Towards a Taxonomy of Schema Changes for NoSQL Databases: The Orion Language

Alberto Hernández Chillón(✉), Diego Sevilla Ruiz, and Jesús García Molina

Faculty of Computer Science, University of Murcia, Murcia, Spain
{alberto.hernandez1,dsevilla,jmolina}@um.es

Abstract. The emergence of NoSQL databases and polyglot persistence demands to address classical research topics in the context of new data models and database systems. Schema evolution is a crucial aspect in database management to which limited attention has been paid for NoSQL systems. The definition of a taxonomy of changes is a central issue in the design of any schema evolution approach. Proposed taxonomies of changes for NoSQL databases have considered simple data models, which significantly reduce the set of considered schema change operations. In this paper, we present a unified logical data model that includes aggregation and reference relationships, and takes into account the structural variations that can occur in schemaless NoSQL stores. For this data model, we introduce a new taxonomy of changes with operations not considered in the existing proposed taxonomies for NoSQL. A schema definition language will be used to create schemas that conform to the generic data model, and a database-independent language, created to implement this taxonomy of changes, will be shown. We will show how this language can be used to automatically generate evolution scripts for a set of NoSQL stores, and validated on a case study for a real dataset.

Keywords: NoSQL databases · Schema evolution · Taxonomy of changes · Schema change operations · Domain specific language

1 Introduction

NoSQL (Not only SQL) systems and polyglot persistence emerged to tackle the limitations of relational databases to satisfy requirements of modern, data-intensive applications (e.g., social media or IoT). The predominance of relational systems will probably continue, but they will coexist with NoSQL systems in heterogeneous database architectures. In this scenario, new database tools are needed to offer for NoSQL systems the functionality already available for relational systems. Moreover, these tools should support several data models, as four

This work has been funded by the Spanish Ministry of Science, Innovation and Universities (project grant TIN2017-86853-P).

kinds of NoSQL systems are widely used: columnar, document, key-value, and graphs, with interest in polyglot persistence continuously growing.

Among those aforementioned tools are schema evolution tools. Schema evolution is the ability to make changes on a database schema and adapting the existing data to the new schema. This topic has been extensively studied in relational [1,6] and object-oriented databases [10]. So far, the attention paid to NoSQL schema evolution has been limited, and building robust solutions is still an open research challenge.

Most NoSQL systems are "schema-on-read", that is, the declaration of schemas is not required prior to store data. Therefore, schemas are implicit in data and code. In addition, no standard or specification of NoSQL data model exists. This means that schema evolution approaches for the same type of NoSQL store could be based on different definitions of data models. The proposals published are limited in three aspects: (i) The data models considered do not cover all the possible elements to be subject to evolution; (ii) This leads to taxonomies of changes that do not include some schema change operations potentially useful; (iii) Except for [3,7], the proposals do not embrace the NoSQL paradigm heterogeneity. Two features not considered in these proposals are structural variations and the existence of aggregation and reference relationships. Taking into account these modeling concepts, new operations can be included into the taxonomy of changes.

In this paper, we present a NoSQL schema evolution proposal based on a unified data model that includes aggregation and reference relationships, and structural variations [2]. This paper contributes in the following: (i) A taxonomy of changes that includes valuable operations not included in previous proposals; (ii) A *domain-specific language* (DSL), *Orion*, was developed to implement the set of *schema change operations* (SCOs) defined in the taxonomy; (iii) Our approach is based on a unified data model with more expressive power than other proposals [3,7]; and (iv) A non-trivial refactoring case study for a real dataset used to improve query performance. Given an Orion script, the operations that adapt databases to the schema changes are automatically generated. Also, the inferred or declared schema is automatically updated from Orion scripts.

This paper is organized as follows. First, we present the unified data model for the four NoSQL paradigms considered. Then, we will define the taxonomy of changes and present the Orion language. Next, the validation process will be described, finally ending with the discussion of the related work and drawing some conclusions and future work.

