Query-based metrics for evaluating and comparing document schemas

Evandro Miguel Kuszera[1,2], Letícia M. Peres[2], and Marcos Didonet Del Fabro[2]

[1] Federal University of Technology - Paraná, Dois Vizinhos, PR, Brazil
evandrokuszera@utfpr.edu.br
[2] Federal University of Paraná, Curitiba, PR, Brazil
{lmperes,marcos.ddf}@inf.ufpr.br

Abstract. Document stores are frequently used as representation format in many applications. It is often necessary to transform a set of data stored in a relational database (RDB) into a document store. There are several approaches that execute such translation. However, it is difficult to evaluate which target document structure is the most appropriate. In this article, we present a set of query-based metrics for evaluating and comparing documents schemas against a set of existing queries, that represent the application access pattern. We represent the target document schema and the queries as DAGs (Directed Acyclic Graphs), which are used to calculate the metrics. The metrics allow to evaluate if a given target document schema is adequate to answer the queries. We performed a set of experiments to calculate the metrics over a set of documents produced by existing transformation solutions. The metric results are related with smaller coding effort, showing that the metrics are effective to guide the choice of a target NoSQL document structure.

Keywords: RDBs · Document stores · Metrics · Evaluation.

1 Introduction

Relational databases (RDB) are widely used to store data of several types of applications. However, they do not meet all requirements imposed by modern applications [18]. These applications handle structured, semi-structured and unstructured data, and RDBs are not flexible enough, since they have a predefined schema. In these scenarios, NoSQL databases [16] emerged as an option. They differ from RDB in terms of architecture, data model and query language [16]. They are generally classified according to the data model used: document, column family, key-value or graph-based. These databases are called schema-free, since there is no need to define a schema before storing data. This flexibility facilitates rapid application development and makes it possible to structure the data in different ways. One of the most used NoSQL format are document stores.

RDB and document stores will be used together for a long period of time, being necessary to investigate strategies to convert and migrate schema and data between them. Different approaches have been presented to convert RDB to

NoSQL document stores [17,20,4,9,8]. Some of them consider just the structure of the RDB in the conversion process [17,20]. While others also consider the access pattern of the application [4,9,8]. However, none of the approaches is concerned with the evaluation and the comparison of the output document structure against the existing queries that need to be adapted and then executed. Often, expert user knowledge is the only guarantee that the produced document is appropriate.

The work from [5] presents eleven metrics to evaluate the structure of document oriented databases. These metrics were based on those proposed in [14] and [10] to evaluate XML schemas. Such evaluation is important to guide the choice of an adequate document structure. However, the approach has no specific metrics for assessing the queries access pattern against document structure. Despite not having a formal schema, a document has a structure used by the queries to retrieve data. We consider that the document structure can be used as an abstraction to represent a schema.

In this paper, we present a set of query-based metrics to evaluate a NoSQL document schema in relation to a set of queries that represent the application access patterns. We use DAGs (Directed Acyclic Graphs) to represent the queries and the document schemas. DAGs as schema have already been used in a previous approach to converting RDB to NoSQL nested models [11]. Since it is possible to produce different data schemas in an RDB to NoSQL conversion scenario, it is important to have a procedure to choose the most appropriate one. To help with this, we show how to use the query-based metrics alone or in combination to evaluate and compare candidate NoSQL schemas.

We performed a set of experiments to validate the metrics, using as input a set of NoSQL schemas generated from an existing RDB, and a set of queries previously defined on the same RDB. All NoSQL schemas were generated using RDB to NoSQL conversion rules already proposed in the literature. After computing the metrics, we migrate the RDB data to a MongoDB instance, according to the schemas previously generated. Then, we measure the necessary query coding effort/maintenance. The conversion showed that the metrics are effective to guide on the choice of an appropriate document structure.

The contributions of this paper are summarized as follows:

– A set of metrics to evaluate and compare document schema against a set of queries that represent the application access patterns, **prior to the execution** of a RDB to document transformation.
– A query score *QScore* and a schema score *SScore*, enabling to combine related metrics.
– A comparison of 4 different RDB to NoSQL document conversion approaches from the literature, through the query-based metrics and a set of previously defined queries.

The remainder of this paper is organized as follows: section 2 presents background about RDB to NoSQL conversion approaches and how we represent NoSQL document schemas as DAGs. In section 3 we present our query-based metrics. Section 4 shows how to combine the query-based metrics to evaluate and compare NoSQL document schemas. Section 5 deals with experiments and results. Related work is given in section 6. Finally, conclusions and future work are provided in section 7.

