Computing (2020) 102:221–246
https://doi.org/10.1007/s00607-019-00736-1

Bringing SQL databases to key-based NoSQL databases: a canonical approach

Geomar A. Schreiner[1] · **Denio Duarte**[2] · **Ronaldo dos Santos Mello**[1]

Received: 14 December 2017 / Accepted: 24 June 2019 / Published online: 29 June 2019

Abstract

Big Data management has brought several challenges to data-centric applications, like the support to data heterogeneity, rapid data growth and huge data volume. NoSQL databases have been proposed to tackle Big Data challenges by offering horizontal scalability, schemaless data storage and high availability, among others. However, NoSQL databases do not have a standard query language, which bring on a steep learning curve for developers. On the other hand, traditional relational databases and SQL are very popular standards for storing and manipulating critical data, but they are not suitable to Big Data management. One solution for relational-based applications to move to NoSQL databases is to offer a way to access NoSQL databases through SQL instructions. Several approaches have been proposed for translating relational database schemata and operations to equivalent ones in NoSQL databases in order to improve scalability and availability. However, these approaches map relational databases only to a single NoSQL data model and, sometimes, to a specific NoSQL database product. This paper presents a canonical approach, called *SQLToKeyNoSQL*, that translates relational schemata as well as SQL instructions to equivalent schemata and access methods of any key-oriented NoSQL database. We present the architecture of our layer focusing on the mapping strategies as well as experiments that evaluate the benefits of our approach against some state-of-art baselines.

Keywords Data interoperability · Cloud computing · Relational-cloud mapping · NoSQL · Big data

✉ Geomar A. Schreiner
schreiner.geomar@prosgrad.ufsc.br

Denio Duarte
duarte@uffs.edu.br

Ronaldo dos Santos Mello
r.mello@inf.ufsc.br

1 Federal University of Santa Catarina, Florianópolis, Brazil

2 Federal University of Fronteira Sul, Chapecó, Brazil

Mathematics Subject Classification 68P15

1 Introduction

Relational databases (RDB) and SQL have been the preferred technologies to store and manage data for decades. However, we have witnessed a tremendous growing of the size of data sets in several application domains over the last years. These data sets are often loosely structured, schemaless and heterogeneous—the so-called *Big Data*—, and their management is challenging since, in general, high availability and scalability are required. Social networks, sensor networks, and healthcare are examples of data-centric applications that produce this kind of data. Cloud computing-based approaches for data management are raising as a promising solution to deal with Big Data [1]. Distributed data centers accessed through the Internet are a typical system architecture in this context. Although RDB are very popular for data management, they are not suited to Big Data-centric applications because they respect the ACID properties for data manipulation, which are orthogonal to the availability and scalability requirements for Big Data manipulation. In fact, the overhead introduced to guarantee ACID transactions may be prohibitive when a large volume of data must be handled. Besides, the fixed record format of relational data, known as *schema first, data later* paradigm [16], also introduces modeling and storage challenges for data instances that do not respect a schema. This variety of representation is also a typical Big Data issue.

NoSQL databases (NoSQL DB) have been proposed to overcome these problems [21]. NoSQL DB support scalability and elasticity requirements to efficiently manipulate data sets that can increase in size quickly. They are based on new data models that better represent heterogeneous (and complex) data instances (also known as *data first, schema later* paradigm [16]) and provide horizontal elasticity instead of the (limited) vertical elasticity supported by most of RDB management systems (RDBMS). Horizontal elasticity leverages performance for Big Data management since new machines can be added or removed based on the application storage needs [7,21]. Usually, the data models of NoSQL DB are organized into four categories: (i) document-oriented (e.g., Mongo DB and SimpleDB), (ii) column-oriented[1] (e.g., Cassandra and Cloudy), (iii) key-value (e.g., Voldemort and Redis), and (iv) graph (e.g., Neo4J).

Based on this motivation, many organizations have been moving their relational data to DB in the cloud (*DB-as-a-service - DBaaS*), in particular, NoSQL DB. However, the cost of this moving is high due to the new paradigm that must be faced. The database access interface is the most challenging one: developers are used to define and manipulate data using the RDB SQL language. Instead, NoSQL DB provide different access methods and access languages depending on their data model or specific product, and usually have limited (or not) support to the SQL standard. As a consequence, the learning curve to start using NoSQL DB is very steep due to these differences in terms of data representation and data accessing.

[1] Some authors, like [7], use the terms *extensible record stores*, *wide column stores* or simply *columnar* as synonyms to *column-oriented databases*.

To adapt applications to new computational environments brings some risks to the organizations, and the existing solutions that deal with this problem may be organized into three categories [6]: (i) *redevelopment*, which rewrites the existing applications from the scratch; (ii) *wrapping*, which provides a new interface to a software component, making it more easily accessible by other components; and (iii) *migration*, which moves the application to the new environment, while retaining the original system's data and functionality. The choice for one of these solutions depends on an evaluation of the costs, like the amount of required changes, as well as the involved risks. The first solution is more expensive since it requires a whole system re-implementation. The third one requires less effort than the first one considering that not all the system will be recoded. Instead, the second solution is the less costly one as it usually provides a faster moving strategy. In this case, the *wrapping component* acts as an interface to a service that performs some processing required by an external client that does not need to know how the service is implemented.

The approach we propose in this paper minimizes the cost of moving of relational-based applications to NoSQL DB in the cloud by following this *wrapping* strategy. Although *wrapping* is a provisional solution, it offers a fast way to deploy applications to new platforms while, for example, a new application is developed (*redevelopment* category).

Related work for *SQL-to-NoSQL* mapping based on *wrapping* usually adopt one of these two strategies: (i) to modify the storage system of a RDBMS kernel, allowing the RDBMS to store data in a NoSQL DB [3,10,24]; or (ii) to develop a layer that translates SQL instructions to corresponding access methods to be executed at the target NoSQL DB [2,8,9,15,19]. Our approach, called *SQLtoKeyNoSQL*, fits into the second strategy. We propose a canonical model that maps a subset of SQL instructions to a hierarchical structure organized as a tree T. T, in turn, can be mapped to any key-based NoSQL DB (document-oriented, column-oriented and key-value). We argue that we have a *canonical approach* because we provide a transparent interoperability of an application with a RDB access interface to one or more key-based NoSQL DB. The existing approaches that provide interoperability between RDB and NoSQL DB do not offer a comprehensive solution, i.e., they focus on a specific NoSQL DB product. We give more details about the differences between our approach and the related work in Sect. 4.

This paper extends several points of our previous work that introduces SQLtoKeyNoSQL [22]. First of all, we review and detail the formal definitions for the canonical model and all the schema and SQL instruction mappings in order to let them more precise. This revision is essential mainly for reproducibility purposes. Secondly, we enhance our access methods for NoSQL DB by adding a new (and optimized) method for retrieving blocks of data. In the previous version of SQLtoKeyNoSQL, data were accessed row by row (*i.e.*, for each retrieved row, we should make an access to the NoSQL DB). Despite being a more technical contribution, this new access method reduces the number of effective access to the target NoSQL DB by retrieving a block of rows on each access, which increases the performance of our solution. We show that through our experiments (Sect. 5). Thirdly, we provide join queries for our approach. We allow the join of data coming from different target NoSQL DB (*e.g.*, join over three tables, one stored in Cassandra database, and the other two in Redis

database) by executing the efficient and well-known *merge join* and *hash join* algorithms. Besides, developers can implement their own join algorithm and set it as the default join strategy at SQLtoKeyNoSQL. The last contribution is a new set of experiments. In our previous work, we evaluate the overhead of our approach. In this paper we conduct new experiments that compare our approach against two state-of-the-art baselines: *SimpleSQL* [9] and *Unity* [15]. We provide a fair comparison by considering the same types of SQL instructions, NoSQL target, computational environment and configurations for each approach, as detailed in Sect. 5. The results show that our approach outperforms *SimpleSQL* and *Unity* concerning processing time to query relational data stored in NoSQL DB.

The remaining of this paper is organized as follows. Next section presents the fundamentals of key-based NoSQL data models. Section 3 details the SQLtoKeyNoSQL approach. Section 4 discusses related work, Sect. 5 presents an experimental evaluation, and Sect. 6 is dedicated to the final considerations and future work.

2 Key-based NoSQL data models

NoSQL DB have been proposed to manage highly heterogeneous and voluminous data efficiently. They can be defined as a database that is not relational and have six properties [7,21]: (i) horizontal scaling; (ii) ability to store complex data in a distributed way; (iii) simple access interface or protocol for manipulating data; (iv) relaxed/non-existent ACID support; (v) high availability; and (vi) optional and flexible schema.

NoSQL DB have independent designs, each one with specific data models that support complex data. In the literature, we find different taxonomies related to the data models of the NoSQL DB [7,21]. In this paper, we consider the four categories of NoSQL data models defined in [21]: (i) document-oriented; (ii) column-oriented; (iii) key-value; and (iv) graph.

