The Journal of Systems & Software 203 (2023) 111743

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

CoDEvo: Column family database evolution using model transformations☆

Pablo Suárez-Otero [a,*], Michael J. Mior [b], María José Suárez-Cabal [a], Javier Tuya [a]

[a] University of Oviedo, Campus de Viesques, Gijon, Asturias, Spain
[b] Rochester Institute of Technology, Henrietta, NY, USA

ARTICLE INFO

Article history:
Received 1 August 2022
Received in revised form 4 February 2023
Accepted 3 May 2023
Available online 7 May 2023

MSC:
0000
1111

Keywords:
Software requirements
Consistency
MDE
Model transformation
NoSQL
Evolution

ABSTRACT

In recent years, software applications have been working with NoSQL databases as they have emerged to handle big data more efficiently than traditional databases. The data models of these databases are designed to satisfy the requirements of the software application, which means that the models must evolve when the requirements of the software application change. To avoid mistakes during the design and evolution of these NoSQL models, there are several methodologies that recommend using a conceptual model. This implies that consistency between the conceptual model and the schema must be maintained when either evolving the database or the software application. In this work, we propose CoDEvo, a model-driven engineering approach that uses model transformations to address the evolution of a NoSQL column family DBMS schema when the underlying conceptual model evolves due to software requirement changes, aiming to maintain consistency between the schema and conceptual model. We have addressed this problem by defining transformation rules that determine how to evolve the schema for a specific conceptual model change. To validate these transformations, we applied them to conceptual model changes from 9 open-source software applications, comparing the output schemas from CoDEvo with the schemas that were defined in these applications.

1. Introduction

NoSQL databases have been growing in importance due to the advantages they provide in the processing of big data (Moniruzzaman and Hossain, 2013). The database data models are designed to satisfy the requirements of a software application, with different design strategies depending on the characteristics of the database. For instance, in relational databases, models are designed based on the data application domain. On the other hand, the data models of NoSQL column family DBMSs are more oriented to the software application needs, as they follow a query-driven approach to schema design, where each table is designed to satisfy a query of the software client application (Carpenter and Hewitt, 2020). This means that if the same data are requested in more than one query from the application, the data may be duplicated in multiple tables. One possible way to avoid designs that may result in data inconsistencies is to design these schemas based on a conceptual model that provides information such as the entities of the system, their relationships and conceptual constraints in a normalized model. There are several methodologies that propose the use of a conceptual model for schema design in NoSQL column family DBMSs (Chebotko et al., 2015; de la Vega et al., 2020; Mior et al., 2017).

The requirements of a software application can change during the lifetime of an application, meaning the database evolves through changes in the following components: (1) the conceptual model, (2) the database schema, (3) the application queries and (4) the data (through migrations). The use of a conceptual model and queries for schema design implies a reflexive dependency where a change in any of them affects the others, requiring further changes to maintain the consistency between components. As data are stored according to the schema and constraints of the conceptual model, it means that these changes affect the data as well. The most typical evolution of a database comes from a direct change to the schema due to a application requirement change, which implicitly modifies the conceptual model. Directly modifying the schema may jeopardize the quality of the application; if one component evolves without considering the others, several problems can occur. For example, there may be inconsistencies between the conceptual model and the schema that allow future storage of data that contradicts the conceptual model constraints. Queries can be affected as well, as they may fail due to sudden changes in the schema. Inconsistencies also increase the probability of mistakes in the schema design, such as allowing future storage of data that contradicts constraints

☆ Editor: Doo-Hwan Bae.

* Corresponding author.
E-mail addresses: suarezgpablo@uniovi.es (P. Suárez-Otero), mmior@mail.rit.edu (M.J. Mior), cabal@uniovi.es (M.J. Suárez-Cabal), tuya@uniovi.es (J. Tuya).

https://doi.org/10.1016/j.jss.2023.111743

specified in the conceptual model (Suárez-Otero et al., 2021). This last problem is significantly more difficult to approach in NoSQL column family DBMSs, as the same data can be duplicated in several tables.

Our objective is to provide an automated approach that addresses how to evolve a NoSQL column family DBMSs database from a conceptual model change considering the schema, the data and the application queries. In this work, we address problems related to schema evolution by proposing an MDE approach named CoDEvo that specifies the required actions to be performed on a schema to maintain consistency between a conceptual model and schema for changes in the conceptual model created by software requirement changes. The contributions of this paper are:

1. The determination of how the database schema must evolve to maintain consistency with the conceptual model. We additionally provide an evolved database schema with the aforementioned transformations applied to the source database schema.
2. A set of experiments to validate CoDEvo, where we apply CoDEvo to several changes in the conceptual models of real projects. We then compare the schemas that are in the repository of the projects against the schemas generated by CoDEvo to determine if these last schemas satisfy the project requirements.

The remainder of this paper is structured as follows. Section 2 contains the background of our work and related definitions. Section 3 details the design of CoDEvo. Section 4 contains the experimentation details and the threats to validity. Section 5 details the related work and Section 6 contains the conclusion and future work.

2. Background and definitions

In this section, we introduce the background of CoDEvo and some basic terminology used in the rest of the paper. In prior work (Suárez-Otero et al., 2021, 2020) we focused on two objectives: (1) understanding how NoSQL databases evolve and (2) how a conceptual model can help during this evolution. We studied several projects to identify changes in the conceptual model that occurred across versions of the project. We organized these changes in categories depending on the conceptual model structures that were created, removed, or modified: entities, attributes and relationships. In the case of the entities, the identified changes were **AddEntity**, **AddPK**, **RemovePK** and **AddWeakEntity**. In the case of attributes, they were **AddAttribute**, **RemoveAttribute** and **SplitAttribute**. In the case of relationships, they were **AddRelationship** and **UpdateCardinality**.

In the following paragraphs, we define several terms related to model transformation and NoSQL databases that we will use during the description of CoDEvo.

Column-attribute mapping: Connections between each column of the database schema with an attribute of the conceptual model. This mapping is obtained from the schema based on the convention used for the names of the columns (NameEntity+NameAttribute).

Data integrity: Generally, data integrity can be defined as "the maintenance of, and the assurance of the accuracy and consistency of, data over its entire life-cycle" (Liu et al., 2017). However, as in this paper we are only working with NoSQL column family DBMSs, where data is usually denormalized, we specifically refer to data integrity as "the maintenance of consistency between information repeated across the database tables".

Inter-model consistency: A guarantee that the database schema conforms to the constraints defined in the conceptual model, mainly the relationships between entities and the attributes that are primary key. Fulfilling these constraints makes it possible to guarantee that data cannot be lost due to poor schema design. It also guarantees that the conceptual model is representing a normalized version of the schema. Any evolution of either the conceptual model or the database schema may affect the other.

Metamodel: A model of a model that represents the type of classes that compound the model, their constraints and their relationships between each other.

Metaclass: A class of a metamodel. The instances of these metaclasses are named **elements**. These elements are contained in the models that conform to the metamodels.

Evolution: One or more changes in either the conceptual model or the database schema that are required to maintain the inter-model consistency. A change is a single insertion, removal, or update of a single element from either the conceptual model or the schema.

Transformation: Generation of a target model or textual artifacts given one or more source models (Mens and Van Gorp, 2006). A transformation is executed based on a trigger condition and performs a set of actions to generate a specific target model or text, that together comprise a transformation rule. A transformation definition is composed of several transformation rules that describe how the target models or texts are generated.

Transformation model: A model that contains the changes that are going to be applied in a transformation. Higher-order model transformations (HOTs) use this type of model, with different types depending on the amount of source models and target models (Tisi et al., 2009). In this work we will use the (de)composition transformation type, which requires having at least one transformation as an input model and one transformation as an output model.

3. CoDEvo approach description

There are several issues regarding NoSQL databases evolution when the conceptual model evolves through one or more changes due to changes in the requirements:

1. **Evolution of the database schema**: Inter-model consistency may break if the database schema does not evolve according to the conceptual model changes.
2. **Ensure data integrity**: As the database schema evolves to solve issue 1, the data may lose their integrity. To maintain this integrity, the data must be migrated to new tables as well as repairs made to data that do not comply with new conceptual constraints.
3. **Updates in application queries**: Queries may reference database structures that were modified during the resolution of issue 1. These queries must be adapted to the new schema without changing the original semantics in the application context.

In this work we focus on issue 1 by proposing CoDEvo, a model-driven engineering approach that addresses the evolution of the database schema using model transformations (Bézivin et al., 2006). It uses higher-order transformations for this, handling the changes in the conceptual model, the database schema, and the data as models, which are named transformation models. In addition to these transformation models, CoDEvo provides output models with the transformations applied.

The metamodels, models that conform to them, database procedures and transformation definitions that compose CoDEvo are illustrated in Fig. 1 with the following notation: Metamodels as rounded corner rectangles, models as blue (input) or green (output) rectangles, database procedures as magenta rectangles

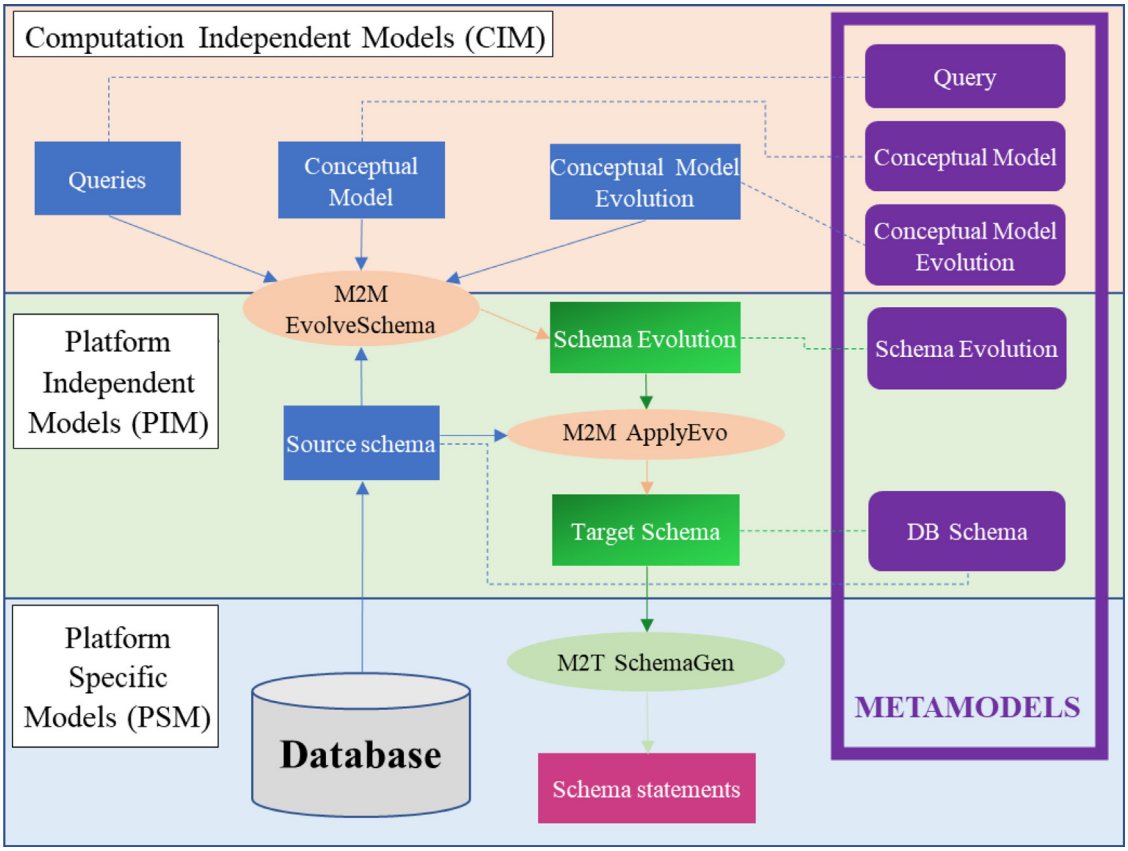

Fig. 1. General process of CoDEvo. Solid arrows display the inputs and outputs of Model-To-Model and Model-To-Text transformations. Dotted arrows display the metamodel that a model conforms to.