2 Background

Different works present approaches for converting RDB to NoSQL document [17,20,4,8,9]. The works [17,20] are automatic solutions that receive as input the RDB metadata and E-R diagrams, respectively. [17] presents an algorithm that considers the dependencies between tables and the number of PKs and FKs during the conversion process. [20] presents an algorithm that uses the dependencies between tables, however, the data nesting is performed only in the FK to PK direction.

The approaches [4,8,9] are semi-automatic. The user needs to provide additional information about the RDB to guide the translation. In these approaches, in addition to the E-R diagram, the user provides a kind of table classification. This table classification is used to decide which tables should be embedded together. The work from [4] develops an algorithm and four kinds of tables: *main*, *subclass*, *relationship* and *common*. The conversion algorithm uses this classification to nest the subclass and common tables in the main table. Relationship tables are converted using references. The approach from [9] has a different classification composed of four classes: *codifier*, *simple entity*, *complex entity* and *N:N-link*. Besides the classification, the user must provide the "table in focus", that represents the target NoSQL entity. The algorithm builds a tree with the related tables. Finally, in [8], they are based on the creation of tags to guide the process. The user annotates the E-R diagram with tags (*frequent join*, *modify*, *insert* and *big size*) that represent the data and query characteristics. From these tags the algorithm decides to use embedded documents or references in the conversion.

In a previous work, we created an approach to convert RDB to NoSQL nested models [11]. We use a set of DAGs (Directed Acyclic Graphs) to capture the source RDB and the target NoSQL document structure, which is an abstraction used to represent a NoSQL schema. Each DAG lists a set of RDB tables which are transformed into one NoSQL entity (document structure). Through our approach we use the DAGs to represent the process of converting RDB to NoSQL document from the works cited at the beginning of this section.

A DAG is defined as $G = (V, E)$, where the set of vertices V is related with the tables of the RDB and the set of edges E with the relationships between tables. The direction of the edges defines the transformation flow. Each DAG may be seen as a tree, where the root vertex is the target entity. The path from one leaf vertex to the root vertex defines one transformation flow. Each vertex contains the metadata of its respective RDB table, including the table name, fields and primary key. The edge between two vertices encapsulates relationship data between two tables, including primary and foreign keys and which entity is on the *one* or *many* side of the relationship. Through the DAG, we specify the de-normalization process from a set of related tables to produce a NoSQL entity. There are works with similar idea, but with different strategies [8,20].

Similarly, a NoSQL entity (document structure) is also represented by a DAG. The root vertex is the first level of the collection and the remaining vertices are the nested entities. The direction of the edges defines the direction of nesting between entities. Besides that, the edge encapsulates nesting type information, including embedded objects or array of embedded objects types. Through a set of DAGs it is possible to represent a NoSQL schema, where each DAG represents the structure of a collection. We define a NoSQL schema as $S = \{DAG_1, ..., DAG_n | DAG_i \in C\}$, where C is the set of collections of S.

3 Query-Based Metrics

This section presents our query-based metrics. We define six metrics to measure the coverage that a particular NoSQL document schema has in relation to a set of queries representing the application access pattern. The metrics are used to identify which schema has the appropriate access pattern required by the application. First, we present the key definitions and terminologies required to introduce the query-based metrics. Then, we present the set of metrics.

3.1 Queries and paths

The following are the key definitions and terminologies used in this paper.

Query as DAG A query 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 $q \in Q$, where Q is the set of queries.

We define two rules to convert an SQL *SELECT* statement into a DAG. SQL statements including sub-queries and *full outer join* clauses are not supported.

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

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.
- **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.

Path, Sub Path and Indirect Path We define the types of paths considered in the metrics to evaluate the schemas and queries DAGs:

- **Path**: a path p is a sequence of vertices $v_1, v_2, ..., v_j$, such that $(v_i, v_{i+1}) \in V_q$, $1 \leq i \leq j-1$, v_1 is the root vertex and v_j is the leaf vertex of the DAG. This sequence of vertices may be called the path from the root vertex to the leaf vertex.
- **Sub Path**: considering a path $p = (v_1, v_2, ..., v_k)$ and, for any i and j, such that $1 \leq i \leq j \leq k$, a subpath of p is defined as $p_{ij} = (v_i, v_{i+1}, ..., v_j)$, from vertex i to vertex j.
- **Indirect Path**: Considering a path $p = (v_1, v_2, ..., v_k)$ and, for any i, y and j, such that $1 \leq i \leq y \leq j \leq k$, an indirect path relative to p is defined as $p_{ind} = (v_i, v_{i+1}, ..., v_j)$, where $\exists v_y \in p : v_y \notin p_{ind}$. That is, an indirect path p_{ind} is the one where all its vertices and edges are contained in path p, but there are additional intermediate vertices in p that separate one or more vertices of p_{ind}.

