Rel2Doc: Migrating Data from Relational to Document-Oriented Databases

Tainam Spagnollo Garbin[1], **Denio Duarte**[1], **Geomar A. Schreiner**[1]

[1]Universidade Federal da Fronteira Sul (UFFS)
Campus Chapecó
Chapecó – SC – Brazil

tainamgbn@gmail.com, {duarte,gschreiner}@uffs.edu.br

Abstract. *This paper presents Rel2Doc, a tool to migrate data from relational to document-oriented databases. From normalized concepts, Rel2Doc implements aggregation on the document side using foreign keys and join tables to avoid references in the document-oriented database. We build an interface where the user can choose the relational database source and some parameters to define how primary and foreign keys are implemented on the document-oriented side. We conducted experiments using PostgreSQL and MongoDB, and Rel2Doc migrates all data regarding correctness and completeness, ensuring that queries have the same answers in both databases.*

1. Introduction

Relational databases (RDB) and SQL have been the preferred technologies for decades for storing and managing data. However, over the last years, we have witnessed tremendous growth in the size and variety of data sets in several application domains. This growth brings other challenges in data management, and NoSQL DBs have been proposed to deal with volume, variety, and velocity of data (aka, the 3 V's) [Abadi 2009].

In this scenario, some legacy data-centric applications must be modernized to meet new user's demands like availability and scalability. Modernizing legacy applications is complex and challenging, especially when it involves moving data across platforms [Thalheim and Wang 2013]. This complexity should be acknowledged and factored into the planning and execution of the modernization process. Data can be lost or partially moved, causing inconsistency in the new application.

Migrating data from a relational model to any NoSQL model can be costly in terms of computational resources and manual effort because it involves mapping the source database structure to the target database structure [Thalheim and Wang 2013]. Relational data are generally normalized and spread across tables. On the other hand, NoSQL data models often store denormalized data and specific models (e.g., document-oriented and column-oriented) aggregate data.

The most used NoSQL model is document-oriented[1], and it is generally implemented using JSON data structure. When migrating relational data to a document-oriented model, a problem arises: how to model relationships into collections. The intuition is keeping all related data in the same collections to avoid joining collections.

[1]db-engines.com/en/ranking

This feature, called aggregation, means related information stays together without being joined, as in the relational model. An essential question in this scenario is: When does a table become an aggregation? That is, a table T_1 encompasses another correlated table T_2 generating a collection where T_2 tuples are inserted as documents inside T_1 tuples. More precisely, a collection built from T_1 encompasses a collection built from T_2.

This demo (`Rel2Doc`[2][3]) intends to migrate relational data into document-oriented data trying to create the minimum collections as possible. We consider all types of relationships and their associated tables to accomplish this. All the keys (`K`) and foreign keys (`FK`) of the database are used to identify whether or not a table becomes a collection or is encompassed into another collection. We run our experiments using PostgreSQL and MongoDB, and the results show that `Rel2Doc` is effective migrating data, and the new document-oriented database is complete and correct. Regarding the computational performance, we note that the number of `FK` in the relational databases directly influences the processing time since `FK` is used to check if a table becomes a new collection or is aggregated to another collection.

2. Theoretical Background

This section presents the theoretical foundations supporting this study on the migration from Relational Databases (RDB) to document-oriented NoSQL databases. It briefly reviews the relational model, the concepts of normal forms and (de)normalization, and then the document-oriented data model.

The relational model is composed by relations, commonly known as tables. Each table comprises rows and columns, where columns represent attributes and rows represent tuples. A primary key uniquely identifies each row/tuple within a table. Crucially, tables can establish relationships with one another through foreign keys, which are attributes that create links between tables.