and transformation definitions as ellipses. The models conform to metamodels with the same name, except for the Source schema and Target schema, that both conform to the metamodel DB schema. The models are connected through properties that are defined in the following subsection (e.g., a column is associated with the attribute mapped to it through the identifier of the attribute). These metamodels, models, database procedures are organized (MDA, OMG, 2008) in three layers:

1. **Computationally-Independent Models (CIM)**: Models independent of the data model of the database schema and the technology:*Conceptual Model, Queries* and *Conceptual Model Evolution*.
2. **Platform Independent Models (PIM)**: Models that depend on the data model of the database schema, but not on a specific technology: *Source schema*, *Schema evolution*, and *Target schema*. These models will be either inputs or outputs in the following M2M (model-to-model) transformation definitions:
   - **EvolveSchema**: Receives as inputs all the models from CIM and *Source schema* and generates the model *Schema evolution* that specifies all the changes of the *DB schema* required to reflect the conceptual model change.
   - **ApplyEvo**: Receives as inputs the *Source schema* and the *Schema evolution* and generates the *Target schema* which contains the schema structure detail after the changes defined in *Schema evolution* are applied to the schema.
3. **Platform Specific Models (PSM)**: Models that depend on the database technology. This layer contains the M2T model-to-text) transformation definition *SchemaGen*, that generates database procedures specific to a particular database technology (Schema statements). These database procedures contain the database statements and instructions required to perform the changes specified in the models *Schema evolution*.

In the rest of this section, we describe these models and transformation definitions. The metamodels will be explained and illustrated separately in following subsections, depending if the model that conform to them are inputs or outputs. The relationships between classes of these metamodels are displayed graphically in Fig. A.9 in the Appendix and textually in the next sections.

3.1. Input models

The input models of CoDEvo are the *conceptual model*, *queries*, *source schema*, and *conceptual model evolution*. The metamodels of these four models are based on the metamodels defined by de la Vega et al. (2020) which they based on the work of Chebotko et al. (2015).

The *conceptual model* metamodel is composed of the metaclasses **Entity**, **Weak entity**, **Attribute** and **Relationship**. The entities and weak entities are associated with one or more attributes. The property 'Key' from an attribute specifies if it is part of the primary key. A relationship associates two entities (can be weak) with a specific cardinality: 1:1 (one to one), 1:n (one to many) or n:m (many to many). The conceptual model metamodel is displayed in Fig. 2.

The metaclasses of the *Query* metamodel are **Requirement Query**, **Selection**, and **Filter**. Note that these requirement queries

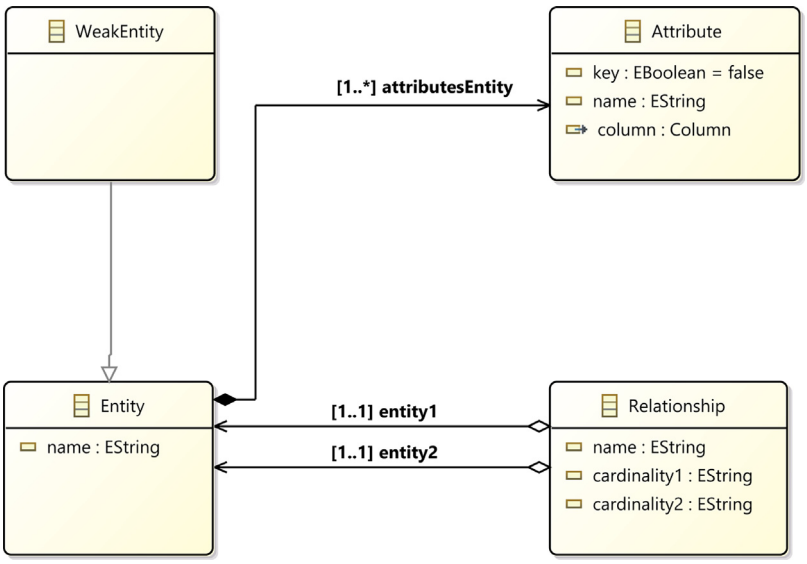

Fig. 2. Metamodel of the conceptual model.

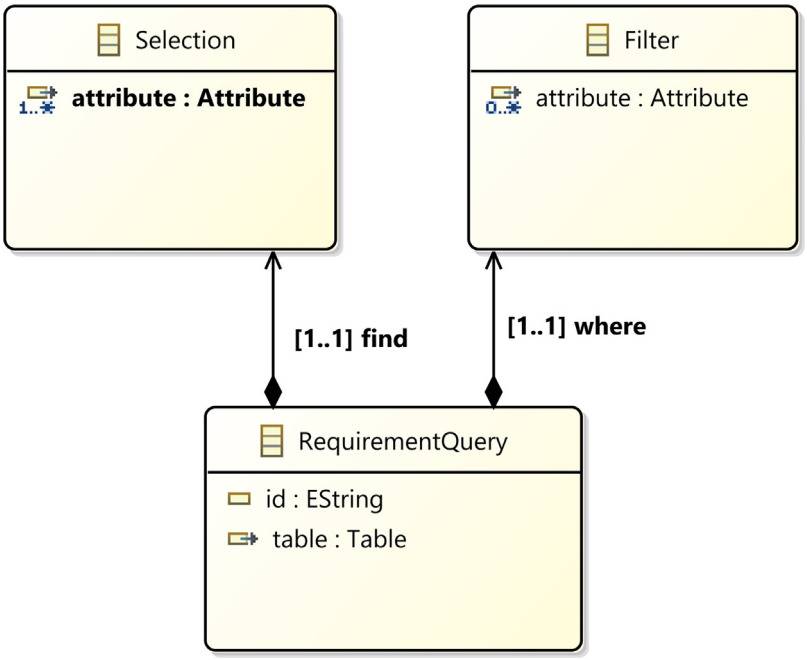

Fig. 3. Metamodel of the queries.

are the conceptual queries used for designing a table of the database and they do not have a direct relation with the application queries (`SELECT`, `INSERT`, `DELETE`, `UPDATE`). Thus, a requirement query element is associated with a table element from the model *source schema*. A requirement query element is also associated with a selection element and a filter element. Each selection element is associated with several attributes from the model *conceptual model*, which specifies the attributes that are requested by the query. Each filter element is also associated with several attributes, which are used to define restrictions in a query, similar to the `WHERE` clause of an SQL query. The query metamodel is displayed in Fig. 3.

Source schema conforms to the metamodel *DB schema*, which contains the design details of a NoSQL column family DBMS schema with the metaclasses **TableFamily**, **Table**, **Custom Type**, and **Column**. All the tables of a schema are part of a single TableFamily, which determines the extent of the schema. A table or a custom type element are associated with one or more column elements. Additionally, a column contains a property named 'Key' to indicate if it is part of the primary key. It also contains the properties 'partition' and 'clustering', which are the two types of key columns that can be specified in a Cassandra database and an association with an attribute from the *conceptual model*, which we refer to as mapping. A column is associated to a type. The DB schema metamodel is displayed in Fig. 4.

The last input is the *conceptual model evolution* model that contains the detail of changes performed in the *conceptual model*. The scenarios that we consider for changes in the conceptual model are obtained from two types of sources: from research works that address the evolution of different databases (Curino et al., 2010; Noy and Klein, 2004; Bonifati et al., 2019; Herrmann et al., 2017) and empirically from detection of cases in real case

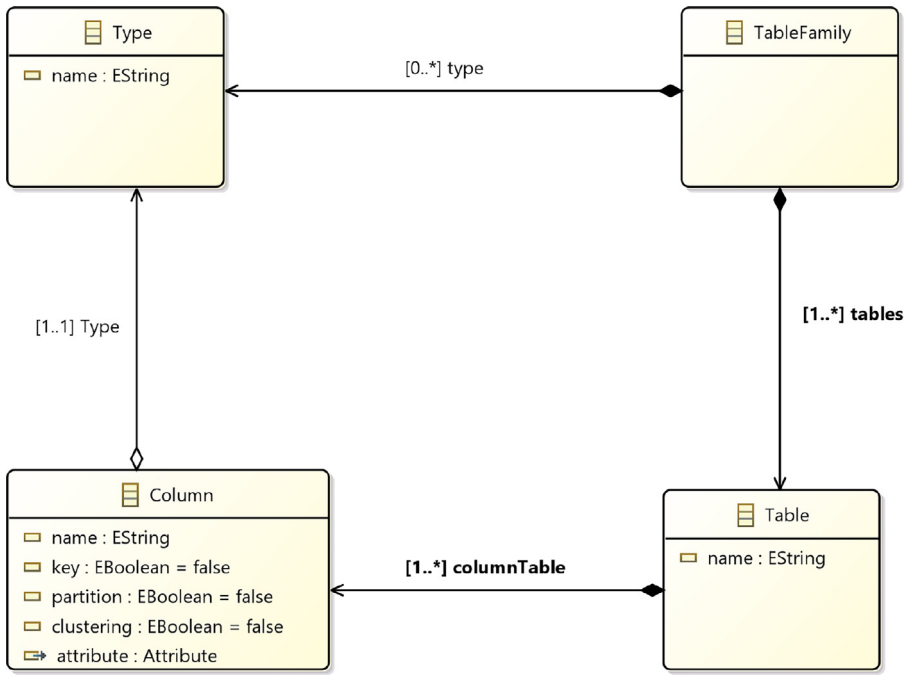

Fig. 4. Metamodel of the database schema.

Table 1
Conceptual model changes detected in projects and scientific works.

