Draft: Technical Report
Formal Semantics of NoSQL Evolution Operations for Different Data Heterogeneity Classes

Mark Lukas Möller\textsuperscript{1}, Meike Klettke\textsuperscript{2}, Uta Störl\textsuperscript{3}

Abstract:
An evolution of a NoSQL database consists of two parts, an evolution of its schema and migrating datasets according to this new schema version. The applied migration operations have to consider the characteristics of the NoSQL source data. In this article, we define the semantics of evolution and data migration operations and their inverse operations, distinguishing between different heterogeneity classes (ranging from regular datasets up to completely unstructured NoSQL datasets). We are going to show the consequences for NoSQL query rewriting, for handling of a lazy NoSQL migration, and we sketch the consequences for a NoSQL migration adviser which proposes a suitable migration strategy for a concrete scenario.

Keywords: NoSQL Schema Evolution, Data Heterogeneity Classes, Query Rewriting

1 Introduction

All successful software products underlie permanent changes. This entails frequent evolutions of data structures and the necessity to adapt data onto the new structures. In database research, schema evolution for relational databases has been studied in detail; in [Ro92], several publications on this matter have been collected. The development of a similar approach for schema evolution and data migration for NoSQL databases becomes much more complicated due to the structural heterogeneity of the input datasets.

Most NoSQL database systems are schemaless, they do not predefine structures and semantic constraints. This is the reason why these systems can be applied for storing homogeneous structured data as well as heterogeneous data. In homogeneous structured NoSQL databases, all datasets have exactly the same structure and certain integrity constraints hold. In this case, the application generates the NoSQL datasets and the application logic guarantees the validity of these datasets. However, in heterogeneous NoSQL databases, datasets with

\textsuperscript{1} University of Rostock, mark.moeller2@uni-rostock.de
\textsuperscript{2} University of Rostock, meike.klettke@uni-rostock.de
\textsuperscript{3} University of Applied Sciences Darmstadt, uta.stoerl@h-da.de
different structures are stored in the same collection. A NoSQL database evolution method must be able to handle all variants of input datasets that can be available in the different classes of NoSQL databases.

Before we introduce the semantics of the evolution operations, we classify the different degrees of NoSQL heterogeneity. Figure 1(a) visualizes three independent dimensions that have to be considered.

![Three dimensions of the NoSQL Heterogeneities](image)

(a) Three dimensions of the NoSQL Heterogeneities

![Heterogeneity Layers of the Evolution Operations](image)

(b) Heterogeneity Layers of the Evolution Operations

Fig. 1: NoSQL Heterogeneity Classes

Our evolution language defines two multi-entity operations, Move and Copy. Both operations specify matching conditions between entities. Because the NoSQL databases do not check semantic constraints in advance, we first have to distinguish whether all properties have a matching partner, or whether there are dangling tuples (x-axis in Figure 1(a)). The next dimension is the cardinality between two entities in case of a Move or a Copy operation (y-axis). The last dimension regards the heterogeneity of entities of the same version. Here we are distinguishing between datasets in which all entities of the same version have homogeneous or heterogeneous structures (z-axis in Figure 1(a)). These three dimensions influence the semantics. We are deriving different heterogeneity classes (HC), starting from the most structured up to unstructured datasets.

**HC1:** In this class, the database can contain datasets in different structural versions, but all datasets in the same version have exactly the same structure. Further, we can assume 1:1 cardinalities and no dangling tuples between two entity types of the matching conditions.

**HC2:** The second class are datasets with the same structure in each version. This class deals with arbitrary cardinalities. Dangling tuples can occur and have to be considered during data migration.

**HC3:** The third class are NoSQL databases that can have different structures within the same version. Here, optional properties (that are available in some entities of a concrete structural version and are missing in other entities of the same version) can occur. NoSQL databases allow these heterogeneities even if an explicit schema
description is available. In this case, there are optional properties in a JSON schema, the set of belonging JSON datasets belong to this class.

If we correlate these NoSQL heterogeneity classes with relational normal forms, we can consider the heterogeneity class 1 (HC1) as normal form -1, because in contrast to relational databases, we combine datasets in different structural versions within the same collection. Accordingly, HC2 can be seen as normal form -2 and HC3 as normal form -3.

We want to motivate with a small and simple example, why heterogeneity influences the process of data migration in NoSQL. In Figure 2, three entities in version 1 are given. Their structure is heterogeneous (HC3), two entities have an attribute points and the third has an attribute score instead. By applying a Rename operation and migrating the entities, the entities in version 2 are generated, which are given on the right-hand side of Figure 2.

With an eager migration all entities are immediately updated and are stored in the NoSQL database in version 2. In this case, the following sample query can be executed and assumes that all entities are in version 2.

```
Select * From Player Where score>100 And version=2
```

It delivers as a result the entities 2.2 and 2.3.

In case of lazy migration, entities are still in version 1 and are only migrated if they are accessed. A query has to be rewritten for selecting the entities for migration. Applying the Rename operation onto the above given query, we could apply the following query instead:

```
Select * From Player Where points>100 And version=1
```

In this case, we get the entity 1.2 as the result, the entity 1.3 is getting "lost."

In query rewriting the structural variants for heterogeneous datasets have to be considered. Instead, the following query has to be executed:
Select id, name, points From Player
Where (points > 100 and version = 1)
Union Select id, name, score As points From Player
Where (score > 100 And version = 1)

Only with this query, we get the complete result set 1.2 and 1.3 for datasets of HC3. We see that even for this simple Rename operation query rewriting that takes older versions into account is not easy to realize. And indeed, most evolution steps of real applications have to deal with even more complex schema changes.

We can conclude that query rewriting for querying versioned data has to consider structural heterogeneities of the input datasets. The semantics of schema evolution, the associated data migration operations, and their inverse operations for the different heterogeneity classes have to be defined.

The article makes the following contribution.

- We introduce three heterogeneity classes (HC1-3) for NoSQL and define the semantics of the NoSQL evolution operations and data migrations operations for each HC, and their reverse operations in Section 2.
- We show in Section 3 what query rewriting for the different HCs entails.
- Finally, we sketch the consequences for a data migration adviser in Section 5 and conclude the advantages of a schema management component for handling evolving structures of NoSQL datasets.

2 Semantics of the Evolution Operations

The NoSQL evolution language consists of the three single-type operations, Add, Delete and Rename and of the two multi-type operations, Move and Copy. The operations are defined for the evolution of the schema. Data migration operations can be derived from the NoSQL evolution operations. These operations are used to migrate affected entities into the latest structural version. We start to define the semantics of the operations on regular structures and structured datasets, and we will extend them later to irregular structures and heterogeneous datasets. The effort for the data migration increases for each of these HCs. In order to define the concrete HC, pre- and postconditions are used to check the regularity of the data. The pre- and postconditions are inspired by the Hoare triple.

