Toward RDB to NoSQL: Transforming Data with Metamorfose Framework

Evandro Miguel Kuszera
C3SL Labs, Federal University of Technology of Paraná
Dois Vizinhos, PR, Brazil
evandrokuszera@utfpr.edu.br

Leticia M. Peres
C3SL Labs
Federal University of Paraná
Curitiba, PR, Brazil
lmperes@inf.ufpr.br

Marcos Didonet Del Fabro
C3SL Labs
Federal University of Paraná
Curitiba, PR, Brazil
marcos.ddf@inf.ufpr.br

ABSTRACT

The emergence of applications dealing with structured, semi structured, and non-structured data created demands on new data storage systems. The relational model, widely used to store data from diverse applications, does not meet all the imposed scenarios. In response, NoSQL databases have emerged as an option. As a consequence, new approaches for converting the relational model to NoSQL models have been created. However, most of them target a specific NoSQL model and provide little (or none) support for customization for transformations with different cardinalities. In this paper we present a novel approach to convert relational databases (RDB) to document and column family NoSQLs. Our approach receives as input a set of directed acyclic graphs (DAG) representing the target NoSQL model. The DAGs are used to generate commands to transform the input RDB data. The approach supports different cardinalities, and the commands can be customized. We have developed a tool to interpret the DAGs and that supports different transformation strategies. We performed experiments to validate our solution, with different configurations, where conversions were carried out to document and column-family storage.

CCS CONCEPTS

• **Information systems → Information integration**; **Extraction, transformation and loading**; *Data exchange*;

KEYWORDS

Data transformation, NoSQL databases, Relational databases, Data mapping, Database conversion.

ACM Reference Format:
Evandro Miguel Kuszera, Leticia M. Peres, and Marcos Didonet Del Fabro. 2019. Toward RDB to NoSQL: Transforming Data with Metamorfose Framework. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3297280.3299734

1 INTRODUCTION

Relational Database Management Systems (RDBMS, or symply RDB) are used to store data of various applications. However, with the emergence of new types of applications and the popularization of cloud computing new requirements have arisen on data store systems. In [13], the authors state that these systems, based on the relational model, can not handle all issues encountered by current applications. As an example we have web applications and mobile applications that produce a large amount of data during user interactions. These data can be structured, semi-structured or unstructured. In addition, RDBs are not flexible enough since they have pre-defined schema, which makes interoperability and adaptability difficult.

NoSQL databases emerged as an alternative [8]. They differ from RDBs in terms of architecture, data model, and query language. The relational model is implemented through tables, columns and rows, and the database schema must be defined before storing the data. NoSQL databases can be divided into four categories: key-value, document, column-family and graph. Each of them uses a different data model. However, since NoSQL databases do not follow the relational model and do not support the SQL standard, the migration from a RDB is difficult[2]. In addition, there is a huge base of users familiarized with SQL syntax and migrating to NoSQL has a large learning curve [10]. There is no standard API for accessing different NoSQL, typically users interact with database at the programming level, reducing portability [6]. As both types of databases will coexist it is important to investigate approaches for schema and data migration between them.

There are different solutions that translate the RDB model and the corresponding data to NoSQL, while most of them follows a two step methodology[4]. First, they transform the relational model to the NoSQL model. Second, they perform the data migration. In the transformation step, mappings between concepts and transformation rules are defined to convert the RDB entities to NoSQL entities. In the migration step, it is necessary to connect to both databases, read the RDB data, transform and write the data into the target NoSQL.

In general, the approaches de-normalize the RDB data based on dependencies between tables and/or data access pattern. Some approaches run automatic conversion algorithms and do not support customization of the conversion process [1, 7, 9, 11, 12, 14, 16, 17]. Others allow the user to customize the generated NoSQL model, but these are solutions that migrate RDB to a single NoSQL model [4, 5]. To the best of our knowledge, there are little approaches which allow customization of data conversion process and address more than one NoSQL model.

In this paper we present an approach to migrate RDB to NoSQL nested models - document or column-family. For this, we present a mechanism to represent the RDB to NoSQL transformations through a set of generated commands. This mechanism receives as input a set of directed acyclic graphs (DAG) corresponding to the desired target NoSQL model. From the set of DAGs the framework commands and user-defined functions (UDF) are generated, with the ability to read the RDB data, transform and persist it into JSON format. It supports transformations with different cardinalities. The contributions of this paper can be summarized as follows:

- a customizable framework to convert RDB to NoSQL nested models.
- a command generator method to produce data transformation commands and UDFs through the specification of a DAG representing the output NoSQL model.
- the support of 1:1, 1:N and N:N transformations, through the combination of two kind of functions: *Map* and *MapReduce*.
- an extensible mapping creation process, so the developer can add new algorithms to generate mappings from the DAG and to produce new data transformation operations.
- an execution flow where the output of one transformation can be subsequently used as input by another transformation, without the need to persist partial results.

We validated our approach through the implementation of a tool, on top of Apache Spark [15], supporting all the components of the transformation framework and we have performed experiments to show the viability of the solution.

The remainder of this paper is organized as follows: section 2 describes the approach to convert RDB to NoSQL using our framework. Section 3 deals with the experimental setup and results. Related work is given in section 4. Finally conclusions and future work are provided in section 5.

2 RDB TO NOSQL TRANSFORMATION FRAMEWORK

In this section we present our data transformation framework which allows to load data, to define mappings between source and target fields, to perform transformations and to persist the data. It receives the target NoSQL model and converts it into a set of commands. These commands are executed to read the data from the RDB, transform and persist in NoSQL format. Figure 1 provides an overview of the approach.