Structure	Change	Description	P	RW
Entity	AddEntity	Creation of a new entity along with the attributes associated with it	X	X
Entity	AddWeakEntity	Creation of a new weak entity and its relationship with the primary entity	X	X
Entity	DeleteEntity	Removal of an entity, including all relationships the entity had	X	X
Entity	MergeEntity	Merge of two exiting entities into a resulting one		X
Entity	SplitEntity	Split one existing entity into two		X
Entity	AddPK	Change of a non-key attribute of an entity into a key attribute	X	X
Entity	RemovePK	Change of a key attribute of an entity into a non-key attribute	X	X
Attribute	AddAttribute	Creation of a new attribute and association to an existing entity	X	X
Attribute	RemoveAttribute	Removal of an attribute from an entity	X	X
Attribute	SplitAttribute	Split an existing attribute from an entity into two or more attributes	X	
Relation.	AddRel	Establishment of a new relationship between two entities. It also defines the cardinality of the relationship (1:1, 1:n or n:m)	X	X
Relation.	UpdateCardinality	Change of the cardinality of an existing relationship	X	
Relation.	DeleteRel	Removal of an existing relationship		X

projects, which are detailed in Section 4. In Table 1, we classify the conceptual model changes by the conceptual model structure directly affected by the change. We also specify if the conceptual model change was detected in a project (P) or/and in a research work (RW).

3.2. Output models

The M2M transformation definition EvolveSchema uses the input models described in the previous section to obtain the output model Schema Evolution. Schema Evolution contains classes that describe how the source schema must evolve through associations with elements from other metamodels:

- **Add** which details the tables and columns that need to be added to the schema.
- **Remove** that details the tables and columns that need to be removed from the schema.
- **AddPK** that details the columns that need to be added to the primary key of a table.
- **RemovePK** that details the columns that need to be removed from the primary key of a table.

The model "Target Schema" is obtained through the M2M transformation definition ApplyEvo and conforms to the metamodel "Schema" which was detailed in the previous section.

3.3. M2M EvolveSchema transformation definition and ApplyEvo description

EvolveSchema contains the transformation rules that specify the transformations of the schema required to maintain the inter-model consistency after a certain change in the conceptual model. These transformations are displayed in the output model *schema evolution*. The inputs of EvolveSchema are the four models described in the previous subsection.

Each transformation rule is composed of several predicates, functions, and atomic transformations. A predicate is an expression that provides a Boolean value given specific properties of one or more elements. A function receives one or two elements

as inputs and returns either a property of this element (name, cardinality, etc.) or one or more elements associated with the input elements. An atomic transformation is an operation that, given one or more elements, generates or removes a single element from source schema or creates an association between two elements of source schema. To differentiate between functions, predicates, and transformations, the names of functions and predicates start with lower case letters and the names of transformations start with upper case letters.

Each element (entity, table, etc.) is denoted as follows:

- Conceptual Model:
  - e: An **entity**.
  - a: An **attribute**. It is associated with one entity.
  - r: A **relationship**. It is associated with two entities, establishing a certain cardinality: '1:1', '1:n', or 'n:m'.
- Schema:
  - t: A **table**.
  - ct: A **custom type**. It can be associated with several tables in an aggregation.
  - c: A **column**. It is associated with either a table or a custom type in a composition. Each column is also associated with an attribute (mapping). At least one of the columns associated to a particular table must be part of the primary key.
- Query:
  - q: A **requirement query**. It is associated with one table that was modeled after the query.
  - f: A **filter**. It is associated with a query and a set of attributes.
  - s: A **selection**. It is associated with a query and a set of attributes.

When any of the previous letters are in uppercase, they represent all the elements of that type that are in the model. For instance, "T" represents all the tables that are in the model.

3.3.1. Predicates and functions

The predicates used in the definition of the transformations are as follows:

- **isIn (a, q)**: if attribute a appears anywhere (select or filter) in the requirement query q.

  $\exists q \in Q, \exists attr \in q, a == attr$
- **isKey (c)**: if column c is part of the primary key of any table of the model.

  $c.key \vee c.partition \vee c.clustering$
- **isKey (c, e)**: if the attribute mapped to c is part of the primary key in entity e.

  $c.attribute.key \wedge c.attribute.entity == e$
- **isKey (a, t)**: if the attribute a is mapped to any column that is part of the primary key of table t.

  $\exists c \in t, isKey(c) \wedge c.attribute == a$
- **isMapped (a, c)**: if the attribute a is mapped to the column c.

  $c.attribute = a$
- **inFilter (a, q)**: if the attribute a is part of the filter from the query q.

  $a \in q.filter$
- **mapped (e, t)**: if entity e has at least one attribute mapped with one column associated with table t.

  $\exists c \in C, isKey(c, e)$
- **mappedPK (e, t)**: if the attributes that compose the primary key of entity e are mapped to columns associated with table t that are part of its primary key.

  $\forall a \in e \wedge a.key, \exists c \in t \wedge isKey(c) \wedge c.attribute == a$
- **mapped (a, t)**: if the attribute a is mapped with any column associated to table t.

  $\exists c \in t, c.attribute == a$

The functions used in the transformations, which either return information about the association of an element with another or properties of an element, are the following:

- **attribute (c)**: returns the attribute that the column c is mapped to.

  $c.attribute$
- **cardinality (e_1, e_2)**: returns the cardinality of the relationship between the entities e_1 and e_2: '1:1', '1:n' or 'n:m'.

  $\exists r \in R, r.entity_1 == e_1 \wedge r.entity_2 == e_2 \quad \vee$

  $r.entity_1 == e_2 \wedge r.entity_2 == e_1 : r.cardinality$
- **column (a, t)**: returns the column from table t that is mapped to attribute a.

  $\exists c \in t, c.attribute == a : c$
- **top (r)**: in a relationship r with cardinality '1:n', returns the entity on the side of the cardinality value '1'.

  $r.entity_1$
- **bottom (r)**: in a relationship r with cardinality '1:n', returns the entity on the side of the cardinality value 'n'.

  $r.entity_2$
- **query (t)**: returns the set of queries that are associated to the table t.

  $\forall q \in Q, q.table == t : q$
- **left (r)**: in a relationship r with cardinality '1:1' or 'n:m', returns the entity on the left side.

  $r.entity_1$
- **right (r)**: in a relationship r with cardinality '1:1' or 'n:m', returns the entity on the right side.

  $r.entity_2$
- **pk (e)**: returns the attributes that are part of the primary key of entity e.

  $\forall a \in e, a.key : a$

3.3.2. Transformations

The actions required to evolve the schema while maintaining the inter-model consistency are obtained using transformations of models. These actions are ultimately used in transformations, which we classify as **atomic transformations** and **transformation rules**. Atomic transformations perform at most two actions that can either be creation of elements or association of existing elements. In the case of transformation rules, each one is triggered by a specific conceptual model change and their objective is to maintain inter-model consistency to evolve the schema considering the conceptual model change. We formally define the transformation rules using atomic transformations, predicates and functions.

Atomic transformations. The atomic transformations that we use in the transformation rules are the following:

- **Associate (t, c)**: associates the column c with the table t.
- **Associate (t, ct)**: associates the custom type ct with the table t.
- **CopyAddPK (t, c)**: creates a copy of table t with the column c added to the primary key.
- **CopyRemPK (t, c)**: creates a copy of table t that does not contain c in the primary key. In order to perform this atomic transformation, there must be other columns that are part of the primary key.
- **CreateCol (a)**: creates a column based on attribute a, naming this new column with the same name as a.
- **CreateColPK (a)**: creates a column based on attribute a, naming this new column with the same name of a and sets this column as primary key.
- **CreateTable (e, {c1, c2, ..., cN})**: creates a table with the name of the entity e, and associates the set of columns $c1$, $c2$, ..., cN with this new table. Note that the brace bracket symbols '{ }' denote a set of elements, in this case a set of columns.
- **CreateTable (r, {c1, c2, ..., cN})**: creates a table with the name of the relationship r, and associates the set of columns $c1$, $c2$, ..., cN with this new table.
- **CreateType (e, {c1, c2, ..., cN})**: creates a custom type with name e associated with the set of columns $c1$, $c2$, ..., cN.
- **Remove (c)**: removes column c and its associations (tables and attributes). If c is part of the primary key of a table, there must be other columns that are part of the primary key in the associated table to perform this operation.
- **Remove (t)**: removes table t from the model as well as its associated columns.
- **SplitTable (t, e1, e2)**: creates two new tables from table t to store the data of $e1$ and $e2$ using the Transformation Rule 1 for the conceptual model change AddEntity (e).

Transformation rules. For each of the transformation rules, there is a formal definition of how the database schema must evolve using the functions, predicates and atomic transformation previously defined. In certain transformation rules we use additional predicates that are only used in these rules. For each of these additional predicates there are both an informal explanation of its purpose as well as a formal definition.

In the formal definition of some rules, we use the symbol '|', which in a notation such as "action |condition", represents that the action is performed each time that the condition is met. The transformation rules that we have defined are the following:

Rule 1. AddEntity (e): Add an entity e, composed of several attributes a, to the conceptual model.

$$AddEntity(e) := CreateTable\,(e, CreateCol(a) \mid a \in e)$$

Rule 2. AddWeakEntity (we): Add a weak entity we, composed of attributes a to the conceptual model. This weak entity also contains a relationship with a primary entity pe.

$$AddWeakEntity(we) :=$$
$$let\ ct = CreateType(we, CreateCol(a) \mid a \in we)$$
$$\forall t \in T, mapped(we, t) : Associate(t, ct)$$

The first step of the transformation is the creation of the custom type, which is assigned to a variable ct. This custom type is then associated with tables that store the entity that is related to the new weak entity. This association is only required in one of the selected tables, the developer being the person responsible for choosing in which of the selected tables the association is performed.

Rule 3. AddPK (a, e): Add the attribute a to the primary key of an entity e.

$$AddPK(a, e) :=$$
$$\forall t \in T, mappedPK(e, t) \wedge \neg isKey(a, t) :$$
$$CopyAddPK(t, Create(a)), Remove(t)$$

Rule 4. RemovePK (a, e): Remove the attribute a from the primary key of an entity e. Let $PQ(t, a, c)$ be the predicate that checks if a column c is mapped to an attribute a and this attribute a is not in any of the queries that table t is associated with.

$$PQ(t, a, c) := attribute(c) == a \wedge \neg isIn(a, query(t))$$

$$RemovePK(a, e) :=$$
$$\forall t \in T, mapped(a, t) \wedge \exists c \in t, PQ(t, a, c) :$$
$$CopyRemPK(t, c), Remove(t)$$

Rule 5. AddAttribute (a, e): Add a new attribute a to an entity e.

$$AddAttribute(a, e) := \forall t \in T, mapped(e, t) :$$
$$Associate(t, CreateCol(a))$$

From all the tables t that satisfy the selection criterion, it is only required to associate the new column with one of them. This decision is up to the developer.