2 Defining Schemas for a Unified Data Model

U-Schema is a unified metamodel that integrates the relational model and data models for the four most common NoSQL paradigms [2]. The taxonomy presented here is based on U-Schema. We will describe the elements of *U-Schema* through the *Athena* language. Figure 1 shows an Athena schema for a gamification application. We will use this schema as a running example.

```
Schema Gamification_athena:1

FSet timeData
{
  createdAt:      Timestamp,
  updatedAt:      Timestamp
}

Root entity Stage
{
  +id:            Identifier,
  description:    String,
  name:           String
} + timeData

Root entity Minigame
{
  +id:            Identifier,
  isActive:       Boolean,
  name:           String,
  points:         Integer (0..99),
  stageIds:Ref<Stage as Identifier>+
} + timeData
```

```
Root entity User {
  Common {
    +id:         Identifier,
    email:       String /^.+@.+\.com$/,
    personalInfo: Aggr<PersonalInfo>&
  }
  Variation 1
  Variation 2 {
    games:       Aggr<MinigameSummary>+,
    points:      Integer (0..9999)
  }
}
Entity PersonalInfo {
  name:          String /^[A-Z][a-z]*$/,
  street:        String,
  city:          String,
  ?postcode:     Integer
}
Entity MinigameSummary {
  gameId:        Ref<Minigame>&,
  ?completedAt:Timestamp,
  ?points:       Integer
}
```

Fig. 1. The `Gamification` schema defined using Athena.

An Athena schema is formed by a set of *schemas types* that can be *entity types* to represent domain entities or *relationship types* to represent relationships between nodes in graph stores. In the Gamification example, there are five entity types: `User`, `PersonalInfo`, `Stage`, `Minigame`, and `MinigameSummary`. `User`, `Stage`, and `Minigame` are root entity types, that is, their objects are not embedded in any other object, while `PersonalInfo` and `MinigameSummary` are non-root as their instances are embedded into `User` objects. Each entity type can have a set of structural variations that include a set of features or properties. The features that are *common* to all the variations are separately declared. `User` has three common features: `id`, `email`, and `personalInfo`. Each variation may add optional features: the second `User` variation has `games` and `points`.

A feature declaration specifies its name and type. There are three kinds of features: *attributes*, *aggregates*, and *references*. For example, the `User.personalInfo` feature specifies that `PersonalInfo` objects are embedded in `User` objects, and the `Minigame.stageIds` feature specifies that `Minigame` objects reference `Stage` objects by holding values of their key attributes. A cardinality needs to be specified for references and aggregations, such as *one to one* or *one to many*. The features may have modifiers as *key* ("+") or *optional* ("?").

Attributes have a scalar type as `Number` or `String`, e.g., `email` and `points`. The `Identifier` scalar type is used to declare unique identifiers, e.g., `id` for users. Certain scalar types allow to add restrictions to their possible values, such as numeric ranges, value enumeration or using regular expressions. The most frequently used collection types are offered: such as `Maps`, `Lists` or `Tuples`.

Athena provides mechanism to favor reuse: schema import and inheritance. Also, composition of schemas is possible through a set of operators. In addition to types, the notion of *set of features* is used to group any set of features, that can be later combined to create more complex schemas.

3 A Taxonomy of Changes for NoSQL Databases

A taxonomy determines the possible changes to be applied on elements of the database schema that conforms to a particular data model. Different categories are established according to the kind of schema element affected by the changes. A taxonomy also specifies the change semantics. Here, we present a taxonomy for U-Schema, which includes novel operations related to the abstractions proper to U-Schema, such as aggregates, references, and structural variation. Our taxonomy also incorporates the operations of previous proposals for NoSQL evolution.

As shown in Table 1, our taxonomy has categories for the following U-Schema elements: *schema types*, *features*, *attributes*, *aggregates* and *references*. *Schema type* category groups change operations on *entity* and *relationship types*. The *Feature* category groups the operations with the same semantics for the three kinds of features. Three operations for variations have been included in the *schema type* category: create a union schema, adapt a variation to other, and delete a variation. The former is useful to squash variations into a single variation for a schema type, and the other two with several uses, such as removing outliers [8].