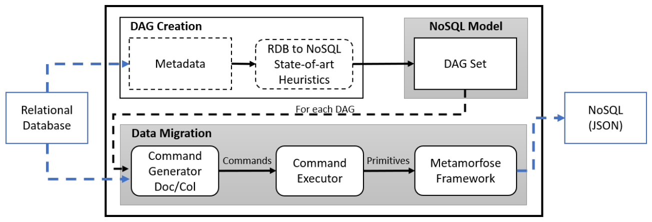


Figure 1: Overview of RDB to NoSQL approach

It is composed of the components: *(i) DAG Set*, *(ii) Command Generator*, *(iii) Command Executor* and *(iv) Metamorfose Framework*. We use a set of DAGs to define the NoSQL model, where each DAG represents a NoSQL entity. The developer provides the set of DAGs, where conversion heuristics from literature can be used in this process. In the following we detail each component.

2.1 Metamorfose Framework

Figure 2 presents the main transformation components: *Entity Set*, *Mapping Definitions*, *Functions* and *Apache Spark*. The *Entity Set* component maintains a collection of the entities loaded or transformed by the framework. Transformations change the data of existing entities or generate new entities. Through the functions *Load*, *Filter*, *Map*, *MapReduce* and *Persist* it is possible to load entities, filter data using SQL, to perform transformations and to persist the results. The *Mapping Definitions* component allows the specification of the source and target schema of a given entity and the transformation functions that are invoked. The functions call *Apache Spark* methods to perform the operations. The choice of using *Apache Spark* is motivated by its support of a wide range of abstractions to manipulate datasets.

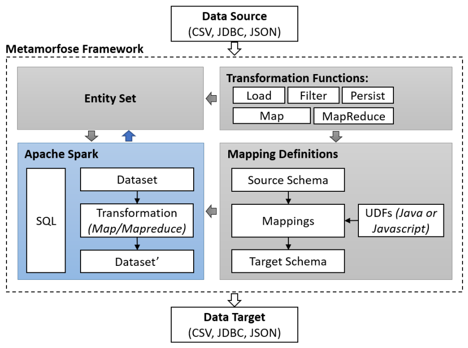


Figure 2: Architecture of the framework

Transformation Functions. The central unit of a transformation is an entity. An entity can be defined as $E = \{I_1, I_2, ..., I_n\}$, where I_i is an instance of E. 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, returning an instance of the target entity (I_{ti}) for each instance of the source entity (I_{si}). The *MapReduce* function has cardinality N:1, returning an instance of the target entity (I_{ti}) for each group of instances of source entity ($I_{s1}, I_{s2}, ..., I_{sn}$). To group instances of the source entity, it is necessary to provide 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.

Figure 3 illustrates the execution flow of the *Map* function. The source instance I_{s1} is transformed into the target instance I_{t1}. The set of mappings ($M_1, M_2, ..., M_n$) defines how the fields of I_{s1} are transformed into fields of I_{t1}. Figure 4 illustrates the execution flow of the *MapReduce* function. The source instances I_{s1} and I_{s2} are transformed into key-value pairs (*map* function) and reduced to the target instance I_{t1}. In the reduction phase the mappings

$(M_1, M_2, ..., M_n)$ are applied to each instance of the group.

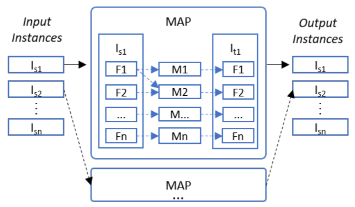


Figure 3: Map Function

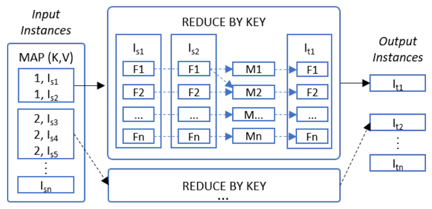


Figure 4: MapReduce Function

Mapping Definitions. Mappings are declarative statements that establish correspondences between source and target fields, which can be used by data transformation functions. An entity has a data schema that can be defined as $S = \{s_1, ..., s_n\}$, where s_i is a data field of the schema S. From a source schema S is possible to derive a target schema T applying transformations on the field values of S. A complete transformation processes a set of mappings $M = \{(s_1, t_1, f_1), ..., (s_n, t_n, f_n)\}$, where s_i represents one or more source fields $\{s_{i1}, ..., s_{ij}\}$, t_i one or more target fields $\{t_{i1}, ..., t_{ij}\}$ and f_i a transformation function.

The mappings are used in *Map* or *MapReduce* functions to transform source instances into target instances. There are three types of user-defined functions (UDF) for translating the input fields: *Casting*, *Java* or *Javascript*. *Casting* are data type conversion (e.g., string to integer and integer to string). Complex transformations are implemented as UDF in *Java* or *Javascript*. Casting transformations allow only one-to-one mappings between source and target fields. Transformations using UDFs allow one-to-one, one-to-many, many-to-one, and many-to-many mappings. Internally, the framework uses JSON to represent the instances of the source and target entities. UDFs receive the input data via JSON and return the results via JSON. For instance, considering the mapping $M_1 = (s(id, fname, lname), t(id, name), f(jscript))$, the f function takes as parameters the *id*, *fname* and *lname* fields encapsulated in a JSON object and returns a JSON object with the *id* and *name* fields.

The use of Javascript UDF mechanism provides flexibility to define complex data mappings and transformations. There is no need to recompile the mappings every time new implementations are provided. In addition, mappings can be persisted as JSON files for future use.

Execution Flow. Figure 5 presents the framework execution flow. In (1) the data is loaded as a Spark dataset. Based on the user mappings the *dataset* is transformed (2) producing (3) *dataset'* as a result. *Dataset'* can be used: (4) as input to a new transformation, (5) as input to SQL query or (6) can be persisted to CSV file, JSON file or relational database table. The SQL query execution on *dataset'* produces new (7) *dataset"*. *Dataset"* can be used for new data transformations (8), for persistence (9) or visualization (10). This flow allows the definition of chains of transformations. Transformations can be applied over the current dataset or can create a new datasets with the resulting data.