The canonical model proposed in this paper supports the first three categories, which are called *key-based NoSQL data models*. Any NoSQL database in this family is able to retrieve an individual data object given an input key, but different NoSQL DB may differ in terms of accessing internal object components [4]. Besides, each key-based NoSQL DB may offer different access methods and protocols. Several of them support the REST API [11], but this is not a standard. We define the common concepts of each key-based NoSQL data model in the following. We illustrate each definition using an extract from a *Cars* RDB presented in Table 1.

The key-value data model is the simplest NoSQL data model. It is composed of a set of key-value pairs, with the value being accessed through a key. A value can maintain a simple or complex content, but this content cannot be queried, *i.e.*, it is a "black-box" content. Because of this, we assume that any value in a key-value data model has an atomic domain. A database based on the key-value model is defined as follows.

Definition 1 *(Key-Value Database)* A key-value database db is a tuple $db = (n_{db}, KV_{db})$, where n_{db} is the name of the database, KV_{db} is the set of key-value pairs, and db is accessed by n_{db}.

Definition 2 *(Key-Value Set)* A key-value $kv \in KV_{db}$ is defined as kv = (key: value), being each $kv.key$ a unique value, and $kv.value$ holds an atomic value.

Example 1 shows an extract of a key-value database based on the first tuple of *Brands* of *Cars* database (Table 1).

Example 1 The Definitions 1 and 2 applied to an extract of Cars RDB results in *db = (Cars,{(Brands.1.id:1), (Brands.1.name:Ford), (Brands.1.year: 1903)})*, where *Cars* is the database name, and *(Brands.1.id: 1)* is one of the key-value pairs.

The document-oriented data model is a specialization of the key-value data model. A document encompasses a set of key-value pairs, and each document is accessed by a unique atomic key. However, a document content is composed of a set of simple or complex attributes. A simple attribute holds an atomic value, and a complex attribute has a list, set or tuple domain.

The document-oriented data model is composed of a database, as well as collections, documents (items), attributes and values [21], as defined in the following.

Definition 3 *(Document Database)* A document database D is a tuple $D = (n_D, \mathcal{C}_\mathcal{D})$, where n_D is the name of D, $\mathcal{C}_\mathcal{D}$ is a set of document collections, and D is accessed by n_D.

Definition 4 *(Document Collection)* A document collection $dc \in \mathcal{C}_\mathcal{D}$ is a tuple $dc = (k_{DC}, DOCS)$, where k_{DC} is the key of dc, $DOCS$ is a set of documents, and dc is accessed by k_{DC}.

Definition 5 *(Document)* A document $d \in DOCS$ is a tuple $d = (k_d, A)$, where k_d is the key of d, A is a set of attributes, and d is accessed by k_d.

Definition 6 *(Attribute)* An attribute $\alpha \in A$ is a pair $(k_\alpha : v)$, where k_α is the key of α, v holds a value whose domain can be atomic, a list, a set, or a tuple, and α is accessed by k_α.

The following example shows a document-oriented modeling based on Table 1.

Table 1 An extract of a database about cars

Table brands			
Id	Name	Country	Founded
1	Ford	USA	1903
2	BMW	Germany	1916
3	Renault	France	1899

Table models				
Id	Name	Prod_begin	Prod_end	Brand_id
1	Clio	1990	–	3
2	Corcel	1968	1986	1
3	E65	2002	2008	2

Example 2 The Definitions 3–6 applied to an extract of Cars RDB results in D = *(Cars, {(Models,{(Model_1(id:"1", name: "Clio"))})})*, where *Cars* is the document database, *Models* is the key of the single document collection in *Cars*, *Model_1* is the key of the single document in *Models*, and *(id:"1")* and *(name: "Clio")* are attribute pairs $(k_\alpha: v)$ of *Model_1*.

Finally, the column-oriented data model represents data properties based on a column-distributed schema. It is composed of a keyspace, column family, column set accessed by a unique key, columns and values [21], as defined in the following.

Definition 7 *(Keyspace)* A keyspace K is a tuple $K = (n_K, F)$, where n_K is the name of K, F is a set of column families, and K accessed by n_K.

Definition 8 *(Column Family)* A column family $f \in F$ is a tuple $f = (n_f, S_c)$, where n_f is the name of f, S_c is a set of column sets, and f is accessed by n_f.

Definition 9 *(Column Set)* A column set $c_s \in S_c$ is a tuple $c_s = (n_{c_s}, Cols)$, where n_{c_s} is the name of c_s, $Cols$ is a set of columns, and c_s is accessed by n_{c_s}.

Definition 10 *(Column)* A column $c \in Cols$ is a tuple $c = (n_c, v)$, where n_c is the name of c, v is an atomic value, and c is accessed by n_c.

Example 3 shows a column-oriented modeling based on Table 1.

Example 3 The Definitions 7–10 applied to an extract of *Cars* database may be represented by a column-oriented database K = *(Cars,{(Models,(row1,{id:"1",name: "Clio"}),Brands(row1,{id:"3",name: "Renault"}))})* where *Cars* is a keyspace, *Models* and *Brands* are column families, *row1* is a column set, and *(id:"1")* and *(name:"Clio")* are columns with their respective values.

Definitions 1–10 are the basis for the definition of our canonical hierarchical model as well as the mapping rules adopted by our approach. We detail them in the next section.

3 The SQLtoKeyNoSQL approach

SQLtoKeyNoSQL is a layer to allow relational access to data stored in NoSQL DB. In order to guarantee a transparent and general access to any key-based NoSQL DB, our approach maps a relational schema to an intermediary canonical model that abstracts the target NoSQL data models. In fact, these data models can be generalized to two concepts (key and $value$), and the canonical model represents keys and values in a simple hierarchical structure, as presented in Sect. 3.1. Our layer also maps SQL instructions to intermediary methods based on the REST API methods (get, put and $delete$), given that most of key-based NoSQL DB supports this API for data accessing.

In the following, we present the canonical model, the mapping strategies accomplished by our layer as well as its architecture.

3.1 Canonical model

The proposed canonical model is composed of a set of key and value nodes organized in a hierarchical structure that is able to represent a relational schema. Besides the root node, it is limited to three key node levels and one atomic value in the leaf nodes. As our model has these specific structural constraints, we decided not to use other available hierarchical data models, like XML[2] and DOM,[3] which are less constrained and more complex at the same time.

The definition of a schema based on our canonical model is given as follows.

Definition 11 *(Canonical Schema)* A canonical schema Can is a tree structure defined as $Can = (n_{root}, S_{L1})$, where n_{root} is a node with a property that holds the RDB name, and $S_{L1} = \{k_{L1_1}, \ldots, k_{L1_n}\}$ is the set of **First Level Keys**, being each $k_{L1_i} \in S_{L1}$ a child node of n_{root} that represents a mapped RDB relation.

Definition 12 *(First Level Key)* A first level key $k_{L1_i} \in S_{L1}$ is a tuple $k_{L1_i} = (n_{L1_i}, S_{L2})$, where n_{L1_i} is a node property that holds a RDB relation name, and $S_{L2} = \{k_{L2_1}, \ldots, k_{L2_o}\}$ is a set of **Second Level Keys**, being each $k_{L2_j} \in S_{L2}$ a child node of k_{L1_i} that identifies uniquely (primary key values concatenation) a tuple of an RDB mapped relation.

Definition 13 *(Second Level Key)* A second level key $k_{L2_j} \in S_{L2}$ is a tuple $k_{L2_j} = (n_{L2_j}, S_{L3})$, where n_{L2_j} is a node property that holds the concatenation of primary keys values of a tuple, and $S_{L3} = \{k_{L3_1}, \ldots, k_{L3_p}\}$ is a set of **Third Level Keys**, being each $k_{L3_k} \in S_{L3}$ a child node of k_{L2_j} that represents an attribute of a mapped RDB relation tuple.

Definition 14 *(Third Level Key)* A third level key $k_{L3_k} \in S_{L3}$ is a tuple $k_{L3_k} = (n_{L3_k}, v)$, where n_{L3_k} is a node property that holds an attribute name of a tuple, and v is the child node of k_{L3_k} with a property that holds the value of the attribute represented by k_{L3_k}. We define this single child node of k_{L3_k} as $child(k_{L3_k})$.

Figure 1 shows the *Cars* RDB schema (Fig. 1a—the same of Table 1) and its corresponding schema in the canonical model (Fig. 1b). n_{root} is named as *Cars*. The tables *Brands* and *Models* are mapped to first level key nodes. The primary keys from both tables (id attributes) are mapped to second level key nodes. Attributes are mapped to third level key nodes, and their values are mapped to leaf nodes.

3.2 Mapping strategies

The core of the SQLtoKeyNoSQL approach comprises mapping strategies to translate relational schemas to corresponding key-based NoSQL schemas as well as basic DDL and DML SQL instructions to compatible API REST access methods. We detail both types of mappings in the following.

[2] https://www.w3.org/XML/.

[3] https://www.w3.org/DOM/.

