A Generic Schema Evolution Approach for NoSQL and Relational Databases

Alberto Hernández Chillón, Meike Klettke, Diego Sevilla Ruiz, and Jesús García Molina

***Abstract*—In the same way as with relational systems, schema evolution is a crucial aspect of NoSQL systems. But providing approaches and tools to support NoSQL schema evolution is more challenging than for relational databases. Not only are most NoSQL systems schemaless, but different data models exist without a standard specification for them. Moreover, recent proposals fail to address some key aspects related to the kinds of relationships between entities, the definition of relationship types, and the support of structural variation. In this article, we present a generic schema evolution approach able to support the most popular NoSQL data models (columnar, document, key-value, and graph) and the relational model. The proposal is based on the Orion language that implements a schema change operation taxonomy defined for the U-Schema unified data model that integrates NoSQL and relational abstractions. The consistency of the taxonomy operations is formally evaluated with Alloy, and the Orion semantics is expressed by translating operations into native code to update data and schema. Several database systems are supported, and the engine built for each of them has been validated by testing each individual SCO and refactoring study cases. A study of relative execution time of operations is also shown.**

***Index Terms*—NoSQL databases, schema evolution, Evolution management, taxonomy of changes, schema change operations.**

I. Introduction

SCHEMA evolution is a classical problem in database research. Database schemas have to be modified during the lifetime of databases due to situations such as the appearance of new functional or non-functional requirements, or database refactoring. When this happens, stored data and code of database applications must be updated to conform to the new schema, as illustrated in Fig. 1. The desirable goal is to automate the co-evolution of data and code for schema changes in order to save effort and to avoid data and application errors.

For relational databases, such automation has formally been addressed in several works that contributed with languages and tools, among which PRISM++ [1] and DB-Main [2] are remarkable. More recently, sophisticated commercial tools are available to support relational schema evolution when agile development is applied by using continuous integration and deployment (CI/DC) [3], for example, Liquibase[1] and Flyway.[2]

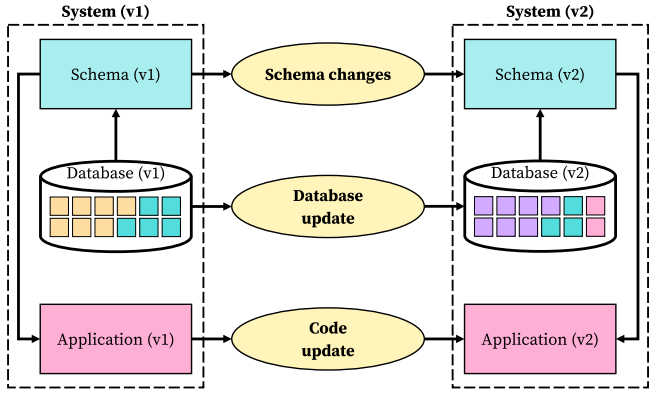

Fig. 1. Data and code must be adapted when the schema changes.

With the advent of NoSQL stores, automating the schema evolution of such stores is also attracting great interest [4], [5], [6], [7]. To provide flexibility, most NoSQL systems do not require developers to specify a schema declaration, but they are *schema-on-read:* no checking against a schema is performed when data is stored. The "*schemaless*" term is commonly used to refer to this characteristic of NoSQL stores. However, not having to declare a schema does not imply the absence of one. Instead, it is implicit in data and code, but not specified explicitly. Data is always stored according to the structure of a schema that can be formally declared, or live implicit in code and data, with developers having to write code that manipulates data by having in mind that schema. Therefore, schema changes also occur for NoSQL stores, and data and code co-evolution is required.

Schemas are needed to implement the functionality offered by most database tools, such as database design, schema visualization, or code generation. This is also the case for tools that automate schema evolution, which require the initial schema and a language to express schema modifications scripts. In schemaless NoSQL systems, the initial schema could either be provided by developers in the format required or automatically inferred from data or code. Several approaches for inferring schemas have been published, such as [8], [9], [10].

There are four primary types of NoSQL stores: columnar, document, key-value, and graph; and there is no existing

Manuscript received 3 May 2023; revised 9 January 2024; accepted 23 January 2024. Date of publication 5 February 2024; date of current version 10 June 2024. This work was supported by MCIN/AEI/10.13039/501100011033 under Grant PID2020-117391GB-I00. Recommended for acceptance by Semih Salihoglu. *(Corresponding author: Jesús García Molina.)*

Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García Molina are with the Faculty of Computer Science, University of Murcia, 30100 Murcia, Spain (e-mail: alberto.hernandez1@um.es; dsevilla@um.es; jmolina@um.es).

Meike Klettke is with the Faculty of Computer Science and Data Science, University of Regensburg, 93053 Regensburg, Germany (e-mail: meike.klettke@ur.de).

Digital Object Identifier 10.1109/TKDE.2024.3362273

[1][Online]. Available: https://www.liquibase.com/.

[2][Online]. Available: https://flywaydb.org/.

specification, standard, or theory that formally establishes the data model for each of them. For property graph stores, an initiative is currently underway to define a standard, known as the GQL language.[3] According to the DB-engines ranking,[4] the most popular NoSQL stores in each category are: MongoDB (document) which ranks 5th, Redis (key-value) 6th, Cassandra (columnar) 12th, and Neo4j (graph) 22nd.

When several data models are widely used, unified or generic data models are commonly proposed to ease the work of tool builders and application developers [2], [11], [12]. In a previous work [9], we presented the U-Schema unified data model to integrate NoSQL and relational data models. Mappings were established between U-Schema and each individual data model, and extractors were developed for the most used NoSQL systems. U-Schema differs from existing generic data models such as ER [13] and ER extensions [2] because it allows representing the particularities of NoSQL data models, such as nested entities, or the existence of structural variation in schemaless systems (i.e., data of the same type can be stored having a different structure).

Using U-Schema, a generic approach to automate schema evolution in both NoSQL and relational databases can be established. We did that by defining a unified schema change taxonomy based on the U-Schema metamodel, and implementing it via a generic language and engine, Orion. In this way, developers would only need to familiarize themselves with a single schema modification language. Additionally, tool builders can leverage this language to reduce the effort required to create schema evolution automation tools. With Orion, developers can specify schema change operations (*SCOs*) in scripts written in a system-agnostic manner. That is, these operations are expressed at logical data model level, and they are independent of a particular system or data model. Orion proves especially beneficial in a *polyglot persistence* setting where schema evolution impacts multiple databases with varying data models. Without a universal language, developers would have to deal with distinct languages or environments tailored to each data model. An Orion engine has been developed for three of the most used NoSQL stores: MongoDB (document), Cassandra (columnar), and Neo4j (graph). As for the validation of our work, we formally verified the schema change semantics of each SCO in our taxonomy using Alloy. Additionally, we tested the three engines by applying every SCO to test scenarios defined by schema examples. We also employed Orion in various refactoring cases for thorough assessment.

Our work contributes to the state of the art as follows:

- We propose a taxonomy for both NoSQL and relational logical schemas. To the best of our knowledge, the heterogeneity of NoSQL data models has been addressed only by Jérôme Fink et al. and Irena Holubová et al. in their approaches defined, respectively, for hybrid polystores [5] and multi-model databases [14]. No unified approaches encompassing both relational and NoSQL data models have been documented. Furthermore, proposed taxonomies in these works lack specific operations related to relevant aspects of NoSQL, such as structural variation for entity types, and the existence of relationship types in graph stores.
- The proposed taxonomy includes a richer set of operations than those previously published [5], [14]. Being based on U-Schema, we have considered changes on relationships, distinguishing between aggregates and references. These changes are frequent in operations such as converting a particular reference into an aggregate or vice versa [15]. Also, we have included changes related to structural variations, which could be very useful, e.g., joining all the variations of a type in a single variation to remove outliers [8].
- *Orion* is a novel language to express the operations of the proposed taxonomy. Orion schema and data updaters have been built for three popular NoSQL database systems, and a study of the data updating cost for each operation has been performed.
- Non-trivial case studies of schema evolution have been carried out by using real datasets.

Preliminary work on our proposal was presented by Hernández Chillón et al. [16], subsequent to the development of the first version of the Orion language and engine, which supported MongoDB and Cassandra. This work has since been extended in several ways. An Orion engine for a graph store (Neo4j) was developed to encompass the two main categories of NoSQL systems: those based on the prevalence of aggregations (document and columnar) and reference-based systems (graph) [17]. We have conducted a new case study of a refactoring to validate the Orion engine for Neo4j, and offer a relative performance study for the three supported systems.

From a more theoretical perspective, we utilized the Alloy formal language to check the consistency of the SCO specifications. Additionally, our study of related work has been broadened, and the approaches have been distributed in two categories: generics and specific to a database system. Moreover, we have considered two influential works on relational and object-oriented schema evolution.

This paper has been organized in the following sections: The next Section is used to introduce our data model. In Section III we define our abstract taxonomy of changes. Section IV shows a formal validation for the taxonomy. In Section V the Orion Language is described as a concrete implementation of the taxonomy. In Section VI, we describe how the Orion engines have been tested, and we measure the relative performance of the different SCOs. Next, in Section VII two case studies are discussed. Section VIII is used to discuss related work, tools and research, and to compare the most promising approaches. Finally, conclusions and future work are drawn in Section IX.

II. U-Schema : A Unified Data Model

U-Schema is a generic metamodel that integrates the relational model and data models from the four most common NoSQL paradigms: columnar, document, key-value, and graph. A detailed description of U-Schema is presented in Fernández Candel et al. [9], where some of its applications are also outlined.

[3][Online]. Available: https://www.gqlstandards.org/.
[4][Online]. Available: https://db-engines.com/en/ranking.

The use of different data models for different needs of persistence is a trend, and U-Schema was devised to build generic database solutions. Here, U-Schema is used to define a generic schema evolution approach.

In this section, we will introduce U-Schema through the *Athena* language [18], which has been built to provide a generic schema definition language with high expressive power. Although most NoSQL systems are schemaless, this language is useful, for example, when designing schemas from scratch, generating data for testing purposes, or schema manipulation when there is no database whose schema can be inferred.

NoSQL data models can be classified into two categories, as noted by Pramod J. Sadalage and Martin Fowler [17]: aggregate-oriented systems (columnar, document, and key-value) where data nesting prevails over references in order to structure data, and graph systems, which are based on graph theory, with the "property graph" being the most popular graph model [19]. In graph systems, schemas are formed by *entity types* (their instances are nodes) related through *relationship types* (their instances are references or arcs), and both types can have *attributes*. In aggregate-based systems, a schema is formed by a set of entity types which can describe root or nested objects, and attributes can act as references. While *reference* is the only relationship in graph schemas, both aggregates and references are present in schemas of aggregate systems.