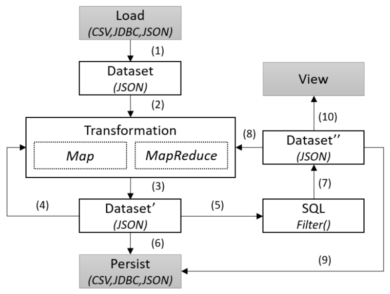


Figure 5: The framework execution flow

2.2 DAG Set

The transformations are represented as DAGs (Directed Acyclic Graphs). A DAG is defined as $G = (V, E)$, where the set of vertices V is related with the entities of the RDB and the set of edges E with the relationships between entities. The direction of the edges defines the transformation flow. Each DAG can be visualized as a tree, where the root vertex is the target NoSQL entity. The path from one leaf vertex to the root vertex defines one transformation flow. Each vertex encapsulates the metadata of its respective RDB entity, including the table name, fields and primary key. The edge between two vertices encapsulates relationship data between two entities, including primary and foreign keys and which entity is on the one or many side of the relationship. The DAG is used as input to denormalize the set of RDB entities to create a NoSQL entity. There are approaches following similar idea, though with different strategies [4, 17].

Figure 6 illustrates three DAGs. The vertices with yellow background color are the target NoSQL entities. DAG 1 is composed of two vertices representing *Table A* and *Table B*. In this case, *Table B* instances are nested in *Table A*. In DAG 2 there are three vertices with *Table B*, *Table C* and *Table D*. The instances of *Table D* and *Table C* are nested in *Table B*. In DAG 3 there is only one vertex. In this case *Table D* is converted as the target NoSQL entity. The conversion process is dependent on target NoSQL model. For each target NoSQL, it is possible to use different conversion strategies.

To summarize, we use the DAG for two purposes. First, for establishing a data path, which is used to define the denormalization process or to define user queries (access pattern). Second, for assisting in the generation of the mappings and UDFs to transform the RDB data into NoSQL entities. The DAGs are used as input to the *Command Generator* component.

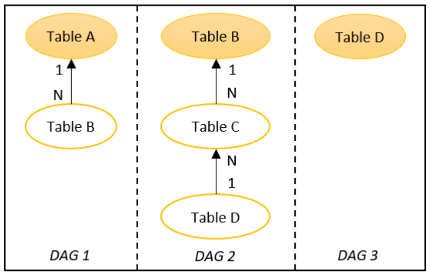


Figure 6: A set of example DAGs

2.3 Command Generator

This component is responsible for converting each DAG into a set of commands. A command encapsulates operations for nesting the leaf vertex entity in a root vertex entity. It is defined as $command = (joinSpec, fieldMaps, function)$, where *joinSpec* is the join operation between entities that are denormalized, $fieldMaps$ is the set of mappings to transform the data, and $function$ is the transformation function to call (*Map* or *MapReduce*). To perform this conversion we create Algorithm 1. It receives as parameters the DAG and target NoSQL type. It iterates through the vertex set and generates a command set as output.

Algorithm 1 Building the transformation commands:

```
Input: The DAG and target NoSQL (document or column family).
Output: The set of transformation commands.
 1: $commands \leftarrow empty$
 2: if $graph.vertexSet = 1$ then
 3:     $simple\_cmd(graph.getVertex(0))$
 4: else
 5:     while $graph.vertexSet > 1$ do
 6:         $leafVertexs \leftarrow graph.getVertexsWithInDegree(0)$
 7:         for all $leaf \in leafVertexs$ do
 8:             if $targetNoSQL = Document$ then
 9:                 $nosql\_doc\_cmd(leaf)$
10:             else if $targetNoSQL = ColumnFamily$ then
11:                 $nosql\_col\_cmd(leaf, graph.vertexSet)$
12:             end if
13:             $graph.remove(leaf)$
14:         end for
15:     end while
16: end if
17: return $commands$
```

In Algorithm 1, if the given DAG has only one vertex (only one entity), then the *simple_cmd* function is executed. This function creates a command with operations to read the data from the RDB entity and to transform each instance into a JSON object. If the target NoSQL is a column family, then the JSON object consists of a column family and *rowkey* field. If the DAG has two or more vertices, the algorithm traverses all paths from the leaf vertices to the root vertex. For each path it calls specific functions (*nosql_doc_cmd* or *nosql_col_cmd*) depending on the target NoSQL. Then, the 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. The framework currently supports Document oriented and Column Family NoSQL stores. For each target NoSQL, the generation commands are explained in Algorithms 2 and 3.

NoSQL Document. To convert RDB to NoSQL document it is necessary to transform RDB entities into JSON objects. Relationships between entities are represented by references (similar to the relational model), embedded in JSON objects, or arrays of embedded JSON objects. Algorithm 2 uses JSON objects and array of embedded JSON objects to represent the relationships between RDB entities. It receives the leaf vertex and returns a command that encapsulates the operations to embed the leaf vertex entity in the successor vertex entity. If the leaf vertex is on side *one* of the relationship it is embedded as a JSON object. If it is on side *many*, then all instances are embedded in an array of JSON objects. This process is repeated until the root vertex entity contains all DAG entities. Function *JScriptGenerator* plays a key role in the transformation process. It receives the vertex representing the RDB entity and UDF type. *JScriptGenerator* returns automatically generated Javascript UDFs with statements to transform an instance of the RDB entity into JSON objects.