Next, the terminology used to define the semantics in the taxonomy is introduced. Let $S = E \cup R$ be the set of schema types formed by the union of the set of entity types $E = \{E_i\}, i = 1 \ldots n$, (and the set of relationship types $R = \{R_i\}, i = 1 \ldots m$, if the schema corresponds to a graph database.) Each schema type t includes a set of structural variations $V^t = \{v_1^t, v_2^t, \ldots, v_n^t\}$, and $v_i^t.features$ denotes the set of features of a variation v_i^t. Then, the set of features of a schema type t is $F^t = \bigcup_{i=1}^{n} v_n^t \cup C^t$, where C^t denotes the set of common features of t. The set F^t will include attributes, aggregates, and references. We will use *dot notation* to refer to parts of a schema element, e.g., given an entity type e, $e.name$ and $e.features$ refer to the name and set of features (F^e), respectively, of the entity type. Finally, the symbol "$\leftarrow$" will be used to express the change of state of a schema element by means of an assignment statement.

For the *Schema type* category, in addition to the atomic operations: add, delete, and rename, three complex SCOs are added to create new schemas types: The *extract*, *split* and *merge* operations. Also three operations to manipulate variations are introduced: The *delvar*, *adapt* and *union* operations. Due to space restrictions, the semantics of these operations is only formally given in Table 1.

The *Feature* category includes complex SCOs to copy a feature from a schema type to another one, either maintaining (*copy*) or not (*move*) the feature copied in the original schema type. Also, it includes SCOs to move a feature from/to an aggregate: *nest* and *unnest*. The *Attribute* category includes operations to *add* a new attribute, change its type (*cast*), and add/remove an attribute to/from a key: *promote* and *demote*. Finally, the *Reference* and *Aggregate* categories include the *morph* operations to transform aggregates into references and vice versa, *card* to change the cardinality, as well as *add* and *cast* commented for attributes.

The operations in Table 1 have attached information regarding the gaining or loss of information they cause. C^+ is used for an *additive change*, C^- for a *subtractive change*, $C^{+,-}$ when there is a loss and gain of information (e.g., *move*

Table 1. Schema change operations of the taxonomy.