In addition, to make the query-based metric definitions more clear, we use the following terms: 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.

3.2 Direct Edge Coverage

Direct Edge (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. Schema coverage (2) is the maximum value found when applying *DirectEdge* metric for each schema collection $c \in C$.

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

$$DirectEdge(q) = Max(C, q, DirectEdge) \quad (2)$$

The function *Max* is a higher-order function that receives a set of elements (e.g., collection set C), the query q and the metric function (e.g., *DirectEdge*). It applies the metric for all elements of the collection and the query q and it returns the higher value. It is used in other metrics in the remaining of the paper.

3.3 All Edge Coverage

All Edge (3) 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. Schema coverage (4) is the maximum value found when applying *AllEdge* metric for each schema collection (c_i).

$$AllEdge(c, q) = |(E_q \cap E_c)|/|E_q| \quad (3)$$

$$AllEdge(q) = Max(C, q, AllEdge) \quad (4)$$

3.4 Path Coverage

The *Path Coverage* metric measures the coverage of query paths in relation to the collection paths. A query may have one or more paths (e.g. q_6 in the Figure 3 has two paths). Through the *Path Coverage* it is possible to measure the coverage of the query paths relative to the collection paths (5). The Path Coverage for all the schema (6) is the maximum value found when applying the metric for each collection.

$$Path(c, q) = |(P_q \cap P_c)|/|P_q| \tag{5}$$

$$Path(q) = Max(C, q, Path) \tag{6}$$

3.5 Sub Path Coverage

Through the *Sub Path Coverage* metric, it is checked if the query paths are present in the collection as subpaths (7). 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. It is possible to measure the sub path coverage of all the schema by applying the metric for each collection (8). The result is the higher value returned.

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

$$SubPath(c, q) = \frac{\sum_{i=1}^{|P_q|} existSubPath(qp_i, P_c)}{|P_q|} \tag{7}$$

$$SubPath(q) = Max(C, q, SubPath) \tag{8}$$

3.6 Indirect Path Coverage

Through the *Indirect Path* metric, it is checked if the query paths are present in the schema as indirect paths (as defined in section 3.1). 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. In (9) we measure the indirect path coverage relative to the collection level, and in (10) relative to the schema, by applying the metric for each collection, where the largest value returned represents the schema coverage.

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

$$IndPath(c, q) = \frac{\sum_{i=1}^{|P_q|} existIndPath(qp_i, P_c)}{|P_q|} \tag{9}$$

$$IndPath(q) = Max(C, q, IndPath) \tag{10}$$

3.7 Required Collections Coverage

The *Required Collections* (11) metric returns the smallest number of collections required to answer a given query. The function $createCollectionPaths(q)$ returns a set of paths that consists of collections that have the entities required to answer the query.

$$ReqColls(q) = min(createCollectionPaths(q)) \tag{11}$$

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

4 Combining the Metrics

In this section we present how to combine the metrics for measuring the overall coverage of a schema with respect to a set of input queries. First, it is necessary to calculate a *QScore*, which denotes a score per metric, or per combination of related metrics, per query. This score enables to set up weights to prioritize the importance of specific metrics. Second, we calculate a *SScore*, which is a score for a set of queries over a given schema. The results are used to rank the input schema. These scores are explained in the following sections.

4.1 Query Score (QScore)

The *QScore* yields a single value per metric, or a value that combines related metrics. The score is calculated for a given metric and a given query q_i.The *QScore* for metrics *DirEdge*, *AllEdge* and *ReqColls* is the same value returned by the metric:

$$QScore(DirEdge, q) = DirEdge(q) \tag{12}$$
$$QScore(AllEdge, q) = AllEdge(q) \tag{13}$$
$$QScore(ReqColls, q) = ReqColls(q) \tag{14}$$

However, the *QScore* for metrics *Path*, *Sub Path* and *Indirect Path* is a unique value and is namely as *Paths*. It returns the highest value among the three metrics, taking into account the depth of each path and an additional weight, as defined below:

$$path_v = Path(q_i) * w_p \tag{15}$$
$$subpath_v = (SubPath(q_i) * w_{sp})/depthSP(q_i) \tag{16}$$
$$indpath_v = (IndPath(q_i) * w_{ip})/depthIP(q_i) \tag{17}$$
$$QScore(Paths, q_i) = max(path_v, subpath_v, indpath_v) \tag{18}$$

First, the metric value is weighted according to its path type, divided by the smallest depth in which the root vertex of path is located in the schema.

