A Generic Schema Evolution Approach for NoSQL and Relational Databases

Alberto Hernández Chillón, Meike Klettke, Diego Sevilla Ruiz, 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 elements. The consistency of the taxonomy operations is formally evaluated with Alloy, and the Orion semantics is expressed by translating operations into native code to change stored data according to the schema changes. Several database systems are supported, and the performance results, highly dependent on the data model of each database, are analyzed.

Index Terms—NoSQL databases, Schema evolution, Evolution management, Taxonomy of changes, Schema change operations

1 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 Figure 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.

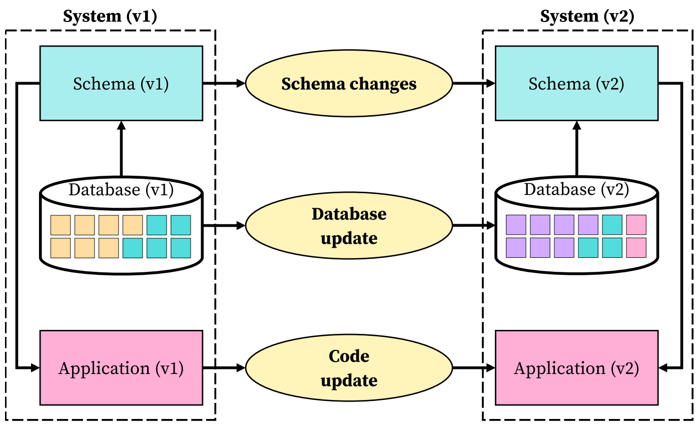

Figure 1. Data and code must be adapted when the schema changes.

• Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García Molina are with the Faculty of Computer Science, University of Murcia, Murcia, Spain.
{alberto.hernandez1, dsevilla, jmolina}@um.es
• Meike Klettke is with Faculty of Computer Science and Data Science, University of Regensburg, Regensburg, Germany.
meike.klettke@ur.de

This work has been funded by the Spanish Ministry of Science and Innovation (project grant PID2020-117391GB-I00).

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]

With the advent of NoSQL stores, automating the schema evolution of such stores is also attracting great interest [4], [5], [6]. To provide flexibility, most NoSQL systems are schema-on-read: no checking against a schema is performed when data are stored (schemaless databases). However, this does not mean the absence of schema. 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 also require of a language to express schema modification scripts. With schemaless NoSQL systems, the schema could be either provided by developers or automatically inferred from data or code. Developers should represent the designed schema in the format required by the automation tool, and several approaches have been published to infer schemas [7], [8], [9].

There are four main kinds of NoSQL stores: columnar, document, key-value, and graph; and there is no specification or standard that defines the data model of any of them. In fact, the data model can be differently supported in systems

1. https://www.liquibase.com/.
2. https://flywaydb.org/.

of the same kind. This also makes schema evolution different for each data model. In a previous work [8], 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 as ER [10] and ER extensions [2] because it allows representing the particularities of NoSQL data models, such as nested entities, or the existence of structural variations, i.e., data of the same type that can be stored having a different structure.

Using U-Schema, a generic approach to automate NoSQL and relational schema evolution may be established, which saves effort to developers and tool builders. In this paper, we have defined a schema change taxonomy for U-Schema, which has been implemented by creating the Orion language and engine. With Orion, schema change operations (*SCOs*) can be defined in scripts that are written in a platform-independent way. That is, operations are expressed at logical data model level, and they are independent of a particular system or data model. Thus, Orion is particularly appropriate in a *polyglot persistence* scenario, in which schema evolution affects to databases with different data models, and developers would have to manage a different language/environment for each data model. We have also developed an Orion engine able to update schema and data for three of the most used NoSQL stores: MongoDB (document), Cassandra (columnar), and Neo4j (graph). For each system, we have measured the performance of the taxonomy operations. Also, Orion has been validated through several refactoring cases.

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

- A taxonomy for NoSQL and relational logical schemas is proposed. As far as we know, NoSQL data model heterogeneity has only been tackled in [5], [11], and no unified approaches for relational and NoSQL data models have been published. Furthermore, proposed taxonomies in these works lack of 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 set of operations richer than those previously published [5], [11]. 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 [12]. 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 [7].
- *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.

Here we show all the work that constitutes our proposal for a generic approach to evolve database schemas. An initial work on Orion was presented in [13], once we had developed the first version of the language, which supported MongoDB and Cassandra. Since then, we have built an Orion engine for a graph database (Neo4j), with the aim of covering the two categories of NoSQL systems based on the prevalence of aggregations (document and columnar) or references (graph) [14]. We have also completed the study of performance for the three supported systems, and performed a new case study of a refactoring to validate the Orion engine for Neo4j. From a more theoretical point of view, we have used the Alloy formal language to check the consistency of the SCO specifications. Additionally, we have extended the study of the related work, and several criteria have been identified to compare schema evolution approaches, as shown in Table 3, that summarizes the comparison.

This paper has been organized in the following sections: The next Section is used to introduce our data model. In Section 3 we define our abstract taxonomy of changes. In Section 4 the Orion Language is described as a concrete implementation of that taxonomy. Next, in Section 5 two case studies are discussed. Section 6 shows a formal validation for the taxonomy and a measure of performance of Orion. Section 7 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 8.