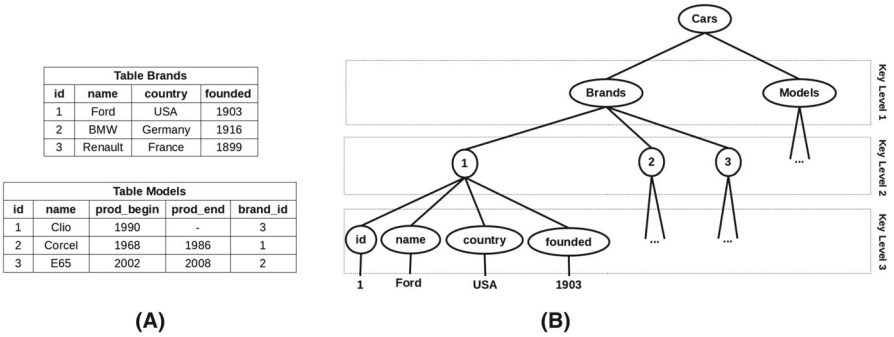

Fig. 1 *Cars* RDB schema (**a**) and the corresponding schema in the canonical model (**b**)

3.2.1 Schema mapping

SQLtoKeyNoSQL supports mappings from the relational model to our canonical model, and from the canonical model to the target NoSQL data model. Since the former is described by the Definitions 11–14, we detail the latter in this section.

As stated before, the canonical model generalizes all key-based NoSQL data models, which simplifies the mapping of canonical schemas to each one of these models. The proposed mappings are stated by the rules that are defined for each one of the three NoSQL data models. Next, we present these rules, beginning with the preliminary definition of the *Node Name* function.

Definition 15 *(Node Name Function)* Given a canonical schema $\mathcal{C}$, the function $name(n)$ returns the property value of a node n that belongs to the tree structure of $\mathcal{C}$.

Rule 01 (Key-Value Mapping) *The mapping of a canonical schema $\mathcal{C}$ to a key-value NoSQL schema proceeds as follows:* (i) *$\mathcal{C}$ maps to a key-value DB bd named $name(n_{root})$;* (ii) *each first level key node $key_1 \in n_{root}.K_{L1}$ and, in turn, each second level key node $key_2 \in key_1.K_{L2_r} (1 \leq r \leq q)$ generate a key $k \in db$ whose name is the concatenation of $name(key_1)$ and $name(key_2)$; and* (iii) *the value of each generated key $k \in db$ is a set of key-value pairs KV_{db}, being each pair $kv \in KV_{db}$ defined as $kv = (name(key_{3l}), v_l)$ where $key_{3l} \in key_2.K_{L3_s} (1 \leq s \leq p)$, i.e., kv has $name(key_{3l})$ as* key *and $name(child(key_{3l}))$ as value v_l.*

Figure 2 shows an example of Rule 01 application considering the canonical schema from Fig. 1b. n_{root} in the canonical schema is mapped to *Cars* (database name). First level keys (table names) are concatenating to each corresponding second level keys (primary key values) to build the key of key-value schema, *e.g.*, *Brands.1* and *Models.2*. Finally, third level keys and their values at the leaf nodes build the value of the key, generating, for instance, (*Brands.1,* {*id: 1; name: Ford; country: USA; founded: 1903*}).

Rule 02 (Document-Oriented Mapping) *The mapping of a canonical schema $\mathcal{C}$ to a document-oriented NoSQL schema proceeds as follows:* (i) *$\mathcal{C}$ maps to a document-oriented DB D named $name(n_{root})$;* (ii) *each first level key node $key_1 \in n_{root}.K_{L1}$*

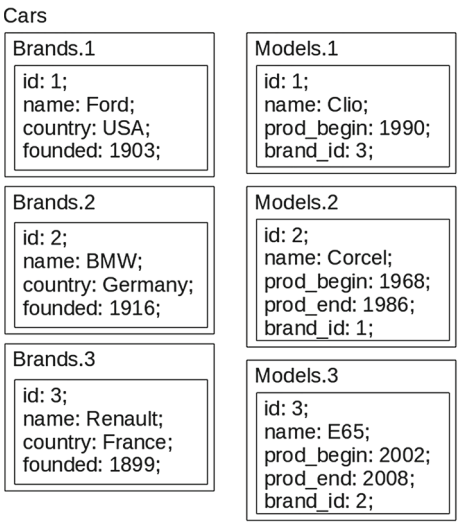

Fig. 2 A key-value schema generated by the mapping of the canonical schema from Fig. 1b

generates a document collection $DC \in D$ *whose access key is* $name(key_1)$*;* (iii) *each second key level node* $key_2 \in key_1.K_{L2_r} (1 \leq r \leq q)$ *generates a document* $d \in DC$ *whose key is* $name(key_2)$*; and* (iv) *each third level key node* $key_3 \in key_2.K_{L3_s} (1 \leq s \leq p)$ *generates an attribute* $a_i \in d$ *whose key is* $name(key_3)$ *and whose value is* $name(child(key_3))$.

Figure 3 presents the document-oriented DB built from Rule 02 over RDB *Cars* (Fig. 1a). The node n_{root} is mapped to the document DB $Cars$. Each first level key ($Brands$ and $Models$) is mapped to a document collection with the same name. Each second level key is mapped to a document in *Cars* DB. For example, $Brands$ collection is composed of three documents with the keys 1, 2 and 3. The third level keys and their child nodes are mapped to document attributes and values. For example, document 1 from $Brands$ is composed of the following attributes: {*id: 1*; *name: Ford*; *country: USA*; *founded: 1903*}.

Rule 03 (Column-Oriented Mapping) *The mapping of a canonical schema* $\mathcal{C}$ *to a column-oriented NoSQL schema proceeds as follows:* (i) $\mathcal{C}$ *is mapped to a* keyspace K *named* $name(n_{root})$*;* (ii) *each first level key node* $key_1 \in n_{root}.K_{L1}$ *generates a column family* $f \in K$ *whose key is* $name(key_1)$*;* (iii) *each second key level node* $key_2 \in key_1.K_{L2_r} (1 \leq r \leq q)$ *generates an access key* $ch \in f$ *whose name is* $name(key_2)$*; and* (iv) *each third level key node* $key_3 \in key_2.K_{L3_s} (1 \leq s \leq p)$ *generates a column* $c \in f$*, indexed by* ch*, whose name is* $name(key_3)$ *and whose value is* $name(child(key_3))$.

Figure 4 exemplifies the application of Rule 03. The root node of the canonical schema n_{root} is mapped to a $keyspace\ Cars$. First level keys are mapped to column families, *e.g.*, $Brands$ and $Models$. Each second level key is mapped to a row key, like row key 1 in the column family $Brands$. Each third level key and its child nodes are mapped to a column and its respective values, *e.g.*, {*id: 1; name: Ford; country: Ford; founded: 1903*} for row key 1 of column family $Brands$.

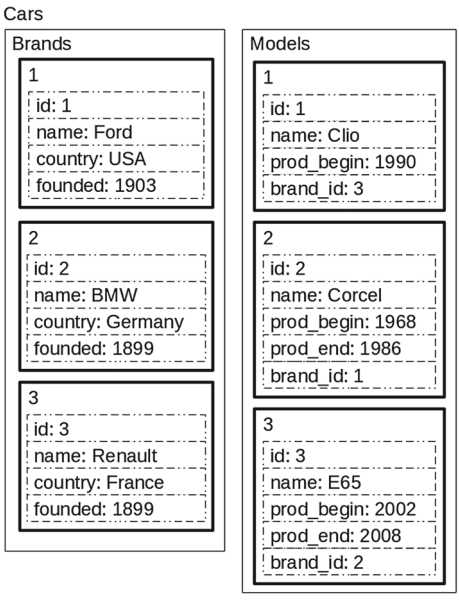

Fig. 3 A document-oriented schema generated by the mapping of the canonical schema of Fig. 1b

Rules 01–03 are considered for organizing and storing data in NoSQL DB. Notice that the canonical data model acts as a simple intermediary schema abstraction. This abstraction is used as a standard to store and access data in different key-based NoSQL DB through a single data model.

SQLtoKeyNoSQL also provides the mapping of SQL instructions in order to support SQL-to-NoSQL interoperability. In next section, we describe the mapping of the main SQL DDL and DML instructions.