The depth is obtained by a specific function for each metric. The weights w_p, w_{sp} and w_{ip} are used to set up a priority between *Path*, *Sub Path* and *Indirect Path* metrics. The method for calculating *QScore* is inspired by the results of [6], where the authors state that the depth of the required data in the collection and the need to access data stored at different levels of the collection produce negative impact. As the *Path* metric denotes the exact match between query path and collection path (with *depth* = 1), one possibility is to set the w_p with the highest value, followed by smaller values for w_{sp} and w_{ip}. In this way, schemas with exact match are prioritized. Another aspect is related to data redundancy that NoSQL schemas may present: a query path can be found as a *Path*, *Sub Path*, and *Indirect Path* in the schema. Then, by defining distinct weights and using path depth, we can prioritize a particular type of path coverage. For better readability, we assign each calculation to a specific variable (15-17), which is then used to calculate the *QScore* (18).

4.2 Schema Score (SScore)

SScore denotes the schema score for a given metric (except *ReqColls*) as the sum of the *QScore* values for all the queries, where each query q_i has a specific weight w_i, and the sum of all w_i is equal to 1. Following the same idea of the *QScore*, it has a single value for *Path*, *SubPath*, and *IndPath*, which is the sum of its corresponding *QScore*. It is defined below:

$$SScore(metric, Q) = \sum_{i=1}^{|Q|} QScore(metric, q_i) * w_i \tag{19}$$

The *SScore* for *ReqColls* metric 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* equal to 1. It decreases when the number of collections increases. It is defined as follows:

$$NC = \sum_{i=1}^{|Q|} QScore(ReqColls, q_i)$$
$$SScore(ReqColls, Q) = \frac{|Q|}{NC} \tag{20}$$

NC is the number of collections required to answer all input queries, which is the sum of all *QScore*. The expression for calculating *SScore* above is based on the schema minimality metric presented in [3].

These scores show the coverage provided by the schema for each query, where we can 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. Since the metrics are not independent, we do not define a single expression to calculate the overall score of the schema. The goal here is to provide the user with a methodology for evaluating NoSQL schema using the proposed metrics. Still, the user can use the metrics independently, according to their needs and application requirements.

5 Experimental evaluation

In this section we present the experiments to evaluate our query-based metrics in an RDB to NoSQL documents conversion scenario, where different NoSQL schemas are generated from the input RDB. In order to generate the candidate NoSQL schemas we select four RDB to NoSQL conversion approaches from the literature. The input queries represent the application's access pattern over the RDB, and are then known a priori. The goal is 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 the following sections we detail the execution of each of these steps.

5.1 Creating NoSQL Schemas from Conversion Approaches

We select four RDB to NoSQL document approaches from the literature, that define different ways to convert relational data to nested data [17,20,9,8]. These approaches were chosen because they contain the most diverse set of translation rules. We apply the translation rules on the RDB of Figure 1 to generate a set of NoSQL schemas using our DAG approach to represent them. So, we create one schema for each approach and set a label from A to D to identify them.

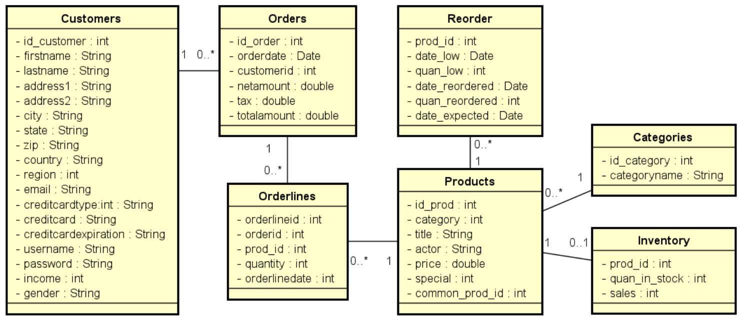


Fig. 1: Input RDB

Figure 2 shows the graphical representation of the generated NoSQL schemas. The vertices with gray background color represent the collections of the schemas (root vertex). We can see that schemas differ in number of collections and arrangement of entities. As a result, we have all approaches represented by the same format, which allows us to evaluate and compare them objectively.