2 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 [8], where some of its applications are also outlined. 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 [15], 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 in two categories, as noted in [14]: *aggregate-based systems* where aggregations prevail over references to connect objects, and *graph systems* where only references are present. In graph systems, references are instances of *relationship types*, and aggregate-based systems only have *entity types*, and references are managed as identifier-based joins. U-Schema allows both kinds of NoSQL systems to be represented.

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 structural variations which contain a set of features. There are

```
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
}
```

Figure 2. The `Sales_department` schema defined using Athena.

four kinds of features: *attributes* hold a value of a primitive type (e.g., Number or String); *aggregates* hold an object or an 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.

Figure 2 shows the `Sales_department` schema, which will be used as a running example. This schema includes five entity types, but no relationship types. Three entity types are root, that is, their objects are not embedded in any other object, and two are non-root as their instances are embedded into `Salesperson` objects. The `Salesperson` root entity type has two variations. These variations share a set of common features (`id`, `teamCode`, `email`, and `personalData`), and then `Variation 1` does not add any feature while `Variation 2` adds the `sales` and `profits` features.

Regarding the syntax of features, each feature declaration is specified by a name and a 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*, *Boolean*, and *TimeStamp*) or structured (*Set*, *List*, *Map*, and *Tuple*). In the case of aggregates, the type is a non-root entity type. For example, `Salesperson.email` is an attribute of type String, and `Salesperson.personalData` specifies that `PersonalData` objects are embedded in `Salesperson` objects.

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, such as *one to one* or *one 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.

3 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 [16], [6]. 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 1. In a similar way to [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^s 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 in [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 1. 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 (where a specific variation is deleted), but they hold different database update semantics.

As Table 1 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 indistinctly on *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 defined to manipulate them. The *Delvar* operation deletes a given variation, *Adapt* deletes a given variation but also migrates data belonging to the deleted variation to a new variation, and *Union* joins all the variations of a schema type into a single one. The first two operations could be useful, for example, to remove outliers (i.e., variations with a small number of elements) [7].

Similarly to the *Schema Type* category, the *Feature* category groups the operations with the same semantics for attributes, aggregates, and references. It includes operations to (i) copy a feature from a schema type to another one, 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 *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*. The *Reference* category includes the *Add* and *Cast* operations commented for attributes, *Mult* to change the multiplicity, and the *Morph* operation to transform a reference to an aggregate. Finally the *Aggregate* category includes operations *Add*, *Mult* and *Morph* commented for references. Please note that there is not *Key* category because in U-Schema a *Key* is a *logical feature* that is always bound to an attribute (a structural feature), and 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*. This means that these basic SCOs cannot be implemented as a combination of two or more other SCOs. On the other hand, *Split* and *Move* are *non-atomic operations* because they can be implemented by using other SCOs (*Move* is composed of a *Copy* and *Delete* feature operations, and *Split* can be defined as two *Extract* and a *Delete* schema type operations). These two operations have still been added to the taxonomy because they are recurrent operations in refactoring scenarios, and other approaches have considered them.

4 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 database system, for example 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.

Metamodeling has been applied to create Orion: First, we defined an object-oriented metamodel to express the abstract syntax, and then a notation or concrete syntax and semantics were defined on the metamodel [17]. The complete metamodel is not shown here because we consider the grammar notation is enough to understand the contribution of the language. The notation will be explained by showing an Orion script for the running example, and then semantics will be illustrated by indicating the generated code.

4.1 Concrete Syntax

The ordered nature of SCOs can be easily expressed similarly to commands of a command-line interface (CLI) language. Therefore, the syntax of Orion is very simple, as illustrated in Figure 3, 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

Table 1
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	$(c^{+\|-})$	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	$(c^{+\|-})$	Given a schema type $t \in T$, a reference $rf \in F^t$, and a scalar type st	$rf.type = st$
Mult	$(c^{+\|-})$	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	$(c^{+\|-})$	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, $rf \notin F^e$	$ag \notin F^e \wedge rf \in F^e \wedge rf.name = ag.name \wedge rf.type = ag.type$

operations can define a list of parameters as in `DELETE Sales::types, isActive, description`. Operations can also be applied to specific variations of a schema type, as in `RENAME *(v1,v3)::phone TO newPhone`. Finally, a specific operation, `ADD REF`, allows to define optional parameters depending on whether the target database is an aggregate-based system (defining the primitive type of the reference value) or a graph system (which can include a set of attributes to embed in the reference). In both cases, the operation also allows to indicate a target entity type and a *join* condition.

Figure 4 shows an Orion script that applies changes on the `Sales_department` schema of the running example in Figure 2. As observed in Figure 4, 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 4.2. The script shows changes on several entity types of the schema, illustrating most of the different changes in the taxonomy: casting on attributes (`*::profits`, `PersonalData::postCode`, and `SaleSummary::isCompleted`), deleting attributes (`Sale::isActive`), nesting attributes to an aggregate (`Salesperson::email` and `PrivateData::city, postcode, street`), morphing an aggregate to a reference (`Salesperson::personalData`), renaming entity types (`Salesperson`) and features (`SaleSummary::completedAt`), and adapting a variation (`Salesperson::v1`). Operations to create entity types and aggregates are also shown (`Company`, `Company::media` and `PersonalData::address`).