Algorithm 2 NoSQL Doc Command:

```
Input: Leaf vertex.
Output: Transformation Command.
 1: $fieldMaps \leftarrow empty$
 2: $succr \leftarrow leaf.getSuccessor()$
 3: for all $field \in succr.fields$ do
 4:     $fieldMaps.oneToOne(field, field, casting)$
 5: end for
 6: if $leaf.isOneSide$ then
 7:     $jsUDF \leftarrow JScriptGenerator(leaf, docEmbedded)$
 8:     $function \leftarrow' map'$
 9: else
10:     $jsUDF \leftarrow JScriptGenerator(leaf, arrayEmbedded)$
11:     $function \leftarrow' mapreduce'$
12: end if
13: $fieldMaps.manyToOne(leaf.fields, leaf.name, jsUDF)$
14: $succr.addField(leaf.name)$
15: $joinSpecAdd(leaf, succr)$
16: return $Command(joinSpec, fieldMaps, function)$
```

NoSQL Column family. The RDB entities are transformed into tables composed of column families. RDB entities are de-normalized and grouped before migrating into NoSQL. Each instance must receive a unique identifier called *rowkey*. We use JSON objects to represent the NoSQL entity. Each JSON object consists of the *rowkey* field and embedded JSON objects to represent the column families. Algorithm 3 receives the leaf vertex, number of vertices of the DAG, and returns a command encapsulating the operations to nest the leaf vertex entity into the root vertex entity. First it creates the join operation between leaf vertex and root vertex entities, including intermediate vertices. In the remaining statements, the leaf vertex entity is nested as one column family of the root vertex entity. The *nesting_key* variable is used to rename the fields of the leaf vertex entity. This mechanism allows to insert the *many* side instances of the relationship inside a column family (JSON object). When the number of vertices is equals to two, it means that the last command is created. In this case some extra operations are executed. The first one is the creation of the field *rowkey*. The primary key of the root vertex entity is mapped as the *rowkey* field. The second one is the creation of the column family of the root vertex entity. The *JScriptGenerator* function returns UDFs to transform RDB entities into column families. It uses the *nesting_key* variable for nesting RDB entities without duplicating the field names in JSON object.

Algorithm 3 NoSQL Col Command:

Input: Leaf vertex and vertices number of graph.
Output: A transformation Command.

```
1: $fieldMaps \leftarrow empty$
2: $root \leftarrow getRootVertex()$
3: $joinSpecAdd(leaf, root)$
4: if $verticesNumber = 2$ then
5:     $fieldMaps.oneToOne(root.PK, 'rowkey', casting)$
6:     $jsUDF \leftarrow JScriptGenerator(root, docEmbedded)$
7:     $fieldMaps.manyToOne(root.fields, root.name, jsUDF)$
8: end if
9: for all $field \in root.fields$ do
10:    $fieldMaps.oneToOne(field, field, casting)$
11: end for
12: $nesting\_key = leaf.PK$
13: $jsUDF \leftarrow JScriptGenerator(leaf, nesting, nesting\_key)$
14: $fieldMaps.manyToOne(leaf.fields, leaf.name, jsUDF)$
15: $root.addField(leaf.name)$
16: return $Command(joinSpec, fieldMaps, 'mapreduce')$
```

2.4 Command Executor

The *Command Executor* component receives a set of commands to convert one DAG to one NoSQL entity. This set of commands is converted to the framework functions. It encapsulates relational database connection parameters and controls the execution order of the commands. These commands are executed by the framework to read, transform, and persist the data.

3 EXPERIMENTS

To validate our approach we executed experiments to convert RDB to NoSQL document and column family.

3.1 Experimental Setting

We use the database of Dell's DVD Store test application [3]. It provides a database schema, load scripts and an application that simulates an online e-commerce site. Through the load scripts it is possible to configure the size of the database in terms of number of records. Figure 7 shows the database schema. It is composed of seven tables, but to illustrate our approach we consider only *Orders*, *Orderlines* and *Products* tables.

From these three tables we build the *Orders*, *Orderlines* and *Products* DAGs presented in Figure 8. Each DAG has a root vertex (yellow background color) and the remaining vertices representing the entities that are nested. For example, in DAG *Orders*, the *Products* and *Orderlines* tables are nested in the *Orders* table. How nesting is performed depends on the NoSQL model. All these three DAGs enable translating the input data into the output data. However, other DAGs, representing only part of the information, could also be created depending on the application domain.

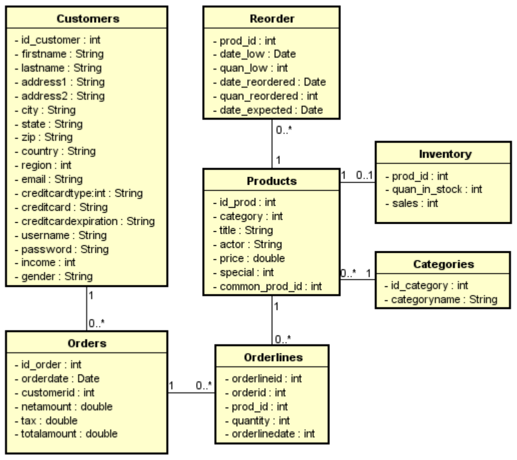


Figure 7: Dell DVD store database schema

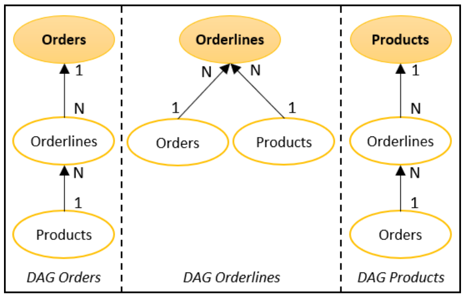


Figure 8: Orders, Orderlines and Products DAGs

We have implemented a tool[1] to validate our approach. It receives as input RDB connection parameters, the DAG and the NoSQL target model. After that, the *Command Generator* converts the DAG to the framework commands. These commands are executed through the *Command Executor* component. As a result, it produces JSON objects. Our approach does not need to persist the data directly into the NoSQL database by default. However, we develop adapters to read and insert the JSON objects into the desired NoSQL database.