5.2 Defining the Evaluation Scenario

The schemas are evaluated considering the best matching between the queries access patterns and the schema structure. We use *Path*, *SubPath*, *IndPath*,

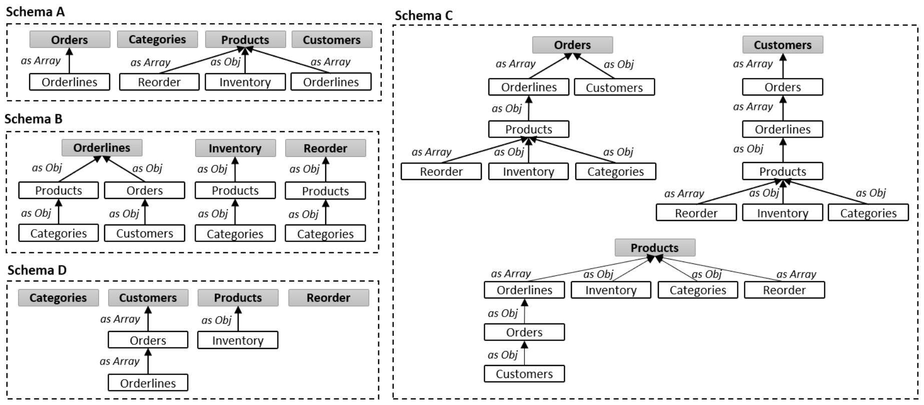
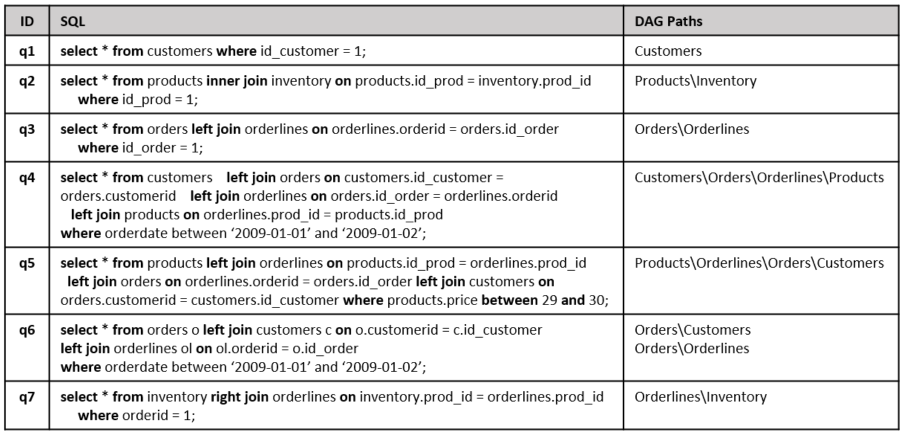


Fig. 2: Generated NoSQL schemas by approach

DirEdge, *AllEdge* and *ReqColls* metrics to check if the entities are (or not) nested according to the access pattern. To calculate *SScore* and *QScore*, we assigned the same weight for all queries and for all *paths* metrics ($w_p = 1; w_{sp} = 1; w_{ip} = 1$). This means that all queries and types of path coverage have the same priority. The depth where query path starts is also considered in the calculation, prioritizing schemas where the entities are closest to the root of the DAG. However, the user can define different weights and turn off the depth of path, according to their needs.



ID	SQL	DAG Paths
q1	**select * from** customers **where** id_customer = 1;	Customers
q2	**select * from** products **inner join** inventory **on** products.id_prod = inventory.prod_id **where** id_prod = 1;	Products\Inventory
q3	**select * from** orders **left join** orderlines **on** orderlines.orderid = orders.id_order **where** id_order = 1;	Orders\Orderlines
q4	**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** orderdate between '2009-01-01' and '2009-01-02';	Customers\Orders\Orderlines\Products
q5	**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** 29 **and** 30;	Products\Orderlines\Orders\Customers
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** orderdate between '2009-01-01' and '2009-01-02';	Orders\Customers Orders\Orderlines
q7	**select * from** inventory **right join** orderlines **on** inventory.prod_id = orderlines.prod_id **where** orderid = 1;	Orderlines\Inventory

Fig. 3: Input queries used to evaluate the NoSQL schemas

The seven (7) queries used to evaluate the schemas are presented in Figure 3. These queries have been chosen because they contain different access patterns.

We show the queries in SQL and also as DAG paths, which are produced according to the translation rules from section 3.1. Each query DAG contains the data path of each SQL statement, but alternative DAGs could be built to represent different access patterns.

5.3 Experimental Results

In this section we present the metrics result and the impact on query coding effort for each schema[3]. Finally, preliminary results on query execution time are presented. Table 1 summarizes all results and is used by the sections below.

Metrics Result. Table 1 shows the *QScore* for the *Paths*, *DirEdge*, *AllEdge* and *ReqColls* metrics, by query. *Paths* is calculating taking into account the *Path*, *SubPath*, and *IndPath* metrics (see section 4.1). The *SScore* is also shown. Considering the paths coverage, schema C has the highest score 0.93. This means C best matches the access pattern of the query set. Following are the schemes A, D and B, with schema B having the worst $SScore = 0.08$.

