Information Systems 105 (2022) 101941

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Exploring data structure alternatives in the RDB to NoSQL document store conversion process

Evandro Miguel Kuszera [a,*], Leticia Mara Peres [b], Marcos Didonet Del Fabro [b]

[a] Federal University of Technology - Paraná, Dois Vizinhos, Brazil
[b] Federal University of Paraná, Curitiba, Brazil

ARTICLE INFO

Article history:
Received 22 December 2020
Received in revised form 10 August 2021
Accepted 5 November 2021
Available online 20 November 2021
Recommended by Holger Pirk

Keywords:
Relational databases
NoSQL
Metrics
Schema quality
Evaluation

ABSTRACT

NoSQL databases have emerged as an alternative to relational databases, which do not meet all the currently imposed scenarios. Large applications that handle a variety of data formats often use several types of databases and the need to migrate data between them is common. There are several approaches that perform this type of conversion. However, the process of choosing the ideal data structuring for the application requirements is not a trivial task. In this paper, we present a RDB to NoSQL conversion approach composed by steps for defining, evaluating and comparing candidate NoSQL schemas (data structuring) in relation to the applications access pattern, before migrating RDB data to NoSQL document store. We present a set of query-based metrics and scores to assist the user in the schema selection process and a framework to migrate RDB data to NoSQL format. Finally, we present experiments to evaluate the benefits of our approach and the correlation between metrics results and query implementation effort after migrating RDB to NoSQL.

1. Introduction

Most applications that manage data use relational databases (RDB) as their main persistence layer. The ACID (atomicity, consistency, isolation and durability) properties and the existence of the SQL query language are often determining characteristics for the choice of the relational model. However, relational databases have limitations to deal with applications that need to manage data at a large scale (e.g. Big Data) and without a predefined structure [1]. As an option, NoSQL databases [2] emerged to meet the requirements of these scenarios. NoSQL databases differ from RDB in terms of architecture, data model and query language, and are typically classified based on the adjacent data model used to store the data. The main four classes are: document, column family, key–value and graph-based models. NoSQL databases do not require a schema definition before writing the data, and many of them follow the schema-on-read approach. However, even without the obligation to define a schema, there is an implicit schema in the data or application code. We will use the term schema in the rest of the article to denote the structure of data that will be stored in the NoSQL database.

It is not uncommon for applications (e.g. web or mobile) to use different types of databases to achieve certain storage requirements such as consistency, availability or performance. We can also cite requirements such as schema flexibility and code maintainability, which are important in the context of application development and evolution. In these scenarios, data migration between different databases is an important task and requires the use of strategies to convert schema and data in such a way the data is stored in the proper place.

Two data models widely used in such scenarios are RDB and NoSQL document store.[1] There are different approaches presented in the literature to convert RDB to NoSQL document stores. Some of them consider just the structure of the RDB (tables and columns) in the conversion process [3,4], while others also consider the access pattern of the application [5–7]. Likewise, there are works that aimed at defining optimized schemas [7–9], requiring as input the conceptual model of the application and its workload, including expected database size and queries. However, all of these approaches have limitations in dealing with or highlighting conflicts in choosing the best schema when queries have an opposite access pattern between them. In certain cases, the expert knowledge is the only guarantee that a produced document was appropriate (or not).

Another option for converting RDB to NoSQL is through Object-NoSQL mappers (ONM), which are platforms that allow access to NoSQL databases through a standard API. There are different ONMs like Hibernate OGM [10], Impetus Kundera [11], EclipseLink [12], DataNucleus [13] and Spring Data [14] that provide support

* Corresponding author.
E-mail addresses: evandrokuszera@utfpr.edu.br (E.M. Kuszera), lmperes@inf.ufpr.br (L.M. Peres), marcos.ddf@inf.ufpr.br (M. Didonet Del Fabro).

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

https://doi.org/10.1016/j.is.2021.101941
0306-4379/

for CRUD operations and, in some cases, support for complex queries against different NoSQL databases. Migrating from RDB to NoSQL using ONMs can be direct, where ONM is configured to read the data from the RDB and persist in the NoSQL database, following the standard ONM API format for persisting the data, or indirect, where the developer defines and performs transformations on the data before persisting it in the NoSQL database, which is not always a trivial task.

In this paper we present a RDB to NoSQL document store conversion approach composed by steps for defining, evaluating and comparing candidate NoSQL schemas, and finally migrating data from RDB to NoSQL. In the first step, the user defines a set of candidate NoSQL schemas and the application access pattern using the RDB metadata as input. The user can define a schema manually or use a schema produced by existing conversion approaches. We use DAGs (Directed Acyclic Graphs) as an abstraction to represent the structure of the schema entities and the query access pattern. In the second step the approach calculates the coverage provided by each schema in relation with the set of input queries. To do that, we present a set of query-based metrics and scores to measure the coverage between schema and queries. Currently, the approach is limited to read queries, which are previously known to the user. In the third step the user evaluates the candidate NoSQL schemas through metrics and scores, calculated in previous step. Steps 1 to 3 are performed using our tool *QBMetrics*. Finally, in the last step the selected schema is sent to *Metamorfose*, our data transformation framework, that reads RDB data, transforms and persists in NoSQL format. It enables having output documents better fitted to the target application.

The goal of our approach is to identify the NoSQL schema that has the greatest coverage against the query access pattern. When the schema and query have matching access patterns, the number of operations required to retrieve and format the data is less, which positively affects the execution time of queries. However, we do not intent to identify the schema that has the best performance for executing queries, in which it is necessary to consider aspects such as size and cardinality of document collections, which are not always available when migrating the data and are not explored in this work. Our approach is designed to perform a one-way data migration, and we do not address schema and query changes that might arise after migrating data from RDB to NoSQL. There are approaches that address these aspects, such as [15] which defines operators to perform the schema evolution of relational bases, and [16] which addresses the schema evolution of NoSQL bases as the application evolves, which are beyond the scope of our work.

To evaluate our approach we performed a conversion from a relational database to MongoDB. The choice of MongoDB was motivated by the fact that it is a document-oriented database widely used by data-intensive applications.[2] In addition, MongoDB provides an expressive query language called *MongoDB Aggregation Pipeline*, which allows us to perform complex queries on document collections. Based on the set of operators available by the MongoDB query language we define a set of query implementation guidelines that serve as a basis to measure the effort of implementing queries on different NoSQL schemas in an objective way. Furthermore, we believe that the language provided by MongoDB is a promising solution, given that there is no standard language for document-oriented database access.

The overall approach is based on our previous work [17–19], where parts of the process to convert RDB to NoSQL document store is presented. The extensions and contributions of this paper are summarized as follows:

- A RDB to NoSQL document store conversion approach composed by steps for defining, evaluating and comparing candidate NoSQL schemas, and finally migrating data from RDB to NoSQL.
- A new metric (*FArray*) used to identify queries in which its filters (or predicates) are located in arrays of embedded documents, which has a negative impact on query implementation.
- A set of guidelines for implementing input queries (application access pattern) using MongoDB Aggregation Pipeline. This framework allows expressing a query as a pipeline, consisting of a set of stages that represent operations on documents.
- A set of experiments used to evaluate our approach, considering schemas generated from different RDB to NoSQL document strategies presented in the literature. In this paper, we extend our previous evaluation and define three distinct scenarios, considering the i) number of collections required to answer the queries, ii) the schema with best access pattern coverage, and (iii) the selection of preferred queries.
- A detailed comparison of the query implementation effort for each schema, based on the number of lines of code (LoC) and number of stages required to implement the queries in MongoDB using our guidelines. We also present a correlation with our query-based metrics results.

The remainder of this paper is organized as follows: Section 2 presents the motivation and related work about RDB to NoSQL document conversion scenario. In Section 3 we present our RDB to NoSQL conversion approach and how we represent NoSQL schemas and queries as DAGs. Sections 4 and 5 present our query-based metrics and scores, and our data transformation framework, called *Metamorfose*. In Section 6 we show guidelines to implement the input queries in the NoSQL document store, considering the structure of schema and query DAGs. Section 7 deals with experiments and results. Finally, conclusions and future work are provided in Section 8.

2. Motivation and related work

Before presenting our RDB to NoSQL approach, let us introduce a RDB to NoSQL document conversion scenario which is used in the remainder of this article. Fig. 1 shows a RDB schema composed of seven tables [20].

There are different ways to convert the RDB tables to NoSQL documents. One way is to migrate all tables to collections of documents and use references to represent the relationships. Another way is to de-normalize some tables and to use embedded documents or arrays of embedded documents to represent the relationships. We can manipulate the data in NoSQL document as aggregates [2], which consist of a collection of related objects treated as a single unit, which can be retrieved from the database with just one read operation. Furthermore, there is no standard to define the composition of an aggregate, being dependent on how the application accesses the data. In contrast, the concept of aggregates is not supported by relational databases, which are based on tables and relationships between tables, and data should be normalized to avoid redundancy. The relational model allows us to manipulate data in different ways, based on operations on tables, but it does not allow us to nest the data and treat it as aggregates, which is a nice feature when we want to store related data close together to optimize read operations. The conversion of RDB tables and relationships into collections of documents is a process often empirical and depends on several aspects, including the access pattern of the application, the size of generated

[2] https://db-engines.com/en/ranking.

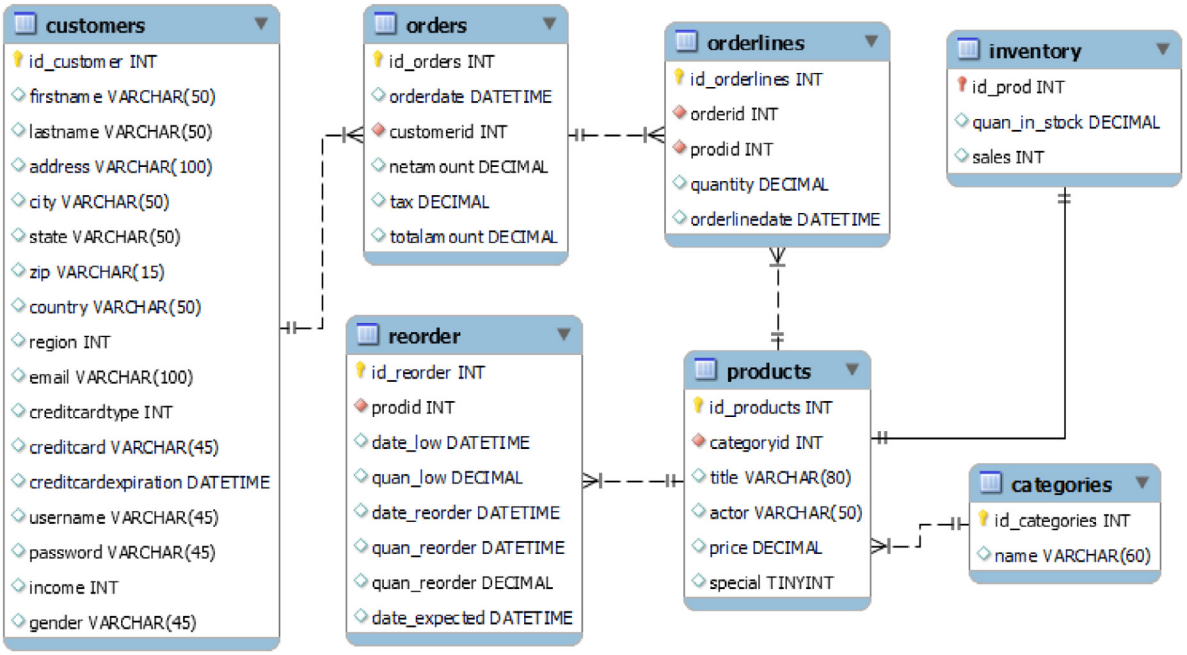

Fig. 1. Example RDB schema.

entities, the entity nesting level and also aspects of application as maintainability and code readability.

Although NoSQL document stores do not have an explicit and fixed schema, a document has a structure that needs to be known in advance to allow the formulation of queries. This structure is used to represent the schema of a database entity. Fig. 2 shows two schema possibilities for converting the RDB tables to NoSQL documents. We represent the schemas through DAGs, in which there is a root vertex (Level 1) that represents a collection of documents and the remaining vertices are embedded documents. *Schema 1* has four collections: *Customers*, *Orders*, *Orderlines*, *Inventory* and *Reorder*. The *Customers* and *Reorder* collections store simple objects with no other nested objects. The *Orders* collection stores *Orders* objects composed of an array of *Orderlines* objects. The *Orderlines* collection stores *Orderlines* objects composed by one *Product* object, which in turn embedded one *Inventory* object. The last collection stores *Inventory* objects that embedded one *Product* object. In contrast, *Schema 2* has a different structure.

Both *Schema 1* and *Schema 2* differ in terms of structure, with different entity nesting order and depth of the DAG. The question is how to choose the best one, considering that we can have queries where the access pattern corresponds to the schema and queries that the access pattern is inverted in relation to the schema, impacting the implementation effort and execution time of the queries. The process of choosing the schema is not easy and often is performed manually by the expert user, who knows the requirements of the application (e.g. access pattern). So, in this work we present an approach to help in the process of choosing the suitable NoSQL schema to migrate the RDB data. In the following sections we will present the works related to our approach.