At first, we start with some definitions. Data with an equal or similar set of properties is called a kind. A kind consists of a schema and of a set of entities. The definition for a kind named $A$ is $\mathcal{K}_A = (S_A, E_A)$.

The schema $S_A$ is defined as a set of property-names, formally $S_A = \{A_1, \ldots, A_n\}$.

The set of entities $E_A$ of kind $A$ over the schema $S_A$ is defined as $E_A := \{e_1, \ldots, e_m\}$ where $m$ is the number of entities in the set of entities and where each entity $e_i$ in $E_A$
consists of up to \( n \) attributes called \( a_{ij} \) with \( i \in \{1, \ldots, m\} \) and \( j \in \{1, \ldots, n\} \). Formally, \( e_i = \{a_{ij} \mid j \in \{1, \ldots, n\}\} \).

Here, \( i \) represents the index for the \( i \)-th entity of \( E_A \) and \( j \) is the \( j \)-th attribute of the corresponding entity. Each attribute \( a_{ij} \) consists of an attribute name and and attribute value: \( a_{ij} = (A_{ij} : v_{ij}) \in S_{A_i} \times D(A_i) \), whereby \( S_{A_i} \subseteq S_A \) and \( D(A_i) \subseteq D(A) \). Here, \( S_{A_i} \times D(A_i) \) represents the domain of the property.

**Example.** To illustrate the definitions, let us consider an example for the representation of customers. The kind is called \( \text{Player} \) and is defined as \( \mathcal{K}_\text{Player} = \{S_{\text{Player}}, E_{\text{Player}}\} \). The schema for a player contains the attributes "id", "name", and "points":

\[
S_{\text{Player}} = \{("id", "name", "points")\}
\]

A set of possible entities is

\[
E_{\text{Player}} = \{(\{("id": 1),("name": "Tanja"),("points": 80)\},
                \{("id": 2),("name": "Hannes"),("points": 130)\})\}
\]

Here, \( E_{\text{Player}} \) is the set of entities that contains two entities (\( e_1 \) and \( e_2 \)) of the kind \( \text{Player} \) each with three attributes. The first property of the first entity is \( a_{11} = (A_{11} : v_{11}) = ("id" : 1) \).

All property names are also available in schema \( S_{\text{Player}} \).

For the definition of the evolution operations, we often have to check whether an entity contains an attribute with a certain attribute name without respect to its value. Because properties are stored as a tuple and not as a set, we define an own operator called \( \varepsilon^+ \) that checks if there is a property available in a given entity or not. For this purpose, we define a projection operation which projects onto the property name:

\[
\pi_A := S_{A_i} \times D(A_i) \rightarrow S_{A_i} \text{ with } (A_{ij}, v_{ij}) \mapsto A_{ij}.
\]

Based on this projection we can define the \( \varepsilon^+ \) operator.

\[
X \varepsilon^+ e_i \triangleq \exists a_{ij} \in e_i : X \in \pi_A(a_{ij})
\]

\[
X \varepsilon^+ E_A \triangleq \forall e_i \in E_A : X \varepsilon^+ e_i
\]

Reconsider the previous example. Here, "id" \( \varepsilon^+ e_1 \) is \( \text{True} \) while "birthday" \( \varepsilon^+ e_2 \) is \( \text{False} \).

We also introduce the Dot-Notation which can be used for reading the value of a given property name. We particularly need it in order to express matching conditions for the multi-entity-operations. For this, we will introduce the following notation:

\[
\forall X \varepsilon^+ e_i : e_i.X := \pi_v(a_{ij}) \text{ with } \pi_v := S_{A_i} \times D(A_i) \rightarrow D(A_i) \text{ with } (A_{ij}, v_{ij}) \mapsto v_{ij}.
\]

In the example, \( e_1.name \) evaluates to "Tanja" and \( e_2.name \) evaluates to "Hannes". In contrast to this, \( e_1.birthday \) does not deliver a result.

Due to migration, we consider the same kind at different points of time. Therefore, we
introduce a notation to state the version. It is written in square brackets. For instance, $S_{A[10]} = \{A_1, \ldots, A_n\}_{[10]}$ describes that the definition for the schema of kind $A$ is valid at schema version 10. In the abstract notation for the evolution and migration operation, we will use $[v_A]$ and $[v_B]$ for the version information of the kinds $K_A$ and $K_B$.

For each operation, we define pre- and postconditions. From the later view of implementation, pre- and postconditions are comparable with the concept of design by contract. Operations are only executed if the preconditions are fulfilled, otherwise they will be rejected. After the execution of an operation, the postcondition is guaranteed. From the formal view, the postcondition is important for the formal chaining of operations and for the query rewriting.

### 2.1 Heterogeneity Class 1

HC1 covers the simplest cases. All datasets are regular, which means that all entities of a kind have the same internal schema. This enables us to provide a reversible semantics for our migration operation with the exception of the Delete operation. We can reconstruct the schema and even the data and therefore rewrite queries with an exact inverse in both directions (cf. [Fa11]). For the Delete operation, a relaxed inverse can be specified. Here, the schema can be reconstructed but not the instances (cf. [Fa11]).

Each evolution operation modifies the schema. On the instance (entity) level, the operation modifies the implicit structure and updates affected instances. The schema and instance modification are described by state transitions. On the left side of the state transitions, there are the states before and on the right side after the execution of the evolution operation. We will define the semantics for one single-entity operation (Add) and for a multi-entity operation (Move).

**The Add Operation** This operation adds a property to all entities of a kind. The kind, the new property name and its default property value are specified. More formally:

\[
\text{Add } A.X = d \\
\text{precond : } \{X \not\in S_{A[v_A]}\} \\
S_{A}(A_1, \ldots, A_n)_{[v_A]} \rightarrow S_{A}(X, A_1, \ldots, A_n)_{[v_A+1]} \\
\forall e_i \in E_{A} : (e_i(a_1, \ldots, a_n)_{[v_A]} \rightarrow e_i((X : d), a_1, \ldots, a_n)_{[v_A+1]})) \\
\text{postcond : } \{X \in S_{A[v_A+1]} \land \forall e_i \in E_{A[v_A+1]} : X \in e_i\}
\]

First, this evolution operation proofs the precondition precond which specifies that the name of the property is currently not available in the explicit schema of kind $K_A$ at version $v_A$. Because we previously defined for the internal schema of the entities that $a_{ij} = (A_{ij} : v_{ij}) \in S_{A_i} \times D(A_i)$ and $S_{A_i} \subseteq S_A$, we know that there is no entity which has a property with the property name $X$. The second line describes how the explicit schema of
kind $\mathcal{K}_A$ evolves. In version $v_A$, the schema $S_A$ consists of $n$ properties $A_1, \ldots, A_n$. After the operation, the schema consists of $n + 1$ properties with the added property and the version number is incremented by 1 to $v_A + 1$. The third line gives the modification of each entity of kind $\mathcal{K}_A$. After the operation, each entity consists of its previous properties $a_1$ to $a_n$ and the added property $(X : d)$ where $X$ is the new property name and $d$ is the specified default value. The version number of each entity is modified to $v_A + 1$. After the operation, when the schema and all entities have been modified, the postcondition $\text{postcond}$ holds. The property name $X$ is part of $S_A$ in version $v_A + 1$ and each entity $e_i$ in the set of entities $E_A$ contains a property with the property name $X$, hence $X \in^* e_i$.