We will first describe the elements of a schema, and then introduce a running example to illustrate the involved concepts. In U-Schema, a *schema* is formed by a set of *schemas types* that can be *entity types* or *relationship types*. The former represents domain entities and the latter relationships between domain entities. In the individual data models integrated in U-Schema, relationship types are only part of graph models. Both schema types are formed by a set of structural variations which contain a set of features. There are four kinds of features: *attributes* hold a value of a primitive type (e.g., Number or String); *aggregates* hold an object or a collection of objects, *keys* are strings or numbers used to uniquely identify objects of a type; and *references* hold an identifier (i.e., a key) to another object. In U-Schema, keys and references are classified as *logical features*, those that hold values that play the role of objects identifiers, and attributes and aggregates are classified as *structural features*. Given an entity type e, it is a *root* entity type if the other entity types in the schema do not include an aggregate feature whose type is e. Besides, features of a schema type can be *common* to all variations, or *specific* to one or more variations.

Fig. 2 shows the `Sales_department` schema, utilized as a running example throughout. This schema includes five entity types, but does not incorporate relationship types. `Salesperson`, `Sale`, and `SeasonExercise` are declared as root entity types, whereas `PersonalData` and `SaleSummary` are non-root entity types, with instances embedded into `Salesperson` objects. An entity type declaration can include variations by specifying both the features common to all variations and the additional features of each variation, as shown by the `Salesperson` type, which has two variations: `Variation 1` does not have additional features while `Variation 2` has the `sales` and `profits` features. An entity type can also be

```
Schema Sales_department:1

Root entity Salesperson {
  Common {
    +id:           String,
    teamCode:      String,
    email:         String /^.+@.+\\.com$/,
    personalData: Aggr<PersonalData>&
  }
  Variation 1 {}
  Variation 2 {
    sales:         Aggr<SaleSummary>+,
    profits:       Integer (0 .. 9999)
  }
}

Entity PersonalData {
  city:            String,
  name:            String /^[A-Z][a-z]*$/,
  number:          Integer,
  street:          String,
  ? postcode:      Integer
}

Entity SaleSummary {
  saleId:          Ref<Sale>&,
  scheduledAt:     Timestamp,
  ? completedAt:   Timestamp,
  ? profits:       Integer
}

Root entity Sale {
  +id:             String,
  types:           List<String>,
  isActive:        Boolean,
  description:     String,
  profits:         Integer (0 .. 9999),
  exercises:       Ref<SeasonExercise as String>+
} + timeData

Root entity SeasonExercise {
  +id:             String,
  name:            String,
  description:     String,
  date_from:       Timestamp,
  date_to:         Timestamp
} + timeData

FSet timeData {
  createdAt:       Timestamp,
  updatedAt:       Timestamp
}
```

Fig. 2. `Sales_department` schema defined using Athena.

declared as a list of features, with some marked as *optional* to indicate that they are specific to one or more variations, while the rest are considered common features.

Regarding the syntax of features, a feature is specified by its name and type. Features may also have modifiers such as "+" (for *keys*) or "?" (for *optionals*). For attributes, the type can be either scalar (e.g., *Number*, *String*, and *TimeStamp*) or structured (e.g., *List*, *Map*, and *Tuple*). In the case of aggregates, the type is a non-root entity type (e.g., `PersonalData` for the `Salesperson.personalData` feature).

Keys and *references* are formed by one or more attributes. For example, `Salesperson.id` is a key, and `Sale.exercises` specifies that `Sale` objects reference `SeasonExercise` objects whose key is the `id` attribute. A cardinality needs to be specified for references and aggregations,

TABLE I
SCHEMA CHANGE OPERATIONS OF THE TAXONOMY

		Precondition	Postcondition
Schema Type Operations (Entity Type and Relationship Type)			
Add	(c^{+})	Let t be a new schema type, $t \notin T$	$t \in T$
Delete	(c^{-})	Given a schema type $t \in T$	$t \notin T$
Rename	$(c^{=})$	Given a schema type $t \in T$ and a string value n, $n \notin T.names$	$t.name = n$
Extract	$(c^{+,=})$	Given a schema type $t \in T$, a set of features $fs \subset F^t$ and a string value $n \notin T.names$	$t \in T \wedge t_1 = T.new \wedge t_1.name = n \wedge t_1.features = fs$
Split	$(c^{=})$	Given a schema type $t \in T$, two sets of features $fs_1 \subset F^t \wedge fs_2 \subset F^t$ and two string values $n_1, n_2 \notin T.names$	$t \notin T \wedge t_1 = T.new \wedge t_2 = T.new \wedge t_1.name = n_1 \wedge t_1.features = fs_1 \wedge t_2.name = n_2 \wedge t_2.features = fs_2$
Merge	$(c^{=})$	Given two schema types $t_1, t_2 \in T$ and a string value $n \notin T.names$	$t_1, t_2 \notin T \wedge t = T.new \wedge t.name = n \wedge t.features = t_1.features \cup t_2.features$
Structural Variation Operations			
Delvar	(c^{-})	Given a schema type $t \in T$ and a variation $v^t \in V^t$	$v^t \notin V^t$
Adapt	$(c^{=})$	Given a schema type $t \in T$ and two variations $v_1^t, v_2^t \in V^t$	$v_1^t \notin V^t$ *(Data is migrated from v_1^t to v_2^t)*
Union	(c^{+})	Given a schema type $t \in T \wedge V^t \neq \{\}$	$V^t = \{v_m\} \wedge v_m.features = \cup_{i=1}^{n} v_i^t.features$
Feature Operations (Attribute, Reference and Aggregate)			
Delete	(c^{-})	Given a schema type $t \in T$ and a feature $f \in F^t$	$f \notin F^t$
Rename	$(c^{=})$	Given a schema type $t \in T$, a feature $f \in F^t$, and a string value $n \notin t.features.names$	$f.name = n$
Copy	(c^{+})	Given two schema types $t_1, t_2 \in T$ and a feature $f \in F^{t_1} \wedge f \notin F^{t_2}$	$f \in F^{t_1} \wedge f \in F^{t_2}$
Move	$(c^{+,-})$	Given two schema types $t_1, t_2 \in T$ and a feature $f \in F^{t_1} \wedge f \notin F^{t_2}$	$f \notin F^{t_1} \wedge f \in F^{t_2}$
Nest	$(c^{+,-})$	Given an entity type $e_1 \in E$, a feature $f \in F^{e_1}$, and an aggregate $ag \in F^{e_1} \wedge ag.type = e_2 \wedge f \notin F^{e_2}$	$f \notin F^{e_1} \wedge f \in F^{e_2}$
Unnest	$(c^{-,+})$	Given an entity type $e_1 \in E$, an aggregate $ag \in F^{e_1} \wedge ag.type = e_2$, and a feature $f \notin F^{e_1} \wedge f \in F^{e_2}$	$f \in F^{e_1} \wedge f \notin F^{e_2}$
Attribute Operations			
Add	(c^{+})	Given a schema type $t \in T$, let at be an attribute, $at \notin C^t$	$at \in C^t$
Cast	$(c^{+\mid-})$	Given a schema type $t \in T$, an attribute $at \in F^t$, and a scalar type st	$at.type = st$
Promote	$(c^{=})$	Given an entity type $e \in E$ and an attribute $at \in F^e \wedge at.key = False$	$at.key = True$
Demote	$(c^{=})$	Given an entity type $e \in E$ and an attribute $at \in F^e \wedge at.key = True$	$at.key = False$
Reference Operations			
Add	(c^{+})	Given a schema type $t \in T$, let rf be an reference, $rf \notin C^t$	$rf \in C^t$
Cast	$(c^{+\mid-})$	Given a schema type $t \in T$, a reference $rf \in F^t$, and a scalar type st	$rf.type = st$
Mult	$(c^{+\mid-})$	Given a schema type $t \in T$, a reference $rf \in F^t$, and a tuple $(l, u) \in \{(0,1),(1,1),(0,-1),(1,-1)\}$	$rf.lowerBound = l \wedge rf.upperBound = u$
Morph	$(c^{=})$	Given a schema type $t \in T$ and a reference $rf \in F^t$, let ag be a new aggregate, $ag \notin F^t$	$rf \notin F^t \wedge ag \in F^t \wedge ag.name = rf.name \wedge ag.type = rf.type$
Aggregate Operations			
Add	(c^{+})	Given an entity type $e \in E$, let ag be an aggregate, $ag \notin C^e$	$ag \in C^e$
Mult	$(c^{+\mid-})$	Given an entity type $e \in E$, an aggregate $ag \in F^e$ and a tuple $(l, u) \in \{(0,1),(1,1),(0,-1),(1,-1)\}$	$ag.lowerBound = l \wedge ag.upperBound = u$
Morph	$(c^{=})$	Given an entity type $e \in E$ and an aggregate $ag \in F^e$, let rf be a new reference, $rf \notin F^e$	$ag \notin F^e \wedge rf \in F^e \wedge rf.name = ag.name \wedge rf.type = ag.type$

such as *one to one* (symbol "&"), *zero to one* ("?"), *one to many* ("+"), or *zero to many* ("*").

Finally, in the running example a *FSet* named `timeData` is used to factor out a set of features appearing in several type declarations. In this case, `timeData` is added to the `Sale` and `SeasonExercice` entity types.

III. A TAXONOMY OF CHANGES FOR U-SCHEMA

In schema evolution approaches, the set of changes that can be applied on a particular data model is usually organized in form of a taxonomy [6], [20]. Several categories are established depending on the kind of schema element affected by a change. Here, we present a taxonomy for the U-Schema data model introduced in the previous section, which includes schema change operations for all of its elements. In this way, our taxonomy includes all the operations proposed in the studied taxonomies, and adds new operations, such as those related to aggregates, references, relationship types, and variations, as shown later.

Next, the terminology used to define the semantics of operations in our taxonomy is introduced. Let T be the set of schema types, and let E be the set of entity types $E = \{E_i\}, i = 1 \ldots n$, $T = E$ in the case of aggregate-based NoSQL stores and relational databases, while $T = E \cup R$ in the case of graph stores, where $R = \{R_i\}, i = 1 \ldots m$ denotes the set of relationship types. Each schema type $t \in T$ includes a set of structural variations $V^t = \{v_1^t, v_2^t, \ldots, v_n^t\}$, with $v_i^t.features$ denoting the set of features of a variation v_i^t. Then, the set of features of a schema type t is $F^t = \bigcup_{i=1}^{n} v_i^t.features$, which will include attributes, aggregates, and references, and $C^t \subset F^t$ denotes the set of common features of a type t. 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.