Table 1: Query-based metrics result, by schema

Schema A

Query	Paths	DirEdge	AllEdge	ReqColls	LoC	Stages	Time
q1	1.0	0.0	0.0	1	5	1	0.0
q2	1.0	1.0	1.0	1	11	2	0.0
q3	1.0	1.0	1.0	1	5	1	0.0
q4	0.0	0.3	0.3	3	92	11	0.06
q5	0.0	0.3	0.3	3	73	11	0.89
q6	0.5	0.5	0.5	2	19	6	0.02
q7	0.0	0.0	0.0	1	27	5	0.06
SScore	0.50	0.45	0.45	0.58	232	37	1.03

Schema B

Query	Paths	DirEdge	AllEdge	ReqColls	LoC	Stages	Time
q1	0.3	0.0	0.0	1	16	3	0.06
q2	0.0	0.0	1.0	1	21	2	0.02
q3	0.0	0.0	1.0	1	48	2	0.05
q4	0.0	0.3	1.0	1	66	7	0.07
q5	0.0	0.66	1.0	1	50	5	0.19
q6	0.25	0.5	1.0	1	41	6	0.11
q7	0.0	0.0	0.0	2	23	4	0.06
SScore	0.08	0.21	0.71	0.88	265	29	0.55

Schema C

Query	Paths	DirEdge	AllEdge	ReqColls	LoC	Stages	Time
q1	1.0	0.0	0.0	1	10	2	0.0
q2	1.0	1.0	1.0	1	12	2	0.0
q3	1.0	1.0	1.0	1	11	2	0.0
q4	1.0	1.0	1.0	1	15	2	0.03
q5	1.0	1.0	1.0	1	15	2	0.06
q6	1.0	1.0	1.0	1	13	2	0.01
q7	0.5	0.0	0.0	1	27	5	0.06
SScore	0.93	0.71	0.71	1.0	103	17	0.16

Schema D

Query	Paths	DirEdge	AllEdge	ReqColls	LoC	Stages	Time
q1	1.0	0.0	0.0	1	10	2	0.0
q2	1.0	1.0	1.0	1	12	2	0.0
q3	0.5	1.0	1.0	1	13	3	0.02
q4	0.0	0.67	0.66	2	55	9	0.08
q5	0.0	0.0	0.66	2	89	12	35.91
q6	0.25	0.5	1.0	1	25	5	0.03
q7	0.0	0.0	0.0	2	37	8	0.09
SScore	0.39	0.45	0.62	0.70	241	41	36.12

We use path metrics to identify which schema best covers the queries. For instance, in schema C, the queries $q_1 - q_6$ are 100% covered by the schema through the *Path*, *SubPath* or *IndPath* metrics. Only q_7 is penalized for starting at level 2 of the *Orders* collection, resulting in lower *QScore* (0.5). In contrast, in the schema B we identify only the queries q_1 and q_6 that match paths metrics, but both queries are penalized by the level where they are located in the schema.

[3] The tool implemented and all the results are available for download at: `https://github.com/evandrokuszera/nosql-query-based-metrics`

For example, to answer q_1 is necessary traverse the *Orderlines* collection to find *Customers* entity, at level 3. If we look at schema B, we notice that the *Orderlines*, *Inventory*, and *Reorder* collections are inverted in relation to the query access pattern. As a result, there is no coverage for the path metrics for queries $q_2 - q_5$ and q_7.

Using the metrics *DirEdge* and *AllEdge*, we verify the degree to which the entities in the schema are related to each other, as required by the access pattern of the queries. For example, schema C has the highest *DirEdge* score and *AllEdge* score, which means that it has the closest access pattern to the query set. In contrast, schema B has the highest *AllEdge* score (same value as schema C) and the smallest *DirEdge* score, which means that entities are related to each other in the schema B according to the query structure, but the relationship direction is inverted, so it does not correspond properly to the access pattern of queries. For example, the collection *Orderlines* has part of the relationships between entities corresponding to the $q_4 - q_6$ query structure ($DirEdge > 0$), but for the remaining queries, schema B is inverted.

For *ReqColls* metric, the schema C has the best result, with $SScore = 1.0$. This means that all queries are answered accessing a single collection. Then, the schemas are ranked as: C, B, D and A. This result is due to schema C being the most redundant schema, in which the three collections encapsulate all RDB entities, but using a different nesting order.

Query Coding Effort. We measured the impact on query coding effort for each schema, to asses if it is related with metrics results. To measure the coding effort, we use the number of lines of code (LoC) required to manually implement the query. While they could be automatically generated, they would need to be maintained during the application life cycle. The goal here is to check whether high *SScore* schemas have less query implementation complexity.