4.2 Semantics: Schema and Data Update

The Orion semantics are determined by the changes each operation causes in the existing schema and stored data. For

```
⟨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
```

Figure 3. EBNF excerpt of the Orion language.

```
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
```

Figure 4. Refactoring of the `Sales_department` schema using Orion.

each SCO, the preconditions and postconditions specified in the taxonomy shown in Table 1 express the schema change semantics, and the data update semantics can be easily deduced from the explanation given of the purpose of each SCO in Section 3. Note that the effect on the database will be different depending on the particular data model.

In our case, the Orion semantics are implemented by means of an interpretation process in which the SCOs in an Orion script are sequentially processed. The Orion engine is then formed by two components: The *Schema Updater* (Schema-U) and the *Data Updater* (Data-U), as shown in Figure 5. The former is in charge of updating the existing schema in each step (and producing the final resulting schema), and the later generates the database-specific code for updating stored data according to the schema modification.

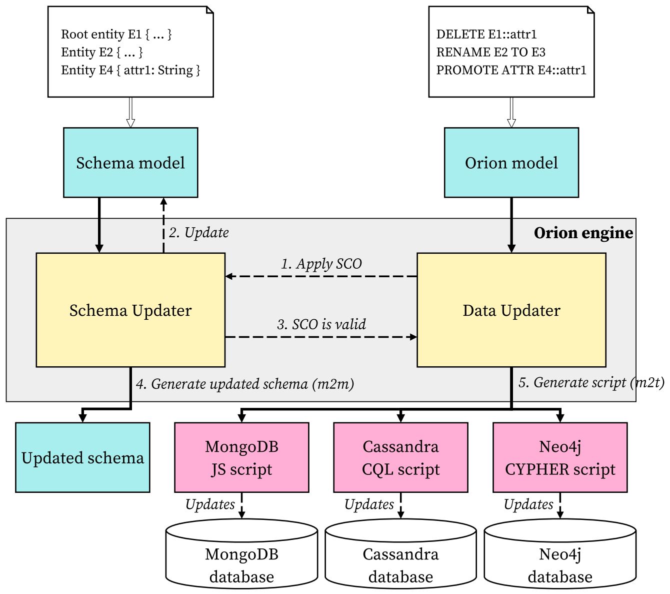

Figure 5. The Orion engine: Components and interpretation process.

As shown in Figure 5, both the input Athena schema and the Orion script are converted into models in order to implement the engine with MDE technology [17]. These models conform to the U-Schema and Orion metamodels, respectively. Using models, model transformations may be written to update schema and data: schema update is carried out by a *model-to-model* (*m2m*) transformation, and database-specific code for updating data is generated by using *model-to-text* (*m2t*) transformations.

The interpretation process is performed in two steps, as shown in 5. First, Data-U sequentially traverses SCOs in the script. For each SCO, Data-U issues a request to Schema-U to check if the visited SCO is valid. Then, if the SCO is safely applicable, Schema-U changes the current schema according to the semantics of the SCO, and Data-U generates the database-specific code to update the stored data. When the considered database allows it, generated code is enclosed in a transaction to achieve atomicity. Note that SCOs must be sequentially processed because they can involve schema changes that affect subsequent SCOs in the script, e.g., a SCO can remove a feature, and another SCO executed later could refer to the removed feature.

Two additional aspects should be taken into account with respect to the Orion engines. First, while the Schema-U component is system-independent because it works at a logical level by using a U-Schema model, Data-U depends on a specific system, as the generated code is specific for each system. This means that where there is one implementation of the Schema-U, a different Data-U must be implemented for each supported system. In our case, we have developed data updaters for MongoDB, Cassandra, and Neo4j. With this choice, we covered both aggregation-based systems (document and columnar) and reference-based systems (graph), supporting three widely used NoSQL stores.[3] Secondly, schema update can be executed as a standalone process to study schema evolution, independently of the data update.

Next, we will comment the data update process for the three supported systems.

In the case of MongoDB, the data updater generates native MongoDB commands in Javascript code. Since MongoDB does not require an explicit schema, in order to apply changes, documents belonging to the desired entity type have to be selected using structural rules. To improve performance, Orion analyzes the scripts and optimizes operations that can be applied sequentially on the same entity type, stacking them together into a single *bulk write*. Some complex operations, however, do not allow that optimization, and must be executed in their own *aggregation pipeline*. An example of this can be seen in Figure 6, where operations `RENAME` and `CAST` are applied on the same entity type and therefore can be stacked.

```
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" }}}}]
 }}
])
```

Figure 6. Example of two operations stacked together in MongoDB.

In Cassandra, CQL (*Cassandra Query Language*) instructions are generated to perform the data update. Due to Cassandra declaring an explicit schema, evolution changes are restricted. To implement some of these operations, it is necessary to export the data to an external file, change the schema, and import the data back.

The Neo4j data updater generates Cypher language code to perform the update. Given its graph nature, Neo4j is also able to handle relationship types and therefore allows the full set of schema type operations to be implemented for relationships. The schemaless nature of Neo4j also allows the database to be updated in a similar way to MongoDB: (i) Selecting all nodes belonging to the entity type to be modified and (ii) applying the desired change. This also allows to stack together changes to the same schema type, reducing the overhead of the change.

Table 2 sums up how each data updater handles each operation. For each specific database, some keywords give insight on how each operation is implemented, and also which operations cannot be executed in a particular database. For example, the `Extract` operation for entity types is implemented as follows:

3. These databases occupy the position 5, 11, and 21 in the DB-engines ranking as of April, 2023: https://db-engines.com/en/ranking.