It is also possible to add a property without default value, e.g. $\text{Add } A.X$. In this case, instead of $(X : d)$, the property $(X : \bot)$ is added.

It can be necessary to reverse the operation. For instance, when a legacy application only knows the schema in version $v_A$ but transparently queries the entities in version $v_A + 1$, which already had been migrated. For this case, we have to define a reverse operation which can be considered as a non-materialized view. Semantically, any reverse operation should restore the schema of the previous version and should reconstruct the instances as far as possible, too. We denote the reversal of any operation $op$ with $op^{-1}$, e.g. $\text{Add}^{-1} A.X$.

Reversing the $\text{Add}$ operation is simple because it only removes the added attribute.

$$\text{Add}^{-1} A.X$$

$$\text{precond} : \{X \in S_{A[v_A]} \land \forall e_i \in E_A[v_A+1] : X \not\in^* e_i\}$$

$$S_A(X, A_1, \ldots, A_n)[v_{A+1}] \rightarrow S_A(A_1, \ldots, A_n)[v_{A}]$$

$$\forall e_i \in E_A : (e_i((X : \bot), a_1, \ldots, a_n)[v_{A+1}] \rightarrow e_i((A_1, \ldots, A_n)[v_{A}])$$

$$\text{postcond} : \{X \not\in S_A[v_A]\}$$

In this case, we can invert the operation by swapping the pre-and the postcondition and the left and the right sides of the rules, because of the lack of optional properties and high homogeneity as characteristics of HC1.

The Rename Operation This operation renames a property of all entities of a kind. Precondition is that the old property name has to be existent, while the new name is not allowed to be present in advance of the operation. Formally, we can define the operation as follows:

$$\text{Rename } A.X \text{ To } Z$$

$$\text{precond} : \{X \in S_A[v_A] : Z \not\in S_A[v_A]\}$$

$$S_A(X, A_2, \ldots, A_n)[v_A] \rightarrow S_A(Z, A_2, \ldots, A_n)[v_{A+1}]$$

$$\forall e_i \in E_A : (e_i((X : x), a_2, \ldots, a_n)[v_A] \rightarrow e_i((Z : x), a_2, \ldots, a_n)[v_{A+1}])$$

$$\text{postcond} : \{X \not\in S_A[v_A+1] : Z \in S_A[v_A+1]\}$$
On the schema level, the property name $X$ is replaced by the property name $Z$. On the entity level, we adapt the property with the property name $X$, in this case ($X : x$). After the operation, the property is ($Z : x$). Here, we changed the property name from $X$ to $Z$ while preserving the property value $x$. After the Rename operation, no entity contains the property $X$ anymore but each entity contains $Z$ as stated in the postcondition.

**The Delete Operation**   The `Delete` operation removes the property name from all entities of a kind. To execute the operation, the property has to be present in advance of the operation.

\[
\text{precond : } \{ X \in S_{A|v_A} \} \\
S_{A}(X, A_2, \ldots, A_n|v_A) \rightarrow S_{A}(A_2, \ldots, A_n|v_A+1) \\
\forall e_i \in E_A : e_i((X : x), a_2, \ldots, a_n|v_A) \rightarrow e_i(a_1, \ldots, a_n|v_A+1) \\
\text{postcond : } \{ X \notin S_{A|v_A+1} \land \forall e_i \in E_{A|v_A+1} : X \notin e_i \}
\]

In contrast to the other operations in Heterogeneity Class 1, we have a loss of information after the `Delete` operation. Nevertheless, we need to provide an inverse operation, e.g. for legacy applications which expects the property to be present for the application logic. We can invert the operation by reconstructing the schema. Instead of the original values, we substitute the property value with a $\bot$ values.

\[
\text{precond : } \{ X \notin S_{A|v_A} \} \\
S_{A}(A_2, \ldots, A_n|v_A+1) \rightarrow S_{A}(X, A_2, \ldots, A_n|v_A) \\
\forall e_i \in E_A : e_i((X : \bot), a_2, \ldots, a_n|v_A) \rightarrow e_i((X : \bot), a_2, \ldots, a_n|v_A+1) \\
\text{postcond : } \{ X \in S_{A|v_A+1} \}
\]

The reverse operation reconstructs a previous schema version of the entities but not its values. This is comparable to a relaxed inverse in the context of the CHASE algorithm (c.f. [Fa11]).

**The Move Operation**   This operation is a multi-type operation which moves a property from one entity of a kind to another entity of a kind based on a matching condition. In HC1, we assume a perfect 1:1 match, which means that every entity of the source kind has exactly
one match with an entity of the target kind, and vice versa. Hence, bijectivity is considered as fulfilled. This presumes that the value of the matching condition is unique for each entity and there is neither an entity on the source side nor on the target side that do not have a matching partner. This restriction creates some implicit assumptions, for instance that both kinds must have the same amount of entities.

For Heterogeneity Class 1, we define the semantics of the evolution operations as follows:

\[ \text{Move } A.X \text{ To } B.Z \text{ Where } A.K = B.F \]

\[ \text{precond} : \{ X \in S_A[v_A], Z \notin S_B[v_B] \} \]
\[ S_A(X, K, A_3, \ldots, A_n)_{v_A} \rightarrow S_A(K, A_3, \ldots, A_n)_{v_A+1} \]
\[ S_B(F, B_2, \ldots, B_m)_{v_B} \rightarrow S_B(Z, F, B_2, \ldots, B_m)_{v_B+1} \]

\[ \forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F : \]
\[ (e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})_{v_A}) \wedge e_j((F : k), b_{j2}, \ldots, b_{jm})_{v_B}) \]
\[ \rightarrow e_i((K : k), a_{i1}, \ldots, a_{in})_{v_A+1} \wedge e_j((Z : x), (F : k), b_{j2}, \ldots, b_{jm})_{v_B+1}) \]

\[ \text{postcond} : \{ X \notin S_A[v_A+1], Z \in S_B[v_B+1] \} \]

In the Move operation, we specify the source and target kinds, in the example \( K_A \) and \( K_B \) and the property names. If these property names differ, the Move operations implicitly realizes a renaming. In the Where clause, the matching condition is specified.