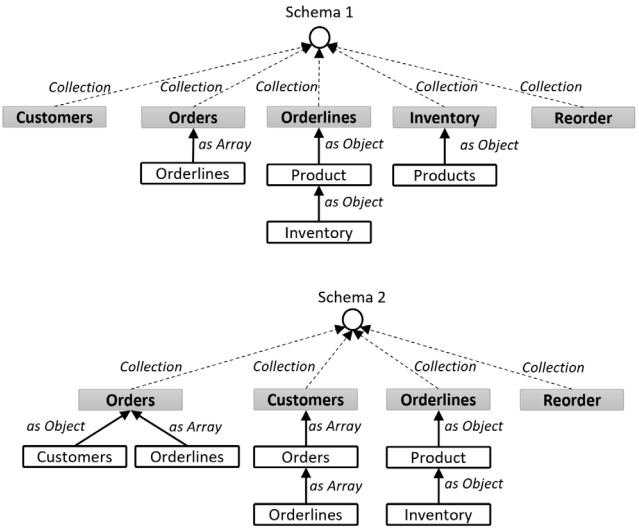

Fig. 2. Two schema possibilities for converting the RDB tables to NoSQL documents.

2.1. RDB to NoSQL document approaches

Different approaches have been presented to convert RDB to document-oriented NoSQL databases [3–7]. These approaches define different ways to convert relational data to nested data. [3,4] are automatic solutions that receive the RDB metadata and E-R diagrams as input, and apply rules that evaluate the dependencies between tables and the direction of relationships (PK-FK order) to decide how to nest data. The approaches [5–7] are semi-automatic and the user provides a kind of table classification to guide the translation. This table classification is used to decide which tables should be embedded together. In [5] the RDB tables should be classified as *main*, *subclass*, *relationship* and *common* by the user. Table classification is used by an algorithm that nest subclass and common tables in the main table, and using references to translate relationship tables. The approach from [6] has a different classification composed of four classes: *codifier*, *simple entity*, *complex entity* and *N:N-link*. An algorithm receives the classified tables and a base table, called "table in focus", and builds a tree that represents the structure of the document. In [7], tags are used to classify RDB tables and relationships, according to the RDB data and characteristics of the queries. The tags are *frequent join*, *modify*, *insert* and *big size* and are applied over the E-R diagram of the RDB. From the annotated E-R diagram, an

algorithm builds a NoSQL schema using embedded documents or references to nest entities.

All these approaches provide different ways to convert RDB to NoSQL document stores, but they do not define ways to evaluate if the produced data and corresponding schema meets the application's requirements, as access pattern or query implementation effort.

2.2. NoSQL Schemas

Our approach has relation to works that define schemas for NoSQL document store. The work [21] defines the NoSQL Abstract Model (NoAM), that uses application's functional requirements to model aggregates (set of entities) to create the NoSQL schema. NoAM provides guidelines to consider application requirements such as scalability, performance and consistency during the modeling process. The work from [22,23] present approaches to transform a conceptual model (UML class diagram) into a NoSQL physical model. Additionally, [23] use a set of queries provided by the user to guide the generation of the schema, storing together related entities. Different from our approach, these works do not provide means to evaluate different schema options in relation to a set of queries that represent the access pattern of the application. Works were defined to generate optimized schemas for column-family [24,25] and document [7–9] oriented NoSQL databases, based on the target application workload (queries and, in some cases, database size.). Our approach can be used to evaluate and compare the output schemas of these works, before migrating data from RDB to NoSQL.

In [26] guidelines are presented for migrating RDB to column-family NoSQL databases. The guidelines address the use of denormalization, translation of SQL queries to NoSQL, use of indexes and column families. However, the work explores only one alternative of a denormalized schema in the experimental evaluation carried out, discarding other alternatives that may impact the results. Guidelines-defining works can be used in our approach to create a set of NoSQL schemas to be evaluated and compared.

2.3. Wrapper-based strategies and object-NoSQL mappers

Another way to migrate RDB to NoSQL is through wrapper-based strategies and object-NoSQL mappers. In [27] a survey is presented on approaches based on wrappers, which map the SQL commands to operations supported by the target NoSQL base. These approaches allow the rapid migration of applications based on relational databases to NoSQL, but limit the options for structuring data (e.g. different forms of nesting) depending on the strategy used to map SQL to NoSQL [28–32]. Our approach focuses on the migration of relational databases to NoSQL databases, where the user can take advantage of all the data structuring flexibility provided by NoSQL databases.

Object-NoSQL mappers or ONMs are data access middleware that allow the developer to access different NoSQL databases through a standard access interface, hiding the complexities of the underlying database. Typically, ONMs provide CRUD operations and limited query features. ONMs can be used to migrate data from RDB to NoSQL, in which the developer should implement the source and target model, and the transformation logic to adapt the RDB data to the target NoSQL model. Uta et al. [33] presented an evaluation of five ONMs (Hibernate OGM [10], Kundera [11], DataNucleus [13], EclipseLink [12] and Morphia [34]), exploring their capabilities and performance overhead regarding the use of native drivers. Rafique et al. [35] presented an evaluation of three ONMs (Playorm [36], String Data [14] and Kundera [11]) in terms of performance overhead and the cost of porting a prototype application from MongoDB to Cassandra and vice-versa, using and not using ONMs. However, the evaluation only considers the LoC necessary to port the application data model to the target NoSQL, without exploring different ways to structure the data in the target NoSQL. In both works, the results showed that ONMs have good support for CRUD operations and provide application portability and interoperability between NoSQL databases. However, there are limitations on query expressiveness and schema evolution support.

Our approach differs from these works because we aim to evaluate different ways to structure the data (use of nesting, order of nesting, use of referencing, etc.) against the access pattern of the application, and select one schema to execute the data migration from RDB to NoSQL. Furthermore, our approach automatically generates operations to transform the RDB data to the target NoSQL model.

2.4. Metrics for evaluating schemas

A metric is a numerical value for an attribute of a given entity used for evaluation or comparison with previously established values or standards. In the literature, different metrics have been proposed to assess the quality of data and corresponding schemas.

For relational [37] and multidimensional [38] schemas, there are objective metrics that count the number of elements in the conceptual schema (entities and relationships, for example), and subjective metrics that involve project stakeholders. These metrics are used to assess whether the schema is considered complete, simple, correct, flexible, easy to implement, among others. Metrics for evaluating XML schemas are usually adaptations of the metrics defined in software engineering. They measure aspects such as structure (number of elements, attributes, entities and notations), clarity, optimality, minimalism, reuse and flexibility of XML schemas [39,40].

The metrics commented above can be used or adapted to assess how the data is structured in NoSQL databases. The work by [41] presented eleven metrics to evaluate data structuring alternatives to assist the user in the process of selecting the most appropriate document store schema. They defined the following metrics: (i) *colExistence* and *DocExistence* to verify the existence of elements in the schema, (ii) *colDepth*, *globalDepth*, *docDepthInCol*, *maxDocDepth* and *minDocDepth* to calculate the depth of schema elements, and (iii) *docWidth*, *refLoad*, *docCopiesInCol* and *docTypeCopies* to calculate the size and number of occurrences of schema elements. These metrics focus on aspects related to the structural complexity of the schema, without considering the access pattern of the application, which is a key aspect in the selection process of the schema. Our approach uses DAGs to represent schemas, similarly to the tree structure used in [41], but differs in the purpose of the metrics, which we are focused on the application's access pattern.

3. Our RDB to NoSQL approach

In this section we present our RDB to NoSQL conversion approach. The approach is composed by steps for defining, evaluating and comparing candidate NoSQL schemas in relation to the application's access pattern.

Fig. 3 shows the steps and architecture of the approach. In step **1. Definition of Schemas and Queries** the user defines a set of candidate NoSQL schemas and the access pattern. We use DAGs as an abstraction to represent the structure of the schema entities and to represent the query access pattern. In step **2. Calculation of Metrics and Scores** is calculated the coverage provided by schema in relation to the set of input queries. In step **3. Schema Evaluation and Selection** the user evaluates the candidate NoSQL

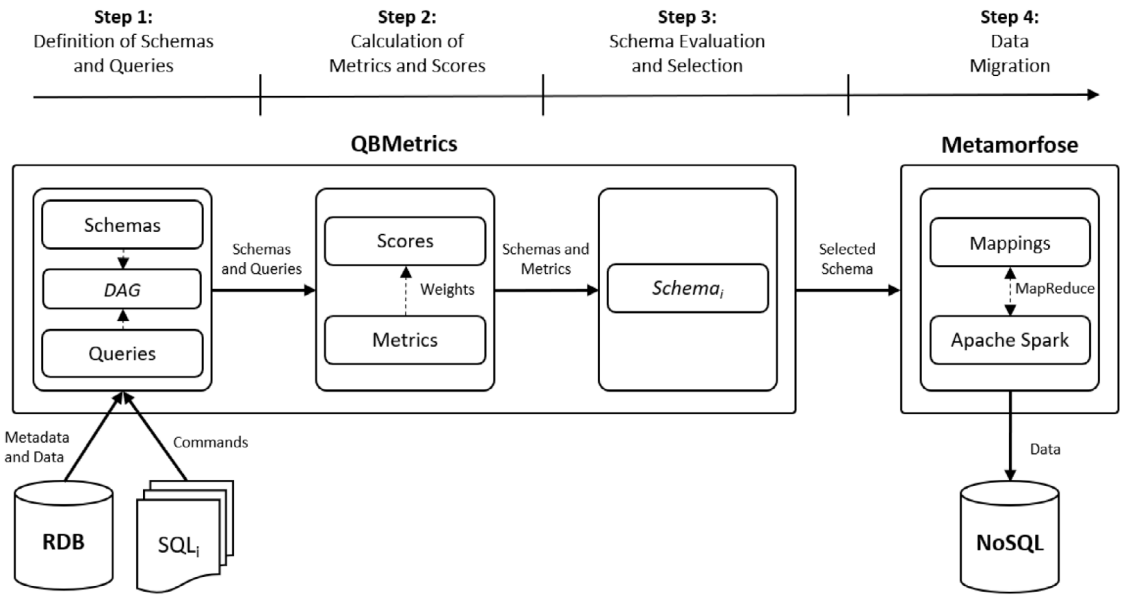

Fig. 3. Steps and architecture of the RDB to NoSQL conversion and data migration approach.

schemas through metrics and scores, calculated in previous step. Steps 1 to 3 are performed using our tool *QBMetrics* [19]. Finally, in step **4. Data Migration** the selected schema is sent to *Metamorfose* [17], our data transformation framework. The next sections show an overview of the steps of the approach.

3.1. Step 1: Definition of candidate schemas and queries

In this step we define one or more candidate NoSQL schemas and the set of queries that represent the application's access pattern. It is worth noting that in a context of conversion from RDB to NoSQL the application queries are known and defined as SQL commands. The aim is to provide a way to represent a NoSQL schema and compare it to the application's access pattern. The user can define a schema manually from scratch or use a schema produced by existing conversion approaches as a template.

In the following we present more details about schema and query definition through DAGs.

3.1.1. NoSQL Schema represented through DAGs

We use DAGs as an abstraction to represent document-oriented NoSQL schemas. A NoSQL schema is defined as a set of entities represented as DAGs. An entity represented as a DAG has a tree structure and relates one or more tables of the input RDB. A DAG is defined as $G = (V, E)$, where the set of vertices V represents the RDB tables and the set of edges E the relationships between tables. The vertices encapsulate the metadata of the respective RDB table, including table name, fields and primary key. The edges encapsulate the metadata of the relationship between two tables, including the primary and foreign keys and which table is on side one or side many of the relationship.

Fig. 4 shows four DAGs (*Customers*, *Orders*, *Orderlines* and *Inventory*) built from the RDB of Fig. 1. The root vertex (gray background color) represents the target NoSQL entity, the other vertices (white background color) represent the nested entities and the edge direction indicates the nesting direction. For example, in DAG *Orders*, *Products* is nested with *Orderlines* which in turn is nested with *Orders*. The target NoSQL entity is materialized as a collection of documents, and the cardinality of relationships (noted on the edges) is used to guide how the RDB data will be transformed to generate the NoSQL entity.

Fig. 5 shows a document-oriented NoSQL schema based on DAGs of Fig. 4. Each DAG is mapped to a collection of documents, where the root vertex represents the first level of the collection and the other vertices represent the nested entities. The direction and cardinality of the edges define the direction and type of nesting between entities. The types of nesting are: embedded document when the relationship is *1:1* or *N:1*, and *array* of embedded documents when the relationship is *1:N*. In this way, a document-oriented NoSQL schema is defined as $S_{doc} = \{DAG(c) | c \in C\}$, such that C is the set of document collections.

3.1.2. Queries represented through DAGs

We use DAGs to denote the data access pattern of queries. We consider only read queries, defined by SQL commands in SELECT-FROM-WHERE format, with support for inner join, left join and right join. Our approach does not support queries with sub-queries, union, aggregations and stored procedures, which we intend to explore in future work.

The set of input queries is defined as Q, and a query $q \in Q$ is defined as $q = (V_q, E_q)$, where V_q is a set of vertices, representing the query tables, and E_q is a set of edges, representing the join conditions between query tables. The query predicates are encapsulated in its respective vertex, and they are defined as $F_q = \{v_i \in V_q\}$, in which v_i represents the RDB table related to the predicate. We represent the query access pattern as a DAG, defining the joins between tables and on which tables the predicates are applied. We create two rules to convert a SQL statement into a DAG.

- **Rule 1**: if the statement has only one table, a DAG with one vertex representing the table is created.
- **Rule 2**: if the statement has two or more tables, it is necessary to define which table is the root vertex of the DAG. The other tables are included into the DAG according to the join conditions.

To identify the join condition in Rule 2, we parse the SQL statement. Then, we apply one of the following subrules to determine which table is the root vertex:

- **Rule 2.1**: if it is a left join, returns the leftmost table in the FROM clause.