- MongoDB (*$project,$out*): An *aggregation pipeline* is used to *project* only the selected features into a new collection (*out* command).
- Cassandra (*2x COPY table, CREATE table*): First the selected features are exported to an external file by using the *COPY* command, then a new table is created and the exported data are copied 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.

Describing how each SCO is implemented in each of the three systems is out of scope of this paper.

5 Case Studies of Orion Applications

5.1 Case Study 1: A StackOverflow Refactoring

Database refactoring is an activity aimed to improve the database design and performance without changing its semantics [18]. A refactoring is a small change on the schema, and several refactorings can be applied to achieve a determined improvement. In our first case study, Orion was used to apply a refactoring to a Neo4j database that imported the StackOverflow dataset.[4]

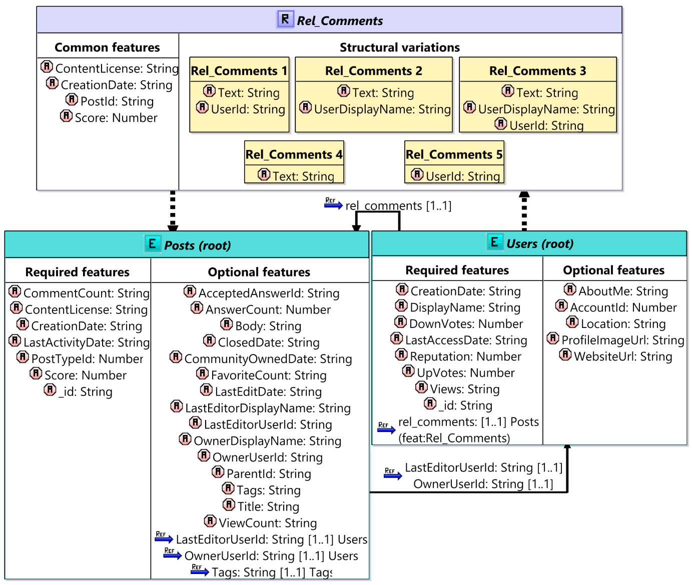

Figure 7. Excerpt of the `StackOverflow` schema injected in Neo4j.

We injected the dataset into Neo4j, but changing it slightly during injection to take advantage of relationship types. In StackOverflow, a `Comment` references its `User` and a `Post` in a 1..1 relation. We injected this database transforming the `Comment` entity type into a relationship type named `Rel_Comments` between `Users` and `Posts`. After the injection, the schema inference strategy from [8] was applied. Figure 7 shows an excerpt of the schema inferred with two of the seven entity types discovered and the already mentioned new relationship type; the schema is visualized with the notation introduced in [19]. Here, `Posts` and

4. https://archive.org/details/stackexchange.

`Users` are shown as *union entity types* listing their required (i.e., common to all variations) and optional (i.e., not present in all variations) features. `Users` references `Posts` by `rel_comments`, a reference with attributes that belongs to `Rel_Comments`. `Rel_Comments` has four required features (`ContentLicense`, `CreationDate`, `PostId`, and `Score`) and five structural variations each with a different set of additional features.

Analyzing the schema, we realized that the newly created relationship type could be improved by casting some types of attributes to specific Neo4j types, adding new fields or copying fields from `Users` and `Posts` to the relationship type. Also injecting an entity type as a relationship type caused some attributes in `Rel_Comments` to turn obsolete and so could be deleted. In this way, by slightly changing the schema, query performance could be improved.

The proposed refactoring can be divided into two blocks: (i) Applying operations to some fields of `Users` and `Posts` to improve query performance over them, and (ii) applying operations over the newly created `Rel_Comments` to improve its expressiveness.

The Orion script to refactor the StackOverflow schema is shown in Figure 8. Firstly, some *Cast* operations are performed to convert certain fields stored as *strings* to *timestamps*. These casts are performed against every schema type containing the `CreationDate` and `LastAccessDate` (which are all three schema types shown). Then a *Mult* operation to allow the possibility for a post to hold more than one tag and two *Copy* operations to move a couple of attributes from `Users` and `Posts` to each `Comment` between them, in order to get quick access to those fields. Operations regarding `Rel_Comments` include a *Union* in order to maintain only a single variation and make all the features mandatory, two *Add Attribute* operations to create new features, a *Cast* over a feature that should be of *double* type, and two *Delete* operations over the two `PostId` and `UserId` carried from the injection 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 API code script to execute the changes on the data.

```
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
```

Figure 8. Operations to be applied to the `StackOverflow` schema and data.

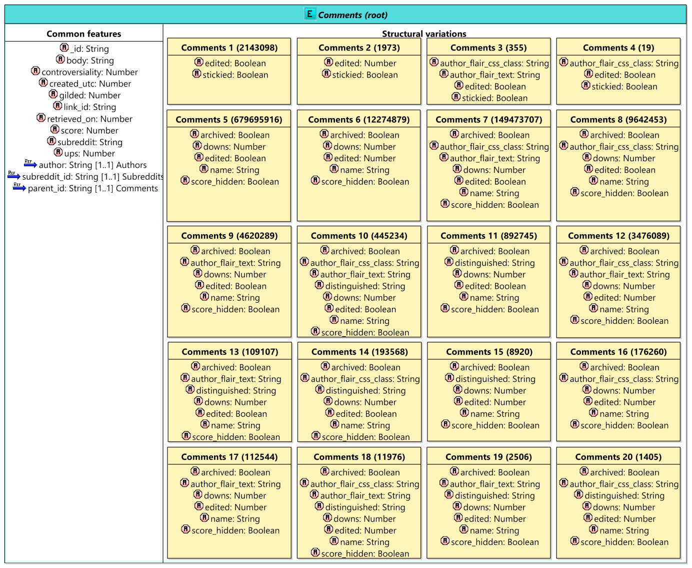

Figure 9. The `Comments` entity type from the *Reddit* schema.