Schema Type Operations		
Add	(C^{+})	Let t be a new schema type, $S \leftarrow S \cup \{t\}$.
Delete	(C^{-})	Given a schema type t, $S \leftarrow S \setminus \{t\}$.
Rename	$(C^{=})$	Given a schema type t and a string value n, $t.name \leftarrow n$.
Extract	$(C^{+,=})$	Given a schema type t, a set of features f of t, $f \subset F^{t}$, and a string value n, then a new type t_1 is created such that $t_1.name \leftarrow n \wedge t_1.features \leftarrow f$ and $S \leftarrow S \cup \{t_1\}$.
Split	$(C^{=})$	Given a schema type t and two sets of features $f_1 \subset t.features$ and $f_2 \subset t.features$, and two string values n_1 and n_2, then two new types t_1 and t_2 are created such that $t_1.name \leftarrow n_1 \wedge t_1.features \leftarrow f_1$ and $t_2.name \leftarrow n_2 \wedge t_2.features \leftarrow f_2$, and $S \leftarrow S \setminus \{t\}$ and $S \leftarrow S \cup \{t_1, t_2\}$.
Merge	$(C^{=})$	Given two schema types t_1 and t_2 and a a string value n, a new schema type t is created such that $t.name \leftarrow n \wedge t.features \leftarrow t_1.features \cup t_2.features$ and $S \leftarrow S \setminus \{t_1, t_2\}$ and $S \leftarrow S \cup \{t\}$.
DelVar	(C^{-})	Given a variation v^{t} of a schema type t, then $t.variations \leftarrow t.variations \setminus \{v^{t}\}$.
Adapt	$(C^{+/-})$	Given two variations v_1^{t} and v_2^{t} of a schema type t, then $v_1^{t}.features \leftarrow v_2^{t}.features$, and $t.variations \leftarrow t.variations \setminus \{v_1^{t}\}$.
Union	(C^{+})	Given a schema type t that has m variations, $t.features \leftarrow t.features \cup_{i=1}^{m} v_i^{t}$.
Feature Operations		
Delete	(C^{-})	Given a schema type t and a feature $f \in t.features$, then $t.features \leftarrow t.features \setminus \{f\}$.
Rename	$(C^{=})$	Given a schema type t, a feature $f \in t.features$, and a string value n, then $f.name \leftarrow n$.
Copy	(C^{+})	Given two schema types t_1 and t_2 and a feature $f \in t_1.features$, then $t_2.features \leftarrow t_2.features \cup \{f\}$.
Move	$(C^{+,-})$	Given two schema types t_1 and t_2 and a feature $f \in t_1.features$, then $t_2.features \leftarrow t_2.features \cup \{f\} \wedge t_1.features \leftarrow t_1.features \setminus \{f\}$.
Nest	$(C^{+,-})$	Given an entity type e_1, a feature $f \in e_1.features$, and an aggregate $ag \in e_1.aggregates \wedge ag.type = e_2$, then $e_2.features \cup \{f\} \wedge e_1.features \setminus \{f\}$.
Unnest	$(C^{-,+})$	Given an entity type e_1, an aggregate $ag \in e_1.aggregates \wedge ag.type = e_2$, and a feature $f \in e_2.features$, then $e_1.features \cup \{f\} \wedge e_2.features \setminus \{f\}$.
Attribute Operations		
Add	(C^{+})	Given a schema type t and an attribute at (name and type), then $t.attributes \leftarrow t.attributes \cup \{at\}$.
Cast	$(C^{+/-})$	Given a schema type t, an attribute $at \in t.attributes$, and a scalar type st, then $at.type \leftarrow st$.
Promote	$(C^{=})$	Given an entity type t and an attribute at, then $at.key \leftarrow True$.
Demote	$(C^{=})$	Given an entity type t and an attribute at, then $at.key \leftarrow False$.
Reference Operations		
Add	(C^{+})	Given a schema type t and a reference rf (name and entity type), then $t.references \leftarrow t.references \cup \{rf\}$.
Cast	$(C^{+/-})$	Given a schema type t, a reference $rf \in t.references$, and a scalar type st, then $rf.type \leftarrow st$.
Card	$(C^{+/-})$	Given a schema type t, a reference $rf \in t.references$, and two numbers $l, u \in [-1, 0, 1] \wedge l \leq u$, then $rf.lowerBound \leftarrow l \wedge rf.upperBound \leftarrow u$.
Morph	$(C^{=})$	Given a schema type t and a reference $rf \in t.references$, then $t.aggregations \leftarrow t.aggregations \cup \{ag\}$, where $ag.name \leftarrow rf.name \wedge ag.type \leftarrow rf.type$ and $t.references \leftarrow t.references \setminus \{rf\}$.
Aggregate Operations		
Add	(C^{+})	Given an entity type e and an aggregation ag (name an entity type), then $e.aggregations \leftarrow t.aggregations \cup \{ag\}$.
Card	$(C^{+/-})$	Given an entity type e, and an aggregation $ag \in e.aggregations$ and two numbers $l, u \in [-1, 0, 1] \wedge l \leq u$, then $ag.lower \leftarrow l \wedge ag.upper \leftarrow u$.
Morph	$(C^{=})$	Given an entity type e and an aggregation $ag \in e.aggregations$, then $e.references \leftarrow e.references \cup \{rf\}$, where $rf.name \leftarrow ag.name \wedge rf.type \leftarrow ag.type$ and $e.aggregations \leftarrow e.aggregations \setminus \{rf\}$.

feature), $C^{=}$ means no change in information, and $C^{+/-}$ adds or subtracts, depending on the operation parameters (e.g., *casting* a feature to boolean).

4 Implementing the Taxonomy: The Orion Language

Once the taxonomy was defined, we created the Orion language to allow developers to write and execute their schema change operations. Orion keeps the system-independence feature of the taxonomy by providing the same abstract and concrete syntax for any database system, although the semantics of each operation must be implemented in a different way depending on each system.

