-It's hot as hell in here.
-You see it, too? For me, it's always like this...
James and Angela - Silent Hill 2

6 Orion: A Schema Evolution Language

Database schemas use to be modified along the lifetime of databases. These schema changes may be caused due to new functional or non-functional requirements, or database refactoring, among other situations. When this happens, stored data and application code must be adapted to the new schema, as illustrated in Figure 6.1. Thus, the automation of schema changes is crucial to save effort and to avoid data and application errors. For relational databases, a great deal of research effort has been devoted to address the schema evolution problem, and a number of tools, from prototypes to workbenches, have been built to facilitate the schema change management [HH03, CMDZ13]. Also, schema evolution of object databases has been extensively studied [Li99].

Unlike relational and object-oriented systems, schema evolution for NoSQL databases has received little attention. This is partly because the lack of a standard or specification for NoSQL data models leads to systems of the same NoSQL category having different features, which adds to the challenge of addressing schema evolution for these systems. Another key factor contributing to the difficulty of handling evolution in NoSQL systems is that in order to provide flexibility to adapt to schema changes, most of NoSQL systems are "schema-on-read", that is, the schema declaration is not required prior to storing data (i.e., they are schemaless).

Although the approaches published do handle data model heterogeneity [FGC20, HKS19] they share some limitations: Schema change operations for relationships have not adequately

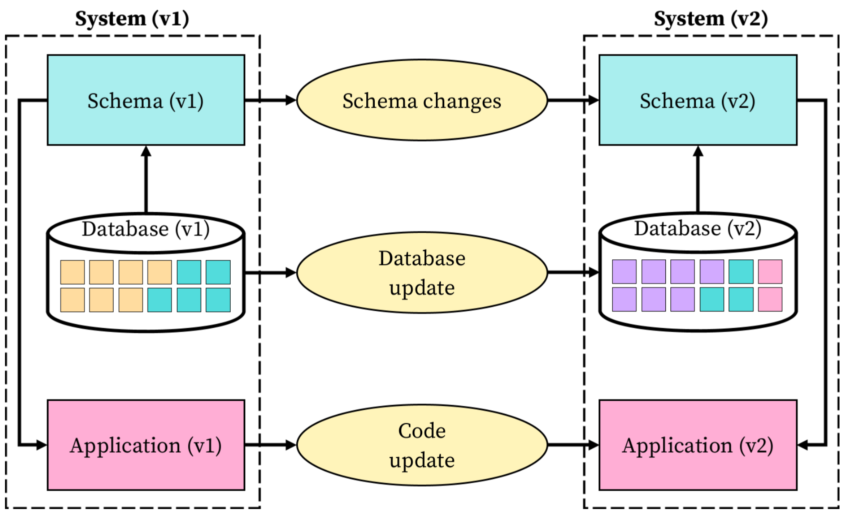

Figure 6.1: Database and code update under schema changes.

been addressed, and the existence of structural variation or relationship types is not considered. Structural variation is possible for schemaless systems since schemas guarantee that data conforms to a particular structure, and relationship types are part of graph schemas.

In this chapter, we show a generic approach aimed to offer automated evolution support for NoSQL and relational systems. It is based on the U-Schema data model and uses Athena schemas as an input. Since U-Schema integrates data models for the four kinds of NoSQL systems and the relational model, it has more expressive power than other proposed generic data models, such as the ones shown in [HEH+94, FGC20]. For example, U-Schema distinguishes between entity types and relationship types, includes aggregation and reference relationships, and allows to represent the structural variation of the schema types. Our approach consists on defining a taxonomy of *Schema Change Operations* (*SCOs*) defined on U-Schema and a domain-specific language (DSL) that implements them. With this language, named *Orion*, database administrators and developers can write scripts that specify sets of SCOs. The Orion engine has been built to update the schema automatically from these scripts, and to translate them to MongoDB (document data model), Cassandra (columnar), and Neo4j (graph) to update the database. On the other hand, code updating is not addressed here, leaving this task to another implements in our toolkit.

This chapter is organized in the following sections: The next section is used to introduce an overview of the approach and the running example used later on. After that, the abstract taxonomy of changes is detailed. Then, the Orion language is described as a concrete implementation of that taxonomy. Next, a formal validation of the taxonomy is shown as well as a measure of performance for Orion. Finally, an evaluation of the Orion language is performed by applying it to three real case scenarios.

6.1 Overview of the Approach

We followed the approach summarized in Figure 6.2. First, we defined the taxonomy of schema changes by sorting the elements susceptible to be changed on the U-Schema metamodel, and then we modeled operations based on those changes. Once the taxonomy has been defined, we provided an implementation of it by creating the Orion metamodel, which conforms the Orion DSL able to express operations. With Orion, developers can create models that specify a set of operations to be executed over a provided schema.

Once Orion models can be created, we implemented the Orion engine. This engine receives an Athena schema and an Orion model, and provides two functions: (i) Updating the schema according to the operations defined, and generating scripts able to update the data in specific databases. These two results contribute to the evolution of the whole system, as was explained at the beginning of the chapter.

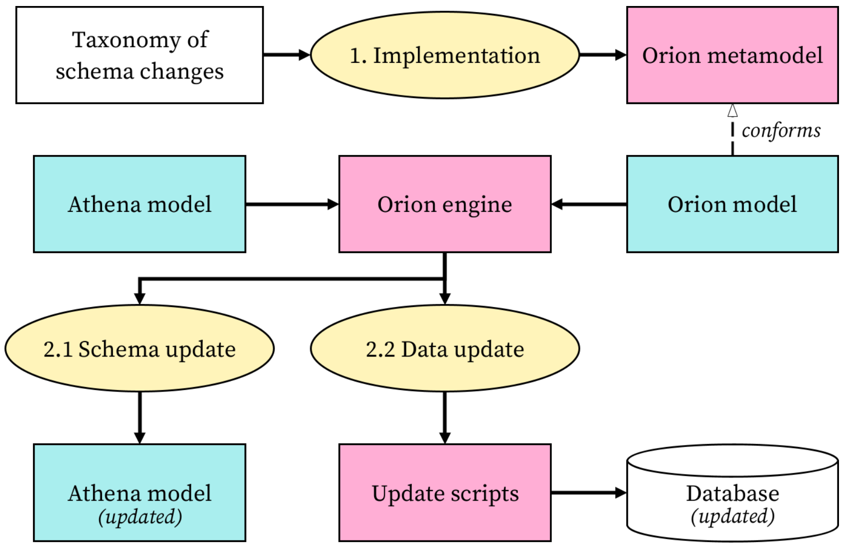

Figure 6.2: The process of design and implementation of Orion.

6.1.1 The *GameTracker* Schema Running Example

To illustrate better the definition of the taxonomy of changes, the Orion language and the usage of the Orion engine, we will introduce an input Athena schema over which Orion operations will be specified. The schema can be seen in Figure 6.3 and is based on a tracker able to follow player progress in videogame achievements.

```
Schema GameTracker:1

Root entity Player {
 Common {
  + id:           Identifier,
  nickname:       String,
  avatar_url:     String  /^https/,
  status:         Integer (0 .. 5),
  last_activity: Timestamp,
  reputation:     Double,
  suspended:      Boolean,
  user_data:      Aggr<Player_Data>&
 }
 Variation 1 {
  experience:     Double,
  hours_played:  Double,
  ach_earned:     Aggr<Ach_Summary>+,
  score:          Integer (0 .. 99999)
 }
 Variation 2 {}
}

Entity Player_Data {
 country_code:   Integer,
 name:           String,
 email:          String /^.+@.+\\.com$/,
 ? birthday:     Timestamp
}
```

```
Entity Ach_Summary
{
 achievement:    Ref<Achievement>&,
 of_the_day:     Boolean,
 ? completed_at: Timestamp,
 ? points:       Integer (0 .. 5000)
}

Root entity Achievement
{
 + id:           Identifier,
 categories:     List<String>,
 description:    String,
 is_active:      Boolean,
 points:         Integer (0 .. 5000)
}

Root entity Game
{
 + id:           Identifier,
 achievements:   Ref<Achievement>+,
 date_from:      Timestamp,
 genres:         List<String>,
 title:          String,
 ? description:  String,
 ? max_players:  Integer
}
```

Figure 6.3: The *GameTracker* Athena schema used as a running example.