5.2 Case Study 2: Outlier Migration in Reddit

A structural variation of an entity type can be considered an outlier if it has a very small number of objects. In a second case study, we looked for outlier variations in the Reddit dataset,[5] 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 injected the Reddit dataset into MongoDB, and inferred its schema by applying the process described in [8]. In the inferred schema, each structural variation of a schema type registers its number of instances (*count* attribute). Figure 9 shows the inferred `Comment` entity type, with more than 860 million comments distributed in 20 structural variations.

In Figure 10, a bar chart with logarithmic axes is shown, in which each variation is represented by its *count* attribute. In a second step, we determine which 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 do 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 Figure 11. 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

5. https://files.pushshift.io/reddit/comments/.

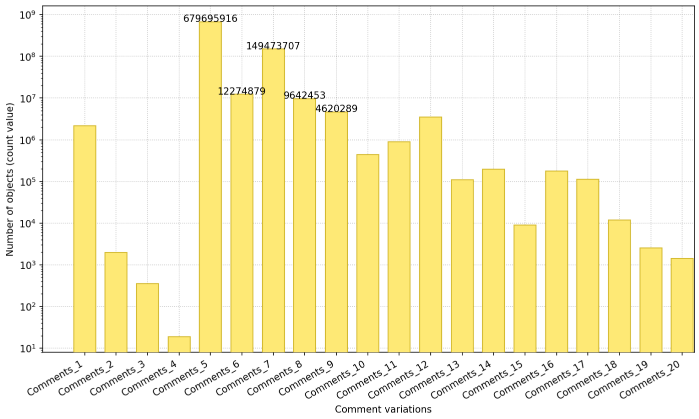

Figure 10. `Comments` variations represented by their *count* property.

the other hand, the variations 1 to 4 are deleted which means that their instances are removed from the database; this could be appropriate because they are obsolete data.

```
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
```

Figure 11. Orion script used to migrate variations on Reddit `Comments`.

Each of these *Delvar* and *Adapt* operations will be translated into Javascript code and will remove or migrate instances from a certain variation. An example of the code generated from one of the *Adapt* operations is shown in Figure 12, 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, variations are removed and the complexity of the schema is reduced as a result.

6 Evaluation

Two kinds of evaluations have been carried out. The schema change semantics of each SCO of the proposed taxonomy has been formally validated using Alloy. Also, the feasibility or applicability of the changes has been evaluated by measuring execution times for the three currently supported NoSQL systems.

```
// 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"]}
  ])
```

Figure 12. Orion script migrating `Comments` variation 11 to 5.

6.1 Validating the Taxonomy

Alloy 5[6] was used to validate each schema change 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*, and consists of two parts: (i) *entities* and *relationships* field declarations, which are a set of *Entity types* and *Relationship types*, and (ii) a set of facts expressing restrictions that any U-Schema model must fulfill, such as: (i) A schema must contain at least one entity type or one relationship type, (ii) there cannot be two different entity types with the same name, and (iii) 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.

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

Figure 13. Alloy definition for the *Rename Entity* operation.

In Figure 13, the definition of the *Rename Entity* operation is shown. Its precondition is declared in the same

6. https://alloytools.org/.

way as it was specified in Table 1, `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 Figure 14. When executing each *Check* operation, no scenarios were found in which the implications (i.e., postconditions) of the operation are not true (counterexample).

```
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
```

Figure 14. Postcondition checking of the *Rename Entity* operation.

We therefore concluded that preconditions were consistent and postconditions were valid. The usage of Alloy also served to refine with additional preconditions certain operations, such as *Extract/Split/Merge Entity*, which were not consistent at the beginning of the process. It also showed the importance of declaring *invariant* expressions in the metamodel.

6.2 Measuring the Performance of Orion Operations

To evaluate the feasibility of each implemented operation we created the following scenario for each database system considered. First, we defined a schema with several root entity types: one entity type per group of operations (features, attributes, references, and aggregates), and one entity type per schema type operation, each one of them with the same number of features. In the second step, a dataset of 150,000 instances per entity type was generated, which conform to the defined schema. This data generation was performed by using the tool described in [20], and data were obtained in JSON format. After that, we inject the dataset into a MongoDB, Cassandra, and Neo4j.

Before measuring the performance of the operations, we applied several standard queries to warm up the database and let it fill its caches. The designed warm up queries do scan the entire database looking for non-existing values on non-indexed fields.

In order to provide a meaningful expression of the feasibility of the implemented SCOs, we did not measure absolute times. Instead, we used the time of a *modification operation* op_{mod} to normalize the obtained times. This op_{mod} operation modifies a field that is not indexed, so the database is not optimized for it. In the third step, op_{mod} is applied over all the instances of a certain entity type. Note that an *update* operation is preferred over a standard query because we are measuring operations modifying the database.