Rule 6. RemoveAttribute (a, e) (Case 1): Remove an attribute a from entity e. The target tables affected by the transformation must not contain in the primary key any column that is to be removed. Let $P(a, c)$ be a predicate that checks if the column c is non-key as well as mapped to a.

$$P(a, c) := attribute(c) == a \wedge \neg isKey(c) :$$

$$RemoveAttribute(a, e) :=$$
$$\forall t \in T, mapped(e, t) \wedge \exists c \in t, P(a, c) :$$
$$Remove(c)$$

Rule 7. RemoveAttribute (a, e) (Case 2): Remove an attribute a from entity e. The target tables affected by the transformation must contain in the primary key any column that is to be removed. Let $P(a, c)$ be a predicate that checks if there is a column c mapped to an attribute a, being c part of the primary key.

$$P(a, c) := attribute(c) == a \wedge isKey(c) :$$

$$RemoveAttribute(a, e) :=$$
$$\forall t \in T, mapped(e, t) \wedge \exists c \in t, P(a, c) :$$
$$CopyRemPK(t, c), Remove(t)$$

Rule 8. SplitAttribute (a, {a1, a2... an}): Split an attribute a to several attributes $a1, a2, \ldots, an$.

$$SplitAttribute(a, \{a1, a2, \ldots, an\}) :=$$
$$\forall t \in T, mapped(a, t) :$$
$$Associate(t, CreateCol(\{a1, a2...an\})),$$
$$Remove(column(a, t))$$

Rule 9. AddRelationship (r) (Case 1): Add a new relationship r between two entities with a cardinality of either 1:1 or n:m.

$$AddRelationship(r) :=$$
$$let\ e_1 = left(r),\ e_2 = right(r)$$
$$CreateTable(r, (CreateColPK(a) \mid a \in e_1 \vee e_2, a.key) \cup$$
$$(CreateCol(a) \mid a \in e_1 \vee e_2, \neg IsKey(a) \vee a \notin e_2))$$

Rule 10. AddRelationship (r) (Case 2): Add a new relationship r between two entities with a cardinality of 1:n.

$AddRelationship(r) :=$
$let\ e_1 = top(r), e_2 = bottom(r)$
$CreateTable(r, (CreateColPK(a) \mid a \in e_2, a.key) \cup$
$(CreateCol(a) \mid a \in e_1 \vee e_2, \neg IsKey(a) \vee a \notin e_2))$

Rule 11. UpdateCardinality (r) (Case 1): Update the cardinality of a relationship r with a new cardinality of 1:n. Let $P(t, e_2)$ be a predicate that checks if all the key attributes of an entity e_2 have columns mapped to the primary key of a table t.

$P(t, e_2) := \forall a \in e_2 \wedge a.key, isKey(a, t)$

$UpdateCardinality(r) :=$
$let\ e_1 = top(r), e_2 = bottom(r)$
$\forall t \in T, mapped(e_1, t) \wedge mapped(e_2, t) \wedge \neg P(t, e_2) :$
$CopyAddPK(t, CreateCol(pk(e_2))), Remove(t)$

Rule 12. UpdateCardinality (r) (Case 2): Update the cardinality of a relationship r to a new cardinality n:m. Let $P(t, e_1, e_2)$ be a predicate that checks if a table t contains key columns for all attributes that are part of the primary key of the entities e_1 and e_2.

$P(t, e_1, e_2) := \forall a \in e_1 \vee e_2 \wedge a.key, isKey(a, t)$

$UpdateCardinality(r) :=$
$let\ e_1 = top(r), e_2 = bottom(r)$
$\forall t \in T, mapped(e_1, t) \wedge mapped(e_2, t) \wedge P(t, e_1, e_2) :$
$CopyAddPK(t, CreateCol(pk(e_2)) \cup CreateCol(pk(e_1))),$
$Remove(t)$

Rule 13. RemoveEntity (e): Remove an entity e from the conceptual model. We consider three scenarios (1, 2.a and 2.b), depending on the characteristics of the table from the schema where transformations need to be performed.

$RemoveEntity(e) :=$
$1.\forall t \in T, mapped(e, t) \wedge (\forall c \in pk(t), attribute(c) \in e) :$
$\quad RemoveTable(t)$
$2.\forall t \in T, mapped(e, t) \wedge \neg(\forall c \in t, attribute(c) \in e) :$
$\quad a.\forall c \in t,\ attribute(c) \in e \wedge \neg isKey(c) :$
$\quad\quad Remove(c)$
$\quad b.\forall c \in t,\ attribute(c) \in e \wedge isKey(c) :$
$\quad\quad CopyRemPK(t, c), Remove(t)$

In scenario 2.b, the creation of a copy table without the columns selected is only done once at the platform level.

Rule 14. MergeEntity ($\mathbf{e_1}$, $\mathbf{e_2}$): Merge two entities e_1 and e_2 into one entity e. We consider two scenarios (1 and 2) depending on the relationship cardinality between e_1 and e_2.

$MergeEntity(e_1, e_2) :=$
$let\ a1pk = pk(e_1), a2pk = pk(e_2), card = cardinality(e_1, e_2)$
$1.\forall t \in T, mapped(e_1, t) \veebar mapped(e_2, t) :$
$\quad CopyAddPK(t, \{CreateCol(a1pk) \cup CreateCol(a2pk)\}),$
$\quad Remove(t)$
$2.\forall t \in T, mapped(e_1, t) \wedge mapped(e_2, t) \wedge card \neq 'n : m' :$
$\quad CopyAddPK(t, \{CreateCol(a1pk) \cup CreateCol(a2pk)\}),$
$\quad Remove(t)$

Rule 15. RemoveRelationship (r): Remove a relationship r from the conceptual model. We consider two scenarios depending on whether the table contains only information of the relationship (scenario 1) or it also contains information from other entities (scenario 2).

$RemoveRelationship(r) :=$
$let\ e_1 = left(r), e_2 = right(r)$
$1.\forall t \in T, mapped(r, t) \wedge$
$(\forall c \in t, attribute(c) \in e_1 \wedge attribute(c) \in e_2) :$
$\quad RemoveTable(t)$
$2.\forall t \in T, mapped(r, t) \wedge$
$(\forall c \in t, attribute(c) \in e_1 \vee \neg attribute(c) \in e_2) :$
$\quad SplitTable(t, e_1, e_2)$

Case for the conceptual model change "Split Entity": There are no changes in the schema. Therefore, no transformation rule is created for this conceptual model change.

Transformation rules with queries. When the requirement queries that need to be executed against the database are provided for the conceptual model changes AddEntity, AddAttribute and AddRelationship, the following transformation rules are executed:

Rule 16. AddEntity (e, q): Add an entity e, composed of several attributes a, to the conceptual model. Query q details how the application will query the data of e.

$AddEntity(e, q) :=$
$CreateTable\ (e, (CreateCol(a) \mid a \epsilon\ e \wedge \neg isInFilter(a, q)) \cup$
$(CreateColPK(a) \mid isInFilter(a, q)))$

Rule 17. AddAttribute (a, e, q): Add a new attribute a to an entity e. Query q details how the application will query the data of the new attribute. Query q details how the application will query the data of a.

$AddAttribute(a, e, q) :=$
$1.\exists t \in T, \forall a \in e \wedge a.key, \exists c \in t, isKey(c) \wedge attribute(c) == a :$
$\quad Associate(t, CreateCol(a))$
$2.\nexists t \in T, \forall a \in e \wedge a.key, \exists c \in t, isKey(c) \wedge attribute(c) == a :$
$\quad CreateTable\ (e, (CreateCol(a) \mid a \epsilon\ e \wedge \neg inFilter(a, q)) \cup$
$\quad (CreateColPK(a) \mid inFilter(a, q)))$

Rule 18. AddRelationship (r, q) (Case 1): Add a new relationship r between two entities with a cardinality of either 1:1 or n:m. Query q details how the application will query the data of r.

$AddRelationship(r, q) :=$
$let\ e_1 = left(r),\ e_2 = right(r)$
$CreateTable(r, (CreateColPK(a) \mid a \in e_1 \vee e_2,$
$a.key \vee inFilter(a, q)) \cup$
$(CreateCol(a) \mid a \in e_1 \vee e_2, \neg a.key))$

Rule 19. AddRelationship (r, q) (Case 2): Add a new relationship r between two entities with a cardinality of 1:n. Query q details how the application will query the data of r.

$AddRelationship(r, q) :=$
$let\ e_1 = top(r), e_2 = bottom(r)$
$CreateTable(r, (CreateColPK(a) \mid a \in e_2, a.key) \cup$
$(CreateColPK(a) \mid a \in e_1 \vee e_2,$
$inFilter(a, q) \wedge \neg(a.key \wedge a \in e_2)) \cup$
$(CreateCol(a) \mid a \in e_1 \vee e_2, \neg a.key \vee a \notin e_2))$

Table 2
Inputs and outputs of the experimentation.

Component	How it was obtained
Input: **Conceptual model** ATL file	**Manually** through reverse engineering analyzing the database schema (tables, columns, primary key of the tables).
Input: **Source schema** ATL file	**Automatically** by reading the CQL script of the repository schema from a particular version. It also contains a **semi-automatically** obtained mapping between attributes of the conceptual model and columns of the schema.
Input: **Queries** ATL file	**Automatically** by using the information provided in the mapping. Each one of the queries contains an association of attributes from the conceptual model that define how each table of the schema is defined.
Input: **Conceptual model evolution** ATL file	**Manually** by defining the conceptual model changes, which are obtained by analyzing the differences that exist between the conceptual model of this version and the previous one.
Output: **Schema evolution** ATL file (CoDEvo schema)	**Automatically** executing M2M transformation **EvolveSchema**.
Output: **Database statements** to perform schema changes	**Automatically** executing M2T transformation **SchemaGen**.
Output: **Target Schema** after the changes are applied	**Automatically** executing the M2M transformation **ApplyEvo**.

By considering the queries for the evolution of the schema, CoDEvo addresses specific requirements related to duplication of data in order to both gain more performance in the queries execution and cover other types of queries where the data is filtered through non-key attributes.

After the transformations are executed, the model schema evolution is generated, containing the changes that need to be performed. After this, the transformation definition **ApplyEvo** begins, which generates the final schema with the changes determined in schema evolution applied to it.

3.4. M2T SchemaGen and M2T DataModifier descriptions

SchemaGen and **DataGen** generate the database statements and actions required to perform in an Apache Cassandra database (Apache Foundation, 2008), the operations specified in models *Schema evolution* and *Data modification*. SchemaGen generates the database statements that evolve the database schema in the correct sequence. DataGen generates a code-like description that details how to migrate the data.