In this example, the schema is formed by a set of five *entity types*: `Player`, `Achievement`, `Game`, `Player_Data`, and `Ach_Summary`. The three former are root entity types, that is, their objects are not embedded in any other object, and the two latter are non-root as their instances are embedded into `Player` objects.

`Player` is an entity type able to register participants collecting achievements. It has two structural variations, where the first one (`Variation 1`) represents active players that contains common and four specific features: `experience`, `hours_played`, `ach_earned`, and `score`, and the second variation (`Variation 2`) represents newly registered players and only contains common features (e.g., `nickname`, `status` or `suspended`, among others). The rest of entity types do not show explicit variations but some of them define optional features (“?”), such as `completed_at` and `points` in `Ach_Summary`, or `description` and `max_players` in `Game`.

A feature declaration specifies its name and type. There are four kinds of features: *keys*, *attributes*, *aggregates*, and *references*. *Attributes* and *aggregates* denote the features that hold the values of a database object. For attributes, the type can be either scalar (*Integer*, *String*, *Boolean*, etc) or structured (*Set*, *List*, *Map*, and *Tuple*). In the case of aggregates, the type is a non-root entity type. For example, `Game.title` is an attribute of type String, and `Player.user_data` specifies that `Player_data` objects are embedded in `Player` objects.

The example schema also shows some *keys* and *references*. Each key is denoted by a “+” symbol, and each root entity contains at least one key feature, such as `Game.id` or `Achievement.id`. References are associated to a root type with a cardinality, such as *one to one* or *one to many*. For example, `Achievement` is referenced by `Game.achievements` and `Ach_Summary.achievement`.

6.2 Design of a Taxonomy of Changes for NoSQL and Relational Databases

In schema evolution approaches, the set of changes that can be applied on a particular data model are usually organized in form of a taxonomy [BKKK87, HVS21], where several categories are established depending on the kind of schema element affected by a change. These taxonomies, however, do not include some of the changes proposed here due to not handling a unified schema representation.

We have defined a taxonomy for the U-Schema data model which includes a set of operations richer than those previously proposed. To do so, first we have considered the elements of the data model susceptible to be changed:

- *Entity types* and *relationship types* are susceptible to changes, and they can both be treated indistinctly. Therefore a *schema type* category is created by grouping both concepts.

- Since our data model considers *structural variations*, changes can be defined to be applied to these variations. For example,joining all the variations in a single variation to remove outliers [KSS15].

- Some changes can be applied to *features* independently of their type (i.e., *attributes*, *aggregates*, and *references*). To denote this, these changes are grouped together on the taxonomy under the *features* category.

- Changes have also been defined for *attributes*, *references* and *aggregates*. Some of them are particularly frequent in certain environments, such as converting a particular reference into an aggregate or vice versa [CW19].

- Changes on *keys* have not been explicitly defined, because in U-Schema a *key* is a *logical feature* that is always bound to an attribute, and therefore keys can be created and deleted by means of their attribute, as will be shown.

- Changes on *dataTypes* have also not been explicitly defined. As with *keys*, types can be modified by applying changes to their corresponding *attributes*.

The terminology used to define the taxonomy of changes will be introduced next. Let T be the set of schema types belonging to a schema, and let E be the set of entity types $E = \{E_i\}, i = 1 \ldots n$, $T = E$ in the case of aggregate-based stores, while $T = E \cup R$, where $R = \{R_i\}, i = 1 \ldots m$ denotes the set of relationship types, in the case of graph stores. 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 6.1. In a similar way to [CMDZ13], we have added operations taking into account a compromise between atomicity, usability, and reversibility. In the case of changes affecting variations, usefulness and atomicity have prevailed on 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^+ denotes a SCO that carries an *additive change*.
  Example: *Add Schema Type*

- C^- is used for SCOs that carry a *subtractive change*.
  Example: *Delete Feature*

- $C^{+,-}$ denotes a SCO in which there is a gain of information for a specific element and a loss of information for other specific element.
  Example: *Move Feature*

- $C^=$ means there is no change in the schema information.
  Example: *Rename Schema Type*.

- $C^{+|-}$ is a change that adds or subtracts information, depending on the specific operation parameters.
  Example: *Cast Attribute* to boolean.

As noted in [CMDZ13], a schema change operation can be considered 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'. In this paper, the schema operation semantics is defined in form of pre and postconditions, which appear in the second and third column. Note that the postconditions only specify the changes on the schema and not the changes on the database, because these depend on the concrete data model. Since the operations semantics would be expressed very similarly to specifying the database change, this semantics is not included here, but we have added a comment to the postcondition of the *Adapt* operation to show that its effect is different to the *Delvar* operation. As Table 6.1 shows, taxonomy operations are classified in six categories corresponding to the U-Schema elements previously listed: *Schema Types*, *Structural Variations*, *Features*, *Attributes*, *References* and *Aggregates*.

For *Schema Type*, the basic, atomic operations *Add*, *Delete*, and *Rename* are provided, along with three complex operations to create new schema types: (i) 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. (ii) 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. And (iii) 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.

		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 $n \notin T.names$	$t \in T \land t_1 = T.new \land t_1.name = n \land t_1.features = fs$
Split*	$C^{=}$	Given a schema type $t \in T$, two sets of features $fs_1 \subset F^t \land fs_2 \subset F^t$ and two string values $n_1, n_2 \notin T.names$	$t \notin T \land t_1, t_2 = T.new \land t_1.name = n_1 \land t_1.features = fs_1 \land t_2.name = n_2 \land t_2.features = fs_2$
Merge	$C^{=}$	Given two schema types $t_1, t_2 \in T$ and a string value $n \notin T.names$	$t_1, t_2 \notin T \land t = T.new \land t.name = n \land t.features = t_1.features \cup t_2.features$
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 $v^t_1, v^t_2 \in V^t$	$v^t_1 \notin V^t$ *(Data is migrated from v^t_1 to v^t_2)*
Union	C^{+}	Given a schema type $t \in T \land V^t \neq \{\}$	$V^t = \{v_m\} \land v_m.features = \cup^n_{i=1} v^t_i.features$
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 $f \in F^{t_1} \land f \notin F^{t_2}$	$f \in F^{t_1} \land f \in F^{t_2}$
Move*	$C^{+,-}$	Given two schema types $t_1, t_2 \in T$ and a feature $f \in F^{t_1} \land f \notin F^{t_2}$	$f \notin F^{t_1} \land f \in F^{t_2}$
Nest	$C^{+,-}$	Given an entity type $e_1 \in E$, a feature $f \in F^{e_1}$, and an aggregate $ag \in F^{e_1} \land ag.type = e_2 \land f \notin F^{e_2}$	$f \notin F^{e_1} \land f \in F^{e_2}$
Unnest	$C^{-,+}$	Given an entity type $e_1 \in E$, an aggregate $ag \in F^{e_1} \land ag.type = e_2$, and a feature $f \notin F^{e_1} \land f \in F^{e_2}$	$f \in F^{e_1} \land f \notin F^{e_2}$
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	$C^{+\vert-}$	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 $at \in F^e \land at.key = False$	$at.key = True$
Demote	$C^{=}$	Given an entity type $e \in E$ and an attribute $at \in F^e \land at.key = True$	$at.key = False$
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	$C^{+\vert-}$	Given a schema type $t \in T$, a reference $rf \in F^t$, and a scalar type st	$rf.type = st$
Mult	$C^{+\vert-}$	Given a schema type $t \in T$, a reference $rf \in F^t$, and a tuple $(l, u) \in \{(0,1), (1,1), (0,-1), (1,-1)\}$	$rf.lowerBound = l \land rf.upperBound = u$
Morph	$C^{=}$	Given a schema type $t \in T$ and a reference $rf \in F^t$, let ag be a new aggregate, $ag \notin F^t$	$rf \notin F^t \land ag \in F^t \land ag.name = rf.name \land ag.type = rf.type$
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	$C^{+\vert-}$	Given an entity type $e \in E$, an aggregate $ag \in F^e$ and a tuple $(l, u) \in \{(0,1), (1,1), (0,-1), (1,-1)\}$	$ag.lowerBound = l \land ag.upperBound = u$
Morph	$C^{=}$	Given an entity type $e \in E$ and an aggregate $ag \in F^e$, let rf be a new reference, $rf \notin F^e$	$ag \notin F^e \land rf \in F^e \land rf.name = ag.name \land rf.type = ag.type$

Table 6.1: The taxonomy of schema changes for NoSQL and relational databases.