In advance of the operation, the schema \( S_A \) of the source kind \( K_A \) contains the property name \( X \), while the schema \( S_B \) of the target kind does not contain the attribute name \( Z \) which was specified in the Move command. On the schema level, it is apparent that the moved property \( X \) is not present anymore in \( S_A \) after the operation execution. Instead, \( S_B \) now contains \( Z \). After the operation, all entities \( e_i \) and \( e_j \) were modified. The property \( (X : x) \) is not present anymore in the source kind while \( (Z : x) \) is. Note that the value of the property remains the same \( (x) \) before and after the operation.

To invert the Move operation in HC1, we can simply "move back" the affected property with the same matching condition in consequence of the given bijectivity (each source entity had exactly one matching partner and vice versa).

The semantics how to invert the Move operation and the definition of the operations Rename, Delete and Copy can be found in the technical report [MKS18].

The Copy operation The Copy operation is quite similar to the Move operation. The difference between both operations is that the copied attribute remains in the source entity. Consequently, the postcondition for the source schema remains unchanged. Formally, we can describe the Copy operation as follows:
Copy $A.X$ to $B.Z$ where $A.K = B.F$

$$\text{precond} : \{ X \in S_{A[A]} : Z \notin B_{S_{[B+B]}} \}$$

$S_{A}(X, K, A_{3}, \ldots , A_{n})_{[A+A]} \rightarrow S_{A}(X, K, A_{3}, \ldots , A_{n})_{[A+A]}$

$S_{B}(F, B_{2}, \ldots , B_{m})_{[B+B]} \rightarrow S_{B}(Z, F, B_{2}, \ldots , B_{m})_{[B+B]}$

$$\forall e_{i} \in E_{A}, e_{j} \in E_{B}, e_{i}, K = e_{j} F :$$

$e_{i}((X : x), (K : k), a_{i_{1}}, \ldots , a_{i_{m}})_{[A+A]} \land e_{j}((F : k), b_{j_{1}}, \ldots , b_{j_{m}})_{[B+B]}$

$\rightarrow e_{i}((X : x), (K : k), a_{i_{1}}, \ldots , a_{i_{m}})_{[A+A]} \land e_{j}((F : k), b_{j_{1}}, \ldots , b_{j_{m}})_{[B+B]})$

$$\text{postcond} : \{ X \in S_{A[A+A]}, Z \notin B_{S_{[B+B]}} \}$$

To invert the Copy operation we need to delete the property from the schema information and from the entity of the previous target kind (here $K_{B}$). Because the copied property is still present in the source kind after the “forward” operation, we do not have any information loss after the operation. The reverse copy operation can be described as:

Copy$^{-1}$ $A.X$ to $B.Z$ where $A.K = B.F$

$$\text{precond} : \{ X \in S_{A[A+1]}, Z \notin B_{S_{[B+1]}} \}$$

$S_{A}(X, K, A_{3}, \ldots , A_{n})_{[A+A]} \rightarrow S_{A}(X, K, A_{3}, \ldots , A_{n})_{[A+A]}$

$S_{B}(F, B_{2}, \ldots , B_{m})_{[B+B]} \rightarrow S_{B}(Z, F, B_{2}, \ldots , B_{m})_{[B+B]}$

$$\forall e_{i} \in E_{A}, e_{j} \in E_{B}, e_{i}, K = e_{j} F :$$

$e_{i}((X : x), (K : k), a_{i_{1}}, \ldots , a_{i_{m}})_{[A+A]} \land e_{j}((Z : x), (F : k), b_{j_{1}}, \ldots , b_{j_{m}})_{[B+B]})$

$\rightarrow e_{i}((X : x), (K : k), a_{i_{1}}, \ldots , a_{i_{m}})_{[A+A]} \land e_{j}((F : k), b_{j_{1}}, \ldots , b_{j_{m}})_{[B+B]})$

$$\text{postcond} : \{ X \in S_{A[A+A]}, Z \notin B_{S_{[B+B]}} \}$$

Until now, we presented the definitions for five basic operations on homogeneous data and 1:1 matches for Multi-Entity operations.

### 2.2 Heterogeneity Class 2

In this class we assume structural homogeneous data within the same version but have to handle non-1:1 Cardinalities, dangling tuples which can be caused by entities that do not have a join partner on the source or on the target side and the side effects of these problems. The single-type operations are the same as in HC1. The multi-entity operations as visualized in Figure 1(b) have to be extended.
Non-1:1 Cardinalities can cause problems in the Move and Copy operation as despicted in Figure 3. Let us extend our running example into an online video game where each gamer has an own account with the characteristic properties mail and passwd. Due to the beta phase of the game, it was possible to create multiple players for testing purposes. For the final release of the game, each gamer can have only one player dataset. To avoid the loss of the progress of the beta phase, the gained points are moved to the account of the gamer.

\[
\text{Player:} \{ \text{id: 2, name: "Mark", score: 120, version: 1} \}\n\text{Account:} \{ \text{id: "1", mail: "mk@example.com", passwd: "$2a$05$LhWv, version: 1} \}
\]

In this context, we see several problems that we need to solve. First, the value of A.2 has to be determined since there are two possible values – 130 and 120. Then, for the account A.3 no players are available. Hence, it is necessary to add the score property with a default value to this entity. The player P.4 does not belong to an existing account. In other examples, even n:m matches are available. All possible cardinalities need to be covered by our semantics.

**Conflict resolution approaches** For the handling of occurring problems – e.g. determining the value of A.2 in Figure 3 – resolution approaches are necessary. We propose two different conflict resolutions approaches: **Overwrite** and **Ignore** and define the semantics for both. The **Overwrite** approach updates a value each time when a matching partner is found in the database. In case the input data are not sorted, a nondeterministic result may be generated. In other cases, e.g. for databases sorted by a timestamp, we can apply this approach.

In Figure 4, we are defining the semantics of the Move **Overwrite** operation. We are here distinguishing between the **global pre- and postconditions** and **case pre- and postconditions**. The first hold for all entities of a kind while the latter ones do not necessarily hold for all affected entities.