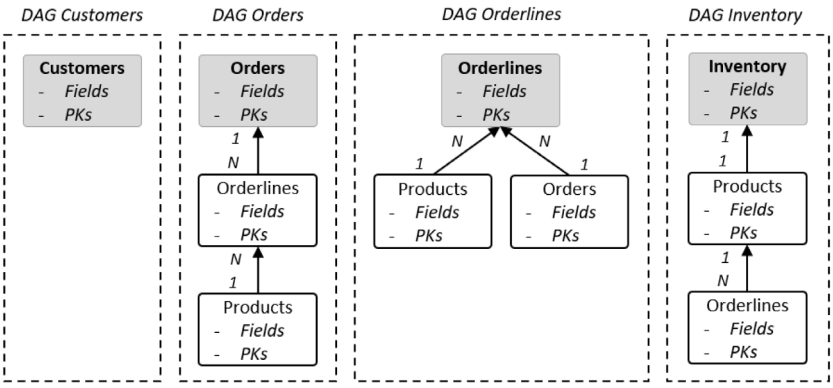

Fig. 4. Set of DAGs used to represent NoSQL entities..

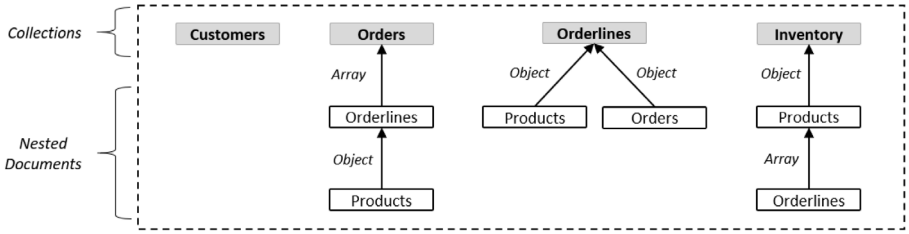

Fig. 5. Document-oriented NoSQL schema composed of four collections of documents..

- **Rule 2.2**: if it is a right join, returns the rightmost table in FROM clause.
- **Rule 2.3**: if it is an inner join, returns the first table in the FROM clause.

The purpose of the rules is to establish a uniform translation mechanism to represent the access pattern of queries as DAGs, considering the hierarchical structure of the document-oriented data model. When we find an SQL command where there is *Customer* LEFT JOIN *Order*, we assume that we should look for a NoSQL schema that has a collection of customer, where each customer document has an array of orders. An SQL command with *Customer* RIGHT JOIN *Order* represents the reverse path. On the other hand, when we have *Customer* INNER JOIN *Order*, we do not have join orientation information and we choose to select the first table right after the FROM clause to define the DAG root. Alternatively, we could check for the presence of the primary key in the join attributes and use the table that contains it as the root DAG, but this strategy would not work correctly for join queries with three or more tables related by primary keys. To illustrate the use of the rules, let us consider the six queries below, which use different numbers of tables and join conditions.

```
q1: select * from orders where id_customer = 1;
q2: select * from orders left join orderlines on id_order =
      orderid left join products on id_prod = prod_id where
      products.price > 29.9;
q3: select * from products left join orderlines on products.
      id_prod = orderlines.prod_id where orderlinedate > '
      01/01/2009';
q4: select * from inventory inner join orderlines on inventory.
      prod_id = orderlines.prod_id;
q5: select * from orders right join customers on id_customer =
      customerid where id_customer = 1 and orderdate > '
      01/01/2009';
q6: select * from orders o left join customers c on o.customerid
       = c.id_customer left join orderlines ol on ol.orderid = o.
      id_order where id_customer = 1;
```

We define a parser that applies the rules and creates a DAG for the input query. Fig. 6 shows the set of DAGs generated after applying the above rules on the SQL queries. *F* represents the query predicates (filter) location in the query DAG.

As a result of this step, the access pattern of schemas and queries are defined as DAGs and are used in the next step to calculate metrics and scores.

3.2. Step 2: Calculation of metrics and scores

In this step the set of candidate NoSQL schemas and the set of queries defined in the previous step. Six metrics have been defined to measure the coverage that a NoSQL schema provides for the application's query set. As schemas and queries are defined through DAGs, it is possible to measure coverage objectively. In addition to the metrics, two scores were defined to assess and compare schemas. The first one defines a score per query and is called *Query Score* (*QScore*). It allows us to combine one or more related metrics and define weights to prioritize the importance of specific metrics. The second is called *Schema Score* (*SScore*) and defines a score considering the coverage of the schema for all queries, by metric.

3.3. Step 3: Schema evaluation and selection

In this step the user performs the analysis and comparison of schemas using the *QBMetrics* tool. *QBMetrics* provides a set of reports in which the user can view the values of the metrics and scores by schema, by query and by metric. The results of the metrics and scores are used to identify which queries are covered and which are not covered by the schema, which queries require two or more collections of documents to produce a result, or to calculate the coverage that the schema provides considering all queries. The user can define different weights for the metrics and queries, prioritizing a certain type of access pattern or prioritizing certain queries (e.g. frequently performed queries).

The goal is to support the user in the process of evaluating and selecting the NoSQL schema that best suits the application requirements.

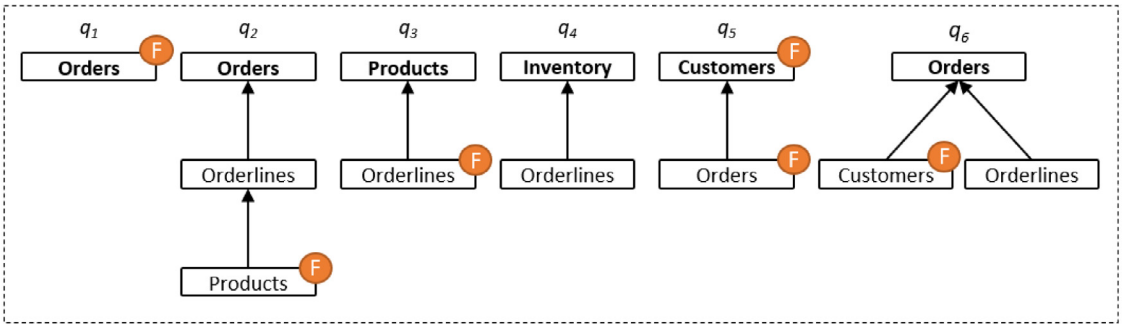

Fig. 6. Set of SQL queries represented as DAGs after applying the conversion rules. *F* represents the query predicates (filter) location in the query DAG.

3.4. Step 4: Data migration

The last step is the migration of data from RDB to NoSQL. The NoSQL schema selected in the previous step is used as an input to the data migration process. Through the *QBMetrics* the user exports the selected schema and sends it to *Metamorfose*, our data transformation framework. The exported schema encapsulates connection parameters from the input RDB, including the server address, username, password and database name.

Metamorfose converts the selected NoSQL schema into commands to read data from the RDB, transform and persist in NoSQL format. These commands encapsulate mappings between the source and destination schema fields, including data transformation functions. Finally, *Metamorfose* performs these mappings using the *map* and *mapreduce* functions implemented in *Apache Spark* [42].

4. Evaluating NoSQL schemas with query-based metrics and scores

This section presents our query-based metrics and scores (steps 2 and 3 of our approach). The metrics and scores are used to measure the coverage that a NoSQL document schema has in relation to the application's access pattern. In this work, the access pattern is represented by the set of paths from query DAG, and the coverage is defined as the correspondence between vertices and edges of the schema and query DAGs. Although the metrics have already been presented in [18], where we performed experiments to evaluate their applicability, in this section we will introduce them again and also introduce a new metric called *FArray*.

4.1. Metrics

The metrics measure the coverage provided from NoSQL schema in relation to query set. To make the query-based metric definitions more clear, we use the following terms: $c \in C$, in which C is the set of collections of the schema, V_q, V_s and V_c are the vertex set of a given query, schema and collection (or DAG), respectively. P_q, P_s and P_c are the path set (all paths from root to leaves) of a given query, schema and collection, respectively. Following are the metric definitions.

$$DirEdge(c, q) = |(E_{dq} \cap E_{dc})|/|E_{dq}| \quad (1)$$

$$AllEdge(c, q) = |(E_q \cap E_c)|/|E_q| \quad (2)$$

$$Path(c, q) = |(P_q \cap P_c)|/|P_q| \quad (3)$$

$$existSubPath(qp, P_c) = \begin{cases} 1 & \text{found } qp \text{ as subpath in } P_c \\ 0 & \text{not found } qp \text{ as subpath in } P_c \end{cases}$$

$$SubPath(c, q) = \frac{\sum_{i=1}^{|P_q|} existSubPath(qp_i, P_c)}{|P_q|} \quad (4)$$

$$existIndPath(qp, P_c) = \begin{cases} 1 & \text{found } qp \text{ as an indirect path in } P_c \\ 0 & \text{not found } qp \text{ as an indirect path in } P_c \end{cases}$$

$$IndPath(c, q) = \frac{\sum_{i=1}^{|P_q|} existIndPath(qp_i, P_c)}{|P_q|} \quad (5)$$

$$ReqColls(q) = min(createCollectionPaths(q)) \quad (6)$$

$$FArray(c, q) = \begin{cases} 1 & \text{if any } q.filter \text{ is located in array field of } c \\ 0 & \text{if any } q.filter \text{ is not located in array field of } c \end{cases} \quad (7)$$

DirEdge (Eq. (1)) measures query edge coverage against the edges of a given schema collection, considering the direction of edges (e.g. $a \rightarrow b$). E_{dq} and E_{dc} denote the set of query and collection edges considering the direction of the edges. *AllEdge* (Eq. (2)) measures edge coverage between the query and schema collection, regardless of edge direction (e.g. $a \rightarrow b$ or $a \leftarrow b$). E_q and E_c denote the query and collection edges, respectively.

Path (Eq. (3)) metric measures the coverage of query paths in relation to the collection paths. It is worth noting that a query may have one or more paths. *SubPath* (Eq. (4)) metric checks if the query paths are present in the collection as subpaths. We define the *existSubPath* function that receives as parameters a query path ($qp \in P_q$) and a set of paths, where the set of paths is the paths of a given collection (P_c). The function returns 1 if the query path was found or 0 if it was not found as a subpath. *IndPath* (Eq. (5)) metric checks if the query paths are present in the schema as indirect paths. An indirect path is a route between two or more vertices where there are intermediate vertices separating them. To find indirect paths in the schema we define the function *existIndPath*, that receives as parameters the query path (qp) and a set of collections' paths (P_c). If there is an indirect path in the collection that matches the query path, the function returns 1, otherwise it returns 0.

It is possible to calculate the schema coverage for the metrics *DirEdge*, *AllEdge*, *Path*, *SubPath* or *IndPath*. To do that, we apply the metric for all $c \in C$ and the maximum value found is defined as the coverage for the schema.

ReqColls (Eq. (6)) metric returns the smallest number of collections of schema required to answer a given query. We define the function *createCollectionPaths*(q), that returns a set of paths that consists of collections that have the entities required to answer the query. *FArray* (Eq. (7)) metric returns 1 if any filter of the query is locate in an array of embedded document in the collection. Otherwise, *FArray* return 0.

The metrics presented above enable to independently evaluate the queries. In the next section we describe how to combine them to provide a broader evaluation.

4.2. Scores

We define two scores to combine the metrics for measuring the overall coverage of a schema with respect to a set of

input queries. The first one is a score per query called *Query Score* (*QScore*), that yields a score per metric, or per combination of related metrics. *QScore* enables to set up weights to prioritize the importance of specific metrics. Second score is called *Schema Score* (*SScore*) and calculates an aggregate value by metric, considering all the input queries.

QScore is calculated for a given metric and a given query *qi*. For metrics *DirEdge*, *AllEdge*, *ReqColls* and *FArray*, the *QScore* is the same value returned by the metric:

$$QScore(DirEdge, q) = DirEdge(q) \tag{8}$$

$$QScore(AllEdge, q) = AllEdge(q) \tag{9}$$

$$QScore(ReqColls, q) = ReqColls(q) \tag{10}$$

$$QScore(FArray, q) = FArray(q) \tag{11}$$

We define a single *QScore* value for the metrics *Path*, *SubPath* and *IndPath*. It is called *QScore*(*Paths*) and returns the highest value among the three metrics:

$$path_v = Path(q_i) * w_p \tag{12}$$

$$subpath_v = (SubPath(q_i) * w_{sp})/depthSP(q_i) \tag{13}$$

$$indpath_v = (IndPath(q_i) * w_{ip})/depthIP(q_i) \tag{14}$$

$$QScore(Paths, q_i) = max(path_v, subpath_v, indpath_v) \tag{15}$$

The weights w_p, w_{sp} and w_{ip} are used to set up a priority between *Path*, *SubPath* and *IndPath* metrics and are configured by the user, with values ranging between 0 and 1. When it is not desired to define different priorities, the user assigns weight equal to 1 to all of them. Path depth is used as a sort of penalty when matching occurs deeper in the DAG (or tree). (Eq. (15)) returns the best coverage when the schema has redundant entities stored at different collections of documents and at different depths. We define this method inspired by the results of [43], that show the negative impact of accessing data stored at different levels (depth) of the collection.

SScore (Eq. (16)) is the sum of the *QScore* values for all the queries, for a given metric (except *ReqColls*). Each query q_i has a specific weight w_i, and the sum of all w_i is equal to 1. For the *FArray* metric the calculation is different, in which the weight of the queries is fixed and defined as $wi = 1/|Q|$, because this metric aims to identify whether the query filter is located in an array field, regardless of the priority of the query. Similar to *QScore(Paths)*, we define *SScore*(*Paths*, *Q*), which is the sum of corresponding *QScore*(*Paths*, q_i).

$$SScore(metric, Q) = \sum_{i=1}^{|Q|} QScore(metric, q_i) * w_i \tag{16}$$

The calculation of *SScore*(*ReqColls*) (Eq. (17)) is inspired on the schema minimality metric presented in [44]. It is a ratio between the number of queries and the number of collections required to answer them. A schema that answers each input query through only one collection has *SScore*(*ReqColls*) equal to 1 and it decreases when the number of collections increases. *NC* is the number of collections required to answer all input queries, which is the sum of all *QScore*.

$$NC = \sum_{i=1}^{|Q|} QScore(ReqColls, q_i)$$

$$SScore(ReqColls, Q) = \frac{|Q|}{NC} \tag{17}$$

Through these scores we can evaluate and compare candidate NoSQL schemas related to access pattern of application. *QScore* shows the coverage provided by schema per metric and query, enabling identify which queries require the most attention or are not covered by the schema. The *SScore* field provides an overview of how well the schema fits the query set. Through the definition of weights for metrics and queries, it is possible to configure different evaluation scenarios, and use the results obtained to rank candidate schemas in relation to the application's access pattern. In addition, since the metrics are not independent, we do not define a single expression to calculate the overall score of the schema.