Three operations have been defined to manipulate *Structural Variations*: (i) The *Delvar* operation deletes a given variation from a schema type, (ii) *Adapt* deletes a given variation but also migrates data belonging to the deleted variation to a new variation, and (iii) *Union* joins all the variations of a schema type into a single one. Since *Delvar* and *Adapt* cause the same changes on the schema (a variation is deleted), a comment has been added in the *Adapt* postcondition to indicate the effect on the database.

The *Feature* category groups basic *Delete* and *Rename* operations. It also includes operations to: (i) *Copy* a feature from a schema type to another one, maintaining the feature copied in the original schema type; (ii) *Move* a feature from a schema type to another, similar to *Copy* but deleting the feature in the original schema type after copying; (iii) *Nest* to move a feature to an aggregate; and (iv) *Unnest* to move a feature from an aggregate to the embedding schema type.

The *Attribute* category includes operations to: (i) *Add* a new attribute, (ii) *Cast* its type to a new type, (iii) *Promote* an attribute to key, and (iv) *Demote* an attribute from a key. As stated before, there is not a *Key* category because keys can be created and deleted by means of the attribute operations *Add*, *Promote*, and *Demote*.

The *Reference* category includes: (i) The *Add* operation, (ii) the *Cast* operation (both similar to the ones defined in the *Attribute* category), (iii) *Mult* to change the multiplicity, and (iv) *Morph* to transform a reference to an aggregate.

Finally the *Aggregate* category includes operations to: (i) *Add* an aggregate to a schema type, (ii) *Mult* to change an aggregate multiplicity, and (iii) *Morph* to swap an aggregate for a reference.

All the listed SCOs, except *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 (*Movie* 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 been added to the taxonomy because they are recurrent operations in refactoring scenarios, and other approaches have considered them.

6.3 Implementation of the Orion Language

Orion is the language created to implement the taxonomy of changes defined on U-Schema. With Orion, developers can declare and execute SCOs in a system-independent way. The Orion metamodel specifies the Orion grammar as an Ecore model [SBPM09]. Here, the

Orion notation will be explained by using the running example and, finally, semantics will be illustrated by indicating the generated code.

6.3.1 Concrete Syntax

The Orion notation allows to express SCOs like commands of a command-line language. Therefore, the syntax of Orion is very simple as illustrated in Figure 6.4, where an excerpt of its EBNF grammar is shown. 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 whole Orion metamodel can be found in Appendix 10.4.1.

The Orion syntax has been defined to let operations be written as concise as possible. The language has been provided with some mechanisms to ease the definitions of those operations, being some of them listed below:

- Operations can be applied over all schema types by using the "*" wildcard.
  Example: `DELETE *::name`
- Operations can define a list of parameters.
  Example: `DELETE Sales::types, isActive, description`
- Operations can be applied to specific variations of a schema type.
  Example: `RENAME *(v1,v3)::phone TO newPhone`
- Differences between aggregate-based systems and graph systems are expressed through optional parameters in some operation commands.
  Example: `ADD REF` requires to indicate a target entity type and a *join* condition, and has two optional parameters: the primitive type of the references values in the case of an aggregated-based store, and a set of attributes for a graph store.
- Operations can also be grouped in `EBlock` structures. This serves two purposes. In databases that allow transactions each `EBlock` will be processed as a single transaction. Also, during code generation, each `EBlock` will be generated in a single file independently from the rest.

```
⟨EntityRenameOp⟩ ::= ‘RENAME’ ‘ENTITY’ ⟨SchemaTypeRenameSpec⟩
⟨SchemaTypeRenameSpec⟩ ::= ID ‘TO’ ID
⟨EntityAdaptOp⟩ ::= ‘ADAPT’ ‘ENTITY’ ⟨SchemaTypeAdaptSpec⟩
⟨SchemaTypeAdaptSpec⟩ ::= ID ‘::’ ‘v’? ⟨VariationIdSpec⟩ ‘TO’ ‘v’? ⟨VariationIdSpec⟩
⟨FeatureDeleteOp⟩ ::= ‘DELETE’ ⟨FeatureDeleteSpec⟩
⟨FeatureDeleteSpec⟩ ::= ⟨MultipleFeatureSelector⟩
⟨FeatureRenameOp⟩ ::= ‘RENAME’ ⟨FeatureRenameSpec⟩
⟨FeatureRenameSpec⟩ ::= ⟨SingleFeatureSelector⟩ ‘TO’ ⟨QualifiedName⟩
⟨FeatureNestOp⟩ ::= ‘NEST’ ⟨FeatureNestSpec⟩
⟨FeatureNestSpec⟩ ::= ⟨MultipleFeatureSelector⟩ ‘TO’ ⟨QualifiedName⟩
⟨AttributeCastOp⟩ ::= ‘CAST’ ‘ATTR’ ⟨AttributeOrReferenceCastSpec⟩
⟨AttributeOrReferenceCastSpec⟩ ::= ⟨MultipleFeatureSelector⟩ ‘TO’ ⟨SinglePrimitiveType⟩
⟨ReferenceAddOp⟩ ::= ‘ADD’ ‘REF’ ⟨ReferenceAddSpec⟩
⟨ReferenceAddSpec⟩ ::= ‘?’? ⟨SingleFeatureSelector⟩ ‘:’ ( ⟨SinglePrimitiveType⟩ ( ‘(’ ⟨DefaultValue⟩ ‘)’ )?
                     |  ‘feat’ ‘by’ ID ( ‘(’ ‘v’? ⟨VariationIdSpec⟩ ( ‘,’ ‘v’? VariationIdSpec )* ‘)’ )?
                     |  ‘{’ ( ⟨SimpleDataFeature⟩ ( ’,’ ⟨SimpleDataFeature⟩ )* )? ‘}’ ) ( ‘?’ | ‘&’ | ‘*’ | ‘+’ ) ‘TO’ ID
                        ( ‘WHERE’ ⟨ConditionDecl⟩ )?
⟨SingleFeatureSelector⟩ ::= ( ID ( ‘(’ ‘v’? ⟨VariationIdSpec⟩ ( ‘,’ ‘v’? ⟨VariationIdSpec⟩ )* ‘)’ )?
                     |  ‘*’ ) ‘::’ ⟨QualifiedName⟩
⟨MultipleFeatureSelector⟩ ::= ( ID ( ‘(’ ‘v’? ⟨VariationIdSpec⟩ ( ‘,’ ‘v’? ⟨VariationIdSpec⟩ )* ‘)’ )?
                     |  ‘*’ ) ‘::’ ⟨QualifiedName⟩ ( ‘,’ ⟨QualifiedName⟩ )*
⟨ConditionDecl⟩ ::= ⟨QualifiedName⟩ ‘=’ ⟨QualifiedName⟩
⟨QualifiedName⟩ ::= ID ( ‘.’ ID )*
⟨VariationIdSpec⟩ ::= INT
```

Figure 6.4: EBNF excerpt of the Orion language.

Figure 6.5 shows an Orion script that applies changes on the *GameTracker* schema of Figure 6.3. An Orion script starts with a `USING` statement that indicates the schema on which the changes are applied. This declaration allows runtime checking of the validity of each operation on the current schema. Note that the schema has to be updated after the execution of each operation of the script, so that the checking can be correctly performed. The operations are, therefore, sequentially executed.

The aforementioned script of Figure 6.5 shows changes on several entity types of the schema in a refactoring scenario, illustrating most of the different changes in the taxonomy. To ease its understanding, operations have been separated according to their categories. First, we used `Add` to create a new root entity type, `Guild`, with four initial attributes. Then we used `Rename` to change the name of `Player` to `GamePlayer`. Please note that from now on, operations will need to refer to this entity type by its new name, as in the following `Adapt` operation used to reduce the number of variations in `GamePlayer` to a single variation, updating the data belonging to the variation dissapearing.