3.2.2 SQL instruction mapping

Our layer is also able to map a subset of SQL DDL and DML instructions to corresponding REST API access methods. These mappings are supported by a *dictionary* (see Sect. 3.4) that maintains relevant metadata. In fact, the execution of SQL DDL instructions triggers updates in the dictionary, and during the processing of SQL DML instructions the dictionary is queried to generate suitable REST API methods to be executed at the target NoSQL DB. The considered SQL DDL instructions, as well as their capabilities, are the following:

- `CREATE TABLE` it creates a table definition in the dictionary. Only the table name, attributes, primary and foreign key constraints are considered.
- `ALTER TABLE` it can add, rename or remove attributes. If an attribute is removed, its definition is removed from the dictionary, and all third key level in the canonical schema that corresponds to it are also removed. This action also propagates to the NoSQL DB, *i.e.*, all corresponding attributes and their data are also removed. The same reasoning applies in case of an attribute renaming, i.e., if an attribute name is modified, this operation is accomplished in the dictionary and in all corresponding

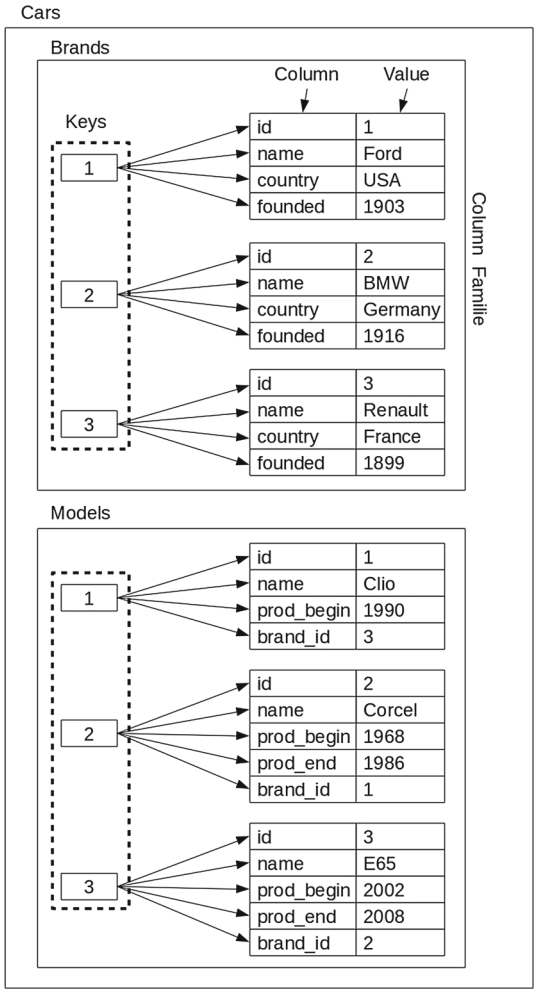

Fig. 4 A column-oriented schema generated by the mapping of the canonical schema of Fig. 1b

data stored in the NoSQL DB. If a new attribute is added, it is just created in the dictionary. Primary key changes are not allowed because the primary key is the basis for data accessing in NoSQL DB.

– `DROP TABLE` it removes table definition from the dictionary, as well as corresponding first level key in canonical schema and corresponding data in the NoSQL DB. Tables can be removed only if other tables do not reference them.

The mapping of SQL DML instructions generates one or more of the following primitive REST API methods: *put* (stores a value based on a key), *get* (retrieves a value based on a key) and *delete* (deletes a value based on a key). The following definitions show how NoSQL DB are accessed by our approach.

Definition 16 *(Canonical Key)* A canonical key is a key obtained by concatenating a first level key $key1 \in n_{root}.K_{N1_m}(1 \leq m \leq o)$ with one of its children nodes key2 $\in key_1.K_{N2_r}(1 \leq r \leq p)$.

As an example of Definition 16, the canonical key $Brands.1$ is the concatenation of the first level key $Brands$ with its child 1.

Definition 17 *(Record)* A record is a set of key-value pairs, each pair is obtained from a third level key $key3$ and its leaf nodes for a given key from second level key $key2$, representing key-value pairs of a given canonical key.

The concept of a record is similar to a tuple in the relational model. For example, $\{id : 1; name : Ford; country : USA; founded : 1903\}$ corresponds to a record in our canonical model, while (1, Ford, USA, 1903) is a tuple in the corresponding relational model. Given these preliminary definitions, we now define the considered primitive REST API methods.

Definition 18 *(Put)* The primitive method $put(k_{put}, v)$ stores a record v corresponding to a canonical key k_{put}.

For example, to store the first tuple of table $Brands$ (Fig. 1a), we issue $put(Brands.1, \{id : 1; name : Ford; country : USA; founded : 1903; \})$.

Definition 19 *(Get)* The primitive method $v = get(k_{get})$ returns a record v from a target NoSQL DB corresponding to a canonical key k_{get}.

The *get* method searches for a given canonical key in a NoSQL DB and retrieves its corresponding value (if exists). For example, $get(Brands.1)$ returns the record $\{id : 1; name : Ford; country : USA; founded : 1903\}$.

Definition 20 *(Delete)* The primitive method $delete(k_{del})$ removes a record identified by the canonical key k_{del}.

The $delete$ method removes a record given a canonical key as the input parameter. For example, $delete(Brands.1)$ removes $Brands.1$ from the $Cars$ DB.

From the get primitive method, we propose the $getN$ method that retrieves a set of records given a set of canonical keys. We use this method to optimize data retrieving by considering NoSQL DB that return blocks of records.

Definition 21 *(GetN)* A method $\mathcal{R} = getN(CK_{all}, FL)$ returns a set of records $\mathcal{R}$ based on a set of canonical keys CK_{all} and a stack of filters FL.

The *getN* method returns a set of records that are stored in a NoSQL DB. The retrieved records must match with the given canonical keys and (possibly empty) filters. For example, $getN(\{Brands.1, Brands.2\}, null)$ returns the records *{id:* 1; *name: Ford; country: USA; founded:* 1903} and *{id:* 2; *name: BMW; country: Germany; founded:* 1916*}*.

Based on Definitions 18–21, we now detail the mapping of the basic SQL DML instructions to NoSQL DB access methods as follows:

- `INSERT` it is translated to a set of *put* methods based on the canonical schema and the dictionary (see Sect. 3.4). Nested queries are not supported. The values are stored based on the given input attributes, and primary key values are required.
- `UPDATE` it is translated to *getN* and *put* methods. One or more tuples can be updated, simple filters over attributes (predicates linked by AND or OR logical connectors) can be used, but nested queries are not supported.
- `DELETE` it is translated to *getN* and *delete* methods. In the same way of the `UPDATE` instruction, one or more tuples may be deleted based on simple filters without nested queries.
- `SELECT` it is translated to a set of *getN* methods. Projections, selections, and joins are allowed, but nested queries, aggregations and ordering are not implemented yet.

In the following, we present the layer architecture that manages all of these mappings as well as the dictionary.

3.3 Architecture

The architecture of the SQLtoKeyNoSQL layer is composed of seven modules, as illustrated in Fig. 5. Each module, in the current version, is implemented using JAVA 8 language.

The first module is the *Access Interface*, which receives SQL instructions from a relational-based application (*Relational App*) or an *Ad-Hoc query* and sends them to the *SQL Parser* module. It also receives results from by the *Execution Engine* module (result sets and/or messages) and, in turn, forwards them to the external components.

The *SQL Parser* module receives an SQL instruction and accomplishes syntactic and semantic analysis with the support of the *Dictionary* module. If the instruction is a `SELECT`, `DELETE`, or `UPDATE`, it further sends it to the *Query Planner* module. Otherwise, it sends the instruction directly to the *Translator* module. The *Query Planner* defines a plan that optimizes query execution. Currently, this module can optimize queries that use filters connected by the `AND` operator. The optimization guarantees that filters over specific tables are executed with high priority to reduce the data volume

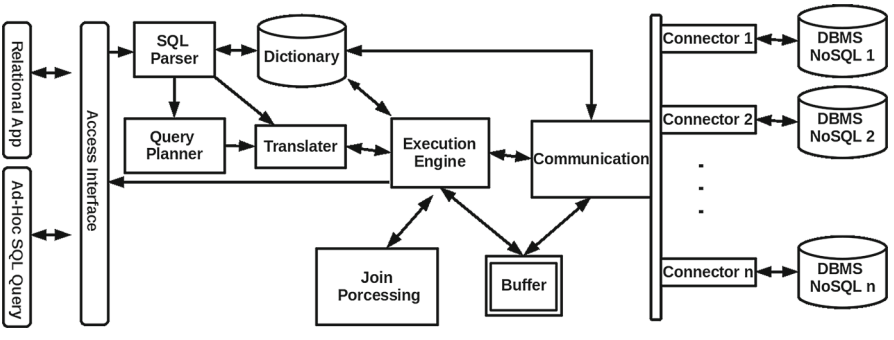

Fig. 5 SQLtoKeyNoSQL layer architecture

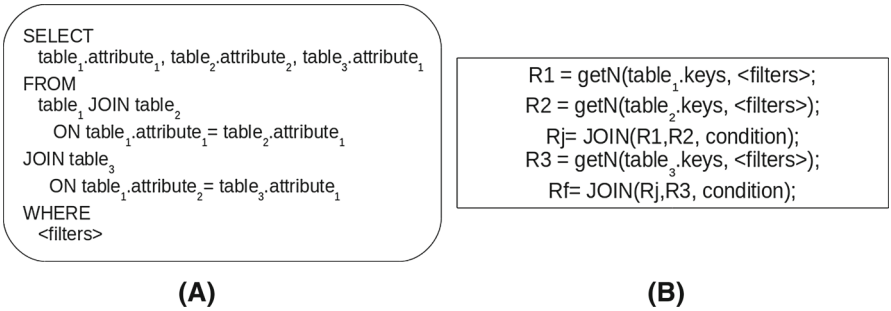

Fig. 6 An input SQL query (**a**) and the set of primitive methods generated as output (**b**) by the *Translator* module

to be processed by further (and more expansive) join operations. The output of this module is a query tree that is translated by the *Translator* module and further executed by the *Execution Engine* module.