The final step consists on executing each operation independently to measure its execution time. To do so, we defined three blocks to execute: (i) Entity type operations, (ii) feature, attribute, reference, and aggregate operations, and (iii) relationship type operations, if applicable. Since operations are executed individually, it is not possible to take advantage of certain mechanisms such as stacking operations together, which is relevant in the case of, for example, MongoDB. This whole process was repeated five times to get a reliable mean time. Given the different nature of each database system considered, the op_{mod} operation is slightly different for each one of them.

Table 2 shows the different execution times for each taxonomy operation performed over each of the considered NoSQL systems. The table includes two columns for each system, one shows a summary of the native code performed, and the other the execution time multiplied by a factor that is 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.

MongoDB operations performed as expected because the majority of them scan over a single entity type (*Delete*, *Unnest*, or *Cast*), so their ratio is close to $1 \times t_M$ and only a couple of operations such as *Copy* or *Morph* do require additional scans (or an explicit *join*) and therefore are more costly. As was explained in Section 3, 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 between them, although the ones with the `COPY` command are the most costly. As explained in Section 4, 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 than in MongoDB, although certain relationship operations (*Split*, *Merge*, and *Union*) performed worse than other relationship operations because they not only affect single relationships but also involve creating new relationships between nodes and filling their fields.

7 Related Work

In this section, we will contrast our proposal to some relevant schema evolution approaches presented for relational and object-oriented databases, and to most of research work done on NoSQL systems.

The works of Jean-Luc Hainaut et al. [2], [21] and Carlo Curino et al. [1], [22] are some of the most influential contributions on automating relational schema evolution. While the interest of Curino et al. was exclusively focused on relational systems in order to build the PRISM/PRISM++

Table 2
Implementation and performance of operations.

	MongoDB	t_M	Cassandra	t_C	Neo4j	t_N
Entity type Operations						
Add	createCollection(),$addFields	0.01	CREATE *table*	0.21	CREATE,MATCH,SET *node*	0.37
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	—	—	—	—	—	—
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	—	—	—	—

tool, Hainaut et al. defined the DB-MAIN generic approach that involved the main data models existing at the end of nineties.

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 several significant aspects. Firstly, GER did not integrate data models supported by NoSQL systems. We used the U-Schema metamodel instead, which was specially designed to support NoSQL and relational schemas. Secondly, we have taken advantage of Model-driven Engineering (MDE) technology incorporated in the EMF/Eclipse framework, as U-Schema data model is implemented in form of an Ecore metamodel [23]. A detailed comparison between GER and U-Schema data model is given in [8]. Also, it should be noted that no taxonomy was defined for DB-Main. Instead, the taxonomy shown in [24] is adopted.

The PRISM/PRISM++ tool is aimed at automating data migration tasks and rewriting legacy queries. PRISM/PRISM++ provides an evolution language based on *Schema Modification Operators* (SMOs) that preserve information and 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 can be rewritten and stored data are updated. 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-1990s, when OO systems evidenced limitations to become an alternative to relational systems. A survey on that topic was presented in [25]. Banerjee et al. [16] 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 the best of our knowledge, most research efforts on NoSQL schema evolution are considered below. In [26], Meike Klettke et al. propose an approach whose main focus is on efficient data migration for different NoSQL systems.

In that work, a 5-operation taxonomy is defined for a simple data model: schemas are a set of entities that are formed by attributes whose type can be a primitive or collection type or either another entity, but relationships between entities are not considered. The schema serves as an abstraction layer on top of different NoSQL databases, defining additional constraints. Regarding the five schema evolution operations, they are *add/remove/rename* properties, and *copy/move* a set of properties from an entity type to another. This taxonomy was implemented in Darwin [27], [4], a data platform for schema evolution management and data migration. Darwin is also able to extract the implicit schema and version history of NoSQL databases, manage those versions, update data eager, lazily, or with intelligent hybrid approaches and rewrite queries that try to retrieve data that is yet to be updated. It was also implemented on the Google Cloud Platform as part of the Cleager tool [28]. This tool maps operations of the taxonomy to MapReduce functions. Our proposal is based on a more complex unified data model, and this results in a richer change taxonomy, which is applicable for NoSQL and relational data models, and we have implemented and validated the taxonomy for three widely used NoSQL systems.

As noted in [29], heterogeneous database systems are commonly implemented through a unified schema approach or a multi-database approach. Holubová et al. explored schema evolution for multi-database systems in [6] and [30]. They proposed a layered architecture which consists of a model-independent layer and a model-specific layer. The former delegates to the corresponding model-specific components by examining the prefix of the affected entities, thus providing a way to support both intra-model and inter-model operations. Since heterogeneous databases can store entities referencing others stored in a different data model, the proposal also provides the foundation for managing referential integrity between them when modifying the database. Regarding schema evolution, a taxonomy of 10 operations is defined: 5 for entity types (*kinds*) and 5 for properties. These latter correspond to those defined in [26], and the first 5 are *add*, *drop*, *rename*, *split*, and *merge*, which have the same meaning as in our taxonomy. The impact of operations is discussed classifying them as intra-model or inter-model, depending on how many models are affected by changes, and as global and local operations, depending on whether they may be specified over the global union schema, or only over a specific model. We have not tackled the issues related to schema evolution in heterogeneous systems, but we are interested in offering automation for individual stores in a data model-independent way through a unified data model. Also, it is convenient to remark that our Orion approach is especially appropriate for persistence scenarios in which schema evolution affects to different data models, and developers would have to manage different languages and environments. Furthermore, our taxonomy includes operations related to relationships and variations, and we have defined a complete language to define and execute schema change operations.