The relationship has a cardinality that indicates the number of elements that are related/linked between the tables. There are three types of relationships: (*i*) *one-to-one (1:1)*, (*ii*) *one-to-many (1:N)*, and (*iii*) *many-to-many (N:N)* . *One-to-one* relationships indicate that one element is exclusively related to one element of the second table, commonly implemented by a foreign key with a unique constraint. The *one-to-many* indicates that one element is related to many elements in the second table. On the other hand, the *many-to-many* needs to have an extra table, called a join table, to implement the relation. In the join table, each row typically contains foreign keys that reference the primary keys of the tables to be linked, thus enabling the creation of connections between the data entries in these tables.

The relational data model minimizes data redundancy and dependency through normalization using normal forms. These forms are rules based on the functional dependency (FD) between a table's attributes.

An FD is described as $\alpha \rightarrow \beta$, where the set of attributes β functionally depends on the set of attributes α. For example, a dependency $rid \rightarrow name$ indicates that *name* depends on the rid (registration id) value, implying that for every instance of a rid value

[2] `github.com/ttainam/migration_demo`

[3] Demonstration: `www.youtube.com/watch?v=YqynxZp_D6w`

should have a consistent name attribute. Thus, *rid* acts as a key for the tuple <*rid*, *name*>.

There are five normal forms: 1NF, 2NF, 3NF, 4NF, and 5NF. The Boyce-Codd Normal Form (BCNF) is a correction of the 3NF. Typically, only the first three forms (1NF, 2NF, and BCNF) verify if a database is normalized. The First Normal Form (1NF) ensures that all attributes in a table store only atomic (single) values. The Second Normal Form (2NF) builds on 1NF and requires that attributes do not partially depend on the primary key. Last, the Boyce-Codd Normal Form (BCNF) corrects anomalies in 3NF by ensuring that non-key attributes do not functionally depend directly on the primary key.

An RDB that adheres to BCNF is free from redundancy and is thus considered normalized. While normalization is crucial for data integrity in RDBs, it can lead to poor query performance, especially in Big Data scenarios, due to the need for multiple table joins to retrieve data. This context brings up the concept of denormalization, which aims to improve query performance by reducing the number of tables and joins required.

Denormalization involves storing the database's logical design in a weaker normal form, relaxing compliance with 1NF, 2NF, and BCNF. Maintaining a design in weaker normal forms ensures faster query execution and more frequent transaction handling.

NoSQL database models, in general, are based on denormalization, as high availability requires related data to be grouped. The document-oriented model relies heavily on denormalization for efficient data storage.

A document-oriented database uses the document as its primary data structure [Abadi 2009]. Documents can be stored and retrieved in formats like XML or JSON and are self-descriptive and hierarchical, with structures ranging from atomic values (e.g., integers, strings) to complex ones (e.g., dates, subdocuments).

Document-oriented DBs are composed of collections of documents. Each document of a collection can have a unique structure, a flexibility not found in the uniform schema requirement of RDBs [Abadi 2009]. This allows for the common occurrence of subdocuments, such as a user document containing a list of comments.

Document-oriented models, like many NoSQL models, do not support joins. To avoid joins, document-oriented design encapsulates data from related documents within a single document, similar to denormalization in RDBs. When exporting an RDB to a document-oriented database, we denormalize the original database to avoid creating foreign key-like structures requiring joins.

3. Related Work

Several works from the state of the art based their models on an input metamodel. For example, [Karnitis and Arnicans 2015] build a tree-like structure that maps all relational objects to a metamodel. From the built metamodel, they execute queries to migrate data to the document-oriented database. However, the author's lacks to present how the approach migrates N:N, 1:N, or 1:1 relationships. They claim that the metamodel can convert data from relational to document-oriented. Using a similar strategy, [Namdeo and Suman 2021] extracts the schema from relational data and queries to propose a JSON schema but does not export the data to the new format.

Other works, such as [Chen et al. 2022] and

[de Lima and dos Santos Mello 2015], propose an approach to build document-oriented schemas from workloads. They do not show the resulting document-oriented database, and how the relationships were implemented needs to be clarified. As [Karnitis and Arnicans 2015] states, one business object can be stored in several tables, and it is hard to tell whether or not a table represents a business object or is a join table.