4. Experimentation and validation

In this section, we validate CoDEvo through its application to nine case studies of projects stored in public repositories that use the column family DBMS Apache Cassandra. These repositories contain the information of the changes in the schema that have been performed since the creation of the projects, which are recorded through commits in the repository. We considered that each commit where the schema is changed generates a new **version** of the schema, which may contain one or more conceptual model changes.

During the experimentation, for each version, we compared the schema of the repository against the schema generated by CoDEvo. In this section, we refer to these schemas as **'repository schema'** and **'CoDEvo schema'**, respectively.

CoDEvo has been automated by implementing the transformation rules defined in section 3 in an Eclipse project using ATL (Jouault et al., 2008). CoDEvo receives as inputs the models that contain necessary information for the evolution of the schema and generates as outputs the models that define how the schema must be evolved. The detail of these inputs and outputs and how they are obtained is displayed in Table 2.

The projects that we use for this validation are the following:

- **Minds**[1]: A social network
- **PowSyBl**[2]: A framework for the implementation of power system simulations and analysis software
- **Thingsboard**[3]: IoT platform for data collection
- **Wireapp**[4]: Encrypted communication app
- **Blobkeeper**[5]: Distributed file-storage service
- **Reviews-serv**[6]: Module for restaurants
- **Sunbirds**[7]: A project for learning and human development
- **Doudouchat**[8]: No description provided
- **Sop**[9]: A database engine within a code library.

All conceptual model changes detected in these projects are displayed in the Table 3.

4.1. Research questions

In order to validate CoDEvo, we have defined the following research questions:

RQ1. How many differences exist between the CoDEvo schemas and the repository schemas?
RQ2. Are the CoDEvo schemas valid considering the project requirements?

To respond to RQ1 and RQ2, we used the conceptual model changes introduced in each project version as inputs for CoDEvo along with the schema from the previous version. Then, we compared the schema from the repository with the schema generated by CoDEvo for each version to obtain their differences. In the following subsections, we answer each RQ separately and address threats to validity.

4.2. RQ1: How many differences exist between the CoDEvo schemas and the repository schemas?

To respond to this research question, we compared the schema in the project repository with the schema generated by CoDEvo

1 https://gitlab.com/minds
2 https://github.com/powsybl
3 https://github.com/thingsboard/thingsboard
4 https://github.com/wireapp/wire-server
5 https://github.com/sherman/blobkeeper
6 https://github.com/bon-app-etit/reviews-service
7 https://github.com/ekstep-sp/sunbird-devops
8 https://github.com/doudouchat/exemple-integration
9 https://github.com/SharedCode/sop

Table 3
Conceptual model changes detected in each project.

Conceptual model change	Minds	PowSybl	Thingsboard	wireapp	Blobkeeper	Review-service	Sunbirds	Doudochat	Sot	All
AddEntity	15	6	0	8	2	0	9	2	2	44
SplitAttribute	1	0	0	0	0	0	0	0	0	1
AddAttribute	21	35	4	30	2	0	27	11	4	134
AddRelationship	17	11	0	7	0	0	6	0	0	41
RemovePK	2	0	0	0	0	0	0	2	0	4
RemoveAttribute	2	5	0	2	0	0	11	1	1	22
AddPK	2	0	1	0	0	0	0	2	0	5
UpdateCardinality	0	0	0	0	0	1	0	0	0	1
AddWeakEntity	0	14	0	0	0	0	0	0	0	14
RemoveEntity	0	0	0	1	0	0	0	1	0	2
TOTAL Changes	60	71	5	48	4	1	53	19	7	268

Table 4
Version types in each project.

Project	Same	I	II-A	II-B	II-C	TOTAL
Minds	10	9	4	0	2	25
PowSybl	14	0	0	0	2	16
Thingsboard	4	0	0	0	0	4
Wireapp	8	5	3	1	0	17
Blobkeeper	4	0	0	0	0	4
Review-serv	1	0	0	0	0	1
Sunbirds	13	1	0	0	0	14
Doudouchat	8	1	0	0	0	9
Sop	3	0	0	0	0	3
TOTAL	65	16	7	1	4	93
Percentage	69.9	17.2	7.5	1.1	4.3	100

to obtain the differences between them for each project version. We categorized the results of these comparisons as follows:

1. **Same results**: When both the repository schema and the CoDEvo schema are the same.
2. **Type I, More database structures**: When the schema generated by CoDEvo contains more database structures than the ones in repository schema. These differences were detected when adding both a new entity (conceptual model change "AddEntity") and a new relationship of this entity with another one (conceptual model change "AddRelationship") in the same version.
3. **Type II, Different database structures**: When new database structures in the repository schema and the CoDEvo schema are different. There are three possible Type II subtypes depending on the conceptual model change that triggers the difference:
   (a) *Type II-A*: New non-boolean attribute (conceptual model change "AddAttribute")
   (b) *Type II-B*: New boolean attribute (conceptual model change "AddAttribute")
   (c) *Type II-C*: New relationship (conceptual model change "AddRelationship").

Table 4 displays the number of repository versions in accordance with the aforementioned types.

In the following subsections we analyze the differences detected between the CoDEvo schema and the repository schema for both types.

4.2.1. Analysis of Type I schema differences

Type I results were detected when the CoDEvo schema contained more database structures than the repository schema. These results were only detected when both an entity and a relationship of this entity with another one are added in the same version. Both the CoDEvo schema and the repository schema contained a table for the relationship (Rules 9 and 10). However,

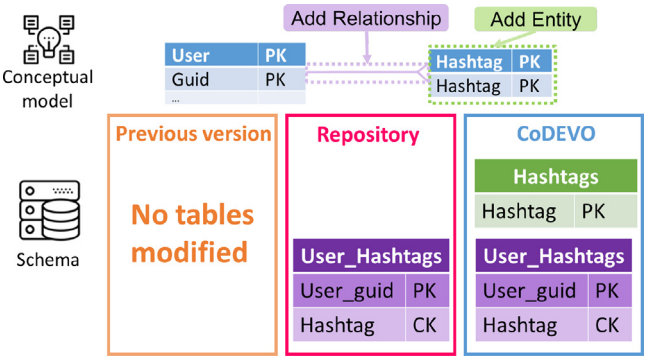

Fig. 5. CoDEvo schema vs repository schema version for the insertion of an entity and a relationship. Green: New table for adding an entity. Purple: New table for adding a relationship. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 5
Type II vs Same schema or Type I for the same conceptual model change.

Conceptual model changes	Type II	Same or I	All
AddAttribute	9	95	104
AddRelationship	4	38	42
TOTAL Changes	13	133	146
Percentage	8.90	91.1	100

CoDEvo additionally created a table to store only information of the entity (Rule 1).

Fig. 5 depicts the insertion of a new entity 'Hashtags' and its relationship with entity 'User' in a version of the project Minds, highlighting the differences between the schemas from the repository and CoDEvo.[10] Both the repository schema and the CoDEvo schema contain the table "User_Hashtags" for the new relationship. However, as described before, CoDEvo additionally added a table "Hashtags" for the new entity.

4.2.2. Analysis of Type II schema differences

There are three variants of Type II results: two for the addition of a new attribute that depends on the data type of the attribute: a non-Boolean attribute (Type II-A) or a Boolean attribute (Type II-B). The third variant is the addition of a new relationship (Type II-C). Table 5 displays the comparison of the number of these conceptual model changes that were detected in Type II results against Type I and same schema results.

It is important to note that in most additions of attributes or relationships the CoDEvo schema either was the same as the repository schema or contained more database structures. Only

[10] https://gitlab.com/minds/engine/-/commit/eaf3d4738ed216f931016cbb238baeae695eef6d

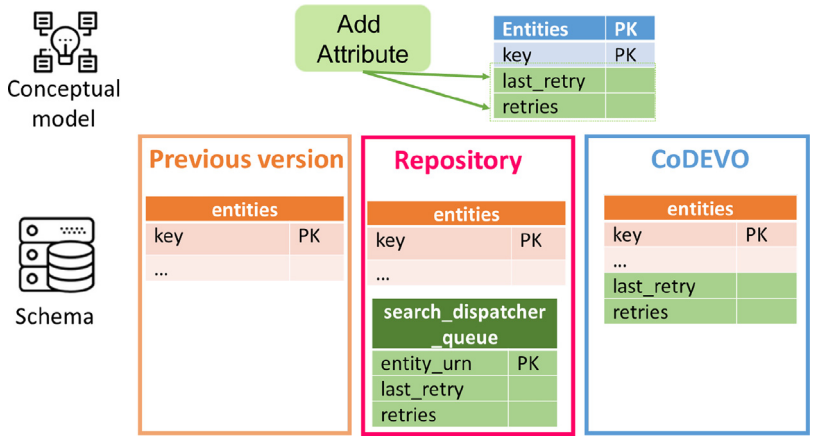

Fig. 6. Comparison between the CoDEvo schema and the repository schema for the addition of attributes "last_retry" and "retries" to entity "Entities". Orange: Table in previous version. Green: New table or columns for adding the attributes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

in **8.90%** of these changes in the conceptual model both schemas were different. In the following sub-sections we describe these differences depending on the Type II variant.

Analysis of Type II-A and Type II-B schema differences. Type II-A and II-B results were detected when adding a new attribute. In them, CoDEvo added a column mapped to the new attributes to the tables that store information about the entity of the attribute. However, the repository schema evolved by creating a new table to store this new attribute. There are two variants of this new table depending on the data type of the attribute: a non-Boolean attribute (Type II-A) or a Boolean attribute (Type II-B).

Type II-A schema differences: New non-Boolean attribute Type II-A results were detected when, for the addition of a non-Boolean attribute, the repository schema contained a table with the following columns: (1) key columns mapped to the primary key of the new attribute's entity and (2) a non-key column mapped to this attribute. Fig. 6 depicts the differences of the schemas of CoDEvo and the repository from a version of the project Minds where the attributes "last_retry" and "retries" were added to the entity "Entities".[11] To address this addition, the project developers added to the repository schema the table "search_dispatcher_queue", which contained a column mapped to the primary key of entity "Entities" as well as two columns for the new attributes. On the other hand, in the CoDEvo schema two columns with the same as the new attributes were added to the table "entities".

Type II-B schema differences: New Boolean attribute Only one Type II-B result was detected, specifically in a version of the project WireApp, where the Boolean attribute "whitelist" was added to the entity 'Team'. Fig. 7 depicts the differences between the schemas of CoDEvo and the repository in this version. In the repository schema, a table that only contains a column mapped to the primary key of entity 'Team' was added.[12] If the key value of a 'Team' is added to the table it means that "whitelist" is true for that particular 'Team'. On the other hand, CoDEvo added a column named 'Whitelist' to the table 'team', which stores information about the entity of the same name, in order to store the value of 'Whitelist' alongside the rest of the information of a particular 'team'.