Then, we `Deleted` a single feature and `Renamed` another one. After that, we `Nested` and `Unnested` some features to/from `GamePlayer` and its `user_data` aggregate.

After that, we applied some *Attribute* operations by using `Add` to create two new features in `Player_Data` (`surname` and `homepage`), we used `Cast` to change two features to a Double type for every entity type (`score` and `points`), another `Cast` for `Ach_Summary.is_completed` to Boolean, and a final `Promote` in `Guild.code` to add a feature to an entity type key.

Finally we applied some *Aggregate* operations: We `Added` a new aggregate for `Guild` with five attributes creating a new non-root entity type named `Realm`, and a new non-root entity type named `Address` embedded by `Player_Data`. After that, we changed the multiplicity (`Mult` operation) of this last aggregate from *one to one* to *one to many*, and then we used `Morph` to change the embedded `GamePlayer.user_data` to a reference, changing the corresponding non-root entity type to a root entity type and renaming the feature to `user_private_data`.

6.3.2 Semantics: Schema and Data Update

The Orion semantics can be expressed through the changes that each operation causes in the existing schema and stored data. In our case, these semantics are given by the Orion engine that modifies the U-Schema model and translates Orion scripts into database operations for updating data to the modified schema. The Orion engine has a component for each of these two tasks: The *Schema Updater* and the *Data Updater*, as shown in Figure 6.6.

The *Schema Updater* takes an Athena model (i.e., a schema) and an Orion model as input, and outputs the updated schema. To do so, the input schema is taken as a starting point, and each Orion operation is sequentially applied to that schema, with the semantics shown in Section 6.2. This process is implemented as a *model to model* transformation which assures platform-independence. It also has to be executed along with the *Data Updater*, to assure

```
GameTracker_ops operations

USING GameTracker:1

// Schema Type operations
ADD    ENTITY Guild: { +id: Identifier, code: String, name: String, num_players: Number }
RENAME ENTITY Player TO GamePlayer
ADAPT  ENTITY GamePlayer::v2 TO v1

// Feature operations
DELETE        Achievement::is_active
RENAME        Ach_Summary::completed_at TO is_completed
NEST          GamePlayer::reputation, suspended TO Player_Data
UNNEST        GamePlayer::user_data.email

// Attribute operations
ADD ATTR      Player_Data::surname: String
ADD ATTR      Player_Data::homepage: String
CAST ATTR     *::score, points TO Double
CAST ATTR     Ach_Summary::is_completed TO Boolean
PROMOTE ATTR  Guild::code

// Aggregate operations
ADD AGGR      Guild::realm: {
  num_guilds: Number, max_guilds: Number, num_players: Number, max_players: Number, type: String
}& AS Realm
ADD AGGR      Player_Data::address: { country: String, city: String }& AS Address
MULT AGGR     Player_Data::address TO +
MORPH AGGR    GamePlayer::user_data TO user_private_data
```

Figure 6.5: Operations applied to the *GameTracker* schema using Orion.

the correctness of the data to be adapted, or can be executed as a standalone process to study schema evolution.

While we implemented a single *Schema Updater*, as it is independent of any database and works at a logical level, the *Data Updater* is bound to a specific database, so a different data updater 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 document, columnar, and graph NoSQL data models, and we support the three most widely used NoSQL stores on their own paradigm.

A data updater receives an Athena and Orion model as input, and generates the piece of database-specific code that applies the necessary changes on the database according to the operations specified in the Orion script. Therefore, this process consists of a *model to text* transformation.

In the case of MongoDB, the data updater generates native MongoDB commands and stores them in a Javascript file. Since MongoDB does not have to declare an explicit schema,

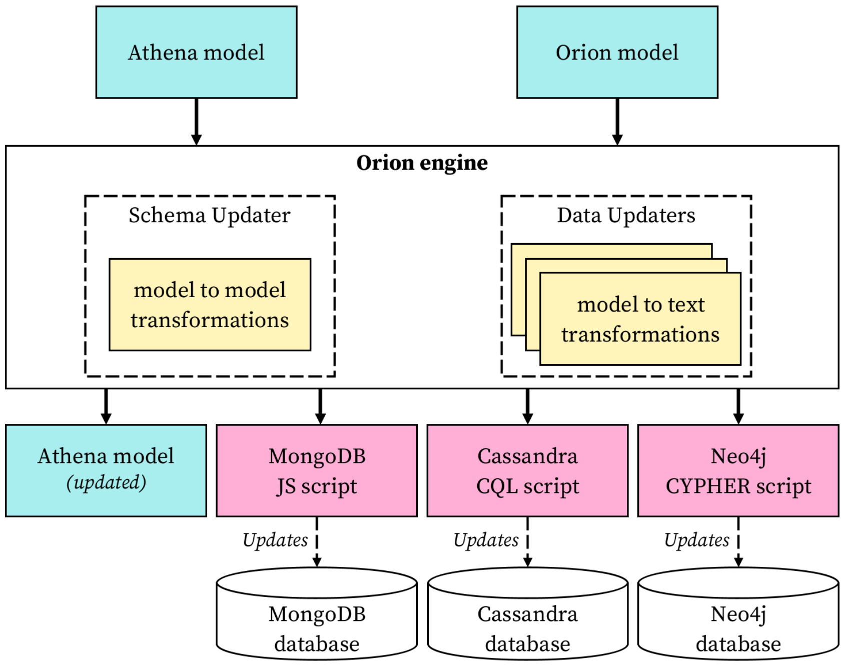

Figure 6.6: The Orion engine developed to handle schema and data updates.

in order to apply changes, documents belonging to the desired entity type have to be selected. 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* instruction. Some complex operations, however, do not allow that optimization, and they must be executed in their own *aggregation* pipeline. An example of this can be seen in Figure 6.8, where operations `Nest` and `Unnest` are applied on the same entity type (`GamePlayer`) and therefore can be stacked, then a `Cast` over all entity types is applied, and finally an `Add` aggregation and a `Mult` aggregation are also stacked together in a single *bulk write*.

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.

In Figure 6.9 some generated operations for Cassandra are shown. Here, the `Rename entity` instruction stores the table on an external file before recreating that table with a new name. However other instructions such as `Delete feature` are performed directly. Finally, a `Promote attribute` operation also relies on storing the table elsewhere but this time

```
Schema GameTracker:2

Root entity GamePlayer {
 Common {
  + id:               Identifier,
  nickname:           String,
  avatar_url:         String /^https/,
  status:             Integer ( 0 .. 5 ),
  last_activity:      Timestamp,
  user_private_data:  Ref<Player_Data>&,
  player_Data:        Aggr<Player_Data>&,
  email:              String /^.+@.+\\.com$/
 }
 Variation 1 {
  experience:         Double,
  hours_played:       Double,
  ach_earned:         Aggr<Ach_Summary>+,
  score:              Double
 }
}
Root entity Player_Data {
  + id:               Identifier,
  country_code:       Integer,
  name:               String,
  ? birthday:         Timestamp,
  reputation:         Double,
  suspended:          Boolean,
  surname:            String,
  homepage:           String,
  address:            Aggr<Address>+
}
Entity Ach_Summary {
  achievement:        Ref<Achievement>&,
  of_the_day:         Boolean,
  ? is_completed:     Boolean,
  ? points:           Double
}
```

```
Root entity Achievement {
 + id:            Identifier,
 categories:      List<String>,
 description:     String,
 points:          Double
}

Root entity Game {
 + id:            Identifier,
 achievements:    Ref<Achievement>+,
 date_from:       Timestamp,
 genres:          List<String>,
 title:           String,
 ? description:   String,
 ? max_players:   Integer
}

Root entity Guild {
 + id:            Identifier,
 + code:          String,
 name:            String,
 num_players:     Number,
 realm:           Aggr<Realm>&
}

Entity Realm {
 num_guilds:      Number,
 max_guilds:      Number,
 num_players:     Number,
 max_players:     Number,
 type:            String
}

Entity Address {
 city:            String,
 country:         String
}
```

Figure 6.7: The updated *GameTracker* Athena schema after applying the SCOs.

columns need to be stored in an appropriate order to assure the primary key is created successfully.