The proposed taxonomy is shown in Table I. In a similar manner to that presented by Carlo Curino et al. [1], we added SCOs taking into account a compromise between atomicity, usability, and reversibility. In the case of changes affecting

variations, usefulness and atomicity have prevailed over reversibility. Each SCO is defined by an identifying name, together with information regarding the gaining or loss of information the operation causes on the schema, denoted by a c^{σ} notation as follows:

- c^{+} if an operation carries an *additive change*, e.g., *Add Schema Type*.
- c^{-} if a *subtractive change* occurs, e.g., *Delete Feature*.
- $c^{+,-}$ denotes an operation in which there is a gain and a loss of information, e.g., *Move Feature*.
- $c^{=}$ means no change in information, e.g., *Rename Schema Type*.
- $c^{+|-}$ adds or subtracts information, depending on the operation parameters, e.g., *casting* a feature to boolean.

As noted by Carlo Curino et al. [1], a SCO can be considered as a function whose input is a schema S and a database D conforming to it, and produces as output a modified schema S' and the database D' that results of updating D to conform to S'. As a matter of fact, the code of applications using the schema S should also be updated, but this is not considered here. In this paper, the SCO semantics are defined in form of pre and postconditions, which appear in the second and third column of Table I. Note that the postconditions only specify the changes to the schema but not to the database, because these depend on the concrete data model. Therefore, the database update semantics are not included here. However, we added a comment to the *Adapt* postcondition to show that its effect on the database is different from that of the *Delvar* operation: Both operations share the same schema updating semantics (a specific variation is deleted), but they hold different database update semantics.

As Table I shows, taxonomy operations are classified in 6 categories that correspond to U-Schema elements: *schema types*, *variations*, *features*, *attributes*, *references*, and *aggregates*.

The *Schema type* category groups operations that can be applied to both *entity* and *relationship types*. In addition to the *Add*, *Delete*, and *Rename* atomic operations, three complex operations are added to create new schema types from existing ones. The *Extract* operation creates a new schema type by copying some of the features of an existing schema type, and leaving the original schema type unmodified. The *Split* operation divides an existing schema type into two new schema types by separating its features into two subsets, and the original schema type ceases to exist. The *Merge* operation can be understood as the inverse of the previous operation: a new schema type is created as the union of two existing ones, which are removed afterwards.

The *Structural Variations* category groups three operations: *Delvar* which deletes a given variation, *Union* which merges all the variations of a schema type into a single one, and *Adapt* which migrates data from a variation to another one. Given two variations v_1 and v_2 of a schema type, *Adapt* creates a new instance of v_2 for each instance of v_1 by: (i) copying the features present in both variations from the migrated instance of v_1 to the new instance of v_2, (ii) the features only present in v_2 are added to the instances created and they are initialized to default values, and (iii) the features only present in v_1 are ignored. *Delvar* and *Adapt* could be useful, for example, to remove outliers (i.e., variations with a small number of elements) [8], while *Union* would be interesting when a union type is desired instead of maintaining variations.

The *Feature* category groups operations with identical semantics for attributes, aggregates, and references. It includes operations to (i) copy a feature from one schema type to another, either maintaining (*Copy*) or not (*Move*) the feature copied in the original schema type; and (ii) move a feature from/to an aggregate: *Nest* and *Unnest*. The taxonomy also incorporates a category for each kind of feature, each containing operations with semantics that differ between categories. 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*. Both the *Reference* and *Aggregate* categories include the *Add* operation, along with two operations specific to relationships: *Mult* to change the multiplicity, and *Morph* to transform a reference to an aggregate or vice versa. The *Reference* category also has the *Cast* operation. It is worth noting the absence of a *Key* category because keys are logical features in U-Schema, and a *Key* is always bound to one or more *Attribute*s (a structural feature). Therefore, keys can be created and deleted by means of the attribute operations *Add*, *Promote*, and *Demote*.

All the listed SCOs, except for *Split* and *Move*, are atomic operations: they cannot be implemented as a combination of two or more other SCOs. Instead, *Move* results from applying the *Copy* and *Delete* feature SCOs, in that order, and *Split* can be defined by combining two *Extract* SCOs and a *Delete* schema type SCO.

IV. Validation of the Taxonomy

Alloy 5[5] was used to validate each schema change operation based on its pre and postconditions. This has been achieved by applying a three step process in which (i) U-Schema concepts and their restrictions have been modeled, (ii) operations implementing the taxonomy have been defined, and then (iii) *checks* for contradictions have been declared for each operation. Each step will be detailed below.

The U-Schema metamodel has been modeled in Alloy using *signatures*. It is divided into two parts: (i) *entities* and *relationships* field declarations, which are a set of *Entity types* and *Relationship types*, and (ii) A set of facts that express constraints a U-Schema model must fulfill, such as: A schema must have at least one entity type or one relationship type; No two distinct entity types can have the same name; and each reference to a schema type must belong to the same schema as that schema type.

The next step is to model the change operations in the taxonomy as Alloy operations, using *predicates* that may be applied on instances of U-Schema elements. Each operation shows the same structure: (i) it checks that input parameters do meet the preconditions, and then (ii) it matches the changes to be reflected on the output parameters.

In Fig. 3, the definition of the *Rename Entity* operation is shown. Its precondition is declared in the same

[5][Online]. Available: https://alloytools.org/.

```
pred Operation_RenameEntity [
    schemaI, schemaO: USchema,
    entityI, entityO: EntityType,
    newName: SchemaTypeName]
{
 entityI in schemaI.entities and
 entityO not in schemaI.entities
 // Precondition check: n ∉ T.names
 newName not in schemaI.entities.name

 entityO.name       = newName
 entityO.root       = entityI.root
 entityO.parents    = entityI.parents
 entityO.variations = entityI.variations
 schemaO.entities   = schemaI.entities - entityI +
     entityO
 schemaO.relationships = schemaI.relationships
}
```

Fig. 3. Alloy definition for the *Rename Entity* operation.

```
Check_Operation_RenameEntity: check
{
 all schemaI, schemaO: USchema,
     entityI, entityO: EntityType,
     newName: SchemaTypeName |
  Operation_RenameEntity[
   schemaI, schemaO, entityI, entityO, newName] =>
    // Postcondition check: t.name = n
    entityO.name = newName
    // Invariant: Everything else remains the same.
    and CheckSchemaEquality[
      schemaI.entities - entityI,
      schemaO.entities - entityO,
      schemaI.relationships,
      schemaO.relationships]
} for 10
```

Fig. 4. Postcondition checking of the *Rename Entity* operation.

way as it was specified in Table I, `newName not in schemaI.entities.name`, and then several statements are defined to be fulfilled by the output schema. When this operation is invoked in Alloy, the engine looks for scenarios in which the supplied preconditions remains true, showing the feasibility of the operation.

We have defined Alloy *Check* operations to find contradictions. For instance, the check operation for the *Rename Entity* operation is shown in Fig. 4. When executing each *Check* operation, no scenarios were found in which the implications (i.e., postconditions) of the operation are not true (counterexample).

We therefore concluded that preconditions were consistent and postconditions were valid. The usage of Alloy showed the importance of declaring *invariant* expressions in the metamodel, and also helped us refining with additional preconditions certain operations. For example, we initially defined the *Split* operation and its complementary *Merge Entity* operation. However, upon modeling their pre and postconditions, Alloy found scenarios in which an entity type was split into two entity types that shared the same name and feature set. This led us to adding a restriction to assure name uniqueness for the new entity types created by the *Split* operation. Furthermore, we realized that it could be useful to perform *Split* only providing a single set of features and getting a single entity type instead of two. Ultimately, this modified

```
⟨RenameOp⟩       ::= 'RENAME' 'ENTITY' ⟨RenameSpec⟩
⟨RenameSpec⟩     ::= ⟨EName⟩ 'TO' ⟨EName⟩
⟨ExtractOp⟩      ::= 'EXTRACT' 'ENTITY' ⟨EName⟩ 'INTO' ⟨EName⟩ ⟨SplitFeats⟩
⟨AdaptOp⟩        ::= 'ADAPT' 'ENTITY' ⟨EName⟩ '::' 'v' ⟨VarId⟩ 'TO' 'v' ⟨VarId⟩
⟨DeleteFeatOp⟩   ::= 'DELETE' ⟨MultipleFSelector⟩
⟨RenameFeatOp⟩   ::= 'RENAME' ⟨SingleFSelector⟩ 'TO' ⟨QName⟩
⟨NestFeatOp⟩     ::= 'NEST' ⟨MultipleFSelector⟩ 'TO' ⟨QName⟩
⟨CastAttrOp⟩     ::= 'CAST' 'ATTR' ⟨MultipleFSelector⟩ 'TO' ⟨PrimitiveType⟩
⟨PromoteAttrOp⟩  ::= 'PROMOTE' 'ATTR' ⟨MultipleFSelector⟩
⟨AddRefOp⟩       ::= 'ADD' 'REF' ⟨SingleFSelector⟩ ':' (⟨PrimitiveType⟩ | '{' (
                     ⟨SimpleDataF⟩ ( ',' ⟨SimpleDataF⟩ )* )? '}' ) ('?' | '&' | '*'
                     | '+' ) 'TO' ⟨EName⟩ ('WHERE' ⟨ConditionDecl⟩)?
⟨MultipleFSelector⟩ ::= ( ⟨EName⟩ ( '(' ⟨VarId⟩ ( ',' ⟨VarId⟩ )* ')' )?
                     | '*' ) '::' ⟨QName⟩ ( ',' ⟨QName⟩ )*
⟨SingleFSelector⟩ ::= ( ⟨EName⟩ ( '(' ⟨VarId⟩ ( ',' ⟨VarId⟩ )* ')' )?
                     | '*' ) '::' ⟨QName⟩
⟨SplitFeats⟩     ::= '(' ⟨QName⟩ ( ',' ⟨QName⟩ )* ')'
⟨ConditionDecl⟩  ::= ⟨QName⟩ '=' ⟨QName⟩
⟨QName⟩          ::= ID ( '.' ID )*
⟨EName⟩          ::= ID
⟨VarId⟩          ::= INT
```

Fig. 5. EBNF excerpt of the Orion language.

version of the *Split* operation was realized by introducing the *Extract* operation.

V. Implementing the Taxonomy in Orion

Orion is the language created to implement the taxonomy defined on U-Schema. With Orion, developers and database administrators can specify and execute SCOs independently of the data model and the database system. For example, the same operation *ADD* is applied to add a new column to an existing table in a relational database, a new collection in a document database, or a new relationship type between nodes in a graph database. In this section, we will first introduce the syntax of the language by presenting an Orion script for the running example, followed by a description of the structure and behavior of the Orion engine developed for MongoDB, Cassandra, and Neo4j.