The *Translator* module receives an SQL instruction or a query plan as input and translates it as described in Sect. 3.2. The output access methods are then sent to the *Execution Engine* module. This output is a stack of primitives or extended methods. For example, Fig. 6a shows an input SQL query and Fig. 6b shows the respectively output generated by the *Translator* module. The stack of output methods first filters data on each pair of tables (*getN* methods) before joining them.

The *Execution Engine* module handles the execution of methods with the support of the *Communication* module and metadata stored in the *Dictionary*. It is responsible for processing filters over returned data, sending (and receiving) data sets to (from) the *Join Processing* module, and generating the result set or messages to be sent to the *Access Interface* module. In the stack of methods of Fig. 6b, for example, the *Execution Engine* first execute the operation *getN* for $table_1$, then *getN* for $table_2$. The two result sets $R1$ and $R2$ are sent to *Join Processing* module with the join condition. After receiving the join result (Rj), it executes the *getN* for $table_3$, which produces the result set $R3$. Finally, Rj and $R3$ are sent to the *Join Processing* module with the next join condition, and the result set Rf is sent to the *Access Interface* module.

The *Join Processing* module executes joins between data sets under the control of the *Execution Engine* module. Each *getN* operation returns a set of records. These sets of records and the join condition are passed to the *Join Processing* module. The *Join Processing* module, in turn, executes a join algorithm that combines the records based on the join condition. Finally, the resulting join records are returned to the *Execution Engine*. Multiple joins are supported and are executed in a left to right order.

The current version of SQLtoKeyNoSQL implements a join operation in two flavors: *Merge-Join* (for data that do not fit in the main memory), and *Hash-Join* (for data that fit in the main memory). Our implementations are based on classical join algorithms [17].

Finally, the *Communication* module executes requested access methods over one or more NoSQL DB. It is composed of connectors (*wrappers*) that translate the *getN*, *put* and *delete* methods to the specific signature of the target NoSQL DB access methods.

Such translations usually perform little syntactic adjustments in the method parameters. If a NoSQL database does not support the retrieval of a set of data at a time, the *getN* method is first mapped to a set of *get* methods. Under the hood, each NoSQL target is represented by a java class connector that needs to implement a uniform interface called *Connector*. The *Connector* interface provides a method for each primitive/extend operation (*get, set, delete*, etc), and each NoSQL target needs to implement the methods of the *Connector* class using its specific API. The *Communication* module accesses data through the *Connector* interface using polymorphism with the specific connector for each NoSQL target. Data returned from the NoSQL DB are sent back to *Execution Engine* through the *Buffer* component of the SQLtoKeyNoSQL layer.

Another important component of the SQLtoKeyNoSQL architecture is the *Dictionary*. It holds the metadata necessary to perform all the mappings, and it is detailed in the next Section.

3.4 Dictionary

The *Dictionary* maintains metadata for each considered RDB schema (attributes, primary keys, foreign keys, among others), as well as information about the target NoSQL DB where the data are stored. The dictionary is defined as follows.

Definition 22 *(Dictionary)* A dictionary $\mathcal{D}$ is a tuple $\mathcal{D} = (\mathcal{T}, \mathcal{N})$, with $\mathcal{T}$ being a set of RDB table metadata and $\mathcal{N}$ a set of target NoSQL DB.

Definition 23 *(Table Metadata)* A table metadata $t \in \mathcal{T}$ is a tuple $t = (name, ATT, PK, FK, KEYS, db)$, where $name$ is the table name, ATT is the set of attribute names of the table, PK is the primary key of the table, $FK = \{(att_1, tname_1), \ldots, (att_n, tname_n)\}$ is a set (possibly empty) of foreign keys of the table, where each pair $(att_i, tname_i) \in FK$ holds the attribute name of a foreign key and the name of the referenced table, $KEYS$ are the set of the keys (third level key node names in the canonical schema) of the table, and db is the alias of the target NoSQL DB.

Definition 24 *(Target NoSQL)* A target NoSQL $bd \in \mathcal{N}$ is a tuple $bd = (alias, user, psw, url)$, being $alias$ a unique name for the NoSQL DB, $user$ and psw the user and password to connect to the NoSQL DB, respectively, and url the address of the NoSQL DB.

Notice that the canonical model maps the relational data in a hierarchical model using a tree view of the database (tree root), table, tuple identifier (the concatenation of the primary key values of each tuple), columns and values (tree leaves). The relationships between tables or tuples are not explicitly defined in the canonical model. They are maintained only in the dictionary.

Figure 7 presents an example of SQLtoKeyNoSQL dictionary corresponding to the database from Fig. 1a. It shows, for example, that the metadata of the *Models* table is *(Models, ATT: {id, name, prod_begin, prod_end, brand_id}, PK: id, KEYS:* {1, 2, 3}, *FK: {(brand_id, Brands)}, DB: DB2)*. Notice also, from Fig. 7, that table *Brands* is stored at NoSQL DB *DB1* and table *Models* in NoSQL DB *DB2*. It means that RDB may be distributed among several NoSQL DB.

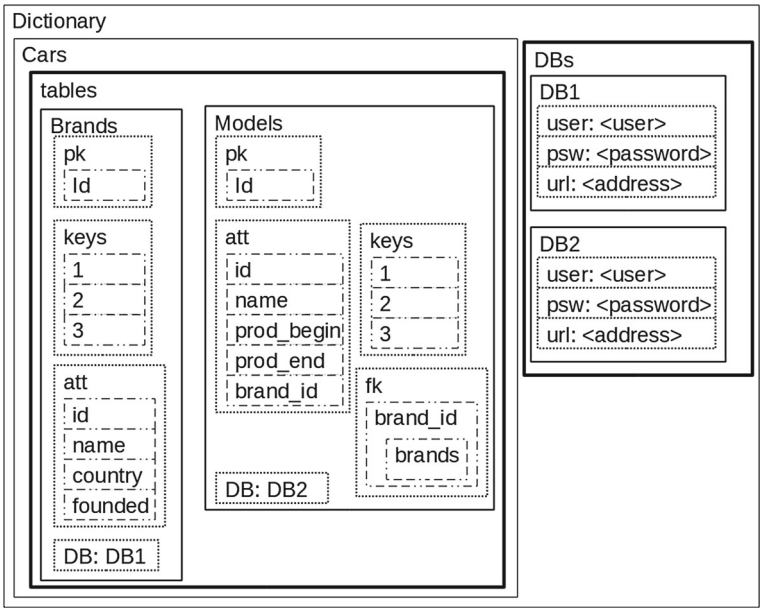

Fig. 7 Example of the SQLtoKeyNoSQL dictionary schema

4 Related work

Many works are dealing with the mapping of RDB or SQL instructions to NoSQL DB. These works follow several approaches. Some of them build a unified layer over different NoSQL DB introducing SQL-like languages [25] or using other query languages to access data [4,5,23]. There are also works that present migration techniques to move a relational-based application to NoSQL DB helping the users to rewrite their SQL instructions to the NoSQL API methods [12–14,20].

Different from the approaches mentioned above, our solution fits into related work that offers a way to query/update NoSQL DB using traditional SQL instructions. As stated in Sect. 1, the related work on which our solution belongs falls into two categories: *layer* and *storage engine*. The first one comprises approaches based on a software layer that provides a schema and operation abstraction over NoSQL DB, allowing users to define and manipulate relational data through SQL instructions. The second one comprises approaches that modify the kernel of a RDBMS to store relational data in a NoSQL DB.

Due to the limited horizontal space, we present two tables to show the related approach features. Table 2 focuses on the: (i) approach category; (ii) target NoSQL DB; and (iii) data model of the target NoSQL DB. Table 3 focuses on the: (i) SQL instructions supported; (ii) dictionary support; and (iii) type of supported join (if considered).

According to Table 2, four approaches of the category *Layer* are found. *SimpleSQL* [9] is a layer that supports the mapping of a subset of SQL DDL and DML instructions and stores data in SimpleDB, a document-oriented NoSQL DB. *JackHare*

Table 2 Related work comparison (Part 1)

Approach	Category	NoSQL DB	Data model
SimpleSQL (2013)	Layer	SimpleDB	Document
JackHare (2013)	Layer	HBase	Column
Unity (2014)	Layer	Cassandra/MongoDB	Column/document
Rith et al. [19]	Layer	Cassandra/MongoDB	Column/document
Apache Phoenix (2014)	Layer	HBase	Column
Phoenix (2011)	Storage engine	Scalaris	Key-value
CloudyStore (2009)	Storage engine	Cloudy	Column
DQE (2013)	Storage engine	HBase	Column
SQLtoKeyNoSQL	Layer	Key-oriented	Key-oriented

Table 3 Related work comparison (Part 2)

Approach	SQL support	Dictionary	Join
SimpleSQL (2013)	DDL + DML subset	Yes	By similarity
JackHare (2013)	DDL + DML subset	Yes	Map-reduce
Unity (2014)	DML subset	Yes	Hash-Join
Rith et al. [19]	DML subset	–	–
Apache Phoenix (2014)	DML + DML	Yes	Hash-Join
Phoenix (2011)	DDL + DML	No	RDBMS-dependent
CloudyStore (2009)	DDL + DML	Yes	RDBMS-dependent
DQE (2013)	DDL + DML	Yes	RDBMS-dependent
SQLtoKeyNoSQL	DDL + DML subset	Yes	Merge-Join/Hash-Join

[8] is also a relational layer, but different from *SimpleSQL*, it provides mappings to HBase, a column-oriented NoSQL DB. Another work that considers a target document-oriented DB is *Unity* [15], but its SQL support is limited to a subset of DML instructions. Rith et. al. [19] is capable of accesses data stored in the Cassandra and MongoDB. The last layer approach is *Apache Phoenix* [2], which accesses and stores data in HBase.