The Neo4j data updater uses the Cypher language to generate code for updating data. 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.

Figure 6.10 shows how some operations would be generated for Neo4j. First, a `Rename`

```
GameTracker.GamePlayer.bulkWrite([
 // NEST GamePlayer::reputation, suspended TO Player_Data
 {updateMany: {filter: {}, update: {$rename: { "reputation": "user_data.reputation",
                                               "suspended":  "user_data.suspended"}}}},
 // UNNEST GamePlayer::user_data.email
 {updateMany: {filter: {}, update: {$rename: { "user_data.email": "email"}}}}
])

// CAST ATTR *::score, points TO Double
GameTracker.getCollectionNames().forEach(function(collName) {
 GameTracker[collName].updateMany({}, [
   {$set: { "score" : { $convert: { input: "$score",  to: 1 }},
            "points": { $convert: { input: "$points", to: 1 }}
   }}])
})

GameTracker.Player_Data.bulkWrite([
 // ADD AGGR Player_Data::address: { country: String, city: String }& AS Address
 {updateMany: {filter:{}, update: [{$addFields: {"address": {"country": "", "city": ""}}}], upsert: true}},

 // MULT AGGR Player_Data::address TO +
 {updateMany: {filter: {}, update: [{$set: { "address": [ "$address" ] }}]}}
])
```

Figure 6.8: An example of operations generated by the Orion engine for MongoDB.

```
// RENAME ENTITY Player TO GamePlayer
COPY player TO 'tmp.csv' WITH HEADER = TRUE;
DROP TABLE player;
CREATE TABLE gameplayer (
 id            uuid,
 ach_earned    list<frozen<ach_summary>>,
 avatar_url    text,
 experience    double,
 hours_played  double,
 last_activity timestamp,
 nickname      text,
 reputation    double,
 score         int,
 status        int,
 suspended     boolean,
 user_data     player_data,
 PRIMARY KEY   (id)
);
COPY gameplayer FROM 'tmp.csv' WITH HEADER = TRUE;
```

```
// DELETE Achievement::is_active
ALTER TABLE achievement DROP ( is_active );

// PROMOTE ATTR Guild::code
COPY guild (
 id,
 code,
 name,
 num_players
) TO 'tmp.csv' WITH HEADER = TRUE;
DROP TABLE guild;
CREATE TABLE guild (
 id          uuid,
 code        text,
 name        text,
 num_players varint,
 PRIMARY KEY (id, code)
);
COPY guild FROM 'tmp.csv' WITH HEADER = TRUE;
```

Figure 6.9: CQL code generated by the Orion engine for some operations in Cassandra.

`entity` and a `Adapt entity` are applied. Since in Neo4j aggregations are not handled, the field `GamePlayer.ach_earned` is ignored in Cypher. Then, a `Delete feature` operation is applied to each node matching a certain label containing the name of the entity type being modified. Finally, a `Cast attribute` operation is applied to each node independently of its label (which indicates its entity type) and also to each relationship in the graph.

```
// RENAME ENTITY Player TO GamePlayer
MATCH (x: Player)
REMOVE x: Player
SET x: GamePlayer
;

// ADAPT ENTITY GamePlayer::2 TO 1
MATCH (x: GamePlayer) WHERE
  x.experience     IS NULL AND
  x.hours_played   IS NULL AND
  x.ach_earned     IS NULL AND
  x.score          IS NULL
SET x.experience   = 0.0,
    x.hours_played = 0.0,
    x.score        = 0
;
```

```
// DELETE Achievement::is_active
MATCH (x: Achievement)
REMOVE x.is_active
;

// CAST ATTR *::score, points TO Double
MATCH (x)
SET x.score  = toFloat(x.score),
    x.points = toFloat(x.points)
;

MATCH ()-[x]->()
SET x.score  = toFloat(x.score),
    x.points = toFloat(x.points)
;
```

Figure 6.10: Cypher code generated by the Orion engine for some operations in Neo4j.

For each Orion operation, the instructions issued by the data updater for particular supported databases are summed up in Table 6.2. There, some keywords give insight of how each operation is implemented, and also which operations cannot be executed in a particular database. For example, since in Neo4j *aggregations* are not handled, operations regarding aggregations are not supported.

6.4 Validation

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

6.4.1 Validating the Taxonomy

Alloy 5[*] has been used to implement 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 implemented for each operation. Each step will be detailed below.

[*]https://alloytools.org/.

The U-Schema metamodel has been modeled in Alloy by using *signatures*. In Figure 6.11 an excerpt of U-Schema is shown, consisting 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 implementing 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. Once the U-Schema specification is defined, Alloy is capable of searching for scenarios that fulfill all the provided restrictions.

```
some sig USchema
{
 entities: set EntityType,
 relationships: set RelationshipType
}
{
 some entities or some relationships
 entities.parents in entities
 relationships.parents in relationships

 some e: entities           | e.root = True
 all e1, e2: entities       | e1.name = e2.name => e1 = e2
 all r1, r2: relationships | r1.name = r2.name => r1 = r2
 all ref:  entities.variations.features + relationships.variations.features
                            | ref.refsTo in entities
 all aggr: entities.variations.features + relationships.variations.features
                            | aggr.aggregates in entities.variations
}
```

Figure 6.11: An excerpt of the U-Schema definition in Alloy.

Figure 6.12 shows a scenario found by Alloy in which a U-Schema instance is composed of two entity types, *EntityType0* and *EntityType1*, both of them *roots*. The first entity type contains a structural variation with two features, *Attribute0* and *Attribute2*, named *FeatureName1* and *FeatureName0*, and both of them of a *PrimitiveType*. The second entity type contains also a structural variation but with only one attribute, *Attribute1* with name *FeatureName1*, also of a *PrimitiveType*.

The next step is to model the change operations in the taxonomy as Alloy operations, by using *predicates* that may be applied over 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 Figure 6.13, the *Rename Entity* operation is implemented. Its precondition is declared in the same way as it was defined in Table 6.1, `newName not in schemaI.entities.name`, and

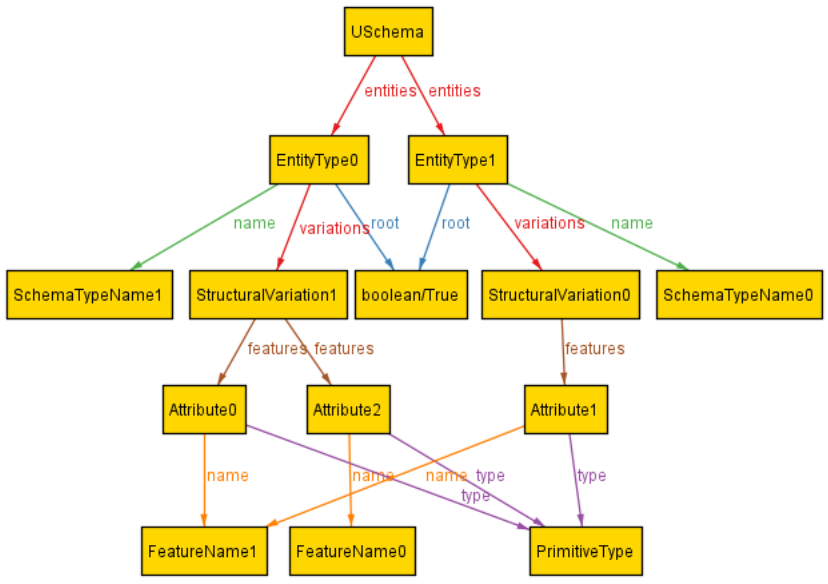

Figure 6.12: A U-Schema instance found by Alloy.

```
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
}
```

Figure 6.13: Alloy definition for the *Rename Entity* operation.

then several statements are defined to be fulfilled by the output schema. When this operation is executed in Alloy to search scenarios in which it is sucessfully applied, a scenario is found, which is shown in Figure 6.14. As can be seen, the input schema only has an entity type whose *name* changes in the output schema but its *root* and *variations* properties are the same.