A. Concrete Syntax

The ordered nature of SCOs can mimic the set of commands of a command-line interface (CLI) language. Therefore, the syntax of Orion is simple, as illustrated in Fig. 5, in which an excerpt of its EBNF grammar is shown. Note that the general format for the majority of operations is a keyword denoting the change operation (e.g., *Add* or *Delete*) followed by another keyword to indicate the kind of schema element it affects (e.g., *Entity* or *Relationship*, *Aggregate* or *Reference*), and finally a list of arguments. The Orion syntax has been defined to let operations to be written as concise as possible, e.g., it is possible to apply certain operations over all schema types by using the "*" wildcard, as in `DELETE *::name`, and operations can accept a list of parameters as in `DELETE Sales::types, isActive, description`. Operations can also be applied to specific variations of a schema type. For example, the

```
Sales_ops operations
Using Sales_department:1

// Sale operations
CAST ATTR *::profits TO Double
DELETE Sale::isActive

// PersonalData operations
CAST ATTR PersonalData::postcode TO String
ADD AGGR PersonalData::address:{country:String}&
    AS Address
NEST PersonalData::city, postcode, street TO address

// Salesperson operations
ADAPT ENTITY Salesperson::v1 TO v2
NEST Salesperson::email TO personalData
MORPH AGGR Salesperson::personalData TO privateData
RENAME ENTITY Salesperson TO Employee

// SaleSummary operations
RENAME SaleSummary::completedAt TO isCompleted
CAST ATTR SaleSummary::isCompleted TO Boolean
RENAME ENTITY SaleSummary TO Summary

// Adding a new type
ADD ENTITY Company: { +id: String, code: String,
    name: String, numOfEmployees: Number }

// Adding new features
PROMOTE ATTR Company::code
ADD AGGR Company::media: { twitterProf: String,
    fbProf: String, webUrl: String }& TO Media
```

Fig. 6. Refactoring of the `Sales_department` schema using Orion.

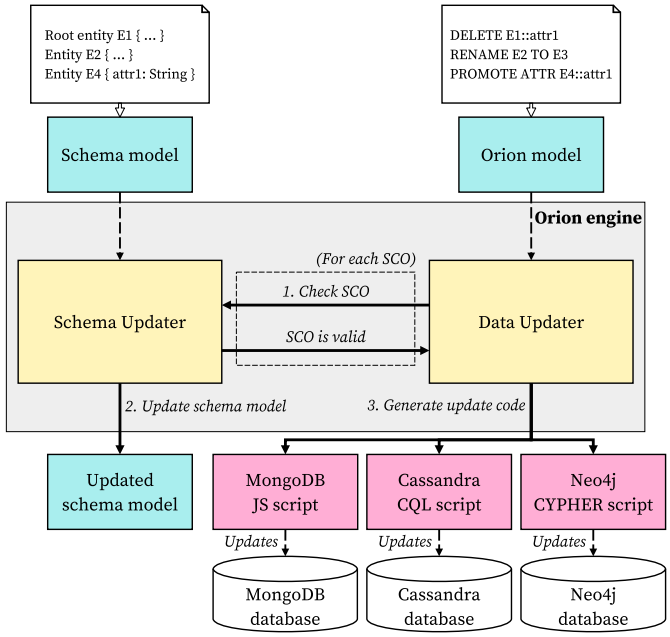

Fig. 7. Orion engine: Components and interpretation process.

`RENAME Task(v1,v3)::duration TO period` operation would only affect variations $v1$ and $v3$ of the `Task` entity type. It is worth noting that the parameters required for the `ADD REF` SCO depend on the type of target database: an aggregate-based system requires defining the primitive type of the reference value, while a graph system may permit inclusion of a set of attributes to be embedded in the reference. In both instances, the operation allows for the indication of a target entity type and a join condition.

Fig. 6 shows an Orion script that applies changes on the `Sales_department` schema of the running example in Fig. 2. As observed in Fig. 6, an Orion script starts with a `Using` statement that indicates the schema on which the changes are applied, and each SCO is validated on the current schema as explained in Section V-B.

The script shows modifications on various entity types within the schema, illustrating most of the changes in the taxonomy: (i) adding new entity types: `Company` (root type), `Media` (aggregated to `Company`), and `Address` (aggregated to `PersonalData`), (ii) attribute casting, the type of `*::profits` and `PersonalData::postCode` are changed, (iii) attribute deletion, `Sale::isActive` is removed, (iv) nesting attributes into an aggregate (`Salesperson::email` into `PersonalData` and `PersonalData::city, postcode, street` into `Address`, (v) morphing an aggregate into a reference (`Salesperson::personalData`), (vi) renaming entity types (`Salesperson`) and features (`SaleSummary::completedAt`), and (vii) adapting the variation `Salesperson::v1` to `Salesperson::v2`). In this case, the adaptation migrates all the objects that only have the common properties to `v2`.

B. Semantics: Schema and Data Update

The Orion semantics are shaped by the modifications each SCO imposes on the existing schema and stored data. For each SCO, the preconditions and postconditions specified in the taxonomy (refer to Table I) convey the schema change semantics. We employ translational semantics [21] to define the data change semantics. Specifically, SCOs are converted into native code, enabling the modification of stored data in accordance with the respective schema changes. Although space limitations prevent an exhaustive discussion of the impact for each SCO, insights can be gathered from the explanation regarding the purpose of each SCO provided in Section III. Next, we explain how the Orion engine implements the semantics, pointing out some specific details for each supported system, and we will also comment the code generated for two representative SCOs.

The Orion engine has been organized in two components: *Schema Updater* and *Data Updater*, as illustrated in Fig. 7. As shown in the figure, a three-step sequential process is applied on each SCO in the input script. The input schema is a U-Schema model obtained in one of three ways: (i) if a schema is present, a format conversion produces the Athena schema, as with Cassandra, (ii) in the case of schemaless stores like Neo4j or MongoDB, a schema extraction process is required, as detailed by Fernández Candel et al. [9], (iii) alternatively, developers can manually draft the schema in Athena. Both the input Athena schema and the Orion script are converted into models, as Orion uses a model-driven engineering (MDE) approach [21].

The *Data Updater* sequentially traverses SCOs in the script. For each SCO, it sends (step 1) a request to the *Schema Updater* to check whether the SCO is safely applicable. This

validation process first determines if the specific database supports the SCO. It permits not only SCOs for natively supported abstractions, but also those simulated by applying the patterns used in the development of the extractors for U-Schema [9]. This includes naming conventions of fields and columns for references (i.e., foreign Keys) in MongoDB and Cassandra, as well as the usage of User Defined Types (UDT) to handle aggregations in Cassandra. Subsequently, it checks that the schema is consistent with the SCO. For example, for `NEST PersonalData::city, postcode, street TO address`, three conditions must hold: (i) the `PersonalData` entity type is present in the schema, (ii) it includes the fields `city`, `postcode`, and `street`, and (iii) it also contains the `address` aggregate feature. If the validation fails, an error is issued and the whole generation process is aborted. If the SCO is safely applicable, the *Schema Updater* modifies the current U-Schema model according to the semantics of the SCO (step 2), and the *Data Updater* produces database-specific code to update the stored data in the order given in the script (step 3). The result of the Orion interpretation of each script is the generated code that implements the changes in the database, which is enclosed in a transaction to achieve atomicity. The *Data Updater* also produces code to update explicitly declared schemas, as is the case of Cassandra.

The *Schema Updater* is therefore a system-independent component because it works at a logical level by using a U-Schema model. In contrast, the *Data Updater* depends on a specific system, as the generated code is specific for each system. For example, this component is responsible for mapping U-Schema data types to the types in each system. This means that while there exists a single implementation of the *Schema Updater*, each supported system requires its own *Data Updater*. We have developed data updaters for MongoDB, Cassandra, and Neo4j. This selection ensures coverage of both aggregate-based (document and columnar) and reference-based (graph) systems, supporting three of the most used NoSQL stores.

It is worth noting that some SCOs do not modify the data. For example, creating a new schema type does not affect data. Others do not even generate code for some systems. Again, creating a new schema type does not generate code for Neo4j, as the schema type is reified when new nodes or arcs of that schema type are created. When features are added (as in *Adding attributes*), they are initialized to default values: 0 for numbers, $false$ for boolean, and $null$ for strings and references.

Next, we will contrast how the data update process is implemented for each of the three supported systems.

MongoDB: The *Data Updater* generates native MongoDB commands in Javascript code. As MongoDB is a schemaless system, the documents targeted for updates are selected based on their structure, described by the variation specified in the SCO. During the interpretation process, operations set to be sequentially applied on the same entity type are identified and grouped together into a single *bulk write* to improve performance. However, some complex operations, such as `MORPH`, do not allow that optimization, and must be executed in their own *aggregation pipeline*. Fig. 8 shows an example of bulk write for the operations `RENAME` and `CAST`.

```
Sales_department.SaleSummary.bulkWrite([
 // RENAME SaleSummary::completedAt TO isCompleted
 { updateMany: {
     filter: {},
     update: {$rename:{"completedAt":"isCompleted"}}
 }},
 // CAST ATTR SaleSummary::isCompleted TO Boolean
 { updateMany: {
     filter: {},
     update: [{$set: { "isCompleted": { $convert:
                       { input: "$isCompleted",
                         to: "bool" }}}}]
 }}
])
```

Fig. 8. Example of two operations stacked together in MongoDB.

Cassandra: This system requires to declare an explicit schema, so schema changes are limited to addition and removal of tables (entity types) and columns (attributes), changes of the type of a column, and the creation of user defined types (they allow the declaration of complex objects, which the *Data Updater* component uses to represent aggregates For the rest of SCOs supported by Cassandra, it is necessary to export the data to an external file, change the schema, and import the data back. Although Cassandra provides a `BATCH` statement that can be used to group several operations, similar to a stacking mechanism in MongoDB, it does not assure that the operations will be executed in the order they appear in the Orion script. This may compromise the consistency between SCOs and the schema (e.g., a SCO applied on a renamed attribute is executed inconsistently before the renaming is actually carried out). Therefore, we have not used the `BATCH` statement for Cassandra. Data updating is performed by generating CQL (Cassandra Query Language) statements.