There are also three approaches of the category *Storage Engine*. Two of them (*Phoenix* [3] and *CloudyStore* [10]) modify the MySQL RDBMS storage engine to provide persistence of relational data in NoSQL DB. As our approach, *Phoenix* is based on an intermediary model called VOEM (*Value-based OEM*), which is an extension of the OEM (*Object Exchange Model*) [18], a data model that is more complex than our canonical model mainly in terms of number of concepts. Besides, it supports the mapping of VOEM schemata only to the key-value data model, specifically, to the Scalaris NoSQL database. Different from *Phoenix*, *Cloudy Store* does not provide an intermediary model, managing the mapping of relational data to the column-oriented NoSQL DB called Cloudy, and considering the MySQL $rowIds$ to optimize data accessing stored in Cloudy. The last approach is *DQE* [24], which modifies the kernel

of the Derby RDBMS, including its query optimization module. Similar to *JackHare*, *DQE* stores relational data in the HBase NoSQL database.

Most of approaches are limited to map to only one NoSQL data model. Only Rith et. al. and *Unity* support more than one target data model. Rith et. al. translates SQL queries to the query language of Cassandra and MongoDB using the query properties of each target DB. *Unity* supports multiple data sources, but its mappings must be coded by hand through wrappers. Besides, *Unity* details mappings only for MongoDB. Our approach is more flexible than those ones since we have support to all key-oriented NoSQL DB.

Table 3 shows that approaches of category *Storage Engine* have full support for SQL DDL and DML instructions. This is justified by the fact they are extensions of existing RDBMS that naturally offer SQL-based access. Despite their limited SQL support, *Layer* approaches are more flexible, since they are not strongly coupled to a particular RDBMS.

A challenging task for SQL-to-NoSQL mapping is join operation support, since NoSQL DB do not have this query capability. Table 3 highlights that *Storage Engine* approaches are dependent of the RDBMS join capabilities for such a task. Only the work of Rith et. al. does not support joins in the *Layer* category. *SimpleSQL* applies a join-by-similarity algorithm to match foreign and primary key values. *JackHare* uses map-reduce jobs to take advantage of parallel processing for improving join operation performance. *Unity* and *Apache Phoenix* execute a hash join algorithm that considers the primary and foreign keys as hash entries. Our approach also supports join operation, providing more than one join algorithm depending on whether the data set fits or not into the main memory.

Next section presents an evaluation of SQLtoKeyNoSQL through a set of experiments that compares it with some related work (baseline approaches).

5 Experiments

This section presents a set of experiments conducted to show the effectiveness of the SQLtoKeyNoSQL approach. We focus our experiments on query operations since it is the most frequent operation performed by RDB.

We refer the readers to [22] for experiments that evaluate the overhead introduced by our approach on considering a data-centric application that directly accesses a relational database. In that paper, we executed a set of SELECT and INSERT instructions on three NoSQL databases with and without considering our layer. The results revealed that our solution is not prohibitive.

To compare the processing time of our approach with two baselines, we execute two sets of experiments. We first compare SQLtoKeyNoSQL with Unity using MongoDB as the NoSQL database target. Then, we compare *SimpleSQL* with SQLtoKeyNoSQL using *Amazon SimpleDB* as the NoSQL database target. Unfortunately, we did not find any available open source *Storage Engine* approach to consider in our experiments. We also check the completeness and correctness of our approach: we execute a set of queries directly over a RDB and using SQLtoKeyNoSQL. We compare the returned tuples and in both cases the tuples are the same.

Table 4 Some metadata of the *Prova Brasil* RDB

Tables	PKs	FKs	#Cols	Original rows	Reduced rows
ts_school	id_school	–	128	79,252	79,252
ts_student_3rdhs	id_student	id_school	98	150,430	200,000
ts_student_5th	id_student	id_school	98	2,720,589	200,000
ts_student_9th	id_student	id_school	98	2,524,126	200,000

5.1 Experiment setup

The experiments were performed on an Intel Core i5-2430M processor with 8 GB DDR3 1066mHz RAM, 240GB Scandisk SSD, running Linux 4.5.5-04 kernel (XUbuntu 16.04 distribution). In the experiments, we use two different NoSQL DB as targets: MongoDB and SimpleDB. MongoDB and SimpleDB are document-oriented NoSQL DB. We ran the experiments for each baseline in the same environment. MongoDB ran as local host as a single node without replicas. SimpleDB ran through Amazon AWS accessed by a REST API. We constrain our experiments to MongoDB and SimpleDB because of the restrictions of the baselines.

5.2 Experiment methodology

Our experiments considered, as a use case, a real RDB. This RDB, called *Prova Brasil* (PB_{db}), stores data about the academic performance of students at compulsory level (elementary and high school) in Brazil. We extracted four tables from PB_{db} to perform the experiments: *ts_student_3rdhs*, which stores results of the test from the third year of Brazilian high school students; *ts_student_5th* and *ts_student_9th*, which stores the results of five and nine years students of the basic school, respectively; and *ts_school*, which stores data about all the Brazilian public schools.

Table 4 shows some metadata of PB_{db}. The first column shows the table names. The second and third columns show the primary key and foreign key attributes. The column *#Cols* presents the number of columns of the tables. The next column presents the original number of rows of each table. Due to the network lag and failures, we could not use the cardinality of the original tables for the experiments with SimpleSQL baseline. After some tests, we decide to export only 200,000 rows to each table. Thus, the last column presents the number of rows considered in our experiments (*Reduced #Rows*).

The main working tables are *ts_student_3rdhs*, *ts_student_5th*, and *ts_student_9th*. The table *ts_school* was chosen because it is the largest table (79, 252 rows and 128 attributes) that can be joined with the other 3 tables.

We considered fifteen SQL queries ($Q1$ to $Q15$) in our experiments, as shown in Figs. 3 and 4. For each query, we changed the number of projections columns and the number of filters. The queries were defined based on the SQL support provided by our approach and the baselines. In short, we avoid aggregations and nested queries. Three of the queries perform join operations ($Q13$, $Q14$ and $Q15$).

Table 5 Queries considered in the experiments (Part 1)

Queries	
Q1	**SELECT** id_uf, id_city, id_area, id_shift, id_grade, tx_resp_q001 **FROM** ts_students_3rdhs;
Q2	**SELECT** id_uf, id_city, id_area, id_shift, id_grade, tx_resp_q001 **FROM** ts_students_3rdhs **WHERE** id_city = 6236282;
Q3	**SELECT**id_uf, id_city, id_area, id_shift, id_grade, tx_resp_q001 **FROM** ts_students_3rdhs **WHERE** id_city = 6236282 AND id_shift = 2;
Q4	**SELECT** * **FROM** ts_students_3rdhs;
Q5	**SELECT** id_uf, id_city, id_area, id_shift, id_grade **FROM** ts_students_5th;
Q6	**SELECT** tx_resp_q001, tx_resp_q002, tx_resp_q003, tx_resp_q004, tx_resp_q005, tx_resp_q006, tx_resp_q007, tx_resp_q008 **FROM** ts_students_5th **WHERE** id_uf = 43;
Q7	**SELECT** tx_resp_q001, tx_resp_q002, tx_resp_q003, tx_resp_q004, tx_resp_q005, tx_resp_q006, tx_resp_q007, tx_resp_q008 **FROM** ts_students_5th **WHERE** id_uf $=' 11'$ AND id_location $=' 1'$;
Q8	**SELECT** id_turma, id_shift, id_city, id_block_1, id_block_2, id_grade, id_students **FROM** ts_students_5th **WHERE** id_uf $=' 15'$ AND id_students $>' 11161931'$
Q9	**SELECT** id_uf, id_city, _id_escola, id_students **FROM** ts_students_9th;
Q10	**SELECT** id_turma, id_shift, id_city, id_block_1, id_block_2, id_grade, id_students **FROM** ts_students_5th **WHERE** id_uf $=' 11'$ AND id_students $>' 10913619'$;

To compare SQLtoKeyNoSQL with the baselines, we organize the experiments in two parts. First, we evaluate the *processing time* of the fifteen queries. Second, we evaluate the *scalability* of all approaches. In this test, we randomly picked a query (Q8) from Table 5 based on table *ts_students_5th* (the table with the higher number of rows). In the *scalability* experiment we decided not to evaluate queries with joins ($Q13$ to $Q15$ from Table 6) because the baselines had a poor performance w.r.t. SQLtoKeyNoSQL in the *processing time* experiment.