5. Data migration with metamorfose framework

This section details the step four of our RDB to NoSQL approach. Fig. 7 shows an overview of the process to translate the NoSQL schema (set of DAGs) to data transformation commands that are executed by Metamorfose, our data transformation framework. The migration process has been presented in a previous work [17]. It is composed by *Command Generator*, *Command Executor* and *Metamorfose* components, detailed below.

5.1. Command generator

This component is responsible for converting each DAG into a set of commands to read RDB data, transform and persist as JSON objects. Fig. 8 shows examples of DAGs supported by the *Command Generator*. Each DAG represents a NoSQL entity, where the root vertex defines the name of this entity. For instance, the DAG1 is composed of two vertices, *Table A* and *Table B*, and in this case the instances of *Table B* (many side) will be nested in the respective instances of *Table A* to generate the new NoSQL instance. In DAG2, the instances of *Table A* (side one) will be nested in the respective instances of *Table B*. In DAG3 there are two vertices representing *Table A*. This situation is allowed to enable the generation of redundant schemas. The instances of *Table A* will be nested in *Table B*, and then in *Table D*. Then, the resulting instances of *Table B* and *Table D* will be nested in *Table C*. DAG4 has only one vertex, and in this case, the instances of *Table D* will be converted directly to the target NoSQL model format.

The format of the nesting is dependent on the direction of the relationship (1-1, 1-N, N-1). *M:N* relationships are not supported directly in the DAG, but can be represented as two *1:N* relationships.

We create an algorithm that receives as parameter the DAG and traverses all paths from the leaf vertices to the root vertex and returns a command that encapsulates the operations to embed the leaf vertex in the successor vertex as an object (if *N:1*) or as an array of object (if *1:N*). A command encapsulates operations for nesting the leaf vertex in the successor vertex and is defined as *command* = (*joinSpec*, *Maps*, *function*), where *joinSpec* is the join operation between tables (vertices) that are denormalized, *Maps* is the set of mappings to transform the data, and *function* is the transformation function that will be call (*Map* or *MapReduce*). After, leaf vertex is removed from the DAG. This process is repeated until the DAG is reduced to one vertex that represents the target NoSQL entity. Fig. 9 illustrates how the algorithm traverses the DAG3 to create the commands *cmd*1 to *cmd*4.

Command Generator component provides a set of strategies to convert relational data to nested data, encapsulated in the *JScriptGenerator*, an UDF generator based in DAG specification. If desired, the generated UDFs can be edited and customized by the data transformation logic, adding or removing fields, defining how null values are handled, and so on. By default, all fields from RDB tables are migrated to the NoSQL database, even if the field value is null. Our approach can be extended by adding new data conversion strategies in the *JScriptGenerator*.

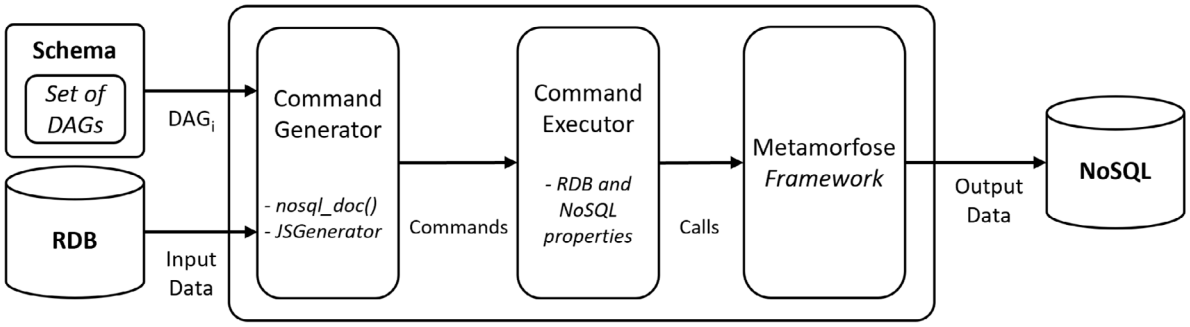

Fig. 7. Data conversion and migration process with Metamorfose framework.

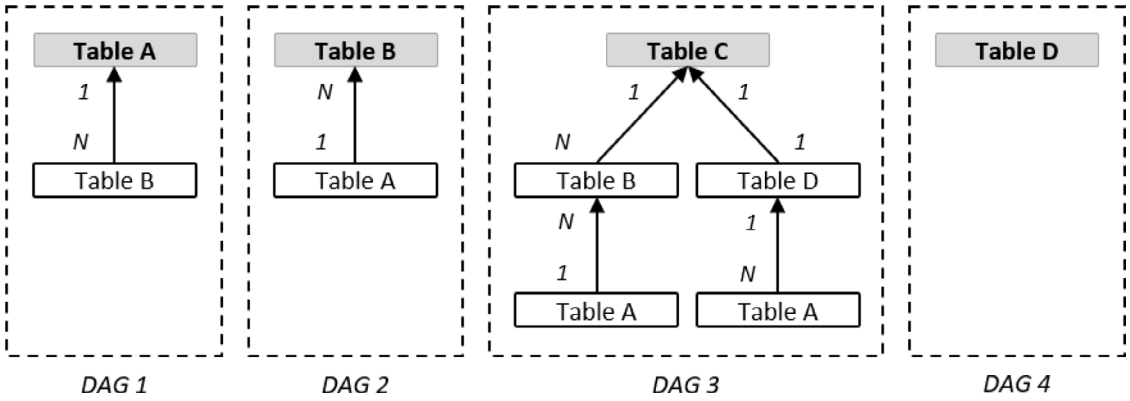

Fig. 8. Examples of DAGs supported by *Command Generator* component.

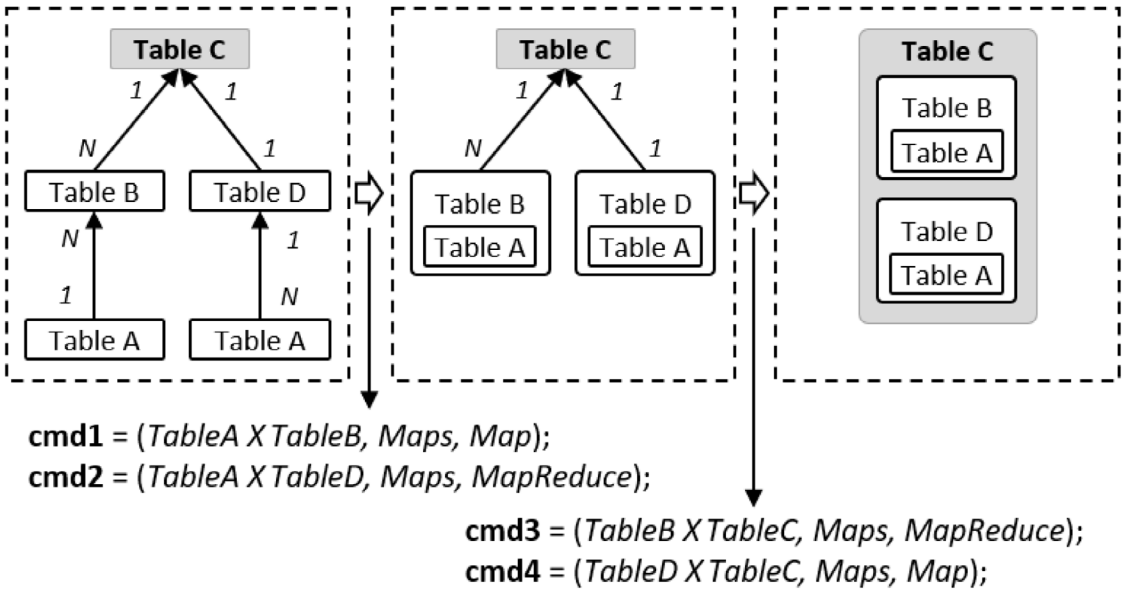

Fig. 9. Example of how the algorithm goes through *DAG3* and creating transformation commands *cmd1* to *cmd4*.

5.2. Command executor

The *Command Executor* component receives a set of commands generated in the previous step, by *Command Generator* component. This set of commands is converted to *Load*, *Map*, *MapReduce* and *Persist* Metamorfose functions. In addition, the *Command Executor* component encapsulates connection parameters with the relational database and NoSQL database, and controls the execution order of the data transformation process.

5.3. Metamorfose framework

Fig. 10 presents the architecture of the Metamorfose, our data transformation framework based on *Apache Spark*. It provides functions for loading, filtering and persisting entities. The functions *Map* and *MapReduce* are used to transform entities, according to mappings and user-defined functions (*Mapping Definitions*). Metamorfose keeps the loaded or transformed entities in the *Entity Set* component, allowing to execute chains of transformations.

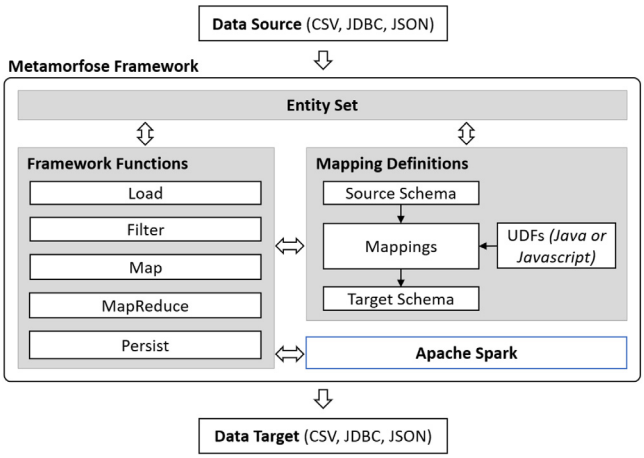

Fig. 10. Architecture of the Metamorfose framework.

Data transformations are executed through the *Map* or *MapReduce* functions. Both functions receive as parameters the source entity and mappings that define the data transformations. The *Map* function has cardinality 1:1 and *MapReduce* function has cardinality N:1. To group instances of the source entity, the *MapReduce* function also receives a grouping key as a parameter. Transformations with N:N cardinality are not supported directly by the framework, but it is possible to define a combination of *Map* and *MapReduce* functions to enable this kind of transformation.

Mappings are declarative statements that establish correspondences between source and target fields, which can be used by data transformation functions. A mapping is defined as $m = (s_i, t_i, f_i)$, in which s_i and t_i are one or more data fields of source and target schemas, and f_i is a transformation function. A complete transformation is defined as set of mappings $M = \{(s_1, t_1, f_1), \ldots, (s_n, t_n, f_n)\}$. Transformation functions are user-defined functions (UDF) for translating the input fields into output fields. There are three types of UDF: *Casting*, *Java* or *Javascript*. *Casting* are data type conversion (e.g., string to integer and integer to string) and allows only one-to-one mappings between source and target fields. *Java* or *Javascript* execute complex transformations, allowing one-to-one, one-to-many, many-to-one, and many-to-many mappings. All kinds of UDFs receive the input data via JSON and return the results via JSON. In addition, mappings can be persisted as JSON files for future use.

5.4. Using metamorfose to convert RDB to NoSQL

To illustrate the conversion process, let us consider the RDB from Fig. 1 and the set of DAGs from Fig. 4. All the DAGs are sent to *Command Generator* component, that create a set of commands according to target NoSQL model. The commands encapsulates the data transformation operations, including the mappings (M) between fields, UDFs (in javascript) and the type of function (Map or MapReduce) that will be used to performs the transformation. Below are listed the mappings and commands produced considering as input the DAG *Orders*.

$$m_{1.1} = (s_{(orderlineid)}, t_{(orderlineid)}, f_{casting});$$
$$m_{1.2} = (s_{(orderid)}, t_{(orderid)}, f_{casting});$$
$$m_{1.3} = (s_{(quantity)}, t_{(quantity)}, f_{casting});$$
$$m_{1.4} = (s_{(orderlinedate)}, t_{(orderlinedate)}, f_{casting});$$
$$m_{1.5} = (s_{(prod\_id)}, t_{(prod\_id)}, f_{casting});$$
$$m_{1.6} = (s_{(id\_prod,actor,title,price)}, t_{(products)}, f_{jscript});$$
$$cmd_1 = {}_{((Products \triangleright\!\!\triangleleft Orderlines),\{m_i \in M\},Map)}$$
$$m_{2.1} = (s_{(id\_order)}, t_{(id\_order)}, f_{casting});$$
$$m_{2.2} = (s_{(customerid)}, t_{(customerid)}, f_{casting});$$
$$m_{2.3} = (s_{(orderdate)}, t_{(orderdate)}, f_{casting});$$
$$m_{2.4} = (s_{(totalamount)}, t_{(totalamount)}, f_{casting});$$
$$m_{2.5} = (s_{(orderlineid,\ldots,products)}, t_{(orderlines)}, f_{jscript});$$
$$cmd_2 = {}_{((Orderlines \triangleright\!\!\triangleleft Orders),\{m_i \in M\},MapReduce)}$$

Command Generator produces two commands (cmd_1 and cmd_2), in which the first one specifies how to nest *Products* in *Orderlines* as an embedded document. The second one (cmd_2) nests the resulting *Orderlines* in *Orders* as an array of embedded document. *Command Generator* also automatically generates the javascript code to translate the input fields data into output fields data.