Neo4j: Being a graph system, Neo4j includes relationship types, thus enabling the implementation of the full set of schema type operations. Given that schema declarations are absent, data to be updated are filtered in the same way as in MongoDB. In this case, both instances of entity types and relationship types can be updated. The *Data updater* generates Cypher statements to perform the update. Stacking is also possible by grouping together several SCOs applied over the same entity type and getting the affected nodes with a single *MATCH* operation. Then, one or more update statements are applied on those nodes.

Table II sums up how each data updater handles each operation. For each specific database, the keywords give insights on how each operation is implemented. We also indicate which operations do not have impact on existing data (for example, creating a new collection in MongoDB), and also which operations cannot be executed in a particular database. To illustrate the notation used in Table II to express how SCOs are implemented, we will detail two operations for each database, one for entity types and another one for references. The `EXTRACT` operation for entity types is implemented as:

- MongoDB (*$project,$out*): An *aggregation pipeline* is used to *project* only the selected features into a new collection (*$out* command).

TABLE II
IMPLEMENTATION AND RELATIVE EXECUTION TIME OF OPERATIONS

	MongoDB	t_M	Cassandra	t_C	Neo4j	t_N
Entity type Operations						
Add	createCollection()	0.01	CREATE *table*	0.21	*(no changes are made)*	—
Delete	drop()	0.01	DROP *table*	0.37	MATCH,DELETE *node*	1.12
Rename	renameCollection()	0	2×(COPY,DROP,CREATE) *table*	2.86	MATCH,REMOVE,SET *node*	1.89
Extract	$project,$out	0.31	2×COPY *table*,CREATE *table*	2.25	MATCH,CREATE *node*	2.92
Split	2×($project,$out),drop()	0.63	(4×COPY,2×CREATE,DROP) *table*	4.44	MATCH,2×CREATE,DELETE *node*	6.55
Merge	2×$merge,2×drop()	4.95	(4×COPY,CREATE,2×DROP) *table*	4.72	2×MATCH,CREATE,2×DELETE *node*	8.29
Delvar	remove()	0.38	—	—	MATCH,DELETE *node*	1.42
Adapt	$unset,$addFields	0.70	—	—	MATCH,REMOVE,SET *node*	1.95
Union	$addFields	1.40	—	—	MATCH,SET *node*	8.44
Relationship type Operations						
Add	—	—	—	—	*(no changes are made)*	—
Delete	—	—	—	—	MATCH,DELETE *rel*	2.43
Rename	—	—	—	—	MATCH,apoc.refactor.setType *rel*	0.22
Extract	—	—	—	—	MATCH,CREATE *rel*	5.43
Split	—	—	—	—	MATCH,2×CREATE *rel*	10.99
Merge	—	—	—	—	2×MATCH,CREATE,2×DELETE *rel*	13.84
Delvar	—	—	—	—	MATCH,DELETE *rel*	0.84
Adapt	—	—	—	—	MATCH,REMOVE,SET *rel*	0.76
Union	—	—	—	—	MATCH,SET *rel*	15.93
Feature Operations						
Delete	$unset	1.08	DROP *column*	0.22	MATCH,REMOVE *field*	0.78
Rename	$rename	1.22	2×COPY *table*,DROP *column*,ADD *column*	2.07	MATCH,SET,REMOVE *field*	2.03
Copy	$lookup,$addFields,$addFields,$out	4.06	2×COPY *table*,ADD *column*	2.21	2×MATCH,SET *field*	2.24
Move	$lookup,$addFields,$addFields,$out,$unset	5.09	2×COPY *table*,ADD *column*,DROP *column*	2.30	2×MATCH,SET,REMOVE *field*	3.31
Nest	$rename	1.27	—	—	—	—
Unnest	$rename	1.30	—	—	—	—
Attribute Operations						
Add	$addFields	1.35	ADD *column*	0.21	MATCH,SET *field*	0.76
Cast	$set	1.31	2×(COPY,DROP,CREATE) *table*	3.06	MATCH,SET *field*	1.50
Promote	—	—	2×(COPY,DROP,CREATE) *table*	3.08	CREATE *constraint* UNIQUE	4.12
Demote	—	—	2×(COPY,DROP,CREATE) *table*	3.08	DROP *constraint*	0.03
Reference Operations						
Add	$lookup,$addFields,$out	4.09	ADD *column*,2×COPY *table*	2.04	2×MATCH,CREATE *rel*	5.39
Cast	$set	1.46	2×(COPY,DROP,CREATE) *table*	3.07	—	—
Mult	$set	1.41	—	—	—	—
Morph	$lookup,$addFields,$out,$unset	4.95	—	—	—	—
Aggregate Operations						
Add	$addFields	1.43	CREATE *type*,ADD *column*	0.24	—	—
Mult	$set	1.45	—	—	—	—
Morph	insert(),save()	34.08	—	—	—	—

- Cassandra (*2x COPY table, CREATE table*): First, the selected features are exported to an external file by using *COPY*, then a new table is created and the exported data are imported back to the new table.
- Neo4j (*MATCH, CREATE node*): A single *MATCH* condition allows to filter nodes of the specific entity type, and *CREATE* a new node for each one of them copying only the selected features.

Then, the `ADD reference` operation is implemented as:

- MongoDB (*$lookup,$addFields,$out*): Implemented with an *aggregation pipeline* whose first operation is a *lookup* to match documents from the referencing collection to the referenced one. Then an *$addFields* operation sets the value of the reference on the affected documents, and write the new field (*$out*).
- Cassandra (*ADD column, 2x COPY table*): A new column that will store the references is created, then for each row on the referenced table, its identifiers and column to be referenced are exported to an external file and imported into the referencing table.
- Neo4j (*2x MATCH, CREATE rel*): Two *MATCH* statements are used to relate nodes of the referencing entity type and nodes of the referenced entity type and match them with a join condition. Then a *CREATE* is used to add a relationship between each node pair. This operation can also create initial features of the relationship, if they are provided on the script.

VI. VALIDATION OF THE ORION ENGINE

We validated the Orion engine through two types of testing. First, we ensured each SCO functioned correctly, and then we addressed two refactoring cases. In this section, we will focus on explaining the first testing approach.

We adopted a unit testing-like approach to validate each SCO. For each SCO, we devised a simple schema to serve as data test, created using the Athena language. Each test was isolated from the rest, and the same context was established for all the tests performed on a specific database (hardware, dataset, and cache). The pre and postconditions of each SCO provide a basis for designing test cases to check data updates. Assertions were employed to automatically check the data updates. The test data were customized based on the specific SCO and the database being targeted. For example, to evaluate operations such as attribute renaming, deletion, or casting, we created a single entity type with attributes of common primitive types (e.g., Integer, String, Boolean, Double). For morph operations, we defined and entity type that references another and includes an aggregate

```
MongoDBSchema.EntityAttributes
  .updateMany({}, { $set: {attr1: "Any11"}})
```

```
UPDATE entityattributes
SET attr1 = 'Any11'
WHERE id IN (...)
```

```
MATCH (x: EntityFeatures)
SET x.attr1 = 1;
```

Fig. 9. op_{mod} operation applied to each database.

feature. Additionally, to test the SCOs that involve variations in MongoDB and Neo4j, we generated several variations of a single entity type.

For each SCO test execution, a dataset comprising approximately 150,000 synthetic instances per entity type present in the schema was created for MongoDB, Cassandra, and Neo4j. We utilized the tool presented by Hernández Chillón et al. [22] to generate these datasets. The test were conducted on an Intel(R) Core(TM) i7-6700 CPU @3.4 GHz with 48 GB of RAM and using SSD storage.

During the SCO tests, we measured execution times to establish a relative benchmark for each operation. This provides developers with a point of reference for the duration of specified operations, compared to a well-known predefined operation executed on the database.

To achieve this, we used a *modification operation*, denoted as op_{mod}, as a normalization factor for the obtained times, as shown in Fig. 9. This op_{mod} operation changes the value of a non-indexed field, so the database is not optimized for it. Prior to measuring, we warmed up the database with a series of non-returning operations, such as looking for non-existent values in non-indexed fields. Then, op_{mod} was executed across all instances of a single entity type. It's important to note that an *update* operation was chosen over a simple query because our focus is on measuring database-modifying operations. Given the different nature of each database system considered, the op_{mod} operation is slightly different for each one of them.

Finally, we executed each operation independently, and organized the test execution in three blocks: (i) Entity type operations, (ii) feature, attribute, reference, and aggregate operations, and (iii) relationship type operations, if applicable. We do not consider bulk groupings, so the times show an upper bound for each operation. We reproduced the experiment five times to get a reliable mean time.

Table II introduced in previous section also shows the different execution times of each taxonomy operation for the three considered systems, The table includes a row for each SCO and two columns for each system: implementation insights and execution time. The execution times are expressed as a factor of the execution time for the op_{mod} operation on MongoDB, Cassandra, and Neo4j, which are denoted as t_M, t_C, and t_N, respectively. The factors on the table can be used to calculate approximate script execution time by summing the factor of each of the operations involved multiplied by the calculated op_{mod} for a specific database.

MongoDB operations performed as expected because the majority of them scan over a single entity type (*Delete*, *Unnest*, *Cast*), so their ratio is close to $1 \times t_M$ and only a couple of operations such as *Copy* or *Morph* require additional scans (or an explicit *join*) and therefore are more costly. As was explained in Section III, although *Delvar* and *Adapt* are semantically equal, they are implemented differently because the former removes instances belonging to a variation and the latter transforms those instances to a new variation by adding and/or deleting fields.

Cassandra operations do not show huge performance differences among them, although the ones with the `COPY` command are the most costly. As explained in Section V, these operations are the ones that were implemented by means of an export/import to an external file. These tables were of only five fields, but it is foreseeable that their performance would drop if tables had more fields. It is also important to note that CSV manipulation on the most costly operations was not included in the measurement.

Finally, Neo4j operations behaved in a similar way as in MongoDB, although certain relationship operations (*Split*, *Merge*, *Union*) performed worse than other relationship operations because they not only affect a single relationship, but also involve creating new relationships between nodes and filling their fields.

VII. Case Studies of Orion Applications

In this section, we will present two of the case studies used to validate Orion. Specifically, we demonstrate how Orion can be employed to refactor two real datasets stored in Neo4j and MongoDB. These schema evolution examples help illustrate the utility of Orion. Database refactoring is the process of applying changes to a database schema in order to improve its design, while retaining its semantics [23]. This is done to adapt the schema to new requirements, improve its performance, or to make it more maintainable, without disrupting the associated applications. A refactoring is a small change on the schema, and several refactorings can be incrementally applied to achieve a given improvement.