The execution of each query in the *processing time* experiment considers an initial *warm-up* phase (we ran each query 3 times), and then, we ran each query 5 times and report the average rates. The results were compared employing statistics significance tests (*paired t-test*) with a 95% confidence interval. *T-test* is a type of inferential statistic used to determine if there is a significant difference between the means of two different groups. We apply the test over two groups of values for each query: one group is our approach and the other one a state-of-art approach (depending on

Table 6 Queries considered in the experiments (Part 2)

Queries	
Q11	**SELECT** id_uf, id_city, _id_escola, id_students, tx_resp_q001, tx_resp_q002, tx_resp_q003, tx_resp_q004, tx_resp_q005, tx_resp_q006 **FROM** ts_students_9th **WHERE** id_uf $>'$ 11′ AND id_uf $<'$ 15′;
Q12	**SELECT** id_uf, id_city, id_escola, id_students **FROM** ts_students_9th **WHERE** id_uf $>'$ 10′ AND id_uf $<'$ 15′ AND id_area $='$ 2′;
Q13	**SELECT** ts_students_3rdhs.id_uf, ts_students_3rdhs.id_city, ts_students_3rdhs.id_shift, id_grade, tx_resp_q001 **FROM** ts_students_3rdhs**NATURALJOIN** ts_uf;
Q14	**SELECT** ts_escola.id_escola, ts_students_9th.id_uf, ts_students_9th.id_city, ts_students_9th.id_shift, ts_students_9th.id_grade, ts_students_9th.tx_resp_q001, ts_students_9th.id_escola **FROM**ts_students_9th**NATURALJOIN** ts_escola;
Q15	**SELECT** ts_escola.id_escola, ts_students_9th.id_uf, ts_students_9th.id_city, ts_students_9th.id_shift, ts_students_9th.id_grade, ts_students_9th.tx_resp_q001, ts_students_9th.id_escola **FROM**ts_students_9th **NATURALJOIN**ts_escola **NATURALJOIN**ts_uf **WHERE**ts_students_9th.id_location = 1;

the experiment). The hypothesis to be verified is that our approach produces a better performance (a reduced execution time) than the baselines.

For the the *scalability* experiment, we execute scripts that insert different numbers of synthetic rows in the *ts_student_5th* table. This experiment was divided into 5 parts based on the inserted rows: 500,000, 1,000,000, 1,500,000, 2,000,000 and 2,500,00 (in the case of SimpleDB: 40,000, 80,000, 120,000, 160,000 and 200,000 rows). We executed query $Q8$ 5 times and got the average rates. Again, the results were compared employing statistics significance tests (paired t-test) with a 95% confidence interval.

5.3 SQLtoKeyNoSQL vs unity

Figure 8 shows the processing time comparison of our approach (SQLtoKeyNoSQL) with *Unity*. The bar graph shows, in the x-axis, the queries and, in the y-axis, the corresponding processing time in seconds.

The results show that our approach obtained better performance w.r.t. *Unity* for all queries. One possible reason for that is our abstract method $getN$, which considers the MongoDB capability for retrieving blocks of data. Instead, Unity fetches only one record at a time.

Notice that the biggest difference is related to the join queries ($Q13$, $Q14$, and $Q15$). Our approach has a significant lower processing time for all of them by running classical (and efficient) join algorithms and prioritizing main memory processing

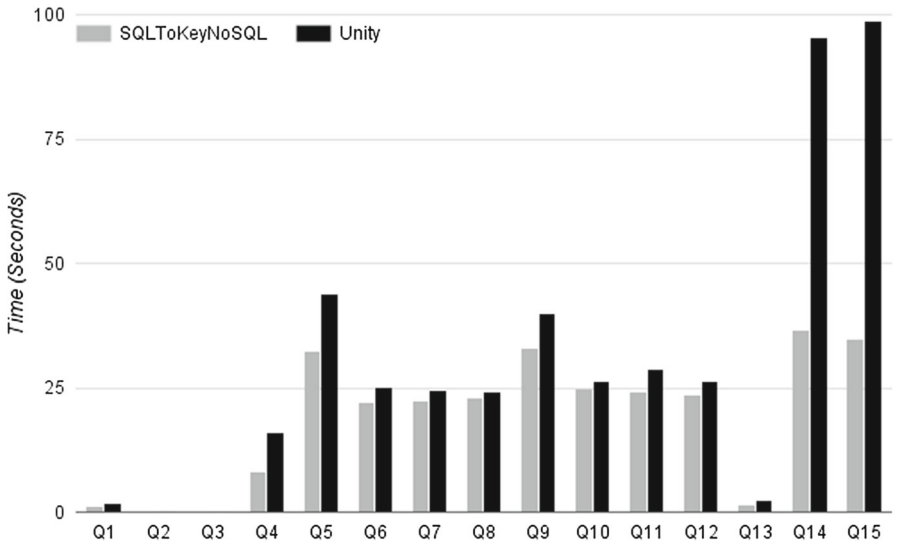

Fig. 8 Processing time comparison between SQLtoKeyNoSQL and *Unity*

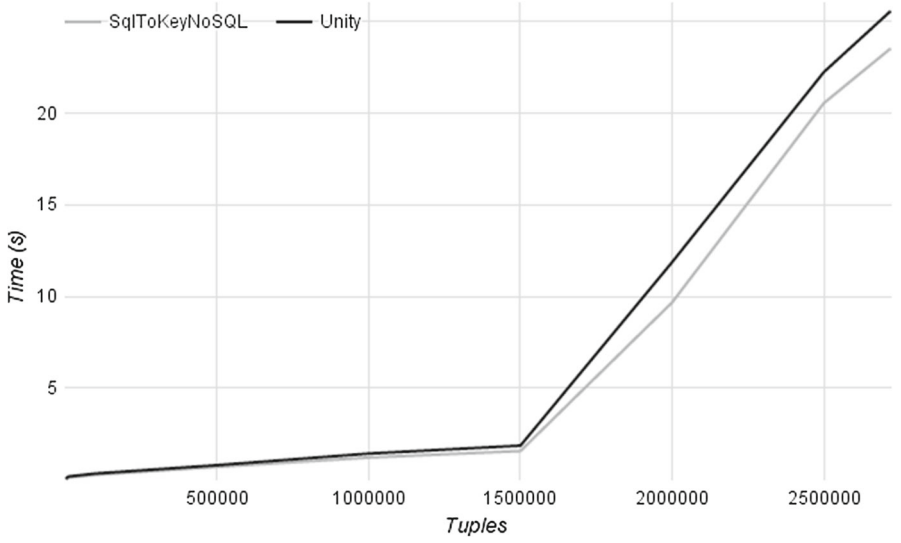

Fig. 9 Scalability comparison between SQLtoKeyNoSQL and *Unity*

when possible. Instead, *Unity* implements a more complex join processing strategy [15].

Figure 9 shows the results for the second part of the experiments, i.e., the scalability evaluation. The line graph presents, in the *x*-axis, the number of rows and, in the *y*-axis, the respective processing time in seconds. Both approaches obtained about the same performance up to 1,500,000 rows. After that, SQLtoKeyNoSQL presents a (lightweight) superior performance. That improvement is probably due to the number

of data requests to MongoDB since SQLtoKeyNoSQL can retrieve blocks of data for each request by executing the *GetN* method. We also notice that both approaches increase significantly the processing time after the mark of 1,500,000 tuples. This is because MongoDB is running in a single node instance, and it is not able to scale.

5.4 SQLtoKeyNoSQL vs SimpleSQL

We accomplished the same set of experiments for *SimpleSQL* by accessing *Amazon SimpleDB* through Amazon AWS cloud. As stated before, for this set of experiments we reduce the cardinality of each table to 200,000 rows.

Figure 10 shows the results for the first part of the experimental evaluation. Notice that SQLtoKeyNoSQL has a very significant lower processing time for all queries. The reason for that is probably due to the two main strategies followed by the approaches. First, *SimpleSQL* executes at least two requests to *SimpleDB* to retrieve the rows: one to get metadata information about the table and another one to get the mapped rows of the table. Instead, SQLtoKeyNoSQL keeps metadata information in main memory and accesses SimpleDB only to get the rows. Besides, *SimpleSQL* has to query each item stored in SimpleDB by filtering a special metadata attribute (*SimpleSQL_TableName*) that maintains the table name. Different from it, SQLtoKeyNoSQL accesses all metadata information in its main memory dictionary.

Figure 11 shows the results for the *SimpleSQL* scalability experiments. The line graph presents, in the *x*-axis, the number of rows for the table and, in the *y*-axis, the respective processing time in minutes. Again, SQLtoKeyNoSQL outperforms *SimpleSQL* in all five experiments. The difference increases drastically with the increase of the number of retrieved rows.