In the HC2, the same pre- and postconditions hold as in the HC1 and on the schema level,
we also have the same schema evolution transitions. On the entity level, we have to consider

\[ \forall e_i \in E_A, e_j \in E_B, e_i, K = e_j, F : \]

\[
\begin{align*}
\text{case precond : } & \{ Z \not\in e_j[v_B] \} \\
& (e_i((X : x)_k, (K : k), a_{i_1}, \ldots, a_{i_n})[v_A] \\
& \wedge e_j((F : k), b_{j_1}, \ldots, b_{j_m})[v_B]) \\
& \rightarrow e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n}[v_A] \\
& \wedge e_j((Z : x)_k, (F : k), b_{j_1}, \ldots, b_{j_m}[v_B])) \\
\text{case postcond : } & \{ Z \not\in e_i[v_B] \} \\
\end{align*}
\]

\[
\begin{align*}
\text{case precond : } & \{ Z \not\in e_j[v_B] \} \\
& (e_i((X : x)_k, (K : k), a_{i_1}, \ldots, a_{i_n})[v_A] \\
& \wedge e_j((Z : x)_k, (F : k), b_{j_1}, \ldots, b_{j_m}[v_B]) \\
& \rightarrow e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n}[v_A] \\
& \wedge e_j((Z : x)_k, (F : k), b_{j_1}, \ldots, b_{j_m}[v_B])) \\
\text{case postcond : } & \{ Z \not\in e_i[v_B] \} \\
\end{align*}
\]

\[
\begin{align*}
\forall e_i \in E_A, e_j \in E_B, e_i, K = e_j, F : \]

\[
\begin{align*}
& (e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n}[v_A] \\
& \rightarrow e_i((K : k), a_{i_1}, \ldots, a_{i_n}[v_A + 1]) \\
& e_j[v_B] \rightarrow e_j[v_B + 1]) \\
& \forall e_j \in E_B, \exists e_i \in E_A : e_i, K = e_j, F \\
& (e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n}[v_A] \\
& \rightarrow e_i((K : k), a_{i_1}, \ldots, a_{i_n}[v_A + 1])) \\
& (e_j((F : k), b_{j_1}, \ldots, b_{j_m}[v_B] \\
& \rightarrow e_j((Z : x)_k, (F : k), b_{j_1}, \ldots, b_{j_m}[v_B + 1])) \\
\end{align*}
\]

Fig. 4: Definition of the Move Overwrite Operation, HC2

we also have the same schema evolution transitions. On the entity level, we have to consider several subcases. At first, we consider 1:1, 1:n, n:1, and n:m matches. The condition that there is at least one matching partner is expressed by \( \forall e_i \in E_A, e_j \in E_B, e_i, K = e_j, F \).

In the definition of the Move Overwrite operation we have to distinguish two cases: The first case expresses that the entity \( e_j \) does not contain the property \( Z \) yet. Then, the property with the name \( Z \) is added to \( e_j \) and its property value is the value of \( X \) of \( e_j \), in our semantics it is \( x \). From the source kind \( e_i \), the property with the name \( X \) and the value \( x \) is removed.

The second case describes what happens when there is already such a property with the name \( Z \) on the target side. This can happen if 1:n or m:n cardinalities are valid. In these cases, we overwrite the property value with the property name \( Z \) of the entity \( e_j \) with the property
value with the property name $X$ of the entity $e_i$. In the previously introduced problem statement in Figure 3, the value of $A.2$ would be 130 after the first match. Afterwards, 130 is overwritten by 120 of the second (and last) match.

Please note that the properties and the version number on the source side have to be kept until the Move operation to all target entities is executed. Only afterwards we can remove the property with the name $X$ of each entity of $e_i$ and increment the version information from the source and target entities. Otherwise, we would run into trouble, for instance with m:n matches. If the property is removed from the source entity after the first match, there is a problem at the second match because the property on the source side is not present anymore. Additionally, we are not able to increment the version information directly because we consider the entities for a specific version only. If we increment the version directly after a match, we would have to check for matching partners in several versions which can cause problems.

The last three lines before the postcondition describe the handling of entities without a join partner: The entity $X$ is deleted from the entity of the source kind. A property with the name $Z$ and with $\perp$ as the property value is added to the entities of the target kind. This ensures schema homogeneity.

The Ignore approach is a similar approach to the Overwrite approach. The only difference is that we do not overwrite the value if it is already existing. Formally, the only change is in the case $Z \in^* e_j[v_B]$. Instead of the value of $X$ of $e_i$, we keep the present value of $Z$ in $e_j$.

Hence, we are replacing the second case block:

\[
\begin{cases}
\text{case precond : } \{ Z \in^* e_j[v_B] \} \\
\langle e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})[v_A] \land e_j((Z : x'), (F : k), b_{j1}, \ldots, b_{jm})[v_B] \rightarrow e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})[v_A] \\
\land e_j((Z : x'), (F : k), b_{j1}, \ldots, b_{jm})[v_B]) \\
\text{case postcond : } \{ Z \in^* e_i[v_B] \}
\end{cases}
\]

Since in both approaches both cases have the same postcondition and we might have overwritten or ignored property values, it is impossible to find an exact query inverse. In case more sophisticated functions are applied for handling n:1 and n:m matches, for instance the aggregate functions sum or avg, we neither can inverse the operations. However, because we assumed structural homogeneity in this Heterogeneity Class, at least finding of a relaxed inverse is possible.

The Copy Operation  This operation is similar to the Move operation in the Heterogeneity Class 2. Formally, we can write the Copy Overwrite operation as:
The necessary changes for the Ignore approach of the Copy operation are comparable to the Move operation. Here, we can replace the semantics block for the second case with the following:

\[
\begin{align*}
\forall e_{i} \in E_{A}, e_{j} \in E_{B}, e_{i} \cdot K & = e_{j} \cdot F, \\
\forall e_{j} \in E_{B} & \exists e_{i} \in E_{A} : e_{i} \cdot F = e_{j} \cdot K, \\
\forall e_{j} \in E_{B} & \exists e_{i} \in E_{A} : e_{i} \cdot F = e_{j} \cdot K,
\end{align*}
\]

where \( e_{i} \cdot F = e_{j} \cdot K \) implies that the preconditions and postconditions of the operations are comparable.
2.3 Heterogeneity Class 3

Operations of HC3 cover the most complicated cases – and unfortunately these cases occur natively in NoSQL databases. Now we assume schema heterogeneity which means that we have to deal with problems such as optional properties.

An example for his HC is visualized in Figure 5. Here, a simple Add operation is executed. The first entity is updated. For adding the property points in the second entity we have to decide if we overwrite the value of points or ignore the operation and keep the already available value 42. Again, we introduce for our approach the additional keywords Overwrite and Ignore for specifying the conflict resolution strategies. In contrast to HC2, this problem also occurs in single type operations (like the Add operation) or for matches with a 1:1 cardinality.

All conflict resolution strategies have in common that without further information we may not be able to rewrite queries without information loss. Consider the left side of Figure 5. We have two entities where one of them has a property points prior to the Add operation. When a Delete operation is executed after the Add operation, we cannot rewrite the operation easily. Compared with the Semantics of HC1 we cannot find a relaxed inverse by adding the property points with a NULL value. Only by using migration meta data it would be possible to rewrite the operations. Figure 5 explains the problem caused by heterogeneous data.