The experiments are performed in one single machine with Windows 10 Professional, Intel Core i7 2.5Ghz, 16 GB of RAM and 1 TB disk. The tool is implemented in Java and Apache Spark 2.3.0. We have configured three database instances of different sizes in Postgres 10. The number of records in the *Orders*, *Orderlines* and *Products* tables is shown in Table 1. In the next section we present the results of the experiments.

3.2 Experiment Results

We select the DAG *Orders* as a guide to describe the artifacts generated during the conversion and examples of JSON objects generated after conversion. We present the run-time results to convert RDB data to NoSQL document and column family.

[1]The tool is available for download at: https://github.com/evandrokuszera/metamorfose

Table 1: Number of records of each Postgres database

RDB Size	Orders	Orderlines	Products
RDB1	12,000	60,350	10,000
RDB2	60,000	301,188	50,000
RDB3	120,000	601,009	100,000

RDB to NoSQL Document. We show bellow the mappings and commands generated by Algorithm 2, using as input the DAG from Figure 8:

$M_{1.1} = (s(\text{orderlineid}), t(\text{orderlineid}), f_{\text{casting}});$
$M_{1.2} = (s(\text{orderid}), t(\text{orderid}), f_{\text{casting}});$
$M_{1.3} = (s(\text{prod\_id}), t(\text{prod\_id}), f_{\text{casting}});$
$M_{1.4} = (s(\text{quantity}), t(\text{quantity}), f_{\text{casting}});$
$M_{1.5} = (s(\text{id\_prod,category,title,actor,price,special}), t(\text{products}), f_{\text{Js1}});$
$cmd1 = ((Products \bowtie Orderlines), fieldMaps, Map)$

$M_{2.1} = (s(\text{id\_order}), t(\text{id\_order}), f_{\text{casting}});$
$M_{2.2} = (s(\text{customerid}), t(\text{customerid}), f_{\text{casting}});$
$M_{2.3} = (s(\text{orderdate}), t(\text{orderdate}), f_{\text{casting}});$
$M_{2.4} = (s(\text{orderlineid,orderid,prod\_id,quantity,products}), t(\text{orderlines}), f_{\text{Js2}});$
$cmd2 = ((Orderlines \bowtie Orders), fieldMaps, MapReduce)$

The *cmd1* command encapsulates operations to transform the instances of *Products* and *Orderlines* into JSON objects and to embed the object *Products* inside *Orderlines*. The *cmd2* command transforms the *Orders* entity into a JSON object and then embeds the resulting *Orderlines* as an array of JSON objects into *Orders*. The process of creating embedded JSON objects is encapsulated in the *Js1* and *Js2* UDFs. Figure 9 shows the code of UDF *Js1* and *Js2*. These UDFs are automatically generated from DAG vertices and edges by *Command Generator* component. If desired, it is possible to can modify this code to customize the translations.

In Figure 10 we show an *Order* instance produced by these commands. The *Order* instance is composed of an array of *Orderlines* and each *Orderline* instance has an associated *Product*.

RDB to NoSQL Column Family. We show bellow the mappings and commands generated by Algorithm 3, using as input the DAG from Figure 8:

$M_{1.1} = (s(\text{id\_order}), t(\text{id\_order}), f_{\text{casting}});$
$M_{1.2} = (s(\text{customerid}), t(\text{customerid}), f_{\text{casting}});$
$M_{1.3} = (s(\text{orderdate}), t(\text{orderdate}), f_{\text{casting}});$
$M_{1.4} = (s(\text{id\_prod,category,title,actor,price,special}), t(\text{products}), f_{\text{Js1}});$
$cmd1 = (((Products \bowtie Orderlines) \bowtie Orders), fMaps, MapReduce)$

$M_{2.1} = (s(\text{id\_order}), t(\text{rowkey}), f_{\text{casting}});$
$M_{2.2} = (s(\text{products}), t(\text{products}), f_{\text{casting}});$
$M_{2.3} = (s(\text{id\_order,customerid,orderdate}), t(\text{orders}), f_{\text{Js2}});$
$M_{2.4} = (s(\text{orderlineid,orderid,prod\_id,quantity}), t(\text{orderlines}), f_{\text{Js3}});$
$cmd2 = ((Orderlines \bowtie Orders), fMaps, MapReduce)$

The *cmd1* command encapsulates operations to transform the *Products* and *Orders* entities into JSON objects and to nest *Products* as column family in the *Orders* entity. The *cmd2* command creates the *rowkey* field from the *id_order* field, copies *Products* object created by *cmd1* to *Orders* entity, encapsulates the *Orders* fields into the JSON object and nests as a column family of the *Orders* entity and, finally, transforms *Orderlines* entity into JSON object and nest it as column family of *Orders* entity. The process of creating column families as JSON objects is encapsulated in UDFs *Js1*, *Js2* and *Js3*, which are shown in Figure 11. These UDFs are automatically generate from DAG vertices and edges by *Command Generator* component. Customizations in the UDFs can be done as well.

Figure 12 shows an *Order* instance after the execution of the commands. The *Order* instance is composed of *rowkey* field and column families *Orders*, *Orderlines* and *Products*. In order to nest the entities *Orderlines* and *Products* in *Orders*, the *orderlineid* and *id_prod* fields are used as *nesting_key*. The *nesting_key* parameter is used to assign an unique column qualifier for each instance field.