In [5], a taxonomy is proposed as part of an approach to rewrite queries for polystore (i.e., heterogeneous databases) evolution. The taxonomy includes six operations applicable to *entity types*, four to *attributes* and four to *relations*. A generic language, called TyphonML, is used to define relational and NoSQL schemas, physical mapping and schema evolution operations. Like our approach, TyphonML is based on a generic metamodel also created with the Ecore metamodeling language. However, U-Schema is a richer data model as discussed in [8], which allowed us to define operations on (i) aggregates and references in a separate way, (ii) structural variations, and (iii) distinguish between entity and relationship types in graph stores.

Suárez-Otero et al. [31] have recently published a work where they address the schema evolution in Cassandra. They define a taxonomy of 7 schema changes defined at a conceptual level and analyze their effect on the physical schema and data, but no automation is addressed. While the work of Suarez-Otero et al. is completely focused on Cassandra, we proposed a generic approach for NoSQL and relational systems, the taxonomy includes a larger number of operations defined at a logical level, and it has been implemented to update schema and data. Note that Cassandra is one of the systems we have chosen to implement our taxonomy.

KVolve [32] is a library that allows for schema evolution in the Redis[7] 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 in [8], where a mapping of Redis to U-Schema is presented. However, we have not built an Orion engine for Redis yet.

In short, the differences between our work and the existing ones can be summarized as follows. We suppose a NoSQL schema represented as a U-Schema model has been extracted from a existing store, this schema can then be changed by writing a Orion script, and the schema and data updates are automatically performed. 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.

Table 3 summarizes the comparison carried out between our approach and other works. Several criteria are defined to compare the schema evolution approaches discussed above: changes operations in the taxonomies, supported database paradigms, schema representation, aim, if operation impact analysis has been performed, and if a tool is available.

8 Conclusions and Future Work

In this paper we have explored the NoSQL schema evolution by using a generic solution: a unified data model with which we defined a taxonomy of schema changes. We presented

7. https://redis.io.

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. Also note that this work presents an application of the unified metamodel presented in [8]. An implementation of Athena and Orion are publicly available on a GitHub repository.[8]

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: (i) If no initial schema is provided, an Orion script can bootstrap a schema by itself; (ii) Differences between Athena schemas may be expressed as Orion specifications; and (iii) 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 on [34], where code analysis is used to detect expressions that need to be updated, and on [35], 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) 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," in *The VLDB Journal*, vol. 22, 2013, pp. 73–98.

[2] J.-M. Hick and J.-L. Hainaut, "Strategy for database application evolution: The DB-MAIN approach," in *International Conference on Conceptual Modeling*. Springer, 2003, pp. 291–306.

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

[4] U. Störl and M. Klettke, "Darwin: A Data Platform for Schema Evolution Management and Data Migration," in *DataPlat 2022: 1st International Workshop on Data Platform Design, Management and Optimization*, 2022.

[5] J. Fink, M. Gobert, and A. Cleve, "Adapting Queries to Database Schema Changes in Hybrid Polystores," in *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 127–131. [Online]. Available: https://doi.org/10.1109/SCAM51674.2020.00019

[6] I. Holubová, M. Vavrek, and S. Scherzinger, "Evolution management in multi-model databases," *Data & Knowledge Engineering*, vol. 136, p. 101932, 2021.

[7] M. Klettke, U. Störl, and S. Scherzinger, "Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores," in *Conference on Database Systems for Business, Technology, and Web (BTW)*, 2015, pp. 425–444.

[8] C. J. F. Candel, D. S. Ruiz, and J. J. G. Molina, "A unified metamodel for nosql and relational databases," *Information Systems*, vol. 104, p. 101898, 2022.

[9] L. Wang, O. Hassanzadeh, S. Zhang, J. Shi, L. Jiao, J. Zou, and C. Wang, "Schema management for document stores," *Proc. VLDB Endow.*, vol. 8, no. 9, pp. 922–933, 2015.

[10] P. P.-S. Chen, "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36, 1976.

[11] I. Holubová, M. Klettke, , and U. Störl, "Evolution Management of Multi-model Data," in *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, 2019, pp. 139–153.

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

[13] A. Hernández Chillón, D. Sevilla Ruiz, and J. Garcia-Molina, "Towards a Taxonomy of Schema Changes for NoSQL Databases: The Orion Language," in *Conceptual Modeling - ER 2021 40th Int. Conf. on Conceptual Modeling, St.John's, NL, Canada*, vol. 13011, 2021, pp. 176–185.

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

[15] A. Hernández Chillón, D. Sevilla Ruiz, and J. Garcia-Molina, "Athena: A Database-Independent Schema Definition Language," in *Advances in Conceptual Modeling - ER 2021 Workshops CoMoNoS, St.John's, NL, Canada*, vol. 13012, 2021, pp. 33–42.

[16] 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.

[17] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

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

[19] A. Hernández Chillón, S. Feliciano Morales, D. Sevilla Ruiz, and J. García Molina, "Exploring the Visualization of Schemas for Aggregate-Oriented NoSQL Databases," in *ER Forum 2017, 36th Int. Conf. on Conceptual Modeling (ER)*, Valencia, Spain, November 2017, pp. 72–85.