A. Case Study 1: A StackOverflow Refactoring in Neo4j

Orion was used to apply a refactoring to a Neo4j database holding the StackOverflow dataset.[6] We imported the dataset into Neo4j, changing it slightly during the process to take advantage of relationship types. In StackOverflow, a `Comment` references both a `User` and a `Post` in a one-to-one relation. During the loading process, we transformed the `Comment` entity type into a relationship type named `Rel_Comments`, which establishes links between `Users` and `Posts`. The database comprises about 20 million `User` nodes, 55 million `Post` nodes, and 75 million `Rel_Comments` relationships connecting them.

After loading the data, we applied the schema inference strategy from Fernández Candel et al. [9]. An excerpt from the inferred schema is shown in Fig. 10, with two of the seven discovered entity types along with the introduced relationship type. The schema is visualized using the notation introduced by Hernández

[6] [Online]. Available: https://archive.org/details/stackexchange.

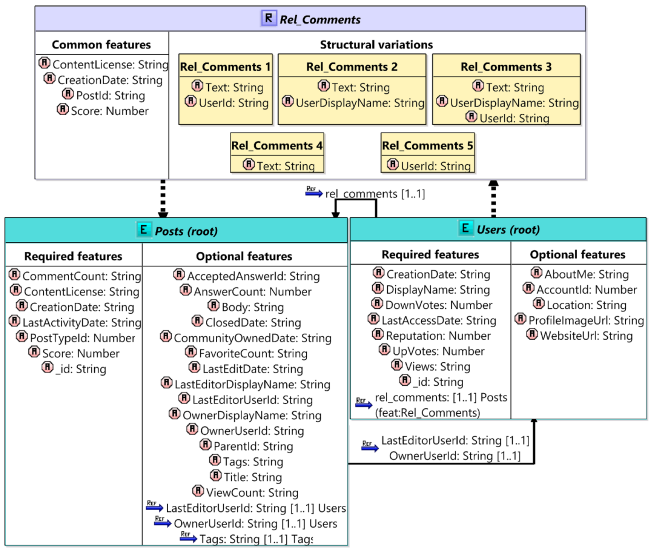

Fig. 10. Excerpt of the StackOverflow schema in Neo4j.

```
StackOverflow_ops operations
Using stackoverflow:1

CAST ATTR *::CreationDate, LastAccessDate
     TO Timestamp
MULT REF Posts::Tags TO +
COPY Posts::PostTypeId
     TO Rel_Comments::CommentTypeId
     WHERE id=PostId
COPY Users::Reputation
     TO Rel_Comments::UserReputation
     WHERE id=UserId

UNION RELATIONSHIP Rel_Comments

ADD ATTR  Rel_Comments::LastEditDate: Timestamp
ADD ATTR  Rel_Comments::KarmaCount:   Number
CAST ATTR Rel_Comments::Score      TO Double
DELETE Rel_Comments::PostId, UserId

RENAME RELATIONSHIP Rel_Comments TO Comments
```

Fig. 11. Operations to be applied to the StackOverflow schema and data.

Chillón et al. [24]. In the schema, both Posts and Users are presented as *union entity types*. Both their required attributes (common across all variations) and optional attributes (those that might not appear in every variation) are listed. The User objects reference Post objects via rel_comment references, which are instances of the Rel_Comments relationship type. This relationship type has four required attributes (ContentLicense, CreationDate, PostId, and Score), and five structural variations, each with a different set of additional features.

In the database, we conducted a refactoring to enhance query performance. In particular, for the Rel_Comments relationship type, several changes were made, as shown in the Orion script in Fig. 11.

Firstly, some *Cast* operations convert certain *string* fields to *timestamps*. These casts are performed against every schema type containing the CreationDate and LastAccessDate (all three schema types shown). Then a *Mult* operation to allow

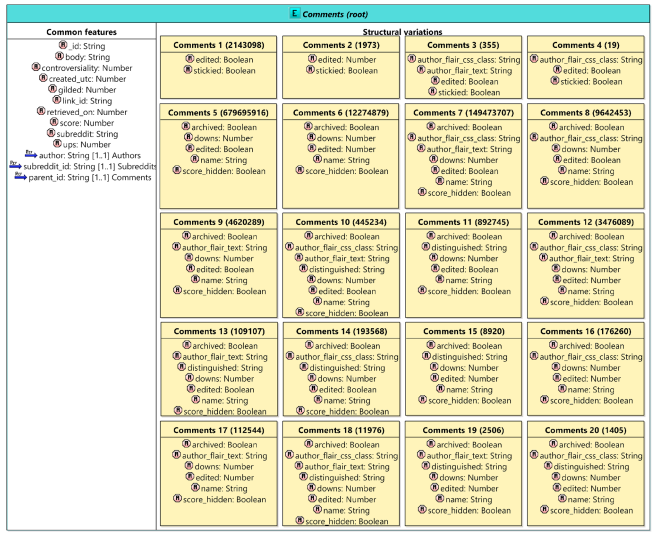

Fig. 12. Comments entity type from the *Reddit* schema.

the possibility for a post to hold more than one tag, and two *Copy* operations to move a couple of attributes from User and Post to each Comment between them, in order to get quick access to those fields. Operations regarding Rel_Comments include (i) a *Union* in order to maintain only a single variation and make all the features mandatory, (ii) two *Add Attribute* operations to create new features, which are initialized to default values as no initialization value is specified, (iii) a *Cast* over a feature that should be of *double* type, and (iv) two *Delete* operations over the two PostId and UserId carried from the loading that now are useless since the relationship stores that information. Finally, we performed a *Rename Relationship* to change the Rel_Comments name to a more suitable Comments name for a relationship. Given this script and the extracted schema, the Orion engine generates the updated schema and the Neo4j Cypher script to execute the changes on the database.

B. Case Study 2: Outlier Migration of Reddit Data in MongoDB

A structural variation of an entity type can be considered an outlier if it has a very small number of objects [8]. In a second case study, we looked for outlier variations in the Reddit dataset,[7] and the documents that belong to these variations were either deleted or migrated to non-outlier or regular variations. This task was performed by applying the following approach. We first loaded the Reddit dataset into MongoDB, and inferred its schema by applying the process described in Fernández Candel et al. [9]. In the inferred schema, structural variations of a schema type register the number of their instances (*count* attribute). Fig. 12 shows the inferred Comment entity type, with more than 860 million comments distributed in 20 structural variations.

In Fig. 13, a bar chart with logarithmic axes shows the *count* attribute for each variation. In a second step, we determine which

[7] [Online]. Available: https://files.pushshift.io/reddit/comments/.

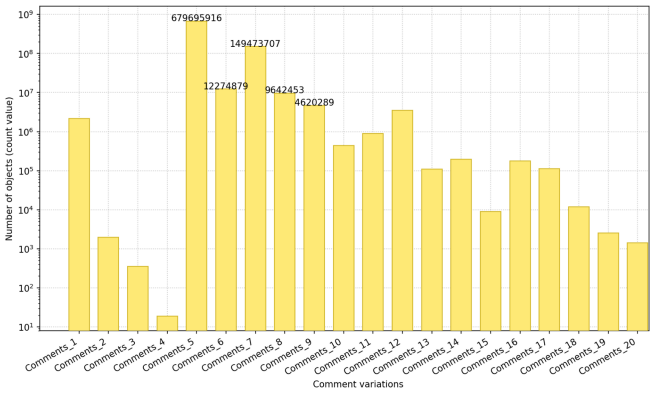

Fig. 13. `Comments` variations represented by their *count* property.

```
Reddit_migration operations

Using reddit:1

DELVAR ENTITY Comments::v1
DELVAR ENTITY Comments::v2
DELVAR ENTITY Comments::v3
DELVAR ENTITY Comments::v4

ADAPT ENTITY Comments::v10 TO v7
ADAPT ENTITY Comments::v11 TO v5
ADAPT ENTITY Comments::v12 TO v7
ADAPT ENTITY Comments::v13 TO v9
ADAPT ENTITY Comments::v14 TO v8
ADAPT ENTITY Comments::v15 TO v6
ADAPT ENTITY Comments::v16 TO v6
ADAPT ENTITY Comments::v17 TO v6
ADAPT ENTITY Comments::v18 TO v7
ADAPT ENTITY Comments::v19 TO v6
ADAPT ENTITY Comments::v20 TO v6
```

Fig. 14. Orion script used to migrate variations on Reddit `Comments`.

variations are outliers. In the case of `Comment`, few variations hold the majority of documents, and we decide to classify the top five most populated variations as *regular*, and the other fifteen variations as *outliers*. These five regular variations cover more than 99% of the comments of the entire dataset, while the remaining outlier variations only cover 1% of the comments.

Once the outlier variations were selected, we decided which ones to migrate and which ones to remove. For those variations to be migrated, we had to determine the target variation. Finally, we write the outlier migration script, by using the *Delvar* and *Adapt* operations as is shown in Fig. 14. Here, adapting variation 11 (an outlier) to variation 5 (a regular variation) means that all instances matching variation 11 will be modified accordingly to fit variation 5, reducing the number of resulting variations. On the other hand, the variations 1 to 4 are deleted which means that their instances are removed from the database; this could be appropriate if, for example, they are obsolete data.

Each of these *Delvar* and *Adapt* operations are translated into Javascript code and remove or migrate instances from a certain variation. An example of the code generated from one of the *Adapt* operations is shown in Fig. 15, where a match that captures only instances of variation 11 is applied and then fields are removed and added with default values as needed. Once the script is executed against the database, data is migrated,

```
// ADAPT ENTITY Comments::11 TO 5
reddit.Comments.updateMany({
  "archived":                {$exists: true},
  "distinguished":           {$exists: true},
  "downs":                   {$exists: true},
  "edited":                  {$exists: true},
  "name":                    {$exists: true},
  "score_hidden":            {$exists: true},
  "author_flair_css_class": {$exists: false},
  "author_flair_text":       {$exists: false}},
  [
    {$unset: ["distinguished"]}
])
```

Fig. 15. Orion script migrating `Comments` variation 11 to 5.

variations are removed and the complexity of the schema is reduced.

VIII. Related Work

Schema evolution has consistently been a central topic in database research, with various approaches proposed for different data models that have emerged over the years [25], [26]. Recently, the advent of NoSQL systems has increased the interest in studying schema evolution in these systems, while simultaneously new tools and approaches for the agile evolution of relational systems have been introduced. In this section, we will compare our proposal with research work centered on NoSQL systems and also consider some influential works published for relational and object-oriented (OO) databases. We will be separating generic approaches from those defined for a particular system.

Generic approaches

At the end of the nineties, when object databases arose as an alternative to relational systems for some kinds of applications, Jean-Luc Hainaut et al. developed DB-Main, a generic approach aimed to support database engineering tasks for the existing data models, such as database design, reverse engineering, and evolution [2], [27]. With the definition of U-Schema, we are pursuing the same objectives as DB-Main but addressing both the NoSQL and relational data models.