We created four (4) target database instances in MongoDB [4] according to schemas A, B, C and D. MongoDB was selected because it is a widely used document store. We use our Metamorfose framework to migrate data from RDB to MongoDB. After that, we implement all queries of Figure 3 using the MongoDB *aggregation pipeline* framework, that uses the concept of data processing pipelines. Each pipeline consists of stages, where each stage transforms the documents what goes through it.

The LoC for each query was obtained by the MongoDB *explain* command, with a standardized query format, facilitating line count. In addition to LoC, we counted the number of stages used in the pipeline to fetch and project documents according to the query DAG structure. Table 1 shows the LoC of each query by schema. Considering the total LoC per schema, schema C has the smallest value (103), followed by schemas A, D, and B. When considering the number of query pipelines, schema C has the lowest value, followed by schemas B, A, and D. In this case, B takes second place because its structure has no nested arrays, so no extra stages are required to unwind arrays of documents.

[4] https://www.mongodb.com

Analyzing *SScore* results for *Paths*, *DirEdge*, and *AllEdge* metrics together with the aggregate LoC and *Stages*, we can verify that schemas with higher *Paths* and *DirEdge* scores require less lines of code when implementing queries. For metric *AllEdge* this is not always true. This metric shows whether the entities in the schema are related as the query access pattern. However, the relationship may exist, but the direction may not match the pattern specified in the query (case of schema B). In this case, more effort is required to project the data according to the query pattern.

To summarize, the expert user can evaluate and compare schema options before executing the translation from a RDB to a NoSQL document stores, by applying the set of defined metrics and scores. Through these metrics we check if the entities are (or not) nested according to the query access pattern. We also check which queries need to fetch data from different collections, so the user can decide which schemas to prioritize. We generated a set of queries and the LoC metrics reflect the results of our metrics.

Query Execution Time. We measured query execution time to verify if it is related with the metrics results. The last column of Table 1 shows the average execution time in seconds (each query was executed 30x). It is worth noting that query time for $q_1 - q_3$ returned zero for some schemas because the search field is the index of collection. The results show that schema C is the most adapted, followed by schemas B, A, and D. Schema B is the second one, even though it did not match the queries access pattern. The reason is due to the execution time of q_5 for schemas A and D. For both schemas, it is necessary to perform the MongoDB *lookup* operation (similar to SQL left join). In A, there is a *lookup* operation between *Orders*, *Products* and *Customers*, and in D between *Products* and *Customers*. In both cases, the fields used in the *lookup* are located in nested arrays, which has significant impact, especially for schema D, where the *lookup* field is located inside two nested object arrays. However, these results are preliminary and need further investigation, where we plan to extend our metrics set to consider the impact on data nesting, document size, collection size, and the use of indexes have on query performance.

6 Related Work

Different works present formal definitions for NoSQL document data models [2,1,19]. In [2], they present NoAM, the NoSQL Abstract Model that use as the main modelling unit the concept of aggregates (set of entities) and is driven by application use cases (functional requirements). [1] and [19] present approaches that transform a conceptual model (UML) into NoSQL physical model. These approaches consist in methodologies for defining NoSQL schemas according to user-supplied parameters. However, they do not provide means to evaluate the schema produced. Our approach aims to evaluate the NoSQL schema in relation to a set of queries that represent the access pattern of the application.

There are works defining optimized schemas for column-family [7,13] and document [15,12,8] oriented NoSQL databases. In [7] they present Graph based Partition Algorithm (GPA) approach, that groups high affinity schema attributes in the same column family to avoid loading unnecessary data to answer queries. In a similar way, [13] describes a cost-based approach to schema design that recommends a column-family NoSQL schema suitable for the application queries. The authors of [12] and [8] present a conversion approach for generating NoSQL document logical schema considering a previously provided conceptual model and the expected workload of the application. Our query-based metrics approach can be used to evaluate and compare the output schemas of these works.

Considering the utilization of metrics, the work from [5] presents a metric set to evaluate schemas of NoSQL document databases. It was based on the works [14] and [10], that present structural metrics to evaluate XML documents. Eleven structural metrics are defined to evaluate NoSQL Document schemas and to assist the user in the process of selecting the most appropriate schema. Our work is partially inspired by [5]. The way the authors represent the NoSQL schema is similar to our approach based on DAGs. However, our approach has a different purpose, which is to evaluate and compare NoSQL schemas based on queries that represent the application access pattern instead of evaluating only the schema structure. To the best of our knowledge, our work is the first that presents a set of query-base metrics used to evaluate and compare NoSQL document schemas using a set of queries.