Zhao et al. [2014] also present an approach to convert NoSQL to Relational databases. They propose to organize the relational schema into a tree where relationships are represented as parent-child nodes. Still, they lack a discussion of the cardinality among tables, mainly the N:N type, i.e., join-tables.

Besides the trivial conversions, our work proposes an approach to deal with candidate join tables: tables created to represent N:N cardinality.

4. Rel2Doc

Unlike those presented in Section 3, our approach does not extract the relational schema before starting the migration process. It reads the schema to compute the table of foreign keys and starts migration. Our approach systematically organizes all tables using a hashing structure based on the number of foreign keys. This systematic organization provides a clear and structured view of the database, reassuring the audience of its manageability.

Algorithm 1 sketches how join tables are found. First, all two-fk-tables (*t2fk*) are retrieved, and if a given *t2fk* is not referenced by any other tables (Line 5). The number of no keys attributes (line 4) is less than a threshold (given by the user - MX_ATT parameter) (Line 7), *t2fk* is a join table. That means it will be aggregated as an object array in one of the tables with outgoing connections.

Algorithm 1: Get Join Table Candidates

```
   Input: Tables, MX_ATT
   Output: Join Table List - JTList
 1 JTList:={} /* Initialize join table list                                  */
 2 Table:=getTables2FK(Tables) /* retrieve tables with only 2 FKs            */
 3 for each t in Table do
 4 |   nbCols:=|getNoKeysColumns(t)| /* number of no key columns             */
 5 |   nbReft:=getNbReferences(t)
 6 |   nbFK:=getNbFK(t)
 7 |   if nbReft=0 ∧ nbFK = 2 ∧ nbCols < MX_ATT then
 8 |   |   JTList:=JTList ∪ t
 9 |   end
10 end
```

After finding the join tables, the migration process starts. Algorithm 2 presents the steps. Four parameters are given: information about PostgreSQL (*PGPar*) and MongoDB (*mongoPAR*), the maximum number of no keys attributes to find join-tables (MX_ATT), and whether or not the documents will have MongoDB's ObjectIDs (insObjID). Line 4 returns the join table list, and from Line 5 to 27, MongoDB collections are created from the relational database given as a parameter. Two kinds of tables are transformed into aggregation: join tables (Line 13) and tables referencing only one table (Lines 17 to 22).

We conduct an experiment to verify Rel2Doc's performance. We use three

Algorithm 2: Migrating Relational to Document-Oriented

```
   Input: PGPar, mongoPAR, MX_ATT, insObjID
   Output: MongoDB Collection Created
 1 ConnectDB(PGPar, mongoPAR)
 2 Tables:=getAllTables(PGPar.database_name)
 3 SortByNumberOfFK(Tables)
 4 Run Algorithm 1 with Tables, MX_ATT returning JTList
 5 for each table in Tables do
 6     if (table in JTList ∨ table is already aggregated) then
 7         continue
 8     end
 9     Create MongoDB Collection Col
10     refTables:=getReferencedTables(table)
11     for each r in table do
12         Insert Document D in Col
13         if (refTables in JTList ∧ r.att is a join attribute) then
14             Insert refTables.table.tuple as an aggregation in D
15             continue
16         end
17         while There is a table tr referencing t do
18             nbFK:=GetFK(tr)
19             if nbFK=1 then
20                 insert tr.tuple as an aggregation in D
21             end
22         end
23     end
24     if MongoPAR.addObjID is True then
25         Add ObjectID in D
26     end
27 end
```

databases: Movies Rental (RENTAL) with 15 tables, 31,722 tuples, and 18 FK; Telephone Voicemail Box (TVB) with 28 tables, 2,806,907 tuples, and 18 FK; and Short Message Service (SMS) with 127 tables, 5,100,409 tuples, and 247 FK. The migration process took 6.5 minutes for RENTAL, 5.77 minutes for TVB, and 1,620 minutes for SMS. It's important to note that the number of foreign keys has a significant impact on the execution time. For instance, TVB's size is nine times bigger than the rental, yet it runs faster. SMS is the largest and most numerous in FKs and the slowest to migrate.