For heterogeneous data sets we need to specify optionality in the preconditions. We denote optional attributes with a question mark. For example, \( X \in S_A \) defines, that \( X \) is an optional property in the schema of kind \( A \) and can (but not necessarily do) appear in an entity. On the schema level, we also use the notation \( S_A(X?) \) for \( X \in S_A \).

The Add operation In HC3, we often have to distinguish a couple of cases. After the Add operation, we can formulate some postconditions for the schema level, no matter if we used the Overwrite or the Ignore approach. We start with the Overwrite approach.

![Fig. 5: Possible problem with heterogenous data with the example operation Add Player.points =25](image)
Add Ignore A.X = d

global precond : \{X \in S_A\}

\forall e_i \in E_{A[X]}

\begin{align*}
 & \text{case : } X \not\in e_i
 \quad \{ \text{case precond : } \{X \not\in e_i\} \\
 & \quad \quad \quad \quad \text{e_i(a_{i_2}, \ldots, a_{i_n})} \rightarrow \text{e_i((X : d), a_{i_2}, \ldots, a_{i_n})} \\
 & \quad \quad \quad \quad \text{case postcond : } \{X \in e_i\} \}
 & \text{case : } X \in e_i
 \quad \{ \text{case precond : } \{X \in e_i\} \\
 & \quad \quad \quad \quad \text{e_i((X : x), a_{i_2}, \ldots, a_{i_n})} \rightarrow \text{e_i((X : d), a_{i_2}, \ldots, a_{i_n})} \\
 & \quad \quad \quad \quad \text{case postcond : } \{X \in e_i\} \}
\end{align*}

global postcond : \{X \in S_{A[X]}, \forall e_i \in E_{A[X]} \subseteq X : e_i\}

The first case conditions defines that the property X is not available. On the instance level, we add the property with the name X and the default value d. The second case describes the case when the property is already present. On the schema level, there are no changes but on the instance level, we overwrite existing values of the property X with the default value d. This operation also can be defined without a default value, denoted with ⊥.

Please note that in HC3 all properties are considered as optional that do not directly affect the operation (here: A₂, \ldots, A_n). For better readability of the operations, we denote the optionality only for properties which have impact on the operation (here: X).

The Add Ignore operation can defined in a similar way. If a property is available, we preserve the value instead of overwriting it:

Add Overwrite A.X = d

global precond : \{X \in S_A\}

\forall e_i \in E_{A[X]}

\begin{align*}
 & \text{case : } X \not\in e_i
 \quad \{ \text{case precond : } \{X \not\in e_i\} \\
 & \quad \quad \quad \quad \text{e_i(a_{i_2}, \ldots, a_{i_n})} \rightarrow \text{e_i((X : d), a_{i_2}, \ldots, a_{i_n})} \\
 & \quad \quad \quad \quad \text{case postcond : } \{X \in e_i\} \}
 & \text{case : } X \in e_i
 \quad \{ \text{case precond : } \{X \in e_i\} \\
 & \quad \quad \quad \quad \text{e_i((X : x), a_{i_2}, \ldots, a_{i_n})} \rightarrow \text{e_i((X : d), a_{i_2}, \ldots, a_{i_n})} \\
 & \quad \quad \quad \quad \text{case postcond : } \{X \in e_i\} \}
\end{align*}

global postcond : \{X \in S_{A[X]}, \forall e_i \in E_{A[X]} \subseteq X : e_i\}

In both cases, it is not trivial to invert the Add operation. It is unclear if we can remove the added property as in the First Class Heterogeneity Semantics because we do not have the information whether the property was present before the information or not.
The Delete operation

The definition of the Delete operation in Heterogeneity Class 3 is quite easy. We can remove the property without considering the individual schemas of the entities. Conflict resolution strategies are not needed. If the property is available, it will be removed. Otherwise, the entity is not modified. Formally, this can be described as follows:

Delete $A.X$

$$\text{global precond : } \{ X \in S_A \}$$

$$\forall e_i \in E_{A[v_t]} :$$

$$\begin{cases} 
\text{case } : X \notin^* e_{i[v_t]} & \begin{cases} 
\text{case precond : } \{ X \notin^* e_{i[v_t]} \} \\
 e_i(a_{i_2}, \ldots, a_{i_n})_{[v_t]} \rightarrow e_i(a_{i_2}, \ldots, a_{i_n})_{[v_{t+1}]} \\
\text{case postcond : } \{ X \notin^* e_{i[v_{t+1}]} \}
\end{cases} \\
\text{case } : X \in^* e_{i[v_t]} & \begin{cases} 
\text{case precond : } \{ X \in^* e_{i[v_t]} \} \\
 e_i((X : x), a_{i_2}, \ldots, a_{i_n})_{[v_t]} \rightarrow e_i(a_{i_2}, \ldots, a_{i_n})_{[v_{t+1}]} \\
\text{case postcond : } \{ X \notin^* e_{i[v_{t+1}]} \}
\end{cases}
\end{cases}$$

$$\text{global postcond : } \{ X \notin S_A[v_{t+1}] \land \forall e_i \in E_{A[v_{t+1}]} : X \notin^* e_i \}$$

The Rename operation

For the Rename operation we have to distinguish between several cases for each entity. Basically, we have to consider if the origin property name exists or not and if the new property name yet exists or not. It is necessary to have a look on all possible combination of cases. In Heterogeneity Class 3, it is required to specify a conflict resolution approach which is needed, if the origin and the new property name both exist in advance of the operation, for instance. The formal semantics for the Rename operation is:
Rename Overwrite $A.X$ To $Z$

\[
global\ precond : \{X \in S_A, Z \notin S_A\} \]

\[
\forall e_i \in E_{A[v]} : \begin{align*}
\text{case } & : X \in e_i[v] \land Z \notin e_i[v] \quad \text{\{case precond : } \{X \in e_i[v] \land Z \notin e_i[v]\}\} \\
& \quad e_i((X : x), (Z : z), a_{i3}, \ldots, a_{in})_{[v]} \rightarrow e_i((Z : x), a_{i3}, \ldots, a_{in})_{[v+1]} \\
& \quad \text{case postcond : } \{X \notin e_i[v] \land Z \in e_i[v]\} \\
\text{case } & : X \notin e_i[v] \land Z \in e_i[v] \quad \text{\{case precond : } \{X \in e_i[v] \land Z \notin e_i[v]\}\} \\
& \quad e_i((Z : z), a_{i3}, \ldots, a_{in})_{[v]} \rightarrow e_i((Z : z), a_{i3}, \ldots, a_{in})_{[v+1]} \\
& \quad \text{case postcond : } \{X \notin e_i[v] \land Z \notin e_i[v]\} \\
\text{case } & : X \notin e_i[v] \land Z \notin e_i[v] \quad \text{\{case precond : } \{X \notin e_i[v] \land Z \notin e_i[v]\}\} \\
& \quad e_i(a_{i3}, \ldots, a_{in})_{[v]} \rightarrow e_i((Z : z), a_{i3}, \ldots, a_{in})_{[v+1]} \\
& \quad \text{case postcond : } \{X \notin e_i[v] \land Z \in e_i[v]\}
\end{align*}
\]

\[
global\ postcond : \{X \notin S_A[v+1], Z \in S_A[v+1]\}
\]

Because we do not know whether the source ($X$) or the target property ($Z$) name is present in advance of the operation, we consider both properties as optional.