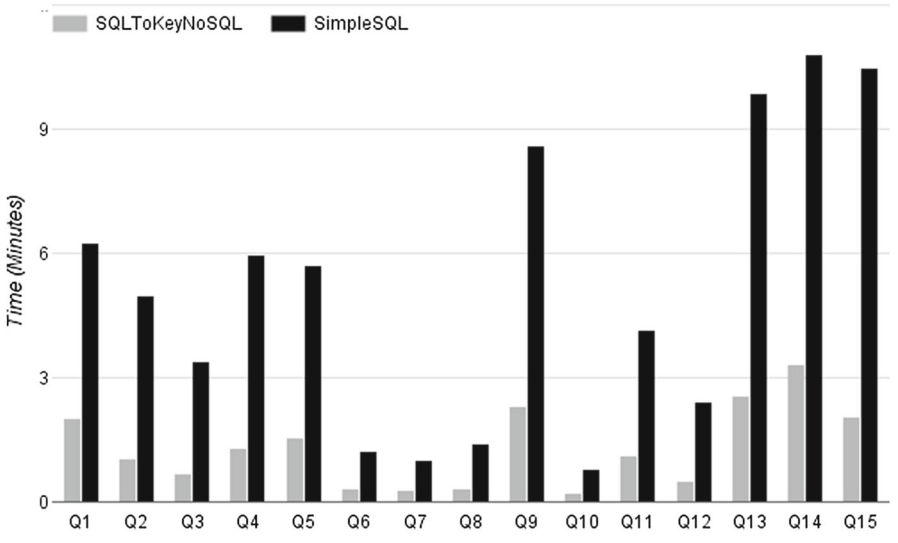

Fig. 10 Processing times comparison between SQLtoKeyNoSQL and *SimpleSQL*

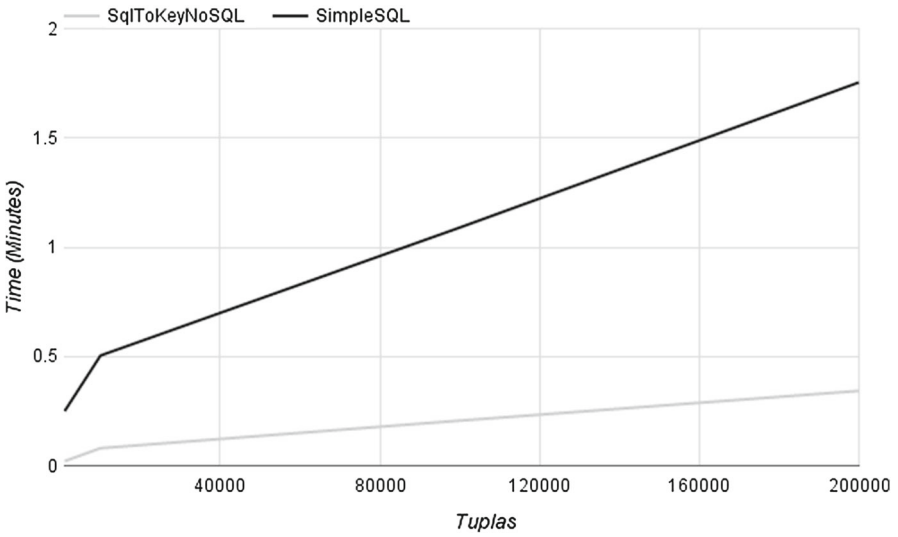

Fig. 11 Scalability comparison between SQLtoKeyNoSQL and *SimpleSQL*

In short, the experiments have shown that SQLtoKeyNoSQL is a promising approach for performing SQL queries over NoSQL data stores. The canonical model and the dictionary help to manage and retrieve the data in an efficient way. Moreover, we offer the flexibility to store data in any key-based NoSQL data model, allowing users to choose the best one for their needs.

6 Conclusion

This paper presents *SQLtoKeyNoSQL*, an approach that provides an SQL-based access interface for data maintained in NoSQL DB. The idea behind this proposal is to offer a solution for relational-based applications that intend to migrate their data to NoSQL DB and do not want to incur in high costs with the learning of new NoSQL DB access methods as well as the changing of their SQL interface to these new access methods. Moreover, it allows the movement of relational data to one or more key-based NoSQL data models.

Our approach is materialized as a layer allowing users to execute a subset of SQL DDL and DML instructions over any key-based access NoSQL DB (document-oriented, column-oriented and key-value NoSQL DB). It supports a canonical data model that works as an intermediate schema between the relational data model and the key-based access NoSQL data models, providing transparent access. Besides, SQLtoKeyNoSQL allows the user to choose the NoSQL target DB where each table is going to be stored.

We evaluate SQLtoKeyNoSQL, regarding processing time, against two baselines available in the literature (*SimpleSQL* and *Unity*), as detailed in Sect. 5. The results of the experiments were considered satisfactory. Our approach outperforms *SimpleSQL*

for all proposed queries, being three times more effective in terms of join processing. The experiments also showed that our approach reached less processing times than *Unity* in terms of scalability tests, with 95% of confidence. Based on the results of the experiments, we also conclude that the new features added to SQLtoKeyNoSQL make it a more robust and scalable approach. SQLtoKeyNoSQL can be a very useful tool for users that intend to migrate their application from the relational data model to a NoSQL key-based-data model with a lower learning curve.

This paper contributes as a basis for a comprehensive and efficient solution for relational-based access to any key-based NoSQL DB. Even so, several future work can be issued as follows: (i) support for index management; (ii) a possible extension of the canonical model to support the graph data model; and (iii) enhancing the SQL subset by adding support to aggregation and subqueries.

References

1. Abadi DJ (2009) Data management in the cloud: limitations and opportunities. IEEE Data Eng Bull 32(1):3–12
2. Apache (2017) White paper: apache phoenix. http://phoenix.apache.org/. Accessed 24 Aug 2018
3. Arnaut DE, Schroeder R, Hara CS (2011) Phoenix: a relational storage component for the cloud. In: 2013 IEEE SICCC 0
4. Atzeni P, Bugiotti F, Rossi L (2012) Sos (save our systems): a uniform programming interface for non-relational systems. In: Proceedings of the 15th international conference on extending database technology. ACM, New York
5. Banerjee S, Goto T, Debnath NC, Sarkar A (2017) Ontology driven query language for nosql databases. In: 2017 IEEE 15th international conference on industrial informatics (INDIN), pp 951–956
6. Bisbal J, Lawless D, Wu B, Grimson J (1999) Legacy information systems: issues and directions. IEEE Softw 16(5):103–111
7. Cattell R (2011) Scalable SQL and NoSQL data stores. SIGMOD Rec 39(4):12–27
8. Chung WC, Lin HP, Chen SC, Jiang MF, Chung YC (2014) Jackhare: a framework for SQL to NoSQL translation using mapreduce. Autom Softw Eng 21(4):489–508
9. dos Santos Ferreira G, Calil A, dos Santos Mello R (2013) On providing DDL support for a relational layer over a document NoSQL database. In: IIWAS. ACM, New York
10. Egger D (2009) SQL in the cloud. Ph.D. thesis, Master Thesis ETH Zurich
11. Fielding RT (2000) Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine
12. Hamouda S, Zainol Z (2017) Document-oriented data schema for relational database migration to NoSQL. In: 2017 International conference on big data innovations and applications (innovate-data), pp 43–50
13. Kim HJ, Ko EJ, Jeon YH, Lee KH (2018a) Migration from RDBMS to column-oriented NoSQL: lessons learned and open problems. In: Lee W, Choi W, Jung S, Song M (eds) Proceedings of the 7th international conference on emerging databases. Springer Singapore, pp 25–33
14. Kim HJ, Ko EJ, Jeon YH, Lee KH (2018b) Techniques and guidelines for effective migration from RDBMS to NoSQL. J Supercomput. https://doi.org/10.1007/s11227-018-2361-2
15. Lawrence R (2014) Integration and virtualization of relational SQL and NoSQL systems including MySQL and MongoDB. In: CSCI, vol 1
16. Liu ZH, Hammerschmidt BC, McMahon D (2014) JSON data management: supporting schema-less development in RDBMS. In: ICMD, SIGMOD
17. Mishra P, Eich MH (1992) Join processing in relational databases. ACM CSUR 24(1):63–113
18. Papakonstantinou Y, Garcia-Molina H, Widom J (1995) Object exchange across heterogeneous information sources. In: 11th CDE. IEEE
19. Rith J, Lehmayr PS, Meyer-Wegener K (2014) Speaking in tongues: SQL access to NoSQL systems. In: 29th ACM SAC, New York

20. Rocha L, Vale F, Cirilo E, Barbosa D, Mouro F (2015) A framework for migrating relational datasets to NoSQL1. Procedia Comput Sci 51(C):2593–2602
21. Sadalage PJ, Fowler M (2012) NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Pearson Education, London
22. Schreiner GA, Duarte D, dos Santos Mello R (2015) SQLtoKeyNoSQL: a layer for relational to key-based NoSQL database mapping. In: iiWAS, ACM, New York
23. Vathy-Fogarassy G, Hugyk T (2017) Uniform data access platform for SQL and NoSQL database systems. Inf Syst 69(C):93–105
24. Vilaça R, Cruz F, Pereira J, Oliveira R (2013) An effective scalable SQL engine for NoSQL databases. In: Dowling J, Taïani F (eds) 13th IFIP, DAIS, Springer, Berlin
25. Xu J, Shi M, Chen C, Zhang Z, Fu J, Liu CH (2016) ZQL: a unified middleware bridging both relational and NoSQL databases. In: 2016 IEEE 14th ICD, ASC, 14th ICPIC, 2nd CyberSciTech, pp 730–737