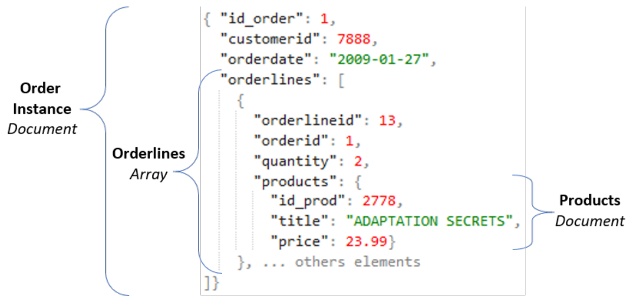
After running the experiments to convert RDB to NoSQL document and column family we checked the consistency of the generated data against RDB data. Through the obtained results we

UDF	Source	Target	Type

Js1	id_prod,
	category,
	title,
	actor,
	price,
	special
		products	function docEmbedded(values) {
			  input = JSON.parse(values);
			  output = JSON.parse('{}');
			  output.products = JSON.parse('{}');
			  output.products.id_prod = input.id_prod;
			  output.products.category = input.category;
			  output.products.title = input.title;
			  output.products.actor = input.actor;
			  output.products.price = input.price;
			  output.products.special = input.special;
			  return JSON.stringify(output);
			}

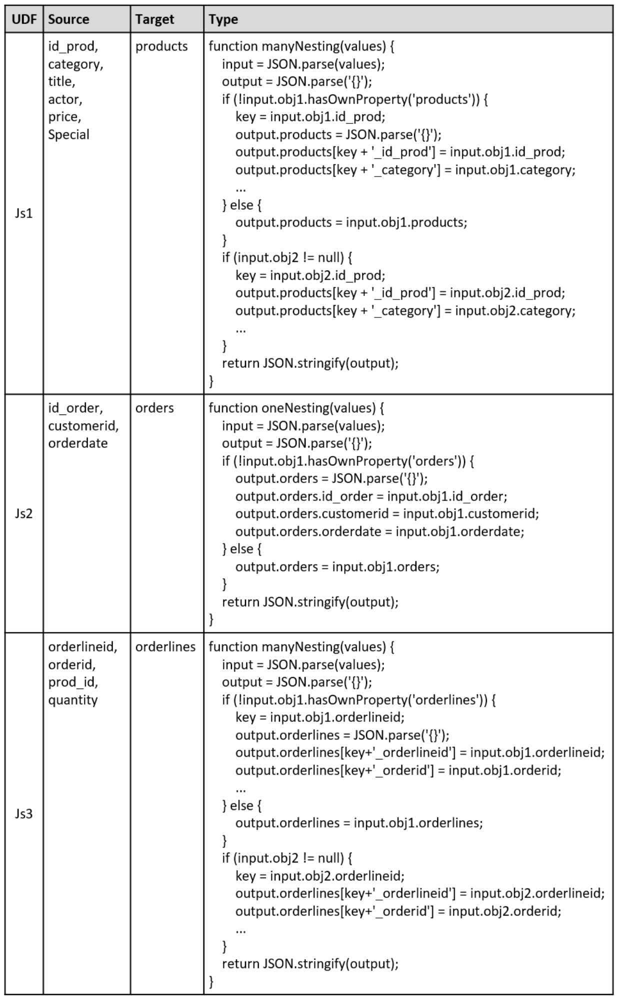
Js2	orderlineid,
	orderid,
	prod_id,
	quantity,
	products
		orderlines	function arrayEmbedded(values) {
			  input = JSON.parse(values);
			  output = JSON.parse('{}');
			  if (!input.obj1.hasOwnProperty('orderlines')) {
			    embed_obj = JSON.parse('{}');
			    embed_obj.orderlineid = input.obj1.orderlineid;
			    embed_obj.orderid = input.obj1.orderid;
			    embed_obj.prod_id = input.obj1.prod_id;
			    embed_obj.quantity = input.obj1.quantity;
			    embed_obj.products = input.obj1.products;
			    if (input.obj2 != null)
			      output.orderlines = [embed_obj, input.obj2];
			    else
			      output.orderlines = [embed_obj];
			  } else {
			    output = input.obj1;
			    if (input.obj2 != null)
			      output.orderlines.push(input.obj2);
			  }
			  return JSON.stringify(output);
			}

Figure 9: RDB to NoSQL Document UDFs



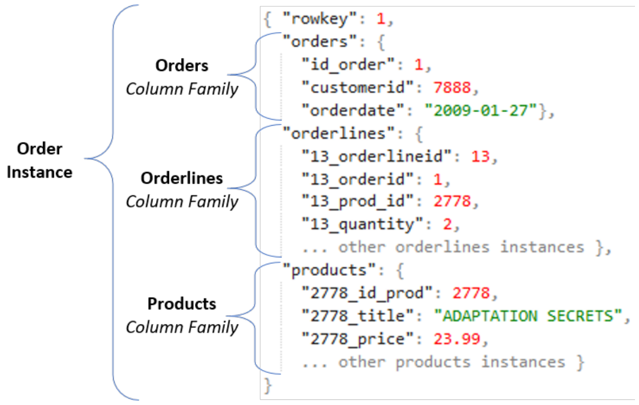
```
{ "id_order": 1,
  "customerid": 7888,
  "orderdate": "2009-01-27",
  "orderlines": [
   {
     "orderlineid": 13,
     "orderid": 1,
     "quantity": 2,
     "products": {
       "id_prod": 2778,
       "title": "ADAPTATION SECRETS",
       "price": 23.99}
   }, ... others elements
]}
```