The commands are sent to *Command Executor* component that converts to a set of Metamorfose calls to load RDB data, transform and persist as JSON objects. Fig. 11 shows examples of instances of *Customers*, *Orders*, *Orderlines* and *Inventory* NoSQL entities generated according to DAGs from Fig. 4. The instances are represented by JSON documents composed by embedded documents and arrays of embedded documents.

It is important to note that the approach migrates all data from the RDB that is related to the root of the DAG (NoSQL entity). In this way, Metamorfose framework can be used to convert and migrate part or all data from RDB. Besides that, the user can configure filters to select a subset of RDB instances.

6. Implementing queries from DAGs

The conversion approach presented in this paper does not define an automatic mechanism for translating SQL queries into the native query language of the NoSQL database. However, to analyze whether there is a relationship between metrics, scores and query implementation effort, we define implementation guidelines. These guidelines aim to standardize the query implementation process, assuming that the query DAG represents the final format in which the data retrieved from the NoSQL database is sent to the application.

We define the implementation guidelines considering the MongoDB Aggregation Pipeline framework[3] that was introduced in MongoDB version 2.2 to do aggregation operations without needing to use map-reduce. The framework provides rich and very expressive query capabilities, and a set of operators that allows us to standardize the operations needed to retrieve and format JSON documents, making it possible to objectively evaluate the effort to implement the queries (LoC and stages). Furthermore, there is no standard query language for document-oriented NoSQL databases (unlike relacional databases that have SQL), and the MongoDB's Aggregation Pipeline is proving to be a prominent query language among developers. However, for other document-oriented databases that do not have such an expressive query language (e.g. CouchDB[4] or previous MongoDB versions), it is necessary to adapt or extend the guidelines on the underlying query language, replacing the use of MongoDB operators for equivalent instructions in the target database.

The MongoDB Aggregation Pipeline framework allows expressing a query as a pipeline, consisting of a set of stages that represent operations on documents. At each stage, it is possible to express selection, grouping and projection operations on documents. In addition, there is a type of stage to perform the joining of documents between two collections (similar to the *LEFT JOIN* used in SQL). Although the framework has twenty eight different

[3] https://docs.mongodb.com/manual/meta/aggregation-quick-reference/.
[4] https://couchdb.apache.org/.

Instance of Customer


```
{ "id_customer": 19465,
  "firstname": "WACHFF",
  "lastname": "XEVRMWNUCN",
  "age": 58,
  "zip": 00000,
  "income": 60000,
  "gender": "F",
  "country": "Russia",
  "city": "TUDOVGJ",
  "address1": "6064622411 Dell Way",
  "phone": "6064622411",
  "email": "XEVRMWNUCN@dell.com",
  "creditcard": 8692041200585663,
  "creditcardtype": 3,
  "creditcardexpiration": "2012/05",
  "username": "user19465",
  "password": "password"
}
```


Instance of Order


```
{ "id_order": 732,
  "customerid": 19465,
  "orderdate": "2009-01-16",
  "totalamount": 319.36
  "orderlines": [
    { "orderlineid": 7324,
      "orderid": 732,
      "prod_id": 8602,
      "orderlinedate": "2009-01-16",
      "quantity": 2,
      "products": {
        "id_prod": 8602,
        "title": "ALABAMA MOURNING",
        "actor": "ROSIE DUKAKIS",
        "price": 14.99}
    },
    ...]
}
```


Instance of Orderline


```
{ "orderlineid": 7324,
  "orderid": 732,
  "prod_id": 8602,
  "orderlinedate": "2009-01-16",
  "quantity": 2,
  "orders": {
    "id_order": 732,
    "customerid": 19465,
    "orderdate": "2009-01-16",
    "totalamount": 319.36
  },
  "products": {
    "id_prod": 8602,
    "title": "ALABAMA MOURNING",
    "actor": "ROSIE DUKAKIS",
    "price": 14.99
  }
}
```


Instance of Inventory


```
{ "prod_id": 8602,
  "quan_in_stock": 205,
  "products": {
    "id_prod": 8602,
    "title": "ALABAMA MOURNING",
    "actor": "ROSIE DUKAKIS",
    "price": 14.99,
    "orderlines": [
      { "orderlineid": 7324,
        "quantity": 2,
        ...
      },
      { "orderlineid": 32998,
        "quantity": 1,
        ...
      }]
  }
}
```


Fig. 11. Examples of instances generated by Metamorfose from the DAGs Customers, Orders, Orderlines and Inventory.

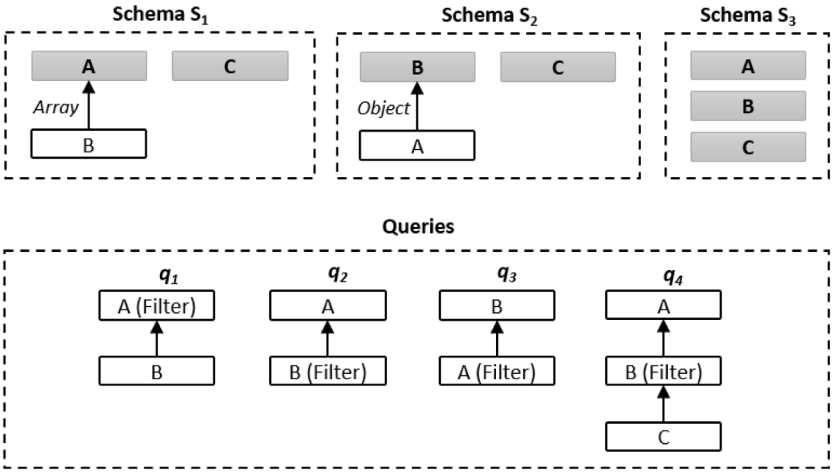

Fig. 12. Schemas and queries used to illustrate query implementation guidelines.

types of stages, our implementation guidelines are based on only seven stages, which are sufficient to carry out the implementation of the queries and reduce different combinations of operations for the same implementation. We use the following operators:

- *AddField*: Remodels each document that passes through the stage, adding new fields to the output document.
- *Group*: Groups input documents according to a specified field, and allows the use of aggregate functions.
- *Lookup*: Performs a left outer join with another collection in the same database, in order to join associated documents for processing.
- *Match*: Filters documents for the next stage of the pipeline.
- *Project*: Reshapes each document, adding new fields or removing existing fields. For each input document, generate an output document.
- *ReplaceRoot*: Replaces the input document with one of its embedded documents. The selected embedded document is promoted as a top-level document.
- *Unwind*: Deconstructs the array field of the input document and produces a new document for each array element. It generates n documents where n is the number of elements in the array and can be zero for an empty array.

Let us consider the schemas and queries of Fig. 12 to introduce the implementation guidelines. The schemas present three ways to structure the entities A, B and C. Schema s_1 uses *1-N* nesting order between entities A and B, while schema s_2 the order is *N-1*. For both schemas, the entity C is not nested with other entities. Differently, in schema s_3 the entities are not nested, and it is necessary to join documents between collections to answer queries.

The implementation of the queries of Fig. 12 on each schema requires operations to retrieve, filter and transform the documents, according to the structure of the DAG and the location of the query filter in the schema. We use (q_i, s_i) to denote the implementation of query q_i on schema s_i. For (q_1, s_1), it is necessary operations to retrieve and filter the documents from the A collection. There is no need to format the returned data due to the structural similarity between the query and schema DAGs. For (q_1, s_2) it is necessary to retrieve, filter and transform the documents from the B collection, according to the query's DAG structure. For (q_1, s_3) it is necessary to retrieve and filter documents from A, then join with related documents from B through the *$lookup* operator. As a result, the B documents are nested in A as an array of embedded documents.

Depending on the structural difference between query DAG and schema DAG it is necessary operators to group or ungroup the documents, before performing formatting operations. The *$group* operator is used to nest documents such as arrays of embedded documents and the *$unwind* operator is used to ungroup arrays of embedded documents. An example where grouping operations are required is (q_1, s_2). First the documents that match the query filter are retrieved, then the B instances are grouped (*$group*) and nested in A as an array of embedded documents.

Another example is (q_3, s_1), where after the filter operation it is necessary to ungroup the instances of B (*$unwind*), to later group the instances of A and nest them as an object embedded in B.

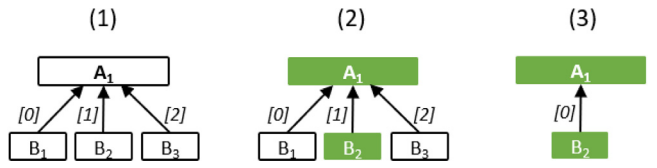

Fig. 13. Impact of query filter location on array of embedded documents in the schema.

Another aspect that affects the implementation of the query is the location of the filter in relation to the schema. When the filter is over an array of embedded documents, additional operations are needed to ensure that only data related to the query filter will be returned. This situation occurs for (q_2, s_1), where the query filter is over B, which is an array of embedded documents. Fig. 13 shows the document A_1 of the collection A from schema s_1 (1). Assuming that the filter of q_2 corresponds to element B_2, then the database returns the entire document A_1 in the filter step (2). However, the elements B_1 and B_3 do not match the query filter and must be removed (3), before the result is sent to the application.

To implement (q_2, s_1), the following operations are necessary: filter the documents from A, then ungroup (*$unwind*) the array of embedded documents (B, in this case) to access the elements individually, then filter the documents of interest again and regroup (*$ group*) the documents according to the query's DAG. As a result, only the data corresponding to the query filter will be returned to the application (3). This example shows how the location of the filter impacts the implementation of the query (additional stages), even when there is a structural match between the query and schema DAGs.

Based on the aspects presented above, we define a set of guidelines to standardize the implementation of queries. In addition, the guidelines are good implementation practices and an initial step to automate the implementation of queries in future work. The guidelines are defined below:

1. *Create DAG*: the query DAG represents the structure in which documents must be formatted before being returned from the database to the application. In Section 3.1.2 we present rules to convert a SQL command to DAG.
2. *Match*: the query implementation must start through the *$match* operator to reduce the number of documents sent to the other stages of the pipeline.
3. *Array filter*: when the query filter is located on an array of embedded documents it is necessary to ungroup (*$unwind*) the documents, re-filter (*$match*) the documents to discard elements related to the root of the document, but not related to the query filter, and then apply the other guidelines below.
4. *Bottom up traverse*: following, the order of implementation of the query is from bottom (leaf vertex) to top (root vertex). For example, the implementation (q_3, s_1) first retrieves the documents from A that match the filter, then performs operations to nest A documents in B documents, according to the query's DAG structure.
5. *Projection*: projection operations on the document fields (*$project, $AddField, $ReplaceRoot*) are pushed to the end of the query pipeline, except when necessary to execute them in intermediate stages.
6. *Join*: when the query relates documents from two or more collections:
   (a) The implementation of the query pipeline must start in the collection where the query filter is located. This strategy aims to reduce the number of documents before performing the *$lookup* operation (similar to the SQL JOIN).
   (b) The nesting order (use of *$lookup*) follows the guideline number four. For example, the implementation (q_4, s_3) must start with the operator *$match* over B, then C is nested in B, and finally, $B(C)$ is nested in A.
   (c) When the junction point is located on an array of embedded documents, it is necessary to first ungroup the documents of the array and then join each document with the documents from the other collection. For example, the implementation (q_4, s_1) needs to ungroup (*$unwind*) B, then nests ($group) C in B, and then $B(C)$ in A.

It is important to note that there is still no standard query language for document-oriented databases, and the guidelines we present here can be adapted to be used in other NoSQL databases. The implementation guidelines will be used in the experiments of Section 7.

7. Experimental evaluation

The experiment scenario is the conversion from RDB to NoSQL document store, where different NoSQL schemas are generated from the input RDB. Among the document-oriented databases, we selected MongoDB (version 4.2) to perform the experiments because it is one of the most used NoSQL databases today[5] and provides a rich query language called MongoDB Aggregation Pipeline. The experiments performed in this paper aim to show how to use the query-based metrics to assist the user in the process of evaluation, comparison and selection of the appropriate NoSQL schema before executing the data migration. In addition, we implemented all the queries in order to check if the metrics results and query implementation effort can be related. In the following sections we detail the execution steps.

7.1. Creating NoSQL schemas from conversion approaches

We create four NoSQL schemas by applying the translation rules from RDB to NoSQL document approaches from literature [3,4,6,7]. We choose these approaches because they define and explore different ways to convert relational data to nested data, encompassing most of the techniques used by similar approaches. We apply the translation rules on the RDB of Fig. 1 to generate one schema for each approach.

Table 1
RDB to NoSQL document approaches, input parameters and output schema.

Author	Input	Output
[3]	RDB Metadata	S_A
[4]	E-R Diagram	S_B
[6]	E-R + Table Classification + Table in Focus: - Orders, Customers, Products	S_C
[7]	E-R + Table/Relationship Classification: - Frequent Join: Customers/Orders - Frequent Join: Orders/Orderlines - Frequent Join: Products/Inventory	S_D

Table 1 shows the approaches and the input parameters used to generate the output NoSQL schemas. We represent the generated NoSQL schemas as DAGs, which are detailed in Fig. 14. We set a label from S_A to S_D to identify them. The nature of the input parameters is dependent on the strategy used by the approach. As a result, we have all approaches represented by the same format, which allows us to evaluate and compare them objectively.

[5] https://db-engines.com/en/ranking.