In order to check the operation for contradictions, we use a *Check* statement. This check is an implication in which if the designed operation is applied to a valid scenario, then some conditions need to be assured, i.e., the operation postconditions. In Figure 6.15 the check statement for the *Rename Entity* operation is shown. Here, it is checked that applying the

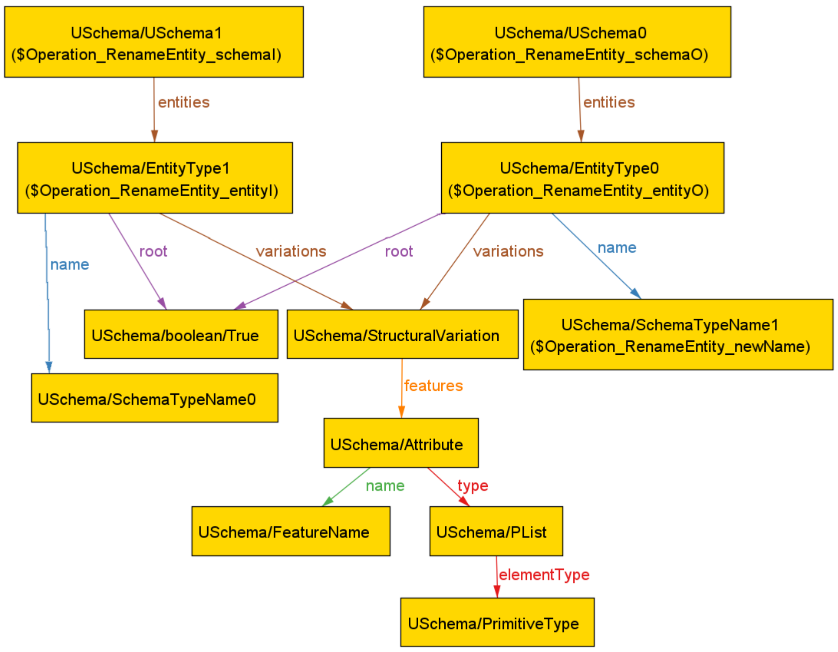
USchema/USchema1 ($Operation_RenameEntity_schemaI)
USchema/USchema0 ($Operation_RenameEntity_schemaO)
entities
entities
USchema/EntityType1 ($Operation_RenameEntity_entityI)
USchema/EntityType0 ($Operation_RenameEntity_entityO)
name
root
variations
root
variations
name
USchema/boolean/True
USchema/StructuralVariation
USchema/SchemaTypeName1 ($Operation_RenameEntity_newName)
USchema/SchemaTypeName0
features
USchema/Attribute
name
type
USchema/FeatureName
USchema/PList
elementType
USchema/PrimitiveType

Figure 6.14: One of the *Rename Entity* scenarios found.

operation to a given scenario, the postcondition, *entityO.name = newName* will be fulfilled. Applying this check means that Alloy tries to find a scenario in which the operation is applied but the implications (i.e., postconditions) are not true (counterexample), which ends with Alloy not finding scenarios.

For each operation some invariants are also checked: (i) For each schema type operation, we check that the involved schemas, without the affected entities, remain unchanged, and (ii) for each feature operation, we check the first invariant and also that the involved entity or entities, without the affected features, also remain unchanged. Applying this check means that Alloy will look for a scenario in which the operation is applied but the implications are not true. In our case, Alloy was not able to find any contradictory scenario.

Therefore, we 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 including invariants in the metamodel.

```
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 check: Everything else remains the same.
    and CheckSchemaEquality[schemaI.entities - entityI, schemaO.entities - entityO,
                            schemaI.relationships,      schemaO.relationships]
} for 10
```

Figure 6.15: Postcondition checking of the *Rename Entity* operation.

6.4.2 Measuring 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. Then, by using the Deimos language described in Chapter 7, we generated a dataset of 150,000 instances per entity type conforming to that schema. After that, we defined a process to inject this dataset into a MongoDB, Cassandra, and Neo4j database. Then, 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 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 and results are more reliable. In the third step, op_{mod} is applied on 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 consisted 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 6.2 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 much more costly. As was explained in Section 6.2, 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 6.3, 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`, 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.

6.5 Evaluation of the Proposed Language

We evaluated Orion and its engine by applying it to three different case studies. In these case studies we will apply refactoring processes to databases, consisting on a succession of SCOs defined in Orion with the purpose of updating schema and data. Database refactoring is an activity aimed to improve the database design and performance without changing its semantics [AS06]. A refactoring is a small change on the schema, and several refactorings can be applied to achieve a determined improvement.

	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	–	–	–	–

Table 6.2: Implementation and performance of the SCOs.

6.5.1 A StackOverflow Refactoring in Neo4j

In this case study, Orion has been used to apply a refactoring to a Neo4j database that imported the StackOverflow dataset [Sta14]. We injected the dataset into Neo4j but changed it slightly during injection to take advantage of relationship types. In StackOverflow, the `Comment` entity type references its `User` and a `Post` in a *one to one* relation, so we injected this database transforming the `Comment` entity type as a relationship type named `Rel_Comments` between `Users` and `Posts`. After injecting the dataset into Neo4j, we inferred the schema, obtained the corresponding U-Schema and transformed it to an Athena schema.

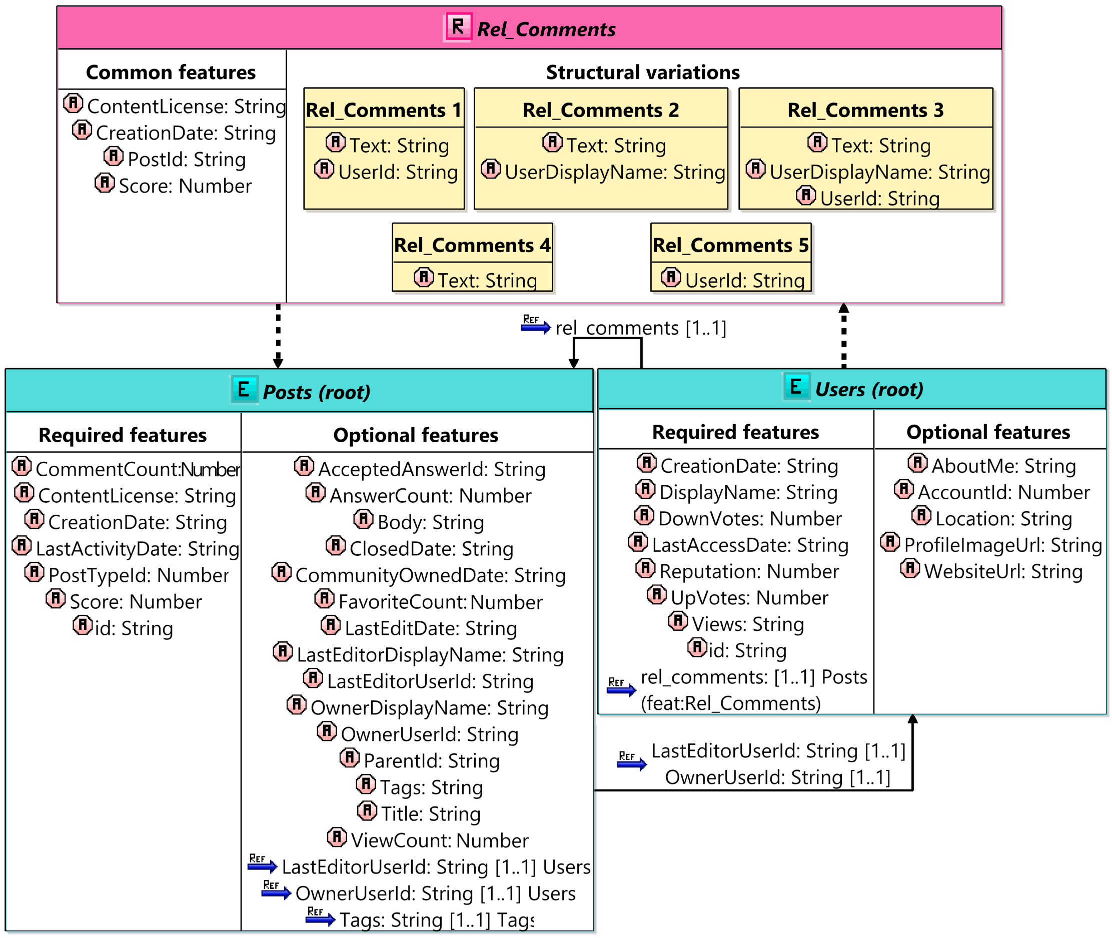

Figure 6.16: Excerpt of the *StackOverflow* schema injected in Neo4j.

