*, 6(3):843–856.
- Reniers, V., Landuyt, D. V., Rafique, A., and Joosen, W. (2019). Object to nosql database mappers (ONDM): A systematic survey and comparison of frameworks. *Inf. Syst.*, 85:1–20.
- Reniers, V., Rafique, A., Landuyt, D. V., and Joosen, W. (2017). Object-nosql database mappers: a benchmark study on the performance overhead. *J. Internet Serv. Appl.*, 8(1):1:1–1:16.
- Sadalage, P. J. and Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition.
- Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. (2007). The end of an architectural era (it’s time for a complete rewrite). In *Proceedings of the 33rd VLDB, University of Vienna, Austria, 2007*, pages 1150–1160.
- Störl, U., Hauf, T., Klettke, M., and Scherzinger, S. (2015). Schemaless nosql data stores - object-nosql mappers to the rescue? In *Datenbanksysteme für Business, Technologie und Web (BTW)*, volume P-241 of LNI, pages 579–599. GI.