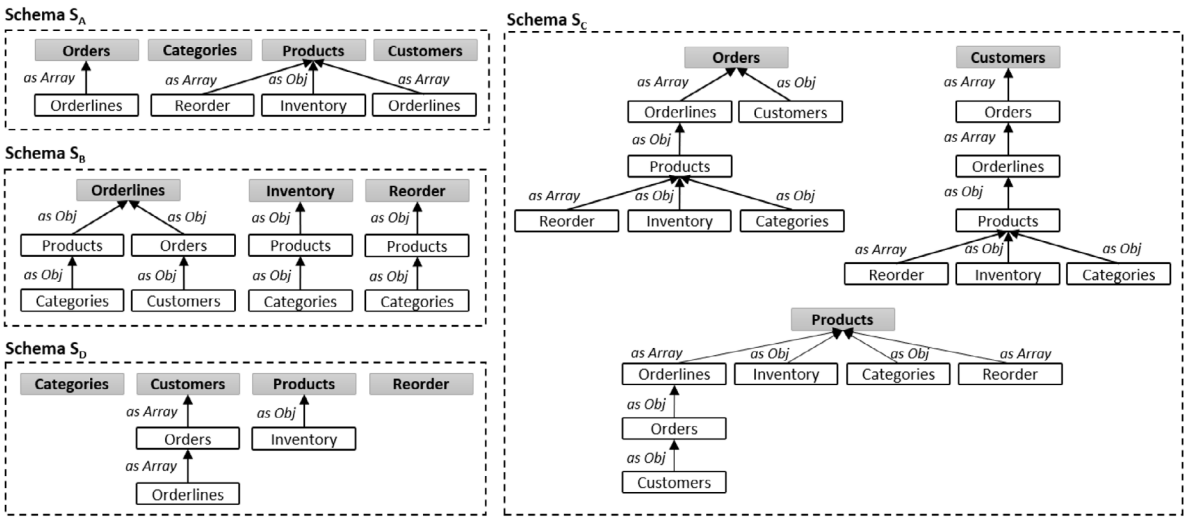

Fig. 14. Generated NoSQL schemas by approach.

Q	SQL	DAG	Q	SQL	DAG
q_1	**select** * **from** customers **where** age between ? and ?;	Customers	q_2	**select** * **from** products **inner join** inventory **on** products.id_prod = inventory.prod_id **where** price between ? and ?;	Products → Inventory
q_3	**select** * **from** orders **left join** orderlines **on** orderlines.orderid = orders.id_order **where** orderdate between ? and ?;	Orders → Orderlines	q_4	**select** * **from** customers **left join** orders **on** customers.id_customer = orders.customerid **left join** orderlines **on** orders.id_order = orderlines.orderid **left join** products **on** orderlines.prod_id = products.id_prod **where** **where** orderdate between ? and ?;	Customers → Orders → Orderlines → Products
q_5	**select** * **from** products **left join** orderlines **on** products.id_prod = orderlines.prod_id **left join** orders **on** orderlines.orderid = orders.id_order **left join** customers **on** orders.customerid = customers.id_customer **where** products.price between ? **and** ?;	Products → Orderlines → Orders → Customers	q_6	**select** * **from** orders o **left join** customers c **on** o.customerid = c.id_customer **left join** orderlines ol **on** ol.orderid = o.id_order **group by** o.orderdate, c.id_customer **where** orderdate between ? and ?;	Orders → Customers, Orderlines
q_7	**select** * **from** inventory **right join** orderlines **on** inventory.prod_id = orderlines.prod_id **where** orderlines.orderlinedate between ? and ?;	Orderlines → Inventory	q_8	**select** * **from** orderlines **where** orderlinedate between ? and ?;	Orderlines

Fig. 15. Input queries used to evaluate the NoSQL schemas.

7.2. Using query-based metrics for evaluating NoSQL schemas

We present three scenarios to evaluate and compare the NoSQL schemas from Fig. 14. The scenarios are a key input, because it is not possible to state that one given translation from RDB to NoSQL document is always the best one: it is necessary to take into account distinct application requirements. Each scenario has the set of requirements, as well as the selection criteria of the target NoSQL.

We present in Fig. 15 the set with eight queries used to evaluate each scenario. These queries have been chosen because they have different read access patterns that are common on the input RDB, including queries on one table or a group of related tables (join), with or without predicates. We show the queries in SQL and also as DAGs, which are produced according to the translation rules from Section 6. Each query DAG contains the data path of each SQL statement, but alternative DAGs could be built to represent different access patterns.

Table 2
Weights used in each scenario.

Scenario	*Path*	*SubPath*	*IndPath*	*Queries*
1	1.0	0.5	0.3	$q_1 - q_8 = 0.12$
2	1.0	0.5	0.3	$q_1 - q_8 = 0.12$
3	1.0	0.5	0.3	$q_1 - q_3 = 0.2$
				$q_4 - q_8 = 0.08$

- **Scenario 1 - number of collections**: the schemas are evaluated considering the number of collections required to answer the queries. Generally, NoSQL databases do not support native join operations between collections, so the developers need to implement joins in the application. To avoid this situation, it is preferable to have schemas that minimize the number of collections. In this case we use the *ReqColls* metrics to evaluate and rank the schemas. The other metrics are not used in this case.
- **Scenario 2 - best access pattern**: the schemas are evaluated considering the best matching between the queries access

Table 3
Metrics Results by Schema.

Schema	Query	Coverage			QScores				
		Path (D)	SubPath (D)	IndPath (D)	Paths	DirEdge	AllEdge	ReqColls	FArray
S_A	q_1	1.0 (1)	1.0 (1)	0.0 (0)	1.0	0.0	0.0	1	0
	q_2	1.0 (1)	1.0 (1)	0.0 (0)	1.0	1.0	1.0	1	0
	q_3	1.0 (1)	1.0 (1)	0.0 (0)	1.0	1.0	1.0	1	0
	q_4	0.0 (0)	0.0 (0)	0.0 (0)	0.0	0.3	0.3	3	0
	q_5	0.0 (0)	0.0 (0)	0.0 (0)	0.0	0.3	0.3	3	0
	q_6	0.5 (1)	0.5 (1)	0.0 (0)	0.5	0.5	0.5	2	0
	q_7	0.0 (0)	0.0 (0)	0.0 (0)	0.0	0.0	0.0	1	1
	q_8	0.0 (0)	1.0 (2)	0.0 (0)	0.3	0.0	0.0	1	1
SScore1:					0.47	0.40	0.40	0.62	0.25
SScore2:					0.68	0.49	0.62	0.62	0.25
Schema	Query	Coverage			QScores				
		Path (D)	SubPath (D)	IndPath (D)	Paths	DirEdge	AllEdge	ReqColls	FArray
S_B	q_1	0.0 (0)	1.0 (3)	0.0 (0)	0.2	0.0	0.0	1	0
	q_2	0.0 (0)	0.0 (0)	0.0 (0)	0.0	0.0	1.0	1	0
	q_3	0.0 (0)	0.0 (0)	0.0 (0)	0.0	0.0	1.0	1	0
	q_4	0.0 (0)	0.0 (0)	0.0 (0)	0.0	0.3	1.0	1	0
	q_5	0.0 (0)	0.0 (0)	0.0 (0)	0.0	0.7	1.0	1	0
	q_6	0.0 (0)	0.5 (2)	0.0 (0)	0.1	0.5	1.0	1	0
	q_7	0.0 (0)	0.0 (0)	0.0 (0)	0.0	0.0	0.0	2	0
	q_8	0.0 (0)	1.0 (1)	0.0 (0)	0.5	0.0	0.0	1	0
SScore1:					0.10	0.19	0.63	0.89	0.00
SScore2:					0.16	0.12	0.64	0.89	0.00
Schema	Query	Coverage			QScores				
		Path (D)	SubPath (D)	IndPath (D)	Paths	DirEdge	AllEdge	ReqColls	FArray
S_C	q_1	0.0 (0)	1.0 (1)	0.0 (0)	0.5	0.0	0.0	1	0
	q_2	1.0 (1)	1.0 (1)	0.0 (0)	1.0	1.0	1.0	1	0
	q_3	0.0 (0)	1.0 (1)	0.0 (0)	0.5	1.0	1.0	1	0
	q_4	0.0 (0)	1.0 (1)	0.0 (0)	0.5	1.0	1.0	1	1
	q_5	1.0 (1)	1.0 (1)	0.0 (0)	1.0	1.0	1.0	1	0
	q_6	0.5 (1)	1.0 (1)	0.0 (0)	0.5	1.0	1.0	1	0
	q_7	0.0 (0)	0.0 (0)	1.0 (2)	0.2	0.0	0.0	1	1
	q_8	0.0 (0)	1.0 (2)	0.0 (0)	0.3	0.0	0.0	1	1
SScore1:					0.55	0.63	0.63	1.00	0.38
SScore2:					0.62	0.64	0.64	1.00	0.38
Schema	Query	Coverage			QScores				
		Path (D)	SubPath (D)	IndPath (D)	Paths	DirEdge	AllEdge	ReqColls	FArray
S_D	q_1	0.0 (0)	1.0 (1)	0.0 (0)	0.5	0.0	0.0	1	0
	q_2	1.0 (1)	1.0 (1)	0.0 (0)	1.0	1.0	1.0	1	0
	q_3	0.0 (0)	1.0 (2)	0.0 (0)	0.3	1.0	1.0	1	1
	q_4	0.0 (0)	0.0 (0)	0.0 (0)	0.0	0.7	0.7	2	1
	q_5	0.0 (0)	0.0 (0)	0.0 (0)	0.0	0.0	0.7	2	0
	q_6	0.0 (0)	0.5 (2)	0.0 (0)	0.1	0.5	1.0	1	1
	q_7	0.0 (0)	0.0 (0)	0.0 (0)	0.0	0.0	0.0	2	1
	q_8	0.0 (0)	1.0 (3)	0.0 (0)	0.2	0.0	0.0	1	1
SScore1:					0.26	0.40	0.54	0.73	0.63
SScore2:					0.46	0.49	0.59	0.73	0.63

patterns and the schema structure. We mainly use *Paths*, *DirEdge* and *AllEdge* metrics to check if the entities are (or not) nested according to the access pattern. We assigned different weights for *Path* (1.0), *SubPath* (0.5) and *IndPath* (0.3) metrics, which means we are prioritizing schemas with greater *Path* coverage. We also assigned the same weight (0.12) for all queries to calculate *SScore* and *QScore* fields. In addition, the depth where the query path starts is used as well. This is because accessing data at level 1 in a collection ($depth = 1$) is more efficient than accessing nested data at deeper levels.

- **Scenario 3 - preferred queries**: the schemas are evaluated in the same way as in *Scenario 2*, however, we set higher weights for queries q_1, q_2 and q_3 (0.20), simulating that they are more important queries in the application (e.g., queries more frequently executed). The remaining queries have the same weight (0.08).

Table 2 summarizes the weights for each scenario. Note that these scenarios are typical examples, but that we can have other scenarios, which depend on the requirements of the application.

7.2.1. Metrics results

Table 3 shows the metrics results by schema, considering all the scenarios.[6] In the first part of the table is showed the metrics coverage by query. We can see which queries are covered or not covered by the schema, considering the metrics *Path*, *SubPath* and *IndPath*, and the depth (*D*) where the match was found in the schema. In the second part of the table is showed the *QScore* values by metric. The *QScore* is calculated by query and by metric, considering the weights defined for the metrics together with the depth where the match between query and schema was found.

[6] The tool used to collect and execute the query metrics, as well as all the results are available for download at: https://github.com/evandrokuszera/nosql-query-based-metrics.

Finally, in the bottom of each schema are showed the *SScore* values by metric. The *SScore* is calculated considering the weights defined by query. In the table we have the *SScore*1 and *SScore*2, where the first one is calculated according to the weights defined in *Scenarios 1* and *2*, and the second one is calculated according to *Scenario 3*.

For example, in schema S_A the queries q_1, q_2 and q_3 have 100% *Path* coverage, the queries q_4, q_5 and q_7 are not covered by the schema for the *Path*, *SubPath* and *IndPath* metrics. Furthermore, the queries q_4, q_5 and q_6 need join operations to answer them, and the filter location of the queries q_7 and q_8 are in arrays of embedded documents. Below are presented the results of the evaluation of the schemas considering the three defined scenarios.

Scenario 1 Results: Table 4 shows the *ReqColls* metric results per query and the corresponding *SScore* per schema. The schemas are ranked as: S_C, S_B, S_D and S_A.

Table 4
ReqColls results by query and respective *SScore* by schema (*Scenario 1*).

Query	S_A	S_B	S_C	S_D
q_1	1	1	1	1
q_2	1	1	1	1
q_3	1	1	1	1
q_4	3	1	1	2
q_5	3	1	1	2
q_6	2	1	1	1
q_7	1	2	1	2
q_8	1	1	1	1
SScore	0.62	0.89	1.0	0.73

Schema S_C is the most redundant schema and all its collections encapsulates all RDB entities, but with distinct entity nesting order. As a result, schema S_C has $SScore = 1.0$ for *ReqColls* metric. This means that all queries are answered accessing a single collection. In the schema S_B, the query q_2 needs to join documents from two collections, resulting in $SScore = 0.89$. For schema S_D is necessary to join documents from two collections to answer the queries q_4, q_5 and q_7, resulting in $SScore = 0.73$. Finally, in the schema S_A is necessary to join documents from three different collections to answer the queries q_4 and q_5 and to join document from two different collections to answer the query q_6, resulting in $SScore = 0.62$.

So, the schemas are analyzed considering only the need to use join operations ($*lookup* operator in MongoDB) to answer the queries. To better support the expert user in the schema evaluation process we can use the others metrics together with *ReqColls*, to provide more information about access pattern coverage.

Scenario 2 Results: Table 5 shows the *SScore* values for the *Paths*, *DirEdge*, *AllEdge*, *ReqColls* and *FArray* metrics. The schemas are ordered by *Paths* metric values. We use *Paths* to identity which schema best cover the queries. Considering the *Paths* coverage, schema S_C has the highest score (0.55). This means S_C best matches the access pattern of the query set. Following are the schemas S_A, S_D and S_B, with schema S_B having the worst $SScore = 0.10$. These results are related to *SScore*1 values in Table 3.