Figure 10: Order entity converted to NoSQL Document



UDF	Source	Target	Type
Js1	id_prod, category, title, actor, price, Special	products	`function manyNesting(values) { input = JSON.parse(values); output = JSON.parse('{}'); if (!input.obj1.hasOwnProperty('products')) { key = input.obj1.id_prod; output.products = JSON.parse('{}'); output.products[key + '_id_prod'] = input.obj1.id_prod; output.products[key + '_category'] = input.obj1.category; ... } else { output.products = input.obj1.products; } if (input.obj2 != null) { key = input.obj2.id_prod; output.products[key + '_id_prod'] = input.obj2.id_prod; output.products[key + '_category'] = input.obj2.category; ... } return JSON.stringify(output); }`
Js2	id_order, customerid, orderdate	orders	`function oneNesting(values) { input = JSON.parse(values); output = JSON.parse('{}'); if (!input.obj1.hasOwnProperty('orders')) { output.orders = JSON.parse('{}'); output.orders.id_order = input.obj1.id_order; output.orders.customerid = input.obj1.customerid; output.orders.orderdate = input.obj1.orderdate; } else { output.orders = input.obj1.orders; } return JSON.stringify(output); }`
Js3	orderlineid, orderid, prod_id, quantity	orderlines	`function manyNesting(values) { input = JSON.parse(values); output = JSON.parse('{}'); if (!input.obj1.hasOwnProperty('orderlines')) { key = input.obj1.orderlineid; output.orderlines = JSON.parse('{}'); output.orderlines[key+'_orderlineid'] = input.obj1.orderlineid; output.orderlines[key+'_orderid'] = input.obj1.orderid; ... } else { output.orderlines = input.obj1.orderlines; } if (input.obj2 != null) { key = input.obj2.orderlineid; output.orderlines[key+'_orderlineid'] = input.obj2.orderlineid; output.orderlines[key+'_orderid'] = input.obj2.orderid; ... } return JSON.stringify(output); }`

Figure 11: RDB to NoSQL Column Family UDFs



```
{ "rowkey": 1,
  "orders": {
    "id_order": 1,
    "customerid": 7888,
    "orderdate": "2009-01-27"},
  "orderlines": {
    "13_orderlineid": 13,
    "13_orderid": 1,
    "13_prod_id": 2778,
    "13_quantity": 2,
    ... other orderlines instances },
  "products": {
    "2778_id_prod": 2778,
    "2778_title": "ADAPTATION SECRETS",
    "2778_price": 23.99,
    ... other products instances }
}
```

Figure 12: Order entity converted to NoSQL column family

verified that generated data is consistent with RDB data and provided DAG.

Performance Results. We measure the performance of the approach to convert data from the RDB1, RDB2, and RDB3 databases according to *Orders*, *Orderlines* and *Products* DAGs. Table 2 shows the results obtained considering only the time to convert the data from one model to another. The time to persist (on disk or NoSQL database) the generated JSON objects is not considered. The main objective of the experiments is to assess the viability and feasibility of the approach, leaving detailed performance evaluations as future work. However, we can see that the conversion to NoSQL column family has higher runtime if compared to the conversion to NoSQL document. This result is due to the fact that the conversion strategy calls more often *MapReduce* functions than the conversion strategy to document stores, and the *MapReduce* function has higher execution time than the *Map* function.

Table 2: Execution time (in seconds) to convert RDB to NoSQL document and column family for Orders, Orderlines and Products DAGs.

RDB	Orders		Orderlines		Products	
	Doc	*Col*	*Doc*	*Col*	*Doc*	*Col*
RDB1	36s	45s	28s	33s	36s	41s
RDB2	75s	120s	40s	61s	83s	129s
RDB3	132s	210s	59s	97s	187s	336s

4 RELATED WORK

There are different related works aiming to convert relational databases to NoSQL document and column family models. The main differences between these approaches, despite their output format, are on how they select the input elements from the relational model, their relations, and on how the output model is produced.

In [7, 9, 11, 14, 16], they present different forms of de-normalization to produce as output column family models. After the denormalization, techniques are used to define the column families and rowkey field of the output table. The approach from [9] presents a dataflow to select the tables to be converted. The dataflow is set up according to the cardinality relationships between the elements. The tables are transformed into column families and grouped into an HBase table. In [7], they analyze primary and foreign keys and add the corresponding tables on linked lists, recursively transforming them, with the concatenation of primary keys of the tables in the linked list. [14] presents an algorithm that traverses all the tables of the RDB and verifies the dependencies between them. For each dependency between two tables, a third denormalized table is created. The approach analyzes the set of tables created and determines which ones have the least cost in terms of storage space and query execution time. In, [16] they also use de-normalization to avoid the use of join queries. A graph conversion model was used to identify and represent the dependencies. Article [11] presented a conversion method composed of four steps: *(i)* relation denormalization, *(ii)* extended table merging, *(iii)* key encoding and *(iv)* views based on indexes. Heuristics were proposed to nest tables with relationships. The techniques for translation are fixed, which means that it is not possible to integrate or to extend these approaches, to choose one or another according to specific needs.

The other set of approaches [1, 4, 5, 12, 17] present translations into document models. Similarly to the column family approaches, they provide ways to choose the input tables and to produce the output documents, but having as output embedded documents and also references to other documents. In [12], the authors implemented a six-step algorithm to represent relationships with different cardinalities. [17] describe a graph-transformation that captures the dependencies between the input tables, where tables are vertices and dependencies are edges. This algorithm is coupled with an extension model to denormalize the data, in three different ways (simple, vertical or horizontal). [1, 4, 5] approaches need additional input. [1] presents the R2NoSQL tool, that enables to create mappings that use a table classification scheme to transform the data. [5] presents a table-like-structure (TLS) that represents the input tables. The elements of the TLS are classified into four groups (codifier, simple or complex entity and N:N link), and the translation algorithm creates a tree based on this classification, which will be used to guide the algorithm. In [4], it is necessary to add description tags (frequent join, big size, frequent modify and insert), which are extracted from the database log. Depending on the tags, a given translation strategy is chosen, and an output graph is produced, which can be manually modified. In addition, [4, 5] use an internal representation to capture the de-normalization process and data conversion. The main advantage is to be able to customize the output after the initial discovery process. Still, they cannot be extended to integrate different methods.