Figure 2 shows a Orion script to refactor the running example schema. `Using` specifies the schema to apply changes, which allows checking the feasibility of all the change operations. This checking requires sequentially executing the operations, and updating the schema before launching the execution of the following operation.

```
 1 Gamification_refactoring operations
 2 Using Gamification_athena:1
 3
 4 // Minigame operations
 5 CAST ATTR *::points TO Double
 6 DELETE Minigame::isActive
 7
 8 // User operations
 9 ADAPT ENTITY User::v1 TO v2
10 NEST User::email TO personalInfo
11 MORPH AGGR User::personalInfo
12     TO privateData
13 RENAME ENTITY User TO Employee
14
15 // PersonalInfoData operations
16 CAST ATTR PersonalInfo::postcode
17     TO String
18 NEST PersonalInfo::city,postcode,
19     street TO Address
20
21 // MinigameSummary operations
22 RENAME MinigameSummary::completedAt
23     TO isCompleted
24 CAST ATTR MinigameSummary::isCompleted
25   TO Boolean
26
27 //Adding a new entity type
28 ADD ENTITY Company
29 {
30   +id: Identifier,
31   code: String,
32   name: String,
33   numOfEmployees: Number
34 }
35
36 //Adding new features
37 PROMOTE ATTR Company::code
38 ADD REF Company::staff+ TO Employee
39 ADD AGGR Company::media:
40 {
41   twitterProf: String,
42   fbProf: String,
43   webUrl: String,
44   ytProf: String
45 }&
```

Fig. 2. Refactoring of the `Gamification` schema using Orion.

Figure 2 shows changes on entity types of the *Gamification* schema: casting on attributes (lines 5, 16, and 24), deleting attributes (line 6), nesting attributes to an aggregate (lines 10 and 18), morphing an aggregate to a reference (11), renaming entity types (line 13) and attributes (22), and adapting a variation (line 9). Entity types and features are also created (lines 28–45).

Implementing a taxonomy of schema changes entails tackling the update of both the schema and the data. The Orion engine generates the updated schema as well as the code scripts to produce the changes required in the database. So far, MongoDB and Cassandra are supported. Orion can also provide an estimation of the time required to perform the change operations.

Table 2 gives insights on the implementation of some operations for each database, as well as a time estimation. The ◷ symbol is used for constant-time operations; ◔ when traversing all the instances of an entity type is required; ◕ is used when traversing one or two entity types and creating a new entity type, and ● is used if all the instances of an entity type have to be serialized and then imported back into the database. ●● is used for operations with very high cost.

Not all operations can be applied to all database systems. For example variation-related operations cannot be applied in Cassandra, since it requires the definition of a schema prior to store data.

Table 2. Excerpt of operation costs in Orion for each database system considered.

	MongoDB operations	Cost	Cassandra operations	Cost
Add schema type	createCollection(),$addFields	◷	CREATE Tab	◷
Split schema type	2x($project,$out),drop()	◕	(4xCOPY,2xCREATE,DROP) Tab	●●
Rename feature	$rename	◔	2xCOPY Tab,DROP Col,ADD Col	●
Copy feature	$set	◔	2xCOPY Tab,ADD Col	●
Add attribute	$addFields	◔	ADD Col	◷
Cast attribute	$set,$convert	◔	(2xCOPY,DROP,CREATE) Tab	●
Card reference	$set	◔	(2xCOPY,DROP,CREATE) Tab	●
Morph reference	$lookup,$out,$unset,drop()	◕	CREATE Type,ADD Col,DROP Tab*	—
Add aggregate	$addFields	◔	CREATE Type,ADD Col	◷
Card aggregate	$set	◔	(2xCOPY,DROP,CREATE) Tab	●

Although the main purpose of the Orion language is to support schema changes in a platform-independent way, it may have other uses: (i) An Orion script can bootstrap a schema by itself if no initial schema is given; (ii) Differences between Athena schemas may be expressed as Orion specifications; and (iii) Orion specifications may be obtained from existing tools such as PRISM/PRISM++ [1].

5 The Validation Process

A refactoring was applied on the StackOverflow dataset[1] in order to validate Orion. We injected this dataset into MongoDB, and its schema was inferred by applying the strategy from [2]. Figure 3 shows an excerpt of four of the seven entity types discovered, visualized with the notation introduced in [5].

Analyzing the schema, we realized that this dataset did not take advantage of constructs typical for document databases such as aggregations (nested documents), and that some features could be casted to specific MongoDB types. By slightly changing the schema we might improve query performance in MongoDB. We designed a scenario to measure the query performance by introducing changes in the schema. We aggregated `Badges` into `Users`, `PostLinks` into `Posts`, and created two new aggregates for the metadata information for `Posts` and `Users`.

[1] https://archive.org/details/stackexchange.