Table 5
SScore results for *Scenario 2*, order by Paths values.

Schemas	SScore values				
	Paths	DirEdges	AllEdges	ReqColls	FArray
S_C	**0.55**	0.63	0.63	1.00	0.38
S_A	**0.47**	0.40	0.40	0.62	0.25
S_D	**0.26**	0.40	0.54	0.73	0.63
S_B	**0.10**	0.19	0.63	0.89	0.00

By analyzing the *Paths* in Table 3 we see that the schema S_C has all the queries covered through the *Path*, *SubPath* or *IndPath* metrics. The *QScore* is calculated considering the metrics weights and depth where the match was found in the schema. For the schema S_C, only the queries q_7 and q_8 are penalized for the depth of the match (location of the *Orders* collection at level 2 in both cases), resulting in lower *QScore* for *Paths* metric. In contrast, in the schema S_B we identify only the queries q_1, q_6 and q_8 that match *Paths* metric, but both queries are penalized by the level where they are located in the schema. For example, to answer q_1 is necessary traverse the *Orderlines* collection to find *Customers* entity, at level 3. If we look at schema S_B, we notice that the *Orderlines*, *Inventory*, and *Reorder* collections are inverted in relation to the query access pattern. For q_2 to q_5 and q_7 there are no coverage for the *Paths* metric.

The metrics *DirEdge* and *AllEdge* show the degree which the schema entities are related to each other according to query access pattern, but without considering the exact match. So, it is interesting to use these metrics together with *Paths* to analyzing the schemas. For example, in Table 5 the S_B has $AllEdge = 0.63$, the same value as the S_C. It means that S_B has the entities related to each other according to the query access pattern, but the direction of the relationships are inverted ($DirEdge = 0.19$ and $Paths = 0.10$). If the schema is inverted in relation to the queries, greater effort will be required to implement the queries.

Finally, the *FArray* metric shows the percentage of the queries that are affected by the filter location in the schema. Here, the schema S_B does not have any query affected by filter location. Schema S_D is the most affected, with $FArray = 0.63$, followed by S_C and S_A. The filter location affects the query implementation and will be explored in the next sections.

Scenario 3 Results: Table 6 shows the *SScore* values for the *Paths*, *DirEdge*, *AllEdge*, *ReqColls* and *FArray* metrics. The schemas are ordered by *Paths* metric. These results are related to *SScore*2 values in Table 3.

Table 6
SScore results for *Scenario 3*, order by Paths values.

Schemas	SScore values				
	Paths	DirEdges	AllEdges	ReqColls	FArray
S_A	**0.68**	0.49	0.62	0.62	0.25
S_C	**0.62**	0.64	0.64	1.00	0.38
S_D	**0.46**	0.49	0.59	0.73	0.63
S_B	**0.16**	0.12	0.64	0.89	0.00

The queries q_1, q_2 and q_3 are prioritized ($weight = 0.2$). Now, the schema S_A has the greater *Paths* value, followed by S_C, S_D and S_B. Analyzing Table 3, the schema S_A has 100% *Path* coverage for queries q_1, q_2 and q_3, which means the schema is structured according to access pattern of the queries. For the schemas S_C and S_D the query q_2 has $Path = 1.0$ and the queries q_1 and q_3 only have *SubPath* coverage. The schema S_B has the smallest coverage for the queries.

So, the metrics are used to identify the most appropriate schema by prioritizing a subset of the queries. Here, the schema S_A is ranked in first place because the access pattern of the queries q_1 to q_3 are found in the schema without penalties related to the depth and weight of the *SubPath* and *IndPath* metrics. Although the schema S_C has the greater *DirEdge* result (0.64), the focus is the *Paths* metric, that appoint the schema with the best access pattern considering the query weights. The results of the *ReqColls* and *FArray* are the same of the *Scenario 2*.

To summarize, through this approach the expert user can evaluate and compare schema options before executing the translation from a RDB to a NoSQL document store, by applying the set of defined metrics and scores.

7.3. Query implementation effort

We migrate RDB (Postgres) to MongoDB to check if the metrics can be used to estimate the query implementation effort, generating one database per schema ($S_A, \ldots, S_D$). After that, we implement all the queries ($q_1, \ldots, q_8$) over all the schemas using our guidelines and the MongoDB Aggregation Framework

Table 7
List and number of stages, and LoC used to implement all the queries by schema.

Query	Schema	List of stages	#Stages	#LoC
q_1	S_A.*Customers*	1.Match	1	8
	S_B.*Orderlines*	1.Match, 2.Group, 3.ReplaceRoot	3	19
	S_C.*Customers*	1.Match, 2.Project	2	13
	S_D.*Customers*	1.Match, 2.Project	2	13
q_2	S_A.*Products*	1.Match, 2.Project	2	14
	S_B.*Inventory*	1.Match, 2.Group, 3.Unwind, 4.Project	4	46
	S_C.*Products*	1.Match, 2.Project	2	15
	S_D.*Products*	1.Match	1	8
q_3	S_A.*Orders*	1.Match	1	8
	S_B.*Orderlines*	1.Match, 2.Project, 3.Group	3	51
	S_C.*Orders*	1.Match, 2.Project	2	14
	S_D.*Customers*	1.Match, 2.Unwind, 3.Match, 4.Project	4	34
q_4	S_A.*Orders*	1.Match, 2.Unwind, 3.Lookup, 4.Group, 5.Lookup, 6.Unwind, 7. Project, 8.Group, 9.Project, 10. Project	10	96
	S_B.*Orderlines*	1.Match, 2.Project, 3.Group, 4.AddFields, 5.Group, 6.Project	6	89
	S_C.*Customers*	1.Match, 2.Unwind, 3.Match, 4.Group, 5.Project, 6.Project	6	65
	S_D.*Customers*	1.Match, 2.Unwind, 3.Match, 4.Unwind, 5.Lookup, 6.Group, 7.AddField, 8.Group, 9.AddField, 10.Project, 11.Project	11	9 2
q_5	S_A.*Products*	1.Match, 2.Unwind, 3.Lookup, 5.Unwind, 5.Lookup, 6.Unwind, 7.Group, 8.Project	8	69
	S_B.*Orderlines*	1.Match, 2.Group, 3.Project	3	41
	S_C.*Products*	1.Match, 2.Project	2	15
	S_D.*Products*	1.Match, 2.Lookup, 3.Unwind, 4.Unwind, 5.Unwind, 6.Match, 7.Project, 8.Group	8	111
q_6	S_A.*Orders*	1.Match, 2.Lookup, 3.Unwind	3	19
	S_B.*Orderlines*	1.Match, 2.Project, 3.Group	3	53
	S_C.*Orders*	1.Match, 2.Project	2	13
	S_D.*Customers*	1.Match, 2.Unwind, 3.Match, 4.Project	4	56
q_7	S_A.*Products*	1.Match, 2.Unwind, 3.Match, 4.Unwind, 5.Project	5	37
	S_B.*Orderlines*	1.Match, 2.Lookup, 3.Unwind, 4.Project	4	27
	S_C.*Orders*	1.Match, 2.Unwind, 3.Match, 4.Project, 5.Unwind	5	37
	S_D.*Customers*	1.Match, 2.Unwind, 3.Unwind, 4.Match, 5.Lookup, 6.Unwind, 7.Project, 8.Unwind	8	52
q_8	S_A.*Orders*	1.Match, 2.Unwind, 3.Match, 4.Project	4	32
	S_B.*Orderlines*	1.Match, 2.Project	4	14
	S_C.*Orders*	1.Match, 2.Unwind, 3.Match, 4.Project	4	32
	S_D.*Customers*	1.Match, 2.Unwind, 3.Unwind, 4.Match, 5.Project	5	36

as target language. Finally, we analyze the results about query implementation effort and data consistency, together with the metrics results from *Scenario 2*. The choice of *Scenario 2* is due to the fact that queries have the same weight (same priority).

7.3.1. Implementation

Table 7 shows the list and number of stages, and LoC used to implement all the queries by schema. The implementation followed the guidelines presented in Section 6. The implementation of all queries start with *$match* operator. The main purpose of using it is to reduce the number of documents to be processed by the pipeline. The next operators are dependent on the schema structure.

We separate the implementation of the queries in two groups. In the first group are the queries q_1, q_2, q_3 and q_8 which are answered by accessing only one collection of documents for all the schemas ($ReqColls = 1$). For the queries q_1, q_2 and q_3 the schema S_A has lower number of stages and LoC follows by the schemas S_C, S_D and S_B. This order corresponds to coverage results of Table 3, where S_A has greater *Paths* value for these queries. The implementation of queries q_1, q_2 and q_3 on S_B has a higher number of stages and LoC than the other schemas because of the depth of the entities and inverted nesting direction of the schema.

The implementation of q_8 on S_B has a lower number of stages and LoC than the other schemas. In this case, the target entity of q_8 is at level one of S_B.*Orderlines* collection, requiring fewer stages and LoC. Besides that, for the schemas S_A, S_C and S_D the filter location of q_8 is over array of embedded documents field ($FArray(q_8) = 1$), which negatively impacts the implementation of the query.

In the second group are the queries q_4, q_5, q_6 and q_7, which are answered by accessing two or more collections of documents for some schemas. To answer queries q_4, q_5 and q_6 it is necessary to access two or more collections of documents in schemas S_A and S_D ($ReqColls < 1$), which impacts the number of stages and LoC. In the opposite way, for schemas S_C and S_B there is no need to join documents from different collections to answer the queries ($ReqColls = 1.0$). Furthermore, S_C has the lowest number of stages and LoC because of its high correspondence with the access pattern of queries (*Paths* value) and S_B is in second place, even with the lowest structural correspondence with q_4, q_5 and q_6. The query q_7 is answered by accessing only one collection of documents in schemas S_A and S_C. Schema S_B requires join documents from two collections ($ReqColls = 2$), but still has fewer stages and LoC than other schemas. This result is related to the depth where the target entities of the q_7 are located in S_B ($level = 1$) and because S_B has $FArray = 0.0$, requiring less stages and LoC to implement the query. The other schemas have $FArray(q_7) = 1$, requiring a greater number of stages and LoC to answer q_7.

7.3.2. Analyzing implementation and metrics together

Table 8 shows the relation between *Paths* and *FArray* score results (*SScore*1) and the number of *Stages* and *LoC* to answer all the queries, by schema. The schemas are ordered by *LoC* field. The schema S_C has the smallest number of *Stages* (25) and *LoC* (204)

Table 8
SScore results for *Paths* and *FArray* metrics, and number of *Stages* and *LoC*, by schema.

Schemas	Paths	FArray	Stages	**LoC**
S_C	0.55	0.38	25	**204**
S_A	0.47	0.25	34	**318**
S_B	0.10	0.00	28	**340**
S_D	0.26	0.63	43	**402**

Table 9
List and number of stages, and LoC used to implement the modified version of the queries q_3, q_5 and q_6.

Query	Schema	List of stages	#Stages	#LoC
q_3'	S_A.*Orders*	1.Match, 2.Unwind, 3.Match, 4.Group, 5.Project	5	44
	S_B.*Orderlines*	1.Match, 2.Project, 3.Group, 4.Project	4	46
	S_C.*Orders*	1.Match, 2.Unwind, 3.Match, 4.Group. 5.Project	5	44
	S_D.*Customers*	1.Match, 2.Unwind, 3.Unwind, 4.Match, 5.Group, 6.Project	6	48
q_5'	S_A.*Products*	1.Match, 2.Unwind, 3.Match, 4.Lookup, 5.Unwind, 6.Lookup, 7.Unwind, 8.Group, 9.Project	9	80
	S_B.*Orderlines*	1.Match, 2.Group, 3.Project	3	41
	S_C.*Products*	1.Match, 2.Unwind, 3.Match, 4.Group	4	51
	S_D.*Customers*	1.Match, 2.Unwind, 3.Unwind, 4.Match, 5.Lookup, 6.Unwind, 7.Project, 8.Group, 9.Project	9	100
q_6'	S_A.*Orders*	1.Match, 2.Unwind, 3.Match, 4.Group, 5.Lookup, 6.Unwind	6	59
	S_B.*Orderlines*	1.Match, 2.Project, 3.Group	3	53
	S_C.*Orders*	1.Match, 2.Unwind, 3.Match, 4.Group, 5.Project	5	56
	S_D.*Customers*	1.Match, 2.Unwind, 3.Unwind, 4.Match, 5.Group, 6.Project	6	70

to implement all the queries, and the best result for the *Paths* (0.55) metric. Following are the schema S_A, S_B and S_D. Although S_B has the lowest value for *Paths* (0.10), it is ranked ahead of S_D, because S_B answers almost queries by only one collection of documents (*ReqColls* $= 0.89$) and it does not have arrays of embedded documents in its structure, not being affected by the filter location of the queries (*FArray* $= 0.0$). In contrast, S_D has *ReqColls* $= 0.73$ and *FArray* $= 0.63$, which increase the number of *Stages* e *LoC* to implement the queries. The results indicate that schemas with higher *ReqColls* and *Paths* scores, and low *FArray* score present less effort to codify the queries.

7.3.3. Filter location impact

In order to explore the impact of filter location in query coding, we modified the filter location for queries q_3, q_5 and q_6 to match arrays of embedded documents in the schemas. Only the schema S_B was not impacted because it does not use arrays in its structure. Table 9 shows the list of stages to coding the new queries q_3', q_5' and q_6'. As a result there is an increase in the number of stages, and consequently in the query implementation effort.