Our approach has a different contribution focus: instead of creating an automatic extraction method, it enables to capture the main methods of the existing approaches in a form of a customizable DAG, which represents the NoSQL entities. The DAG is used as input to generate a set of commands. The commands can also be changed in order to apply specific translations strategies. The current version provides strategies for transforming RDB data into tables composed of column families, embedded documents, embedded document array or references. This mechanism allows our approach to express most of the conversion techniques proposed by related works, for both column families and document models.

5 CONCLUSIONS

In this article we have presented an approach to convert RDB to NoSQL nested models, and we provided an implementation for document and column family stores. We use a customizable directed acyclic graph (DAG) to represent the NoSQL entities, which serves as input to generate a set of commands and UDFs to convert the RDB entities to NoSQL entities. These commands can be modified by the user to describe new strategies for transforming the data. We provide strategies for transforming RDB data into tables composed of column families, embedded documents, embedded document array or references. This mechanism allows us to express the conversion techniques presented by different related works, for both column family and document models. It supports 1:1, 1:N and N:N transformations, through the combination of two kind of functions *Map* and *MapReduce*. We implemented a tool, on top of Apache Spark and we performed experiments to validate the approach, where conversions were carried out to document and column family NoSQLs, which has shown to be effective.

As future work, we will incorporate new strategies to convert RDB to NoSQL, where the user can select which one best suits their requirements. We will also develop a graphical interface to assist the user in building the DAG and to configure parameters of the data conversion process. In addition, we will conduct experiments to measure the performance of our approach relative to other conversion tools and using available Open Data sources.

ACKNOWLEDGMENTS

This work was partially funded by CAPES, Sistema de Monitoramento de Políticas de Promoção da Igualdade Racial (SNPPIR), Simulador de Custo-Aluno-Qualidade (SASE/MEC) and UTFPR.

REFERENCES

[1] M.C. De Freitas, D.Y. Souza, and A.C. Salgado. 2016. Conceptual mappings to convert relational into NoSQL databases, Maciaszek L. Hammoudi S. Maciaszek L. Camp O. Cordeiro J. Missikoff M.M., Cordeiro J. (Ed.). *Proceedings of the 18th International Conference on Enterprise Information Systems* 1 (2016), 174–181.

[2] G. Dos Santos Ferreira, A. Calil, and R. Dos Santos Mello. 2013. On providing DDL support for a relational layer over a document NoSQL database. *ACM International Conference Proceeding Series* (2013), 125–132.

[3] Dave Jaffe and Todd Muirhead. 2005. The open source dvd store test application. Retrieved August 28, 2018 from https://linux.dell.com/dvdstore/

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

[5] G. Karnitis and G. Arnicans. 2015. Migration of Relational Database to Document-Oriented Database: Structure Denormalization and Data Transformation, Merkuryev Y. Al-Dabass D. Romanovs A., Merkuryeva G. (Ed.). *Proceedings - 7th International Conference on Computational Intelligence, Communication Systems and Networks, CICSyN 2015* (2015), 113–118.

[6] R. Lawrence. 2014. 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. 285–290.

[7] C.-H. Lee and Y.-L. Zheng. 2015. SQL-To-NoSQL Schema Denormalization and Migration: A Study on Content Management Systems. *Proceedings - 2015 IEEE International Conference on Systems, Man, and Cybernetics* (2015), 2022–2026.

[8] Pramod J. Sadalage and Martin Fowler. 2012. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence* (1st ed.). Addison-Wesley Professional.

[9] M.Y. Santos and C. Costa. 2016. Data models in NoSQL databases for big data contexts. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9714 (2016), 475–485.

[10] G.A. Schreiner, D. Duarte, and R. Dos Santos Mello. 2015. SQLtoKeyNoSQL: A layer for relational to key-based NoSQL database mapping. *17th iiWAS 2015 - Proceedings* (2015).

[11] D. Serrano, D. Han, and E. Stroulia. 2015. From Relations to Multi-dimensional Maps: Towards an SQL-to-HBase Transformation Methodology. *Proceedings - 2015 IEEE 8th International Conference on Cloud Computing* (2015), 81–89.

[12] L. Stanescu, M. Brezovan, and D.D. Burdescu. 2016. Automatic mapping of MySQL databases to NoSQL MongoDB, Maciaszek L. Ganzha M., Paprzycki M. (Ed.). *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016* (2016), 837–840.

[13] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. 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.* 1150–1160.

[14] T. Vajk, P. Feher, K. Fekete, and H. Charaf. 2013. Denormalizing data into schema-free databases. *4th IEEE Int. Conf. CogInfoCom - Proc.* (2013), 747–752.

[15] Matei Zaharia, Reynold Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65.

[16] G. Zhao, L. Li, Z. Li, and Q. Lin. 2014. Multiple nested schema of HBase for migration from SQL, Ogiela M.R. Xhafa F. Yoshihisa T. Barolli L., Li J. (Ed.). *Proceedings - 2014 9th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2014* (2014), 338–343.

[17] G. Zhao, Q. Lin, L. Li, and Z. Li. 2014. Schema conversion model of SQL database to NoSQL, Ogiela M.R. Xhafa F. Yoshihisa T. Barolli L., Li J. (Ed.). *Proceedings - 2014 9th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2014* (2014), 355–362.