Figure 6.16 shows an excerpt of the inferred schema with two of the seven entity types discovered and the already mentioned new relationship type. Here, `Posts` and `Users` are shown as *union entity types* listing their required and optional features, because they contain too many variations to be visualized. `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 that some attributes in `Rel_Comments` turned obsolete and 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.

```
StackOverflow_neo4j_ops operations

Using stackoverflow_neo4j: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 6.17: Operations applied to the *StackOverflow* schema and data in Neo4j.

The Orion script to refactor the StackOverflow schema is shown in Figure 6.17. 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` features, 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_Comment` 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 itself 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 Athena schema, the Orion engine generates the updated schema and the Cypher code script to execute the changes on the data. In Figure 6.18 an excerpt of the generated code can be seen. First, some `Add` operations are shown, which make use of native Cypher functions to create default values (such as in *datetime()*). Then the `Cast` operation is applied, which uses the function *toFloat* to cast a feature to a type. After that, a `Delete` operation is applied to all entity types and also relationship types, and finally a `Rename Relationship` is implemented by making use of the Neo4j APOC library.[†]

```
// ADD ATTR Rel_Comments::LastEditDate: Timestamp
MATCH ()-[x: Rel_Comments]->()
SET x.LastEditDate = datetime()
;
// ADD ATTR Rel_Comments::KarmaCount: Number
MATCH ()-[x: Rel_Comments]->()
SET x.KarmaCount = 0
;

// CAST ATTR Rel_Comments::Score TO Double
MATCH ()-[x: Rel_Comments]->()
SET x.Score = toFloat(x.Score)
;
```

```
// DELETE *::PostId, UserId
MATCH (x)
REMOVE x.PostId, x.UserId
;
MATCH ()-[x]->()
REMOVE x.PostId, x.UserId
;

// RENAME RELATIONSHIP Rel_Comments TO comments
MATCH ()-[x: Rel_Comments]->()
CALL apoc.refactor.setType(x, 'comments')
YIELD input, output RETURN *
;
```

Figure 6.18: Cypher code generated by the Orion engine for some operations in Neo4j.

6.5.2 A StackOverflow Refactoring in MongoDB

In this second case study, as an alternative to the first refactoring case the StackOverflow dataset [Sta14] was injected into a MongoDB database. We did not apply any change during the injection this time. Unlike the previous case, we could not inject an entity type as a relationship, but instead planned to take advantage of constructs typical to document databases, such as aggregations, since we already knew that this dataset only relies on references between entity types.

Once the dataset was injected, we obtained the U-Schema model and the equivalent Athena schema. As is shown in Figure 6.19, this time we focused on `Posts` and `Users`, their relationships with `Postlinks` and `Badges`, and how a post does not store its links, it is a postlink that stores to which post it is associated, and the same case applies to a user and its badges.

A useful refactoring case would consist on embedding, for each post, its corresponding postlinks, and for each user its corresponding badges. To do so we implemented the Orion script shown in Figure 6.20. In this script, first, the same `CAST` and `MULT` operations are applied as in the previous refactoring case, showing that these operations can be applied to both databases. After that, we used some `NEST` operations on `Posts` and `Users` to create new ag-

[†] https://neo4j.com/developer/neo4j-apoc/.

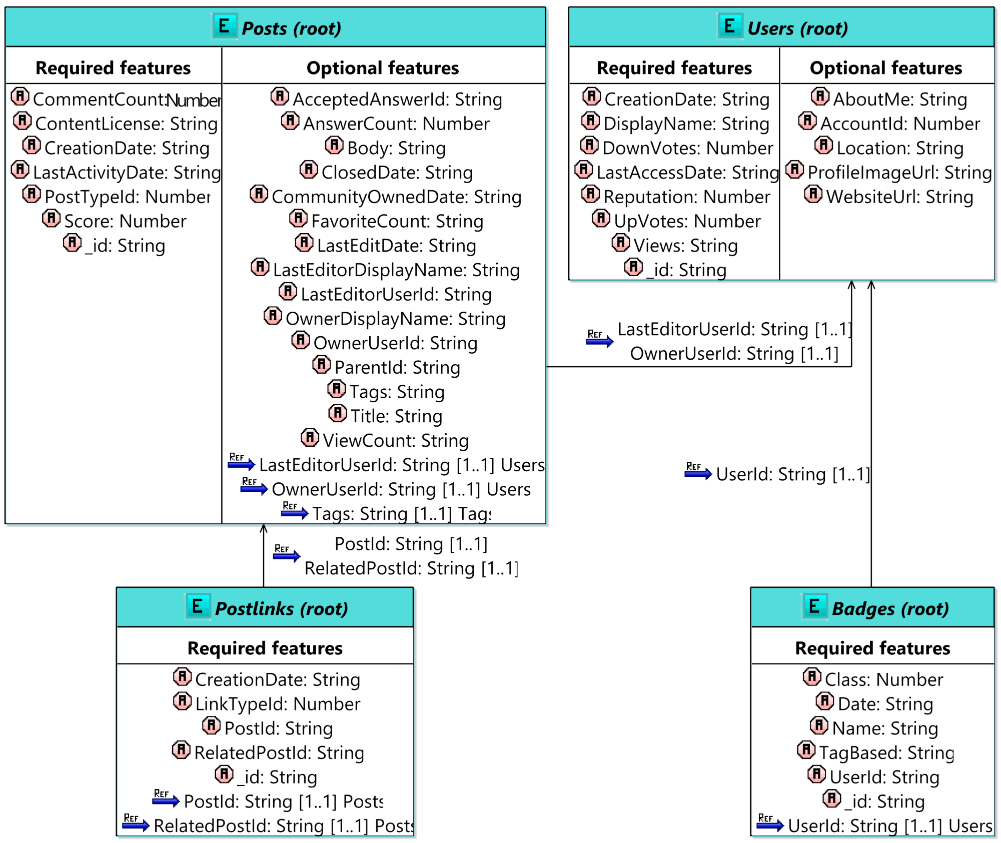

Figure 6.19: Excerpt of the *StackOverflow* schema injected in MongoDB.

gregated objects (`PostMetadata` and `UserMetadata`) where similar features are stored. We also embedded `Postlinks` into `Posts` and `Badges` into `Users` by using `ADD REF` and `MORPH`. This is not a trivial operation since there is no direct reference between a post and its postlinks. Instead, such reference needs to be created by providing a join condition that uses the reference between a postlink and its post. Once this reference is created, it can be swapped for an aggregation, removing the `_id` field from the embedded object and the `Postlinks` collection. The same principle applies to `Users` and `Badges`.

An excerpt of the JS code script generated by the Orion engine is displayed in Figure 6.21. Here, the operations that aggregate `Postlinks` into `Posts` are displayed: (i) An inverse reference between the two entity types is created by using a *lookup* operation. A temporal index is created beforehand to accelerate the process and it is dropped afterwards. (ii) Then this reference is swapped into an aggregation by using another *lookup* operation. The final instructions remove the newly created reference, since in Orion the `rmId` option was specified, and then

```
StackOverflow_mongodb_ops operations

USING stackoverflow_mongodb:1

CAST ATTR *::CreationDate, LastAccessDate TO Timestamp
CAST ATTR Posts::LastActivityDate TO Timestamp
MULT REF Posts::Tags TO +

NEST Posts::CreationDate, LastActivityDate TO PostMetadata
NEST Users::CreationDate, LastAccessDate, DownVotes, UpVotes TO UserMetadata

ADD REF Posts::postlinkIds: String* TO Postlinks WHERE Posts._id = Postlinks.PostId
MORPH REF Posts::postlinkIds (rmId rmEntity) TO postlinks

ADD REF Users::badgeIds: String* TO Badges WHERE Users._id = Badges.UserId
MORPH REF Users::badgeIds (rmId rmEntity) TO badges
```

Figure 6.20: Operations to be applied to *StackOverflow* in MongoDB.

the whole `Postlinks` collection is dropped, as specified by the `rmEntity` option in the Orion script.