Analysis of Type II-C schema differences: New relationship. For the addition of a new relationship, CoDEvo proposed the creation of a new table to store the information about the relationship. However, in two versions from project PowSybl and one from project 'Minds' the repository schema evolved by altering a table that initially stored information of one of the entities. After it was altered, the table stored information of the relationship. This alteration consisted of additions to the table columns which were mapped to the primary key of the other entity of the relationship.

Fig. 8 displays the differences between the schemas of CoDEvo and the repository from a version PowSybl for the addition of a relationship "one to many" between entities 'Busbreaker' and 'lccConverterStation'. CoDEvo created a new table ('Bus_lccConverterStation') to store information about this relationship.[13] On the other hand, in the version the schema evolved by adding the columns "Bus" and "ConnectableBus" to the table "lccConverterStation".

4.2.3. Conclusion for RQ1

From the results obtained by comparing the schemas of the repository and CoDEvo for each version (see Table 2), we were able to determine that 69.9% were the same schema (Type I). Another 17.2% of the total were Type I results, where CoDEvo added more database structures in addition to the ones that were in the repository schema. Altogether, the same schema or a Type I difference was obtained in 87.1% of the total of versions. This shows how CoDEvo was able to generate for these versions the same database structures as the human developers. The remaining results where the modifications of the schema were different (12.9%) were classified as Type II. In RQ2, we analyze if the schemas of CoDEvo are still valid regarding the project requirements when they differ from the repository schema, and, if they are valid, whether they are better or worse than the repository schema.

4.3. RQ2: Are the CoDEvo schemas valid considering the project requirements?

To respond to this research question, we discuss to what extent the differences of database structures between the schema of CoDEvo and the repository affect the following:

[11] https://gitlab.com/minds/engine/-/commit/5262cdb391b0283e95d3ed09fa90bf16dd39938f

[12] https://github.com/wireapp/wire-server/commit/a7586ec34b5844648e78c62cd751370a5567f265

[13] https://github.com/powsybl/powsybl-network-store/commit/3c962d5c8f78f56c7be6e7729be2a2ca910c2bcf

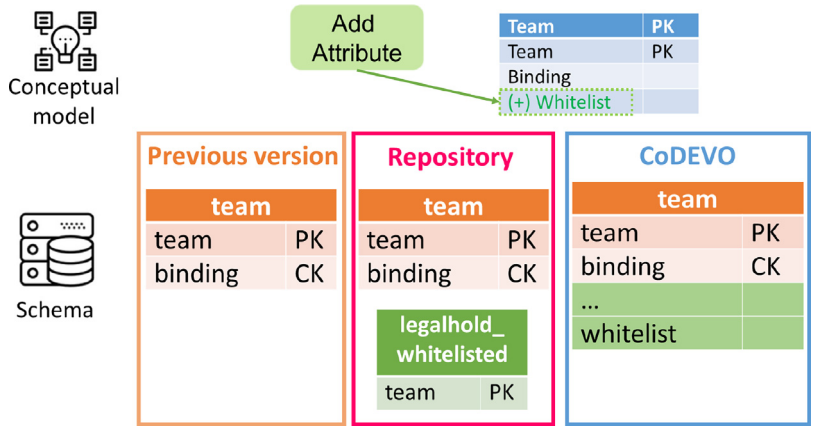

Fig. 7. Comparison between the CoDEvo schema and repository schema from the project WireApp for the addition of a boolean attribute "whitelist;; to entity 'Team'. Orange: Table in previous version. Green: New table or column for adding the attribute. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

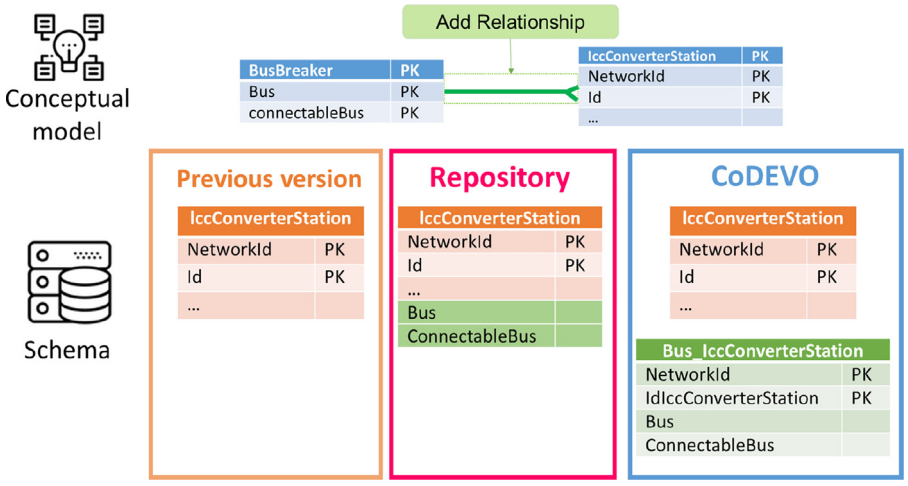

Fig. 8. CoDEvo schema vs repository schema of Type II version from project PowSybl for inserting a new relationship between entities BusBreaker and IccConverterStation. Orange: Table in previous version. Green: New table or columns for adding a relationship. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

1. Fulfilling of the project query requirements: We determine if the CoDEvo schemas fulfill the project requirements of a version by checking if it is possible to query the same data as in the repository schemas. This is done by determining the possible ways of querying data from the repository schema and checking if the CoDEvo schema allows them
2. Maintenance of data integrity: We compare the database operations (`INSERT`, `DELETE`, `UPDATE`) that developers must implement in the client application to maintain the data integrity in the CoDEvo schema against the database operations required in the repository schema.

4.3.1. Differences regarding requirements of the queries

In this subsection we determine if the CoDEvo schema satisfies query requirements of a project version when it differs from the repository schema. In the comparison performed for RQ1, Type I results meant that the database structures from the repository were also in the CoDEvo schema, therefore CoDEvo was able to satisfy the requirements as well and they do not need to be analyzed either. However, Type II results showed that the schemas of CoDEvo and the repository contained different database structures. Table 5 details how many times the CoDEvo schema was different than the repository schema (Type II) compared to how many times it was the same or it contained additional tables. In the following paragraphs we analyze if in each of the Type II subtypes the schema of CoDEvo still satisfies the project requirements.

Type II-A and Type II-B results are obtained when adding a new attribute. In both subtypes, CoDEvo added a new column to a table that stored information about the attribute's entity, while in the repository schema a new table was created. A new column for storing data of the attribute fulfills the requirements of considering the new attribute, as it associates this data with the instance of the entity that it belongs to. On the other hand, if a table is designed by the developers with a different primary key in order to query data in a different way, the CoDEvo schema would not allow the same queries as the repository schema. However, this scenario was not detected in any of the researched projects.

We have also researched the queries executed by the client applications from the project presented in Figs. 6 and 7 to reinforce the decisions that are made by CoDEvo for evolving the schema.

In the scenario from Fig. 6 the information is queried through the values of the PK from 'Entities' which is named 'key' in the table 'entities' and 'entity_urn' in the table 'search_dispatcher_queue'.[14] These queries can be executed against the CoDEvo

[14] https://gitlab.com/minds/engine/-/blob/5262cdb391b0283e95d3ed09fa90bf16dd39938f/Core/Search/RetryQueue/Repository.php

schema by querying the data against the table 'entities' instead of the table 'search_dispatcher_queue'.

In the scenario from Fig. 7 the application checks through the primary key of 'team' if that particular 'team' is whitelisted.[15] The same information can be queried in the CoDEvo schema by checking if the column 'whitelist' stores the value 'true' for a given 'team' value.

Type II-C results were obtained for some of the versions that added a new relationship. CoDEvo added a new table to store information of the new relationship. Note that this option was also chosen by the repository schema in 37 out of the 41 new relationships inserted into the projects. In the other 4 versions where the schema from CoDEvo and the repository are different, the schema provided by CoDEvo did not allow the execution of all possible queries that could be executed against the repository schema. For instance, in the repository schema in Fig. 8 the altered table "lccConverterStation" from the human developers allows to query more data in a single query than the table "Bus_lccConverterStation" obtained by CoDEvo, as the later contains less non-key columns.

The queries executed in the scenario presented in Fig. 8 by the client application filter the data through the PK values of the entity 'lccConverterStation'.[16] The same queries can be executed using the CoDEvo schema, although as we described in the previous paragraph, there is less data returned as the CoDEvo schema table contains less non-key columns.

In conclusion, CoDEvo schemas were able to satisfy the requirements regarding storage of the new attribute or relationship. It is also noteworthy that in most versions, the CoDEvo schema and the repository schema are the same. Of the remaining versions, only the versions with Type II-C results (33.33% of the total of versions with Type II results) show the CoDEvo schema as less suitable than the repository schema for potential queries that could be implemented in the client application due to new requirements.

4.3.2. Differences regarding data integrity maintenance

When there is an update of data implemented in a client application, it may require additional database statements to maintain data integrity in the database. These statements depend on the characteristics of the schema, such as the duplication of data among the tables. In this section, we describe the differences between the schemas of CoDEvo and the repository regarding the database statements that must be implemented in the client applications for updates of data. As Type I results show that both schemas are the same, we only analyze the differences when there was a Type I or Type II result.

Versions with Type I results. Type I results were detected for the addition of both an entity and a relationship in the same version. In these versions, the repository schema contained a table to store the relationship, while, in addition to this table, the CoDEvo schema also contained a table to only store data of the new entity. This additional table increases the development cost, as a new table must be considered when implementing changes in the data, making the CoDEvo schema more costly than the repository one. On the other hand, an additional table does not negatively affect the execution time of data update operations, as only a few database statements are required for updating data in this additional table.

This means that the only cost increase of a new table for storing data of the entity is during the development of the application. On the other hand, this additional table has the advantage of expanding the ways of querying data in the database. As future work, for users who do not want this additional table, we could incorporate interactivity by asking if users prefer a table to store information of the entity or a table that stores the relationship.

Versions with Type II-A and Type II-B results. Type II results for adding an attribute showed that, while CoDEvo altered an existing table by adding a column for the new attribute, in the repository schema a new table was added. If we compare them regarding development cost, performance and query possibilities we obtain the following:

Development cost: The CoDEvo schema is less costly than the ones from the versions as it avoids the creation of a new table. The only changes required in the client application for the CoDEvo schema are those of adding the new attributes in the existing queries of the new tables. On the other hand, for the repository schema, all the code related to the new table must be created from scratch.

Performance: Both schemas are similar, although the CoDEvo schema has the better performance. The CoDEvo schema should be similar in performance to the previous version schema, as it only requires the consideration of new columns in data operations that already existed in the client application. On the other hand, the repository schema is marginally more time consuming due to requiring execution of new database statements for the new table.

Regarding data integrity maintenance, the CoDEvo schema is better in both development cost and performance than the repository schemas. Developers would require less time to update the client application and this application would also have better performance when executing operations against the database.