Table 10 shows the new values for *FArray*, *Stages* and *LoC* score results (*SScore*1), considering the new version of the queries. Since there was no change in the query's DAG, the *Paths* values are the same. Now, the schema S_B is ranked in second place, ahead schema S_A, with the lowest number of *Stages* (29) and the *LoC* (335) value close to S_C (313). Through these results we see that the filter location in the schema impacts the query implementation effort.

Table 10
SScore results for *Paths* and *FArray* metrics, and number of *Stages* and *LoC* by schema, considering the queries q'_3, q'_5 and q'_6.

Schemas	Paths	FArray	Stages	**LoC**
S_C	0.55	0.75	33	**313**
S_B	0.10	0.00	29	**335**
S_A	0.47	0.63	42	**370**
S_D	0.26	0.75	48	**419**

Although the schema S_B is inverted to the query access pattern, the number of *Stages* to implement all the queries is lower than schemas that best match the query access pattern (e.g. schema S_C). This is due to S_B does not have arrays of embedded documents in its structure (*FArray* $= 0$). The schema S_C is the most affected after changing queries, with *FArray* $= 0.75$ and an increase of 34.5% in the number of stages and 24% in the number of LoC. These results show that the evaluation of the schemas cannot be based on just one metric, but that it must consider all of them in the process of selecting the most appropriate schema.

7.4. Discussions

The results presented in this paper showed that the metrics are an effective tool for selecting an appropriate NoSQL schema when migrating RDB to NoSQL. Through the metrics we can identify which schemas best cover the access pattern of the application, which queries are covered and which are not covered by the schema, or which queries need more attention. Besides that, the user can prioritize metrics and queries by assigning different weights to them.

By analyzing query implementation and metrics together, we identified that schemas with high *Paths* coverage and low *ReqColls* values present less query implementation effort (Stages and LoC). However, when the filter of the query is located on array of embedded documents, the implementation effort is negatively impacted, even though the schema has a high structural correspondence with the query access pattern. We can see this impact for the queries q'_3, q'_5 and q'_6, that increased the Stages and LoC of the schemas S_C, S_A and S_D.

The *FArray* metric indicates the queries affected by filter location on an array of embedded documents. Nevertheless, *FArray* does not indicate the degree that the query is affected. So, the user should analyze the schemas considering the best access pattern coverage together with the *FArray* values, where lower values are desired.

Another aspect that is important to mention is about the *ReqColls* metric. This metric does not consider how the collections are combined to answer the query. Some join operations between collections are simple to implement, as in the case of q_7 over schema S_B. In future work we plan to extend the metrics related to join operation to best indicate the query implementation effort.

8. Conclusions

We presented a RDB to NoSQL conversion approach, in which the user defines, evaluates and compares candidate NoSQL schemas against the application access pattern, for finally migrating the relational data. The solution is used in a scenario of transformation of RDBs to NoSQL document stores. In such a scenario, several transformation strategies could be chosen, and frequently the only support available is the knowledge of the expert developer. Our approach is used as a guide on the choice of the most adequate target NoSQL schema.

Our solution uses DAGs to represent schema and query, and it uses this abstraction for two purposes: (i) to evaluate the coverage provided by the schema in relation to access pattern of the queries and (ii) to generate a set of data transformation commands to migrate relational data to document store. We define a set of query-based metrics and scores, which are calculated based on the input DAGs. The metrics enable to identify how the target document schema covers the input original queries. The metrics can be analyzed individually, or collectively, using a score per metric (*QScore*), or a score per schema (*SScore*) guiding the choices of the most appropriate schema. The definition of schemas and the calculation of metrics and scores are performed

by *QBMetrics*, a tool developed to support the process. We also define *Metamorfose*, a framework that receives the input schema and translates into commands to read data from RDB, transform and persist into NoSQL format (JSON).

We evaluated our solution in three different scenarios to show its viability. We applied the metrics on a set of schemas produced by existing transformations approaches, enabling to evaluate which schema is the most adapted for a set of input queries. In addition, if the choice of a given output schema is not possible, the metrics may guide the re-factoring of the existing queries. We also evaluated the effort to code all the queries for each schema and we identified that schemas with high structural correspondence present less query implementation effort (Stages and LoC), but when the filter of the query is located on array of embedded documents, the implementation effort is negatively impacted.

As future work, we can address the extension of our approach by integrating existing conversion algorithms and heuristics to automatically create NoSQL schemas using our schema representation, providing a complete migration solution. We also intend to extend our approach to other NoSQL models and the automatic translation of input queries to NoSQL language after migrating relational data, using Object-NoSQL mappers as a middle layer. Another direction is the integration of our approach with cost based approaches that consider the cost of executing queries during the schema selection process.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] M. Stonebraker, S. Madden, D.J. Abadi, S. Harizopoulos, N. Hachem, P. Helland, The end of an architectural era (it's time for a complete rewrite), in: Proceedings of the 33rd VLDB, University of Vienna, Austria, September 23-27, 2007, pp. 1150–1160.

[2] P.J. Sadalage, M. Fowler, NoSQL Distilled: A Brief Guide To the Emerging World of Polyglot Persistence, first ed., Addison-Wesley Professional, 2012.

[3] L. Stanescu, M. Brezovan, D.D. Burdescu, Automatic mapping of MySQL databases to NoSQL MongoDB, in: 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), 2016, pp. 837–840.

[4] G. Zhao, Q. Lin, L. Li, Z. Li, Schema conversion model of SQL database to NoSQL, in: 2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 2014, pp. 355–362.

[5] M.C.d. Freitas, D.Y. Souza, A.C. Salgado, Conceptual mappings to convert relational into NoSQL databases, in: Proceedings of the 18th International Conference on Enterprise Information Systems, ICEIS 2016, 2016, pp. 174–181.

[6] G. Karnitis, G. Arnicans, Migration of relational database to document-oriented database: Structure denormalization and data transformation, in: 2015 7th International Conference on Computational Intelligence, Communication Systems and Networks, 2015, pp. 113–118.

[7] T. Jia, X. Zhao, Z. Wang, D. Gong, G. Ding, Model transformation and data migration from relational database to MongoDB, in: 2016 IEEE International Congress on Big Data (BigData Congress), 2016, pp. 60–67.

[8] V. Reniers, D. Van Landuyt, A. Rafique, W. Joosen, Schema design support for semi-structured data: Finding the sweet spot between NF and De-NF, in: 2017 IEEE International Conference on Big Data (Big Data), 2017, pp. 2921–2930.

[9] C. de Lima, R. dos Santos Mello, A workload-driven logical design approach for NoSQL document databases, in: Proceedings of the 17th International Conference on Information Integration and Web-Based Applications & Services, iiWAS '15, 2015, pp. 73:1–73:10.

[10] Hibernate, Hibernate OGM, URL http://hibernate.org/ogm/.

[11] Impetus, Impetus Kundera, URL https://github.com/Impetus/Kundera.

[12] EclipseLink, EclipseLink (Online), URL https://www.eclipse.org/eclipselink/.

[13] DataNucleus, DataNucleus Access Plataform, URL https://www.datanucleus.org/products/accessplatform/.

[14] Spring, Spring Data (Online), URL https://spring.io/projects/spring-data.

[15] K. Herrmann, H. Voigt, T.B. Pedersen, W. Lehner, Multi-schema-version data management: Data independence in the twenty-first century, VLDB J. 27 (4) (2018) 547–571, http://dx.doi.org/10.1007/s00778-018-0508-7.

[16] M. Klettke, U. Störl, M. Shenavai, S. Scherzinger, NoSQL schema evolution and big data migration at scale, in: 2016 IEEE International Conference on Big Data (Big Data), 2016, pp. 2764–2774, http://dx.doi.org/10.1109/BigData.2016.7840924.

[17] E.M. Kuszera, L.M. Peres, M.D.D. Fabro, Toward RDB to NoSQL: Transforming data with metamorfose framework, in: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, in: SAC '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 456–463, http://dx.doi.org/10.1145/3297280.3299734.

[18] E.M. Kuszera, L.M. Peres, M. Didonet Del Fabro, Query-based metrics for evaluating and comparing document schemas, in: S. Dustdar, E. Yu, C. Salinesi, D. Rieu, V. Pant (Eds.), Advanced Information Systems Engineering, Springer International Publishing, Cham, 2020, pp. 530–545.

[19] E.M. Kuszera, L.M. Peres, M. Didonet Del Fabro, QBMetrics: A tool for evaluating and comparing document schemas, in: N. Herbaut, M. La Rosa (Eds.), Advanced Information Systems Engineering - CAiSE Forum, Springer International Publishing, Cham, 2020, pp. 77–85.

[20] D. Jaffe, T. Muirhead, The open source dvd store test application, 2005, URL https://linux.dell.com/dvdstore/.

[21] F. Bugiotti, L. Cabibbo, P. Atzeni, R. Torlone, Database design for NoSQL systems, in: E. Yu, G. Dobbie, M. Jarke, S. Purao (Eds.), Conceptual Modeling, Springer International Publishing, Cham, 2014, pp. 223–231.

[22] F. Abdelhedi, A. Ait Brahim, F. Atigui, G. Zurfluh, MDA-based approach for NoSQL databases modelling, in: L. Bellatreche, S. Chakravarthy (Eds.), Big Data Analytics and Knowledge Discovery, Springer International Publishing, Cham, 2017, pp. 88–102.

[23] X. Li, Z. Ma, H. Chen, QODM: A query-oriented data modeling approach for NoSQL databases, in: 2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA), 2014, pp. 338–345.

[24] L. Ho, M. Hsieh, J. Wu, P. Liu, Data partition optimization for column-family NoSQL databases, in: 2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity), 2015, pp. 668–675.

[25] M.J. Mior, K. Salem, A. Aboulnaga, R. Liu, NoSE: Schema design for NoSQL applications, in: 2016 IEEE 32nd International Conference on Data Engineering (ICDE), 2016, pp. 181–192.

[26] H.-J. Kim, E.-J. Ko, Y.-H. Jeon, K.-H. Lee, Techniques and guidelines for effective migration from RDBMS to NoSQL, J. Supercomput. 76 (10) (2020) 7936–7950.

[27] G. Schreiner, D. Duarte, R. Mello, When relational-based applications go to NoSQL databases: A survey, Information 10 (2019) 241, http://dx.doi.org/10.3390/info10070241.

[28] Apache Software Foundation, Apache phoenix, 2020, URL http://phoenix.apache.org/.

[29] W.-C. Chung, H.-P. Lin, S.-C. Chen, M.-F. Jiang, Y.-C. Chung, JackHare: A framework for SQL to NoSQL translation using MapReduce, Autom. Softw. Eng. 21 (2014) 489–508.

[30] G.A. Schreiner, D. Duarte, R. dos Santos Mello, SQLtoKeyNoSQL: A layer for relational to key-based NoSQL database mapping, in: Proceedings of the 17th International Conference on Information Integration and Web-Based Applications & Services, in: iiWAS '15, Association for Computing Machinery, New York, NY, USA, 2015.

[31] G. dos Santos Ferreira, A. Calil, R. dos Santos Mello, On providing DDL support for a relational layer over a document nosql database, in: Proceedings of International Conference on Information Integration and Web-Based Applications & Services, in: IIWAS '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 125–132.

[32] R. Lawrence, Integration and virtualization of relational SQL and NoSQL systems including MySQL and MongoDB, in: 2014 International Conference on Computational Science and Computational Intelligence, vol. 1, 2014, pp. 285–290, http://dx.doi.org/10.1109/CSCI.2014.56.

[33] U. Störl, T. Hauf, M. Klettke, S. Scherzinger, Schemaless NoSQL data stores - object-NoSQL mappers to the rescue? in: T. Seidl, N. Ritter, H. Schöning, K. Sattler, T. Härder, S. Friedrich, W. Wingerath (Eds.), Datenbanksysteme FÜR Business, Technologie Und Web (BTW), 16. Fachtagung Des GI-Fachbereichs "Datenbanken Und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings, in: LNI, P-241, GI, 2015, pp. 579–599, URL https://dl.gi.de/20.500.12116/2432.

[34] MongoDB, Morphia (Online), URL https://github.com/MorphiaOrg/morphia.

[35] A. Rafique, D.V. Landuyt, B. Lagaisse, W. Joosen, 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) (2018) 843–856, http://dx.doi.org/10.1109/TCC.2015.2511756.

[36] D. Hiller, Playorm: Orm for NoSQL with scalable SQL(Online), URL https://github.com/deanhiller/playorm.
[37] D.L. Moody, Metrics for evaluating the quality of entity relationship models, in: T.-W. Ling, S. Ram, M. Li Lee (Eds.), Conceptual Modeling – ER '98, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 211–225.
[38] F. Di Tria, E. Lefons, F. Tangorra, Cost-benefit analysis of data warehouse design methodologies, Inf. Syst. 63 (2017) 47–62.
[39] M. Pusnik, M. Hericko, Z. Budimac, B. Sumak, XML schema metrics for quality evaluation, Comput. Sci. Inf. Syst. 11 (2014) 1271–1289.
[40] M. Klettke, L. Schneider, A. Heuer, Metrics for XML document collections, in: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers, EDBT '02, 2002, pp. 15–28.
[41] P. Gómez, C. Roncancio, R. Casallas, Towards quality analysis for document oriented bases, in: Conceptual Modeling, Springer International Publishing, Cham, 2018, pp. 200–216.
[42] M. Zaharia, R. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: A unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65.
[43] P. Gómez, R. Casallas, C. Roncancio, Data schema does matter, even in NoSQL systems!, in: 2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS), 2016, pp. 1–6.
[44] S.S.-S. Cherfi, J. Akoka, I. Comyn-Wattiau, Conceptual modeling quality - from EER to UML schemas evaluation, in: S. Spaccapietra, S.T. March, Y. Kambayashi (Eds.), Conceptual Modeling — ER 2002, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 414–428.