The conditions of the first case are equal to those we assumed in HC1. Here, $X$ is present while $Z$ is not and it is easy to rename the property name. The second case covers the situation where both $X$ and $Z$ are present in advance of the operation. In this situation, the conflict resolution strategy \textit{overwrite} is applied. The property value of $Z$ is replaced by the value of $X$ and $X$ is removed from the target side. The third case considers the situation that there is no property $X$ which cannot be renamed but a property $Z$. Here, the entity remains unchanged. The last case deals with the case that neither $X$ nor $Z$ is present before the operation. In this case, we introduce a new property with a \texttt{Null} value analogously to the \texttt{Move} and \texttt{Copy} operation in HC2. With respect to the application context, it might make more sense not to introduce a \texttt{Null} value but to remain the entity unchanged. This could be covered by introducing another keyword, e.g. \texttt{Default} \texttt{Null} or \texttt{Default} \texttt{None} which is evaluated if there is no property after the operation. There are several possible variants, whereby we restrict to the introduced one only for the sake of simplicity.

The \textit{Rename Ignore} approach works quite similar to the introduced \textit{Rename Overwrite} approach. Here, the semantics remains with the difference of the second case which is defined as follows.
If the new and the old property name both exist in advance of the operation, we keep the existing value instead of overwriting it.

**The Move Operation** Multi entity operations like Move and Copy are the most difficult cases of the third layer. Here, we need to consider all the cardinalities as in the second layer. Due to schema homogeneity, even for 1:1 cardinalities several cases are distinguished.

Figure 6 depicts all the cases which can occur in the 1:1 case for the Move operation. The first match depicts the case where points is present in Player but not in Accounts and can be easily moved. The second case describes that points is not present in Player but in Account. Here, the already existing value for points in Account is preserved. The third case describes the case that points is neither in Player nor in Accounts and the last case depicts the case where points is part of both entities. All these cases are part of the semantics definition of the Move operation.

On the schema level, we definitely know that after the operation \( S_A \) (in the example Player) does not contain \( X \) (points) anymore and \( S_B \) (Account) definitely contains \( Z \) (in the example in Figure 6: points). For all entities without a matching partner, the property is removed if the entity is on the source side \( (\mathcal{K}_A) \) or the entity gets a property with a Null value on the target side \( (\mathcal{K}_B) \), respectively.

The definition of the Move Overwrite operation for HC3 is given in the Appendix of this paper.
The Copy operation As before, the Copy operation is similar to the Move operation and only differs in keeping the affected property in the entities of the source kind. The definition of the Copy Ignore operation for HC3 is given in the Appendix of this article.

3 Impact of the Heterogeneity Classes on Query Rewriting

In NoSQL databases, datasets can be stored in different versions within the same database. If we want to avoid that the application logic has to be adapted onto several structural versions of the datasets, we need transparent query rewriting to overcome these heterogeneities caused by the different versions. Figure 7 shows forward and backward query rewriting for such applications.

Fig. 7: Forward and backward query rewriting for versioned databases

Rewriting legacy queries to match entities in newer versions is called forward query rewriting. Backward query rewriting has to be applied if an application knows the current structures but the datasets are still available in older versions. Such a scenario is needed if the evolution of NoSQL databases is realized with a lazy or hybrid data migration. Query rewriting has to be adapted onto the concrete HC of the input data. In case of lazy data migration, datasets can be available in different structural versions. For simplicity, we show the query rewriting with two versions: \( v_A \) for the latest version and \( v_A - 1 \) for the previous version. Generally, query rewriting with more than 2 versions is realized in the same way.

In this chapter, we continue with an abstract and much shorter example for focusing on the aspects of query rewriting.

Backward Query Rewriting for the Evolution Operation Add In this example, the version \( v_A \) is generated from the version \( v_A - 1 \) by applying an Add operation:

**Schema evolution operation:** Add \( A.x = d \)

**Query:** Select * From A

This query assumes the schema version \( v_A \). For a backward query rewriting, for integrating entities from version \( v_A - 1 \), the query is rewritten for the different Heterogeneity Classes.
**Query for Heterogeneity Class 1:**

Select * From A Where version = $v_A$
Union Select *, d As X From A Where version = $v_A - 1$.

For datasets in Heterogeneity Class 3, the query rewriting is much more complicated. Here we get the following rewritten query:

**Query for Heterogeneity Class 3 (conflict resolution strategy: Ignore):**

Select * From A Where version = $v_A$
Union Select * From A Where version $v_A - 1$ And Exists(A.X)
Union Select *, d As X From A Where version = $v_A$ And Not Exists(A.X)

The first row selects the entities in the newest version for which we can conclude that they contain the property X. The second row selects all entities that are still in the previous version and contain the property X. Due to the Ignore conflict resolution strategy, we keep the value of X. The third row selects the entities in the previous version that do not have a property X and extends the result with the property X and default value d. The keyword [Not] Exists(...) is not part of native SQL but is lend from XPath. It is necessary for checking the presence or absence of a property.

**Query for Heterogeneity Class 3 (conflict resolution strategy: Overwrite):**

Select * From A Where version = $v_A$
Union Select <PropertyList\{X\}>, d As X From A Where version $v_A - 1$ And Exists(A.X)
Union Select *, d As X From A Where version = $v_A$ And Not Exists(A.X)

The first subquery selects all entities in the latest version. The second query selects all entities which are lazy migrated and where A.X is existent before the operation – due to the Overwrite approach, we need to replace these property values with X. In this case, we can modify our projection clause by omitting the present property X and select d As X. Here, <PropertyList> is not an actual SQL keyword and needs to be expanded to the actual properties of the kind. The last subquery selects all entities where A.X is not present before the operation. In this case, we can simply select all properties and additionally d As X.

**Backward Query Rewriting for the Evolution Operation Delete**

In this example, the version $v_A$ is generated from the version $v_A - 1$ by applying a Delete operation:

**Schema evolution operation:** Delete A.X

**Query:** Select * From A

This query assumes the schema version $v_A$. For a backward query rewriting, for integrating entities from version $v_A - 1$, the query is rewritten for the different Heterogeneity Classes.