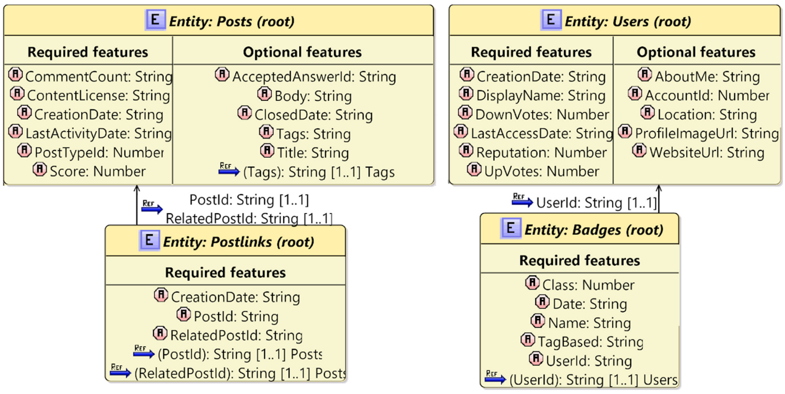

Fig. 3. Excerpt of the `StackOverflow` inferred schema.

The Orion script that specifies the StackOverflow schema changes is shown in Fig. 4. The first operations (lines 4–6) are `CAST`s of attributes to types more suitable for MongoDB. Then some `NEST` operations on `Posts` and `Users` are applied to create aggregated objects (`PostMetadata` and `UserMetadata`, lines 8–9). We also embedded `Postlinks` into `Posts` (lines 13–14) and `Badges` into `Users`. These two last operations are not trivial, as `Postlinks` and `Badges` are the entity types referencing `Posts` and `Users`, and therefore it is needed to first copy certain features and then morph them (lines 16–18).

```
 1  StackOverflow_ops operations
 2  Using stackoverflow
 3
 4  CAST ATTR *::CreationDate TO Timestamp
 5  CAST ATTR *::LastAccessDate TO Timestamp
 6  CAST ATTR Posts::LastActivityDate TO Timestamp
 7
 8  NEST Posts::CreationDate,LastActivityDate TO PostMetadata
 9  NEST Users::CreationDate,LastAccessDate,DownVotes,UpVotes TO UserMetadata
10
11  CARD REF Posts::Tags TO *
12
13  COPY Postlinks::_id TO Posts::postlinkId WHERE Postlinks.PostId = Posts._id
14  MORPH REF Posts::postlinkId ( rmId rmEntity ) TO Postlinks
15
16  COPY Badges::_id TO Users::badgeId WHERE Badges.UserId = Users._id
17  MORPH REF Users::badgeId ( rmId rmEntity ) TO Badges
18  RENAME Users::badgeId TO badges
```

Fig. 4. Orion operations to be applied to the `StackOverflow` schema and data.

Given the Athena schema shown in Fig. 3 and the Orion script of Fig. 4, the Orion engine generates the updated schema and the MongoDB API code script to execute the changes on the data. Queries can then be adapted, and performance can be measured and compared. MongoDB code is implemented using aggregation pipelines and bulk writes to improve the performance.

6 Related Work

A great research effort has been devoted to the database schema evolution problem for relational databases. Some of the most significant contributions have been the DB-Main and PRISM/PRISM++ tools. DB-Main was a long-term project aimed at tackling the problems related to database evolution on relational systems and other paradigms such as hierarchical or object-oriented [4,6]. The DB-Main approach was based on two main elements: The GER unified metamodel to achieve platform-independence, and a transformational approach to implement operations such as reverse and forward engineering, and schema mappings; More recently, Carlo Curino et al. [1] developed the PRISM/PRISM++ tool aimed to automate migration tasks and rewriting legacy queries. PRISM/PRISM++ provides an evolution language based on operators able to modify schemas and integrity constraints. Although much more mature and evolved than our work, neither of these two approaches address the NoSQL database evolution.

A proposal of schema evolution for NoSQL document stores was presented by Stefanie Scherzinger et al. in [9]. The work defines a 5-operation taxonomy for a very simple data model: schemas are a set of entities that have properties (attributes and embedded objects), but relationships or variations are not considered. The operations are add/remove/rename properties, and copy or move a set of properties from an entity type to another. In our proposal, the generic data model is more complex, which results in a richer change taxonomy.