Versions with Type II-C results. Type II-C results were detected when adding a new relationship generating CoDEvo a new table. On the other hand in the repository schema an existing table was altered. If we compare the development cost, performance and query possibilities we obtain the following:

Development cost: The repository schema avoid the creation of a new table, which might make it less costly than the CoDEvo schema. However, in this case we also need to consider that in the repository schema the purpose of the altered table is modified from querying information of one entity to querying information of a relationship. This means that all the code related to that table must be modified in order to include information of the other entity of the relationship. Therefore, the development costs for the repository schema might be higher than for the CoDEvo schema depending on the amount of code that needs to be modified.

Performance: The repository schema would have better performance due to not requiring a new table to consider when executing data operations. It is the opposite scenarios that was described for Type II-A and Type II-B results, where the CoDEvo schema had a better performance for the same reasons.

4.4. Threats to validity

We evaluated CoDEvo with several case studies, comparing the database structures proposed by CoDEvo against the database structures that are actually implemented in the repository schemas. However, there are several threats to the validity of this experimentation. In this subsection we discuss the threats to external and internal validity.

Threats to external validity: 6 out of the 9 projects that we used for the experimentation were also used to define the transformation rules of CoDEvo. These projects were named in prior

[15] https://github.com/wireapp/wire-server/blob/develop/services/galley/src/Galley/Cassandra/Queries.hs

[16] https://github.com/powsybl/powsybl-network-store/blob/3c962d5c8f78f56c7be6e7729be2a2ca910c2bcf/network-store-server/src/main/java/com/powsybl/network/store/server/NetworkStoreRepository.java

work where we proposed the main idea of CoDEvo (Suárez-Otero et al., 2020). This is a threat to the validity of the experimentation, as the CoDEvo schemas might always be the same as the repository schemas, artificially increasing the effectiveness of CoDEvo generating appropriate schemas artificially. To mitigate this threat, we added more projects to be used in the analysis: Sunbirds, Doudouchat and Sop. The results that we obtained in these three projects were successful, although we are searching for more projects to analyze in order to further mitigate this threat.

Threats to internal validity: The projects that we used for the experimentation do not provide an explicit conceptual model. We needed to infer the conceptual model from these projects by analyzing the database structures of the repository schema. The same was required to obtain the conceptual model changes, which were obtained by analyzing the evolution of the schema in each version. This manual determination is a threat as we could have defined these conceptual models and conceptual model changes to perfectly fit into the transformation rules defined in CoDEvo. In order to avoid this, we carefully analyzed each version to obtain the equivalent conceptual model. Then, we compared one version against the previous one to obtain what was changed, identifying the conceptual model changes applied in the version. There are also techniques that can be used to obtain the conceptual model from a column family DBMS schema (Mior and Salem, 2018; Sevilla Ruiz et al., 2015). Using these external techniques, we could also ensure that the conceptual models that we use in the experimentation are not biased towards the transformations defined in CoDEvo. Note that these techniques can obtain different conceptual models as they have different ways of obtaining them.

5. Related work

Schema evolution work has traditionally focused on relational databases with approaches that address issues such as the evolution of the integrity constraints (Curino et al., 2010) and foreign key (Vassiliadis et al., 2019), or guidelines to evolve relational schemas (Delplanque et al., 2020). Another topic that is addressed is how the database schema and the data must evolve after an ontology change (Noy and Klein, 2004), similar to what we propose in this work. However, these works are focused on relational databases, making their use for other systems such as NoSQL databases more difficult. For instance, column family DBMSs do not have relationships between tables, making the works about the evolution of the integrity constraints and foreign keys unfit for these kinds of databases. Even so, we used the ontology from Noy and Klein (2004) as one of the sources for potential conceptual changes to be addressed in our work, focusing in our work on how these conceptual model changes affect a column family schema.

For NoSQL databases, Scherzinger et al. have analyzed different topics related to evolution on NoSQL databases (Hillenbrand et al., 2019, 2021; Möller et al., 2020; Scherzinger et al., 2013; Störl et al., 2018, 2020). They have approached data migrations from direct changes of the schema (Scherzinger et al., 2013; Störl et al., 2020), while we focused on changes in the conceptual model. They have developed the tool Migcast (Hillenbrand et al., 2019), which advises the developers in the process of database evolution by recommending different data migration strategies focusing on the financial cost. These strategies can be determined as they are able to predict the financial cost, depending on the cloud provided that is chosen. They have also researched how to self-detect the optimal migration of data when the evolves (Hillenbrand et al., 2021). Similar to our maintenance of the inter-model consistency for column family DBMSs, Möller et al. have addressed the evolution of document-oriented databases after a change in the conceptual model (Möller et al., 2020).

The conceptual model is used for the design of column family DBMSs as supported by several schema design methodologies (Chebotko et al., 2015; de la Vega et al., 2020; Mior et al., 2017) that propose its use in the database schema design process. Chebotko et al. (2015) were the first that proposed its use to create a logical model following a set of data modeling principles, mapping rules, and mapping patterns. Mior et al. (2017) continued with this approach by also incorporating statistical information about query frequency and expected data volume in developing the NoSE tool. de la Vega et al. (2020) used the previous two works as a starting point to create the Mortadelo tool, which generalized their approaches and added a design strategy for document oriented-databases.

Chillón et al. (2021) have addressed schema evolution for NoSQL schemas, focusing on a general approach for both MongoDB and Apache Cassandra. They provide a taxonomy named Orion of several conceptual model changes, defining how these changes are performed at the conceptual level. Additionally, they provide a brief description of how some of these changes are applied in both MongoDB and Cassandra. This taxonomy is applied to U-Schema (Candel et al., 2022), a unified metamodel to represent the schema of all types of databases. This allows the evolution of any database schema by applying the changes defined through Orion in a model that conforms to U-Schema. The main differences with our approach are that we focus on only column family DBMSs, providing more detail in the changes that must be performed in the database schema to consider a given type of conceptual model change. These types were detected in both real projects and evolution approaches focused on other databases. We have also performed an empirical experimentation of CoDEvo, comparing its output schemas with the schemas generated by real life developers.

Another group has also researched schema design and data evolution focusing on multi-model databases (Svoboda et al., 2021). They propose a transformation algorithm that converts through transformations an ER model into a model of any of all known data models, which they name “schema category”. From a schema category they create “instance categories”, which represent a particular database state with data. They extended their previous work by providing more detail into the transformations and the specifications of the implementation of the approach (Koupil and Holubová, 2022). In another work (Koupil et al., 2022) they describe the tool MM-evocat, which addresses the evolution of the schema, supporting so far PostgreSQL (relational database) and MongoDB (document database). This last work is the most related to our research, although so far they have not addressed evolution of column family DBMSs.

The queries have also been considered to evolve the schema in order to improve the performance of the database (Benats et al., 2022). In this work they propose an approach that analyzes the history of queries executed by the application and provides a recommendation of how to evolve the schema in order to improve the performance of the queries.

Higher-order transformations (HOTs) are widely used in several works such as in the proposal of García et al. (2012) where they handle the transformations as models using HOTs to evolve a given transformation. This evolution is done by taking changes in the model to be transformed that they detected in a previous stage. To the best of our knowledge, our approach is the first one that uses HOTs to evolve NoSQL databases.

6. Conclusions and future work

In this work we have proposed CoDEvo, an MDE approach addressing maintenance of inter-model consistency between the conceptual model and the schema for conceptual model changes.

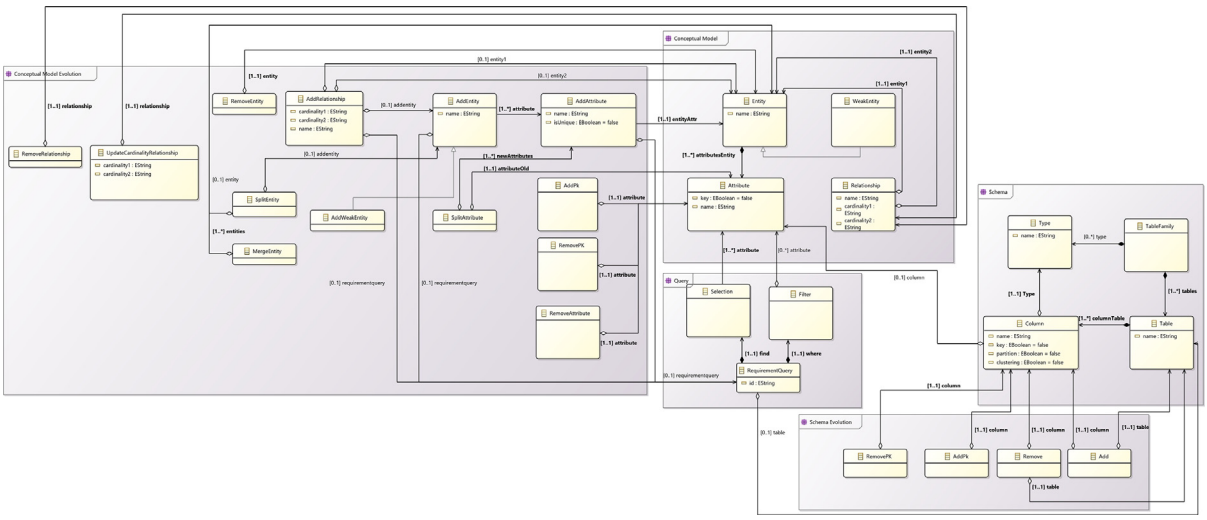

Fig. A.9. Relationships between the metamodels Conceptual Model Evolution, Conceptual Model, Query, Schema and Schema change.

We have focused on addressing both changes detected in real software projects and changes addressed for evolving the schema of other databases from scientific papers. Then, we provided an automatic solution to correctly evolve the database based on model transformations. We validated these transformations through their application for the conceptual model changes detected in real-world projects, obtaining successful results.

CoDEvo was able in most versions to provide a schema that was the same or equivalent to the repository schema created by the project developers. Only in a minority of versions analyzed during the experimentation (12.9%) were the database structures proposed by CoDEvo different. In these versions with different schema, it is possible to query the exact same data using the CoDEvo schema in 66.67% of them. Through the experimentation, we were able to check that CoDEvo was able to avoid the problems related to schema evolution, providing a consistent evolution proposal for the schema.

As future work, we want to devise a model driven engineering approach focusing on how the data must be migrated after the schema is modified in order to maintain data integrity. To achieve this, we plan to use our knowledge from prior work where we addressed the maintenance of data integrity following changes in the data from a conceptual model point of view (Suárez-Otero et al., 2019; Suárez-Otero González et al., 2019; Suárez-Cabal et al., 2023). In these works, we developed a technique to prevent integrity loss (Suárez-Otero et al., 2019; Suárez-Cabal et al., 2023) as well as an oracle that checks if a column family DBMS ensures data integrity (Suárez-Otero González et al., 2019). Another problem that we will address is the evolution of the application queries when the model or the schema changes, which we have not approached in this work.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

Funding: This work was supported in part by projects [TIN 2016-76956-C3-1-R] funded by the Spanish Ministry of Economy and Competitiveness, [PID2019-105455GB-C32] funded by MCIN/ AEI/10.13039/501100011033 and the Severo Ochoa predoctoral grant [PA-21-PF-BP20-184] funded by the Principality of Asturias.