DB-Main was based on two main elements: (i) The Generic Entity/Relationship (GER) metamodel to achieve platform-independence; and (ii) a transformational approach to implement processes such as reverse and forward engineering, and schema mappings. Our proposal is also based on a generic metamodel and a transformational approach, but differs in two significant aspects. Firstly, GER did not integrate data models supported by NoSQL systems. Instead, we used the U-Schema metamodel, which was specially designed to support NoSQL and relational schemas. Secondly, we took advantage of MDE technology incorporated in the EMF/Eclipse framework, as the U-Schema data model is implemented as an Ecore metamodel [28]. Fernández Candel et al. provides a detailed comparison between the GER and U-Schema data models in the article that presents U-Schema [9]. With regard to schema evolution, the taxonomy proposed by John F. Roddick et al. [29] was adopted in DB-Main. This taxonomy was defined for the Entity-Relationship data model. It includes operations to modify entities, attributes

and relationships, which are mapped to operations on the relational model. In addition to the add/remove/rename operations for each element, the taxonomy includes operations to convert attributes into entities (and vice versa), to convert weak entities into regular entities (and vice versa), to change the cardinality of relationships, and to promote/demote attribute into/from key. Our taxonomy covers all the operations of the taxonomy of Roddick et al. either with a direct mapping or through the execution of several operations. Additionally, our taxonomy includes specific operations for NoSQL data models, which are not considered in Roddick's work.

As noted by Sudha Ram [30], heterogeneous database systems are commonly implemented through either a unified schema approach or a multi-database approach. Holubová et al. explored schema evolution for multi-database systems by proposing a taxonomy of 10 operations for a layered architecture which consists of a model-independent layer and a model-specific layer [6], [31]. The former layer delegates to the corresponding model-specific components by examining the prefix of the affected stored objects, thus providing a way to support both intra-model and inter-model operations. The operations of the taxonomy include five for entity types (*kinds*) and five for properties. The former five— *add*, *drop*, *rename*, *split*, and *merge* — have the same meaning as in our taxonomy, and the latter five correspond to those defined by Meike Klettke et al. that are discussed below [32]. The impact of operations is analyzed by classifying them as either intra-model or inter-model, based on the number of models affected by the changes. Additionally, operations are categorized as global or local, depending on whether they may be specified over the global union schema, or only over a specific model. Addressing issues on heterogeneous systems is beyond the scope of what we have tackled with the Orion approach. We are focused on providing a unified solution to automate the schema evolution for systems based on a single data model. Nonetheless, it is noteworthy that Orion is particularly well-suited for polyglot persistence scenarios, where schema evolution extends across diverse data models, compelling developers to manage multiple languages and environments. Moreover, our taxonomy includes operations that involve relationships and variations. Lastly, we provide a comprehensive language designed for the definition and execution of schema change operations.

As part of an approach aimed to rewrite queries for polystore evolution, Jérôme Fink et al. proposed a taxonomy that includes six operations applicable to *entity types*, four to *attributes*, and four to *relations* [5]. A generic language, called TyphonML, is used to define relational and NoSQL schemas, physical mapping, and schema evolution operations. Like our approach, TyphonML also relies on a generic metamodel, which is similarly crafted using the Ecore metamodeling language. However, U-Schema offers a richer data model as discussed in the paper that presents it [9]. This enables us to define operations for abstractions as structural variations and relationship types (in graph stores), and to consider two kinds of relationships: aggregates and references (in aggregate-oriented stores).

Meike Klettke et al. proposed a 5-operation taxonomy in a work focused on efficient data migration for aggregate-oriented NoSQL systems [32]. This taxonomy is based on a very simple data model: a schema is composed of a set of entity types, each formed by attributes whose type can be a primitive, collection, or another entity type, but relationships between entities are not considered. This data model is intended to serve as an abstraction layer on top of different NoSQL databases, thereby defining additional constraints. The schema evolution operations defined include adding, deleting, renaming properties, and copying or moving a set of properties from one entity type to another. This taxonomy was implemented in Darwin [7], [33], a data platform for schema evolution management and data migration, and also on Google Cloud platform whith the Cleager tool [34], which maps operations of the taxonomy to MapReduce functions. Our U-Schema-based taxonomy is clearly more comprehensive in terms of defined operations and the supported data models. Orion includes operations on abstractions such as relationships, variations, and relationship types, and the taxonomy unifies operations for relational and NoSQL systems (aggregate and graph systems).

Approaches for specific NoSQL stores

Neo4j (Graph): Angela Bonifati et al. recently proposed a mathematical framework designed for the validation and evolution of schemas for property graph (PG) stores [4]. Schemas are also represented as PG, and a schema DDL based on OpenCypher is introduced with the purpose of supporting the algorithmic contributions of their work. The paper formalizes the notion of PG schema and shows how schema validation can be applied by using homomorphisms. The proposed taxonomy includes operations applicable to both entity types and relationship types: add, drop and rename, split and join, and also adding, removing and renaming attributes. All these operations are included in our taxonomy with the same semantics. The authors applied rewrite rules to formalize schema and data updates, and they have used the `ReGraph` library to implement a prototype. This implementation aims to prove the conceptual and technical feasibility of the proposal. We have not tackled code updating in this work, but we provide a universal schema evolution language, which has been tested for three popular databases, and a reference execution time has been obtained for each operation. Also, we have formally validated the taxonomy operations with Alloy. Interestingly, the approach of Bonifati et al. aims to support prescriptive and descriptive schemas, i.e., an application could simultaneously use DDL schemas and schemas implicit in data and code, in order satisfy the requirements of applications both in production and in development. Instead, we have considered that schemas can come from schema-on-read and schema-on-write systems, but evolution is applied on prescriptive schema represented in U-Schema.

Cassandra (columnar): Suárez-Otero et al. [35] recently published a work addressing the schema evolution in Cassandra, focusing on maintaining consistency between conceptual and logical schemas. To achieve this, they have developed an MDE solution, CoDEvo, using model transformations to implement inter-schema consistency. They defined a simple conceptual schema metamodel, which represents a schema as a set of *Entities*, which can have *Attribute*s (reference a column of a logical schema model), a set of *Relationship*s between two

entities, and a *Weak Entity* that inherit from *Entity*. Another simple metamodel represents Cassandra schemas. By analyzing changes in conceptual schemas from some real projects, a taxonomy was defined which includes operations on the four kinds of elements above mentioned. All these operations are present in the Orion taxonomy except for *Add Weak entity* and *Split attribute*, which suggest that U-Schema can serve to represent both conceptual and logical schemas. No language was developed for the proposed taxonomy by Suárez-Otero et al, and changes are not propagated to the database. CoDEvo was empirically evaluated by comparing the generated schemas with the schemas declared by developers in nine projects of public repositories.

Redis (Key-Value): KVolve [36] is a library that allows schema evolution in the Redis[8] key-value store. It is restricted to key and value changes for entries sharing a common prefix, and accepts a previously-defined user function written in C with the actions to be performed. Key changes must be done by unambiguous bijections, and value changes can only access the value to update it. This library operates on standalone Redis instances. A lazy strategy is applied to update entries as they are accessed. This solution is limited to Redis, while our approach is generic and we defined a taxonomy of changes expressed at the logical level. In our approach, key-value stores can store aggregate and reference values as described by Fernández Candel et al. [9], where a mapping of Redis to U-Schema is presented. However, we have not built an Orion engine for Redis yet.

Two influential approaches for relational and OO databases: PRISM/PRISM++ [1] is an approach and tool intended to automate data migration tasks and rewrite legacy queries. It defines an evolution language based on *Schema Modification Operators* (SMOs), which preserve information and can be are revertible, and *Integrity Constraint Modification Operators* (ICMO). Given a schema, a new schema, and a set of mappings expressed through SMOs and ICMOs, queries are rewritten and stored data are updated. An assessment of the performance associated with rewriting queries is presented. Although much more mature and evolved than our work, this approach does not address the NoSQL database evolution.

In OO systems, schema evolution is a more complicated problem than in relational systems. This is because OO schemas are composed of classes, hierarchies of inheritance, and aggregation, while relational schemas are sets of tables. In addition, classes have structure (attributes) and behavior (methods). OO schema evolution aroused great interest until the mid-1990 s, when OO systems evidenced limitations to become an alternative to relational systems. A survey on that topic was presented by John F. Roddick [25]. Banerjee et al. [20] published a seminal paper proposing a schema change taxonomy, and discussing the operations whose semantic impact was analyzed. Our proposal is inspired by that work: we have defined a taxonomy for NoSQL databases, the change operations are rigorously specified and its performance is measured.

To conclude this section, it is important to note that schema versioning is intimately related to schema evolution. Schema version management systems enable the creation of a new schema version when schema evolves, track the version history, and keep a database instance for each version. Kai Herrmann et at. have recently conducted notable work on schema versioning by developing the InVerDa tool, which extends relational systems with schema version management [37]. InVerDa includes BiDel, a schema evolution language based on a change taxonomy. However, schema versioning is beyond the scope of our work with Orion.

In short, the differences between our work and the existing ones can be summarized as follows. We start with a NoSQL schema represented as a U-Schema model, which has been either extracted from a existing store, or written using Athena. This schema can then be changed by writing a Orion script, and the schema and data updates are automatically performed in the database. Orion is a system-independent operation language because U-Schema is a unified data model that includes all the typical elements of logical NoSQL and relational schemas, even structural variations are considered, which allows a more complete taxonomy to be defined.

IX. Conclusions and Future Work

In this paper, we have explored the NoSQL schema evolution by offering a generic solution: a unified data model with which we defined a taxonomy of schema changes. We presented the Orion schema operation language implementing this taxonomy. Thanks to the richness of the unified metamodel abstractions, we were able to define changes that affect aggregates, references, and variations. The operations have been implemented for three widely used NoSQL stores, one based in documents and schemaless, other column-based that requires schema declarations, and a third one based in graphs. The usefulness of our proposal has been validated through a refactoring of the StackOverflow dataset and an outlier migration on the Reddit dataset. This work also presents an application of the unified metamodel presented by Fernández Candel et al. [9]. An implementation of Athena and Orion are publicly available on a GitHub repository.[9]

Although the main purpose of the Orion language is to support schema changes in a platform-independent way, it can be used in other cases. For example, if no initial schema is provided, an Orion script can bootstrap a schema by itself through the use of the *Add*, *Nest*, and *Promote* SCOs. These SCOs will create the required collections and schema definition (in case of having an explicit schema). Once the database is provided with the defined schema, it can be populated. In addition to this, differences between Athena schemas may be expressed as Orion specifications; and Orion specifications may be obtained from specifications of existing tools such as the PRISM/PRISM++ operation language [1].