7 Conclusions

We presented a solution to evaluate how adequate a NoSQL document schema is with respect to a set of queries representing the application access patterns. Our approach is used as a guide on the choice of the most adequate target document schema in a scenario of RDB to NoSQL document transformation. The queries and the set of target schemas are represented as DAGs.

We define a set of query-based metrics, which are calculated based on the input DAGs (queries and schemas). 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*), enabling specialized analysis.

We applied the metrics on a set of schemas produced by existing RDB to NoSQL transformations solutions. The evaluation of these different transformation approaches was only possible because we adopted the DAGs as a common unified format. This also means that the approach is technology independent. We executed all transformation scenarios, to confirm that the metrics can be related to the coding effort of the queries, with respect to the LoC measure. In addition, if the choice of a given output schema is not possible, the metrics may guide the re-factoring of the existing queries. As future work, we aim to extend the evaluation to integrate with cost-based approaches.

References

1. Abdelhedi, F., Ait Brahim, A., Atigui, F., Zurfluh, G.: Mda-based approach for nosql databases modelling. In: Bellatreche, L., Chakravarthy, S. (eds.) Big Data Analytics and Knowledge Discovery. pp. 88–102. Cham (2017)
2. Bugiotti, F., Cabibbo, L., Atzeni, P., Torlone, R.: Database design for nosql systems. In: Yu, E., Dobbie, G., Jarke, M., Purao, S. (eds.) Conceptual Modeling. pp. 223–231. Springer International Publishing, Cham (2014)
3. Cherfi, S.S.S., Akoka, J., Comyn-Wattiau, I.: Conceptual modeling quality - from eer to uml schemas evaluation. In: ER 2002. pp. 414–428. Springer (2003)
4. Freitas, M.C.d., Souza, D.Y., Salgado, A.C.: Conceptual mappings to convert relational into nosql databases. In: Proceedings of the 18th ICEIS 2016
5. Gómez, P., Roncancio, C., Casallas, R.: Towards quality analysis for document oriented bases. In: Conceptual Modeling (2018)
6. Gómez, P., Casallas, R., Roncancio, C.: Data schema does matter, even in nosql systems! In: 2016 IEEE Tenth RCIS. pp. 1–6 (June 2016)
7. Ho, L., Hsieh, M., Wu, J., Liu, P.: Data partition optimization for column-family nosql databases. In: 2015 IEEE International Conf. SmartCity. pp. 668–675 (2015)
8. Jia, T., Zhao, X., Wang, Z., Gong, D., Ding, G.: Model transformation and data migration from relational database to MongoDB. In: IEEE BigData. pp. 60–67 (2016)
9. Karnitis, G., Arnicans, G.: Migration of relational database to document-oriented database: Structure denormalization and data transformation. In: 2015 7th ICCI-CSN. pp. 113–118 (2015)
10. Klettke, M., Schneider, L., Heuer, A.: Metrics for xml document collections. In: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering. pp. 15–28. EDBT '02 (2002)
11. Kuszera, E.M., Peres, L.M., Fabro, M.D.D.: Toward RDB to NoSQL: Transforming data with metamorfose framework. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. pp. 456–463. SAC '19 (2019)
12. de Lima, C., dos Santos Mello, R.: A workload-driven logical design approach for nosql document databases. In: Proceedings of the 17th iiWAS
13. Mior, M.J., Salem, K., Aboulnaga, A., Liu, R.: Nose: Schema design for nosql applications. In: 2016 IEEE 32nd ICDE. pp. 181–192 (2016)
14. Pusnik, M., Hericko, M., Budimac, Z., Sumak, B.: Xml schema metrics for quality evaluation. Comput. Sci. Inf. Syst. **11**, 1271–1289 (2014)
15. Reniers, V., Van Landuyt, D., Rafique, A., Joosen, W.: 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). pp. 2921–2930 (2017)
16. Sadalage, P.J., Fowler, M.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 1st edn. (2012)
17. Stanescu, L., Brezovan, M., Burdescu, D.D.: Automatic mapping of mysql databases to nosql mongodb. In: 2016 FedCSIS. pp. 837–840 (Sep 2016)
18. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era (it's time for a complete rewrite). In: Proc. of 33rd VLDB, University of Vienna, Austria, Sept 23-27, 2007. pp. 1150–1160 (2007)
19. Xiang Li, Zhiyi Ma, Hongjie Chen: QODM: A query-oriented data modeling approach for nosql databases. In: 2014 IEEE WARTIA. pp. 338–345 (2014)
20. Zhao, G., Lin, Q., Li, L., Li, Z.: Schema conversion model of sql database to nosql. In: 2014 Ninth 3PGCIC. pp. 355–362 (2014)