```
// ADD REF Posts::postlinkIds: String* TO Postlinks WHERE Posts._id = Postlinks.PostId
stackoverflow_mongodb.Postlinks.createIndex({Postlinks.PostId: 1}, {name: "TEMP_INDEX"})

stackoverflow_mongodb.Posts.aggregate([
  { $lookup: { from: "Postlinks", localField: "_id", foreignField: "PostId", as: "postlinkIds" }},
  { $addFields: { "postlinkIds": "$postlinkIds._id"}},
  { $out: "Posts" }
])

stackoverflow_mongodb.Postlinks.dropIndex("TEMP_INDEX")

// MORPH REF Posts::postlinkIds ( rmId rmEntity ) TO postlinks
stackoverflow_mongodb.Posts.aggregate([
  { $lookup: { from: "Postlinks", localField: "postlinkIds", foreignField: "_id", as: "postlinks" }},
  { $out: "Posts" }
])
stackoverflow_mongodb.Posts.updateMany({}, {$unset: {"postlinkIds": 1, "postlinks.$[]._id": 1}})
stackoverflow_mongodb.Postlinks.drop()
```

Figure 6.21: JavaScript code generated for some operations in MongoDB.

Once the update script is executed, each post contains an aggregation array with its own postlinks, and each user contains an array of its own badges, so there is no need to join these collections to retrieve the information.

6.5.3 Outlier Migration in Reddit

In this last case study, we remove outliers in the Reddit dataset [Red18]. As will be explained in Chapter 8, an *outlier variation* is any variation with an abnormally low number of objects belonging to it compared with the total number of objects belonging to the rest of variations of an entity type. In that same chapter, a process devised to detect outliers is described, and here the outlier variations will be updated to regular variations by using Orion.

This dataset was injected into MongoDB, and its schema was inferred to obtain its corresponding U-Schema, and transformed to an Athena schema to be provided to the Orion engine. For this case scenario, the `Comments` entity type was considered, with more than 850 million comments distributed in twenty structural variations. Figure 6.22 shows the inferred `Comment` entity type, the composition of its variations as well as references to the other entity types: `Authors`, `Moderators`, and `Subreddits`.

By applying the process to detect outliers described in Chapter 8, five *regular variations* and fifteen *outlier variations* were detected. By using Orion, a developer can delete obsolete instances belonging to old variations, as well as adapt some variations to new ones, migrating the data accordingly. The developer can do so by using the `Delvar` and `Adapt` operations, as is shown in Figure 6.23. Here, variations $\{v_{1..4}\}$ will be deleted along with all objects belonging to those variations, by using the `Delvar` operation, and variations $\{v_{10..20}\}$ will be adapted to other variations, updating objects belonging to those variations by using the `Adapt` operation. After applying these Orion operations, the number of `Comment` variations will be reduced from twenty to five.

Each of these `Delvar` and `Adapt` operations will be translated into the corresponding Javascript code to be executed on the MongoDB database. In Figure 6.24, an example of the code generated from a `Delvar` operation and an `Adapt` operations is shown. The structure of both operations is fairly similar and can be divided in the following steps:

- First, the *match* clause only captures instances of variation 4 or variation 11. This can be achieved by checking if certain fields exist, or do not exist, for a particular variation. In this case, for variation 11, features `archived`, `distinguised`, `downs`, `edited`, `name`, and `score_hidden` must exist, and features `author_flair_css_class` and `author_flair_text` must not exist.

- Then, in `Delvar` the entire object is removed, while in `Adapt` features are added or

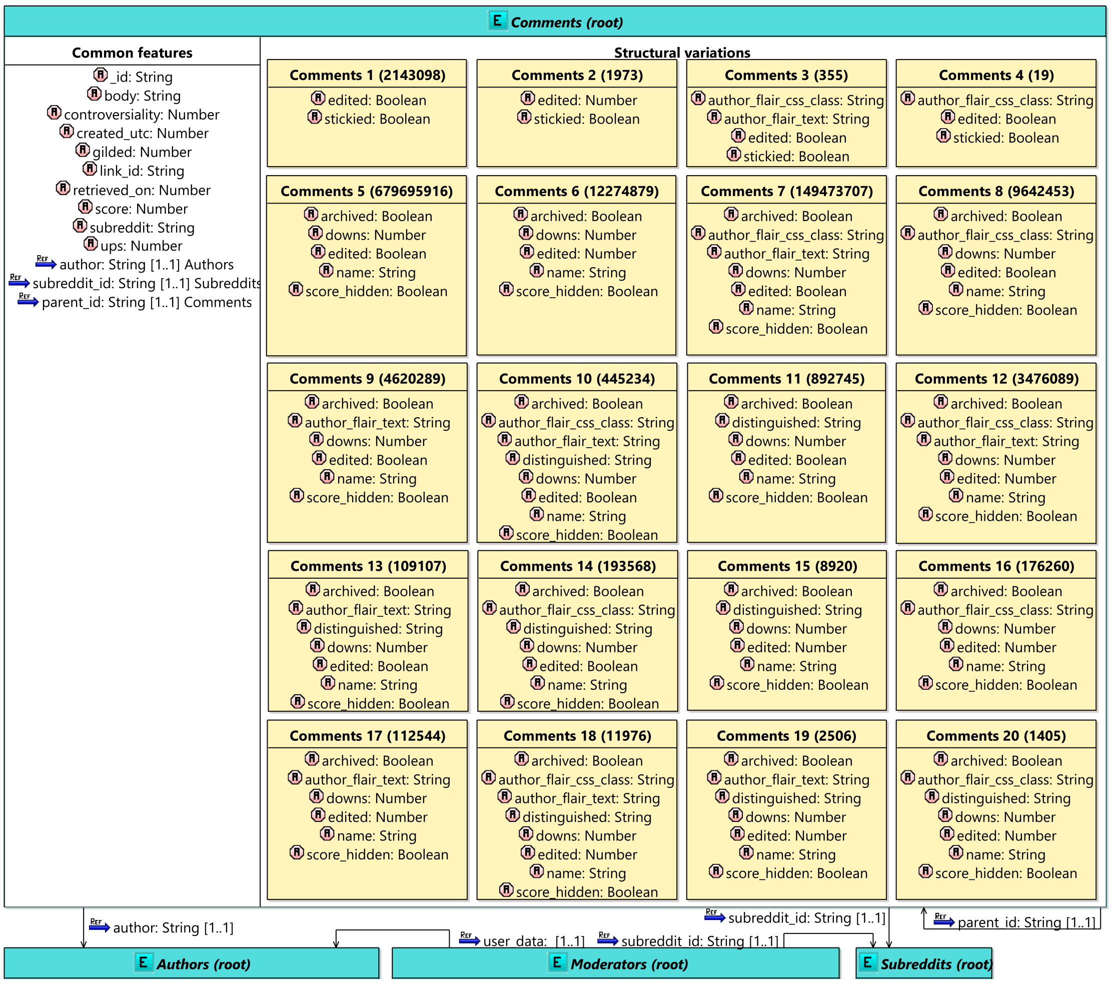

Figure 6.22: The `Comments` entity type from the *Reddit* schema.

removed to make variation 11 conform to variation 5. In this case, it is neccesary to *unset* the `distinguised` feature.

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.

```
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 6.23: Orion model used to update variations on the Reddit `Comments` entity type.

```
// DELVAR ENTITY Comments::4
reddit.Comments.remove({
 "author_flair_css_class": {$exists: true},
 "edited":                 {$exists: true},
 "stickied":               {$exists: true},
 "archived":               {$exists: false},
 "downs":                  {$exists: false},
 "name":                   {$exists: false},
 "score_hidden":           {$exists: false},
 "author_flair_text":      {$exists: false},
 "distinguished":          {$exists: false}
})
```

```
// 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 6.24: Orion script migrating the `Comments` variation 11 to 5.