The future work considered includes: (i) Updating application code that makes use of the retrieved data as well as handling query rewriting. Some preliminary work has been done by Hernández Chillón et al. [38], where code analysis is used to detect expressions that need to be updated, and by

[8][Online]. Available: https://redis.io.

[9][Online]. Available: https://github.com/modelum/uschema-engineering.

Fernández Candel et al. [39], where code analysis is proposed to extract schemas, apply refactorings and provide suggestions of code modifications; (ii) Investigating new operations to be added to the taxonomy, such as operations regarding schema inheritance and type hierarchies, and refining existing ones as needed; (iii) Extending Orion to generate code for specific programming languages, which will allow to implement operations on databases that are not supported natively; (iv) Applying a dependency study on Orion scripts to check the feasibility of reordering SCOs on the same schema type. This way, related SCOs could be stacked together and issued as a single operation to achieve better performance gain; (v) Finally, integrating Orion into a tool for agile migration.

References

[1] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the database schema evolution process," *VLDB J.*, vol. 22, pp. 73–98, 2013.

[2] J.-M. Hick and J.-L. Hainaut, "Strategy for database application evolution: The DB-MAIN approach," in *Proc. Int. Conf. Conceptual Model.*, 2003, pp. 291–306.

[3] P. Sadalage and M. Fowler, "Evolutionary database design," 2016. [Online]. Available: https://martinfowler.com/articles/evodb.html

[4] A. Bonifati, P. Furniss, A. Green, R. Harmer, E. Oshurko, and H. Voigt, "Schema validation and evolution for graph databases," in *Proc. 38th Int. Conf. Conceptual Model.*, Salvador, Brazil, 2019, pp. 448–456.

[5] J. Fink, M. Gobert, and A. Cleve, "Adapting queries to database schema changes in hybrid polystores," in *Proc. 20th IEEE Int. Work. Conf. Source Code Anal. Manipulation*, Adelaide, Australia, 2020, pp. 127–131, doi: 10.1109/SCAM51674.2020.00019.

[6] I. Holubová, M. Vavrek, and S. Scherzinger, "Evolution management in multi-model databases," *Data Knowl. Eng.*, vol. 136, 2021, Art. no. 101932.

[7] U. Störl and M. Klettke, "Darwin: A data platform for schema evolution management and data migration," in *Proc. Workshops EDBT/ICDT Joint Conf.*, M. Ramanath, Themis Palpanas, Eds., Edinburgh, UK, Mar. 29, 2022, vol. 3135.

[8] M. Klettke, U. Störl, and S. Scherzinger, "Schema extraction and structural outlier detection for JSON-based NoSQL data stores," in *Proc. Conf. Database Syst. Bus. Technol., Web*, 2015, pp. 425–444.

[9] C. J. F. Candel, D. S. Ruiz, and J. J. G. Molina, "A unified metamodel for NoSQL and relational databases," *Inf. Syst.*, vol. 104, 2022, Art. no. 101898.

[10] L. Wang et al., "Schema management for document stores," in *Proc. VLDB Endow.*, vol. 8, no. 9, pp. 922–933, 2015.

[11] A. Wang, "Unified data modeling for relational and NoSQL databases," infoq, 2016. [Online]. Available: https://www.infoq.com/articles/unified-data-modeling-for-relational-and-nosql-databases/

[12] "Polyglot data modeling. hackolade web," Accessed: Sep. 2023. [Online]. Available: https://hackolade.com/polyglot-data-modeling.html

[13] P. P.-S. Chen, "The entity-relationship model: Toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, 1976.

[14] I. Holubová, M. Klettke, and U. Störl, "Evolution management of multi-model data," in *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. Berlin, Germany: Springer, 2019, pp. 139–153.

[15] D. Coupal and K. W. Alger, "Building with patterns: The attribute pattern," 2019. [Online]. Available: https://www.mongodb.com/blog/post/building-with-patterns-the-attribute-pattern

[16] A. Hernández Chillón, D. Sevilla Ruiz, and J. Garcia-Molina, "Towards a taxonomy of schema changes for NoSQL databases: The orion language," in *Proc. 40th Int. Conf. Conceptual Model.*, St.John's, NL, Canada, 2021, pp. 176–185.

[17] P. Sadalage and M. Fowler, *NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence*. Boston, MA, USA: Addison-Wesley, 2012.

[18] A. Hernández Chillón, D. Sevilla Ruiz, and J. Garcia-Molina, "Athena: A database-independent schema definition language," in *Proc. Adv. Conceptual Model.*, St.John's, NL, Canada, 2021, pp. 33–42.

[19] R. Angles et al., "PG-schema: Schemas for property graphs," in *Proc. ACM Manage. Data*, vol. 1, no. 2, pp. 1–25, 2023.

[20] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth, "Semantics and implementation of schema evolution in object-oriented databases," *SIGMOD Rec.*, vol. 16, no. 3, p. 311–322, 1987.

[21] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. San Rafael, CA, USA: Morgan & Claypool Publishers, 2012.

[22] A. Hernández Chillón, D. Sevilla Ruiz, and J. García-Molina, "Deimos: A model-based NoSQL data generation language," in *Proc. Adv. Conceptual Model.*, Viena, Austria, 2020, pp. 151–161.

[23] S. W. Ambler and P. J. Sadalage, *Refactoring Databases: Evolutionary Database Design*, Boston, MA, USA: Addison-Wesley Professional, 2006.

[24] A. Hernández Chillón, S. Feliciano Morales, D. Sevilla Ruiz, and J. García Molina, "Exploring the visualization of schemas for aggregate-oriented NoSQL databases," in *Proc. 36th Int. Conf. Conceptual Model.*, Valencia, Spain, 2017, pp. 72–85.

[25] J. F. Roddick, "Schema evolution in database systems - an annotated bibliography," *SIGMOD Rec.*, vol. 21, no. 4, pp. 35–40, 1992.

[26] E. Rahm and P. A. Bernstein, "An online bibliography on schema evolution," *ACM Sigmod Rec.*, vol. 35, no. 4, pp. 30–31, 2006.

[27] J. Hainaut, "The transformational approach to database engineering," in *Generative and Transformational Techniques in Software Engineering*, Braga, Portugal: Springer, 2005, pp. 95–143. [Online]. Available: https://doi.org/10.1007/11877028_4

[28] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, Boston, MA, USA: Addison-Wesley Professional, 2009.

[29] J. F. Roddick, N. G. Craske, and T. J. Richards, "A taxonomy for schema versioning based on the relational and entity relationship models," in *Entity-Relationship Approach — ER '93*, R. A. Elmasri, V. Kouramajian, and B. Thalheim, Eds., Berlin, Germany: Springer, 1994, pp. 137–148.

[30] S. Ram, "Heterogeneous distributed database systems - guest editor's introduction," *Computer*, vol. 24, no. 12, pp. 7–10, 1991.

[31] M. Vavrek, I. Holubová, and S. Scherzinger, "MM-evolver: A multi-model evolution management tool," in *Proc. Int. Conf. Extending Database Technol.*, 2019, pp. 586–589.

[32] M. Klettke, U. Störl, M. Shenavai, and S. Scherzinger, "NoSQL schema evolution and Big Data migration at scale," in *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 2764–2774.

[33] U. Störl et al., "Curating variational data in application development," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 1605–1608.

[34] S. Scherzinger, M. Klettke, and U. Störl, "Cleager: Eager schema evolution in NoSQL document stores," in *Datenbanksysteme Für Bus.*, Technologie und Web (BTW 2015), Bonn: Gesellschaft für Informatik, 2015, pp. 659–662.

[35] P. Suárez-Otero, M. J. Mior, M. J. S. Cabal, and J. Tuya, "Codevo: Column family database evolution using model transformations," *J. Syst. Softw.*, vol. 203, 2023, Art. no. 111743. [Online]. Available: https://doi.org/10.1016/j.jss.2023.111743

[36] K. Saur, T. Dumitraş, and M. Hicks, "Evolving NoSQL databases without downtime," in *Proc. IEEE Int. Conf. Soft. Maintenance Evol.*, Raleigh, NC, USA, 2016, pp. 166–176, doi: 10.1109/ICSME.2016.47.

[37] K. Herrmann, H. Voigt, T. B. Pedersen, and W. Lehner, "Multi-schema-version data management: Data independence in the twenty-first century," *VLDB J.*, vol. 27, no. 4, pp. 547–571, 2018, doi: 10.1007/s00778-018-0508-7.

[38] A. Hernández Chillón, J. García Molina, J. R. Hoyos, and M. J. Ortín, "Propagating schema changes to code: An approach based on a unified data model," in *Proc. Workshops EDBT/ICDT 2023 Joint Conf. 3rd Workshop Conceptual Model. NoSQL Data Stores*, Ioannina, Greece, 2023, vol. 3379.

[39] C. J. F. Candel, "A unified data metamodel for relational and NoSQL databases: Schema extraction and query," Ph.D. dissertation, Faculty of Informatics, Univ. Murcia, Murcia, Spain, 2022.

Alberto Hernández Chillón received the PhD degree in computer science from the University of Murcia, in 2022. During his research, he developed a set of tools based on the U-Schema metamodel to define independent schemas, handle schema and data evolution and generate volumes of random data. Before that, he was a member of the Cátedra SAES team from 2014 to 2019, where he worked on topics related to Model-Driven Engineering, automatic code generation and NoSQL technologies.

Meike Klettke professor for data engineering with the University of Regensburg. She studied computer science with the University of Rostock, received her doctorate with the University of Rostock, in 1997 with the topic "Acquisition of integration constraints in databases" and habilitated in 2007 with a thesis on "Modeling, evaluation and evolution of XML document collections." Since 2022, she has headed the data engineering working group with the Faculty of Informatics and Data Science in Regensburg, Germany.

Diego Sevilla Ruiz received the MSc and PhD degrees in computer science from the University of Murcia. He Associate Professor with the Department of Computer Engineering (DITEC), University of Murcia, Spain. His research interests include NoSQL databases, Distributed Systems, Functional Programming and Model-Driven Engineering and Testing. He has published several journal articles and conference papers on these topics.

Jesús García Molina received the PhD degree from the University of Murcia, in 1987. He is a full professor with the Department of Informatics and Systems, University of Murcia, Spain, where he leads the Modelum group, an R&D group focused on Model-Driven Engineering with a close partnership with industry. His research interests include model-driven development, domain-specific languages, and model-driven modernization.