Vavrek et al. explored schema evolution for multi-model database systems [7,11]. They suppose a layered architecture in which a database engine interacts with individual engines for each data model, instead of defining a unified metamodel. A taxonomy of 10 operations is defined: 5 for entity types (*kinds*) and 5 for properties. These latter correspond to those defined in [9], and the first 5 are *add*, *drop*, *rename*, *split*, and *merge*, with the same meaning as in our taxonomy. We propose a complex engine architecture for a unified model, instead of a layered one, as most databases share a common set of features which can be included in a single core model, which will act as a pivot model to support schema mappings. Our goal is to provide native and broader support for the most popular databases. Also, our taxonomy includes new operations such as transforming aggregations into references and vice versa, joining all the variations, removing a variation, or add/remove/rename references and aggregates.

In [3], a taxonomy is proposed as part of an approach to rewrite queries for polystore evolution. The taxonomy includes six operations applicable to *entity types*, four to *attributes* and four to *relations*. A generic language, TyphonML, is used to define relational and NoSQL schemas. The language also includes physical mapping and schema evolution operations. Our richer data model allowed us to define operations that separately affect aggregates and references, and variations and relationship types are considered, among other differences.

7 Conclusions and Future Work

In this paper, we have taken advantage of the U-Schema unified data model to present a schema changes taxonomy for NoSQL systems, which includes more complex operations than previous proposals. Evolution scripts for this taxonomy are expressed by means of the Orion language.[2] Currently, Orion works for two popular NoSQL stores: MongoDB (document and schemaless) and Cassandra (columnar that requires a schema declaration). Orion has been validated by applying a refactoring on the StackOverflow dataset.

Our future work includes (i) The implementation of generators for different database paradigms, such as Neo4j for graphs and Redis for key-value; (ii) Investigating optimizations of the generated code to evolve databases; (iii) Query analysis and query rewriting; and (iv) Integrating Orion into a tool for agile migration.

References

1. Curino, C., Moon, H.J., Deutsch, A., Zaniolo, C.: Automating the database schema evolution process. VLDB J. **22**, 73–98 (2013)
2. Fernández Candel, C., Sevilla Ruiz, D., García Molina, J.: A Unified Metamodel for NoSQL and Relational Databases. CoRR abs/2105.06494 (2021)
3. Fink, J., Gobert, M., Cleve, A.: Adapting queries to database schema changes in hybrid polystores. In: IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 127–131 (2020)
4. Hainaut, J.-L., Englebert, V., Henrard, J., Hick, J.-M., Roland, D.: Database evolution: the DB-MAIN approach. In: Loucopoulos, P. (ed.) ER 1994. LNCS, vol. 881, pp. 112–131. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58786-1_76
5. Hernández, A., Feliciano, S., Sevilla Ruiz, D., García Molina, J.: Exploring the visualization of schemas for aggregate-oriented NoSQL databases. In: 36th International Conference on Conceptual Modeling (ER), ER Forum 2017, pp. 72–85 (2017)
6. Hick, J.-M., Hainaut, J.-L.: Strategy for database application evolution: the DB-MAIN approach. In: Song, I.-Y., Liddle, S.W., Ling, T.-W., Scheuermann, P. (eds.) ER 2003. LNCS, vol. 2813, pp. 291–306. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39648-2_24
7. Holubová, I., Klettke, M., Störl, U.: Evolution management of multi-model data. In: Gadepally, V., et al. (eds.) DMAH/Poly 2019. LNCS, vol. 11721, pp. 139–153. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33752-0_10
8. Klettke, M., Störl, U., Scherzinger, S.: Schema extraction and structural outlier detection for JSON-based NoSQL data stores. In: BTW Conference, pp. 425–444 (2015)
9. Klettke, M., Störl, U., Shenavai, M., Scherzinger, S.: NoSQL schema evolution and big data migration at scale. In: IEEE International Conference on Big Data (2016)
10. Li, X.: A survey of schema evolution in object-oriented databases. In: TOOLS: 31st International Conference on Technology of OO Languages and Systems, pp. 362–371 (1999)
11. Vavrek, M., Holubová, I., Scherzinger, S.: MM-evolver: a multi-model evolution management tool. In: EDBT (2019)

[2] The implementation of Orion and its specification may be found at https://catedrasaes-umu.github.io/NoSQLDataEngineering/tools.html.