Figure 1 shows an extract how three RENTAL tables are aggregated in the resulting document-oriented schema: *language* and *actor* became part of *film* collection, besides *film_actor* is also an aggregation built as a join-table inside *actor*.

The fundamental concept of our migration approach are the foreign keys. Experiments show the effectiveness of the method. However, it is important to notice that the document dataset generated is generic and does not consider specific details of the original application's workload. Also, the aggregation process creates data redundancy, which can be a problem in some scenarios. To mitigate redundancy problems, our approach

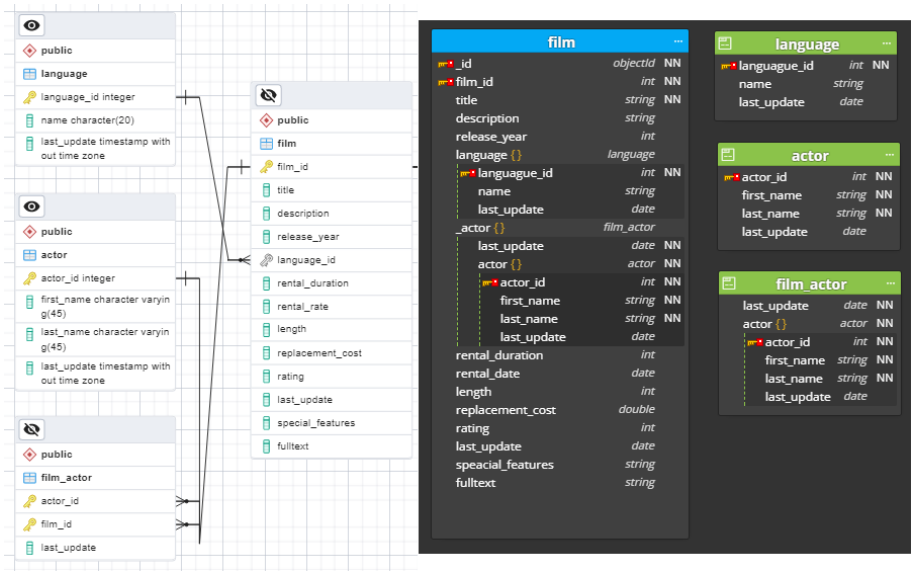

Figure 1. Extract from the RENTAL schema and the resulting MongoDB schema.

considers the aggregate of a table when the table does not have more than one reference.

5. Conclusion

In this demo, we propose a novel approach for migrating data from RDBs to document-oriented NoSQL DBs. Our method minimizes the number of collections created by considering all types of relationships and the tables involved. Keys (`K`) and foreign keys (`FK`) determine whether a table becomes a standalone collection or is embedded within another. To test our approach, we migrated data from PostgreSQL to MongoDB. The experiments demonstrated that our approach is effective in accurately exporting data, with performance primarily influenced by the database's number of `FK`s.

We intend to explore improving the way we find join tables by considering more than two tables involved in the join in future work. We also intend to enhance the approach to considering the application needs when migrating the relational data.

References

Abadi, D. J. (2009). Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.*, 32(1):3–12.

Chen, L., Davoudian, A., and Liu, M. (2022). A workload-driven method for designing aggregate-oriented NoSQL databases. *Data & Knowledge Engineering*, 142:102089.

de Lima, C. and dos Santos Mello, R. (2015). A workload-driven logical design approach for NoSQL document databases. In *Proceedings of the 17th iiWAS*, pages 1–10.

Karnitis, G. and Arnicans, G. (2015). Migration of relational database to document-oriented database: Structure denormalization and data transformation. In *7th CICN*.

Namdeo, B. and Suman, U. (2021). Schema design advisor model for RDBMS to NoSQL database migration. *International Journal of Information Technology*, 13(1):277–286.

Thalheim, B. and Wang, Q. (2013). Data migration: A theoretical perspective. *Data & Knowledge Engineering*, 87:260–278.