Appendix. Metamodel with relationships

See Fig. A.9.

References

Apache Foundation, 2008. Apache cassandra. https://cassandra.apache.org. (Accessed 16 February 2022).

Benats, P., Meurice, L., Gobert, M., Cleve, A., 2022. Query-based schema evolution recommendations for hybrid polystores. In: Proceedings of the 41st International Conference on Conceptual Modeling (ER 2022), Forum Track.

Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A., 2006. Model transformations? transformation models! In: International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 440–453, https://doi.org/10/bbf5wc.

Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., Voigt, H., 2019. Schema validation and evolution for graph databases. In: International Conference on Conceptual Modeling. Springer, pp. 448–456, https://doi.org/10/gmnv7f.

Candel, C.J.F., Ruiz, D.S., García-Molina, J.J., 2022. A unified metamodel for NoSQL and relational databases. Inf. Syst. 104, 101898. http://dx.doi.org/10.1016/j.is.2021.101898.

Carpenter, J., Hewitt, E., 2020. Cassandra: the Definitive Guide: Distributed Data at Web Scale. O'Reilly Media, http://dx.doi.org/10.5555/3006375.

Chebotko, A., Kashlev, A., Lu, S., 2015. A big data modeling methodology for apache cassandra. In: 2015 IEEE International Congress on Big Data. IEEE, pp. 238–245. http://dx.doi.org/10.1109/BigDataCongress.2015.41.

Chillón, A.H., Ruiz, D.S., Molina, J.G., 2021. Towards a taxonomy of schema changes for NoSQL databases: the orion language. In: International Conference on Conceptual Modeling. Springer, pp. 176–185, https://doi.org/10/js5q.

Curino, C.A., Moon, H.J., Deutsch, A., Zaniolo, C., 2010. Update rewriting and integrity constraint maintenance in a schema evolution support system: Prism++. Proc. VLDB Endow. 4 (2), 117–128. http://dx.doi.org/10.14778/1921071.1921078.

de la Vega, A., García-Saiz, D., Blanco, C., Zorrilla, M., Sánchez, P., 2020. Mortadelo: Automatic generation of NoSQL stores from platform-independent data models. Future Gener. Comput. Syst. 105, 455–474. http://dx.doi.org/10.1016/j.future.2019.11.032.

Delplanque, J., Etien, A., Anquetil, N., Ducasse, S., 2020. Recommendations for evolving relational databases. In: International Conference on Advanced Information Systems Engineering. Springer, pp. 498–514, https://doi.org/10/js5g.

García, J., Diaz, O., Azanza, M., 2012. Model transformation co-evolution: A semi-automatic approach. In: International Conference on Software Language Engineering. Springer, pp. 144–163, https://doi.org/10/js5h.

Herrmann, K., Voigt, H., Behrend, A., Rausch, J., Lehner, W., 2017. Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 1101–1116. http://dx.doi.org/10.1145/3035918.3064046.

Hillenbrand, A., Levchenko, M., Störl, U., Scherzinger, S., Klettke, M., 2019. MigCast: putting a price tag on data model evolution in NoSQL data stores. In: Proceedings of the 2019 International Conference on Management of Data. pp. 1925–1928. http://dx.doi.org/10.1145/3299869.3320223.

Hillenbrand, A., Störl, U., Nabiyev, S., Klettke, M., 2021. Self-adapting data migration in the context of schema evolution in NoSQL databases. Distributed and Parallel Databases 1–21, https://doi.org/10/js5k.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., 2008. ATL: A model transformation tool. Sci. Comput. Program. 72 (1–2), 31–39. http://dx.doi.org/10.1016/j.scico.2007.08.002.

Koupil, P., Bártík, J., Holubová, I., 2022. MM-evocat: A tool for modelling and evolution management of multi-model data. In: Proceedings of the 31st ACM International Conference on Information & Knowledge Management. pp. 4892–4896. http://dx.doi.org/10.1145/3511808.3557180.

Koupil, P., Holubová, I., 2022. A unified representation and transformation of multi-model data using category theory. J. Big Data 9 (1), 1–49, https://doi.org/10/js5x.

Liu, B., Yu, X.L., Chen, S., Xu, X., Zhu, L., 2017. Blockchain based data integrity service framework for IoT data. In: 2017 IEEE International Conference on Web Services. ICWS, IEEE, pp. 468–475. http://dx.doi.org/10.1109/ICWS.2017.54.

MDA, OMG, 2008. Object management group model driven architecture.

Mens, T., Van Gorp, P., 2006. A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. 152, 125–142. http://dx.doi.org/10.1016/j.entcs.2005.10.021.

Mior, M.J., Salem, K., 2018. Renormalization of NoSQL database schemas. In: International Conference on Conceptual Modeling. Springer, pp. 479–487, https://doi.org/10/js5r.

Mior, M.J., Salem, K., Aboulnaga, A., Liu, R., 2017. NoSE: Schema design for NoSQL applications. IEEE Trans. Knowl. Data Eng. 29 (10), 2275–2289. http://dx.doi.org/10.1109/TKDE.2017.2722412.

Möller, M.L., Klettke, M., Störl, U., 2020. EvoBench–a framework for benchmarking schema evolution in NoSQL. In: 2020 IEEE International Conference on Big Data. Big Data, IEEE, pp. 1974–1984. http://dx.doi.org/10.1109/BigData50022.2020.9378278.

Moniruzzaman, A., Hossain, S.A., 2013. NoSQL database: New era of databases for big data analytics-classification, characteristics and comparison. arXiv preprint arXiv:1307.0191.

Noy, N.F., Klein, M., 2004. Ontology evolution: Not the same as schema evolution. Knowl. Inf. Syst. 6 (4), 428–440, https://doi.org/10/bt5548.

Scherzinger, S., Klettke, M., Störl, U., 2013. Managing schema evolution in NoSQL data stores. arXiv preprint arXiv:1308.0514.

Sevilla Ruiz, D., Morales, S.F., García Molina, J., 2015. Inferring versioned schemas from NoSQL databases and its applications. In: International Conference on Conceptual Modeling. Springer, pp. 467–480, https://doi.org/10/js5s.

Störl, U., Klettke, M., Scherzinger, S., 2020. NoSQL schema evolution and data migration: State-of-the-art and opportunities. In: EDBT. pp. 655–658. http://dx.doi.org/10.5441/002/edbt.2020.87.

Störl, U., Müller, D., Tekleab, A., Tolale, S., Stenzel, J., Klettke, M., Scherzinger, S., 2018. Curating variational data in application development. In: 2018 IEEE 34th International Conference on Data Engineering. ICDE, IEEE, pp. 1605–1608. http://dx.doi.org/10.1109/ICDE.2018.00187.

Suárez-Cabal, M.J., Suárez-Otero, P., de la Riva, C., Tuya, J., 2023. MDICA: Maintenance of data integrity in column-oriented database applications. Comput. Stand. Interfaces 83, 103642. http://dx.doi.org/10.1016/j.csi.2022.103642.

Suárez-Otero, P., Mior, M.J., Suárez-Cabal, M.J., Tuya, J., 2020. Maintaining NoSQL database quality during conceptual model evolution. In: 2020 IEEE International Conference on Big Data. Big Data, IEEE, pp. 2043–2048. http://dx.doi.org/10.1109/BigData50022.2020.9378228.

Suárez-Otero, P., Mior, M.J., Suárez-Cabal, M.J., Tuya, J., 2021. An integrated approach for column-oriented database application evolution using conceptual models. In: International Conference on Conceptual Modeling. Springer, pp. 26–32, https://doi.org/js5t.

Suárez-Otero, P., Suárez-Cabal, M.J., Tuya, J., 2019. Leveraging conceptual data models to ensure the integrity of cassandra databases. J. Web Eng. https://doi.org/10/js5n.

Suárez-Otero González, P., Suárez Cabal, M.J., Tuya González, P.J., et al., 2019. Verificación del mantenimiento de la consistencia lógica en bases de datos Cassandra. Jornadas Ing. Softw. Bases Datos (JISBD)(24ª. 2019. Cáceres) URL http://hdl.handle.net/11705/JISBD/2019/037.

Svoboda, M., Čontoš, P., Holubová, I., 2021. Categorical modeling of multi-model data: one model to rule them all. In: International Conference on Model and Data Engineering. Springer, pp. 190–198, https://doi.org/10/js5w.

Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J., 2009. On the use of higher-order model transformations. In: European Conference on Model Driven Architecture-Foundations and Applications. Springer, pp. 18–33, https://doi.org/10/d2qf84.

Vassiliadis, P., Kolozoff, M.-R., Zerva, M., Zarras, A.V., 2019. Schema evolution and foreign keys: a study on usage, heartbeat of change and relationship of foreign keys to table activity. Computing 101 (10), 1431–1456, https://doi.org/10/js5p.

Pablo Suárez-Otero received his B.Sc. degree in Computer Engineering in 2015 and in M.Sc. in Computer Engineering in 2017 from the University of Oviedo. He received his Ph.D. from the University of Oviedo in 2023. He is currently a post-doc researcher in Computing at the University of Oviedo as well as an Assistant Professor at the University of Oviedo. He is a member of the Software Engineering Research Group (GIIS, giis.uniovi.es). His research interests include software testing, NoSQL databases and data modeling.

Michael J. Mior received a Masters degree from the University of Toronto in 2011. After spending a few years at a startup company, he received his PhD from the University of Waterloo in 2018. Michael then joined the Computer Science Department at the Rochester Institute of Technology as an Assistant Professor. His research interests include schema design, understanding, and integration for non-relational semistructured data. He and his students build tools to improve understanding and management of this type of data.

María José Suárez-Cabal is an assistant professor at the University of Oviedo, Spain, and is a member of the Software Engineering Research Group (GIIS, giis.uniovi.es). She obtained her Ph.D. in Computing from the University of Oviedo in 2006. Her research has focused on software testing, and more specifically on testing SQL and database applications, being published in high impact international journals and conferences. Her current research interests also include testing NoSQL systems.

Javier Tuya received the Ph.D. degree in engineering from the University of Oviedo, Oviedo, Spain, in 1995. He is a Professor with the University of Oviedo, Oviedo, Spain, where he is the Research Leader of the Software Engineering Research Group. He is the Director of the Indra-Uniovi Chair and worked in the development of the new ISO/IEC/IEEE 29119 Software Testing Standard as member of the ISO WG26 Working Group and as Convener of the corresponding UNE National Body Working Group. His research interests in software engineering software testing for database applications and system testing.