[20] A. Hernández Chillón, D. Sevilla Ruiz, and J. García-Molina, "Deimos: A Model-based NoSQL Data Generation Language," in *Advances in Conceptual Modeling - ER 2020 Workshops CoMoNoS, Viena, Austria*, vol. 12584, 2020, pp. 151–161.

[21] J. Hainaut, "The transformational approach to database engineering," in *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, 2005, pp. 95–143. [Online]. Available: https://doi.org/10.1007/11877028_4

[22] C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful Database Schema Evolution: The PRISM Workbench," in *Proceedings of the VLDB Endowment*. VLDB Endowment, 2008.

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

[24] 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. Springer Berlin Heidelberg, 1994, pp. 137–148.

[25] J. F. Roddick, "Schema evolution in database systems - an annotated bibliography," *SIGMOD Rec.*, vol. 21, no. 4, pp. 35–40, 1992. [Online]. Available: https://doi.org/10.1145/141818.141826

[26] M. Klettke, U. Störl, M. Shenavai, and S. Scherzinger, "NoSQL Schema Evolution and Big Data Migration at Scale," in *IEEE International Conference on Big Data*. IEEE Computer Society, 2016.

[27] U. Störl, D. Müller, A. Tekleab, S. Tolale, J. Stenzel, M. Klettke, and S. Scherzinger, "Curating variational data in application development," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1605–1608.

[28] S. Scherzinger, M. Klettke, and U. Störl, "Cleager: Eager Schema Evolution in NoSQL Document Stores," in *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, 2015, pp. 659–662.

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

[30] M. Vavrek, I. Holubová, and S. Scherzinger, "Mm-evolver: A Multi-model Evolution Management Tool," in *EDBT*, 2019.

[31] P. Suárez-Otero, M. J. Mior, M. J. S. Cabal, and J. Tuya, "An integrated approach for column-oriented database application evolution using conceptual models," in *Advances in Conceptual Modeling - ER 2021 Workshops CoMoNoS, EmpER, CMLS, St. John's, NL, Canada, October 18-21, 2021, Proceedings*, ser. Lecture Notes in

8. https://github.com/modelum/uschema-engineering.

Computer Science, I. Reinhartz-Berger and S. W. Sadiq, Eds., vol. 13012. Springer, 2021, pp. 26–32.

[32] K. Saur, T. Dumitraş, and M. Hicks, "Evolving nosql databases without downtime," 2016.

[33] J. Hainaut, V. Englebert, J. Henrard, J. Hick, and D. Roland, "Database Evolution: the DB-Main Approach," in *Entity-Relationship Approach - ER'94, Business Modelling and Re-Engineering, 13th International Conference on the Entity-Relationship Approach, Manchester, UK, December 13-16, 1994, Proceedings*, ser. Lecture Notes in Computer Science, P. Loucopoulos, Ed., vol. 881. Springer, 1994, pp. 112–131.

[34] 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 *Proceedings of the Workshops of the EDBT/ICDT 2023 Joint Conference, 3rd Workshop on Conceptual Modeling for NoSQL Data Stores (CoMoNoS), Ioannina, Greece*, vol. 3379, 2023.

[35] C. J. F. Candel, "A Unified Data Metamodel for Relational and NoSQL databases: Schema Extraction and Query," Ph.D. dissertation, Faculty of Informatics, University of Murcia, Murcia, Spain, 2022.

Table 3
Comparison of schema evolution approaches.

		Curino et al. [22]	**Hainaut et al.** [33]	**Störl et al.** [26]	**Holubová et al.** [6]	**Fink et al.** [5]	**Hernández et al.**
Operation	**Schema Types**	Create,Drop,Rename Copy,Merge,Join Partition,Decompose	Add,Delete,Rename Change to/from weak Split,Partition,Join,Coalesce	Create,Drop,Rename	Create,Drop,Rename Split,Merge	Add,Remove,Rename Merge,Split,Migrate	Add,Delete,Rename Extract,Split,Merge
	Variations	—	—	—	—	—	Delvar,Adapt,Union
	Features	—	—	Add,Delete,Rename Copy,Move	Add,Delete,Rename Copy,Move	Add,Remove,Rename	Delete,Rename,Copy Move,Nest,Unnest
	Attribute	Add,Drop,Rename Copy,Move	Add,Drop,Rename Type change Promote,Demote	—	—	Type change	Add,Cast Promote,Demote
	Reference	—	Add,Delete,Rename Cardinality change	—	—	Cardinality change	Add,Cast,Mult,Morph
	Aggregate	—	—	—	—	Cardinality change	Add,Mult,Morph
Supported Paradigms		Relational	Relational, hierarchical Object-oriented, network	NoSQL	Multi-model	Multi-model	Relational, NoSQL
Unified Schema Representation		No	Yes (GER)	No (But a Generic Interface)	No	No	Yes (U-Schema)
Data Update		Yes	Yes	Yes (eager, lazy, hybrid)	Yes	Yes	Yes
Code Update		Query rewriting SQL Views	Program modification (hints)	Query adaptation	Query adaptation	Query adaptation	No
Implementing Tool		PRISM/PRISM++	DB-Main	Darwin	MM-evolver	TyphonML	Orion
Semantic Change Analysis		No	No	Yes	Yes	Yes	Yes