**Query for Heterogeneity Class 1 and 3:**
Select * From A Where version = \(v_A\)
Union Select \(<\text{PropertyList}\{X\}>\) From A Where version = \(v_A - 1\).

Rewriting the \texttt{Delete} operation is identical for all heterogeneity classes. We can select all properties from the most recent version because they were migrated eagerly and therefore the deleted property is not present in this version. To select lazy migrated entities, we have to expand all properties from the entities of kind A in the query (the resulting set of properties is denoted as \texttt{PropertyList}) without the property \(X\). The semantics for HC1 and HC3 is identical except for the precondition and there is no difference in query rewriting for different heterogeneity classes.

**Backward Query Rewriting for the Evolution Operation Rename** In this example, the version \(v_A\) is generated from the version \(v_A - 1\) by applying a \texttt{Rename} operation:

\textbf{Schema evolution operation:} Rename \texttt{A.X} To \texttt{Z}

\textbf{Query:} Select * From A

This query assumes the schema version \(v_A\). For a backward query rewriting, for integrating entities from version \(v_A - 1\), the query is rewritten for the different Heterogeneity Classes.

\textbf{Query for Heterogeneity Class 1:}
Select * From A Where version = \(v_A\)
Union Select \(<\text{PropertyList}\{X\}>\), X As Z From A

In HC1, we can select the entities in the latest version as in the operations before. For the lazy migrated entities, we need to select all entities without the renamed one (\(X\)) and substitute the old property name (\(X\)) with the new one (\(Z\)) by using the \texttt{As} clause. Due to the characteristics of HC1, we know that all lazy migrated entities have the property \(X\) and there is no entity which has the property \(Z\) in advance of the operation.

\textbf{Query for Heterogeneity Class 3 (Conflict Resolution Strategy: Ignore):}
Select * From A Where version = \(v_A\)
Union Select \(<\text{PropertyList}\{X\}>\), X As Z From A Where version = \(v_A - 1\) And Exists (A.X) And Not Exists(A.Z)
Union Select \(<\text{PropertyList}\{X\}>\), Null As Z From A Where version = \(v_A - 1\) And Not Exists (A.X) And Not Exists(A.Z)
Union Select * From A Where version = \(v_A - 1\) And Not Exists (A.X) And Exists (A.Z)
Union Select \(<\text{PropertyList}\{X\}>\) From A Where version = \(v_A - 1\) And Exists (A.X) End Exists (A.Z)

\textbf{Query for Heterogeneity Class 3 (Conflict Resolution Strategy: Overwrite):}
Select * From A Where version = \(v_A\)
Union Select \(<\text{PropertyList}\{X\}>\), X As Z From A Where version = \(v_A - 1\)
Where version = \(v_A - 1\) And Exists (A.X) And Not Exists(A.Z)
Union Select <PropertyList\{X}\>, Null as Z From A
   Where version = \(v_A - 1\) And Not Exists (A.X) And Not Exists(A.Z)
Union Select * From A
   Where version = \(v_A - 1\) And Not Exists (A.X) And Exists (A.Z)
Union Select <PropertyList\{X, Z\}>, X As Z From A
   Where version = \(v_A - 1\) And Exists (A.X) End Exists (A.Z)

In both queries, all different cases are present which also occur in the semantics. The first query fetches the eager migrated entities while the other four queries are the analogies to the migration rules. The last subquery in both approaches realize the conflict resolution strategy. Here, both properties \(X\) and \(Z\) are present in advance of the operation. In the Ignore approach, we use the value of \(Z\). Hence, we can simply use a projection operation which selects all properties without \(X\). In the Overwrite approach, we generally select all properties without \(X\) and \(Z\) and additionally the value of \(X\) with the alias name \(Z\).

**Backward Query Rewriting for the Evolution Operation Move**

A more complex example is a version \(v\) generated from the version \(v - 1\) by applying a Move operation:

**Schema evolution operation:** Move A.X To B.Z Where A.A = B.B

**Query:** Select * From B

This query assumes the schema version \(v_B\) for properties of kind B. A backward query rewriting is also integrating entities that are still in version \(v_B - 1\).

**Query for Heterogeneity Class 1:**

Select * From B Where version = \(v_B\)
Union Select * From B, X As Z From A Where A.A = B.B And
   A.version = \(v_A - 1\) And B.version = \(v_B - 1\)

The first Select clause selects all entities in the current version. The second line selects the entities which are still in the previous version. The property \(Z\) is still available in the entity \(A\) as property \(X\). Because we assume 1:1 cardinalities, we do not need to handle dangling tuples or multiple matching partners.

**Query for Heterogeneity Class 2 (conflict resolution strategy: Ignore):**

Select * From B Where B.version = \(v_B\)
Union Select Distinct On (A.A) B.*, A.X From A, B
   Where A.A = B.B And A.version = \(v_A - 1\) And B.version = \(v_B - 1\)
Union Select *, Null as Z From B Where B.version = \(v_B - 1\)
   And B.B Not In (Select A From A Where A.version = \(v_A - 1\))
The first query selects all entities in version $v_B$. This implies that each of the entities in this result set contains the property $Z$. The second, nested query takes entities from the previous version and solves the m:n matches. From the view of each concrete entity, these are a m:1 matches. For each matching property, we avoid duplicates with the Distinct On $(A.a)$ clause to fulfill the Ignore conflict resolution strategy. The third query is responsible for 0:1 and 0:n matches. We select entities without a matching partner in $A$ and substitute $z$ with a Null value. Figure 8 visualizes the effect of the different parts of the rewritten query with an example.

---

**Query for Heterogeneity Class 3 (conflict resolution strategy: Ignore):**

```sql
Select * From B Where B.version = $v_B$
Union Select * From B Where B.version = $v_B - 1$ And Exists(B.$z$)
Union Select Distinct On (A.$a$) B.*, A.$x$ From A, B
  Where A.$a$ = B.$b$ And B.version = $v_B - 1$ And A.version = $v_A - 1$
  And Not Exists (B.$z$) And Exists (A.$x$)
Union Select Distinct * From Null As $X$ From A, B
  Where A.$a$ = B.$b$ And B.version = $v_B - 1$ and A.version = $v_A - 1$
  And B.$z$ Is Null And Not Exists(A.$x$) And A.$a$ Not In (Select Distinct A.$a$ From A, B
  Where A.$a$ = B.$b$ And B.version = $v_B - 1$ and A.version = $v_A - 1$
  And Not (Exists B.$z$) And Exists (A.$x$))
Union Select Distinct B, Null As $Z$ From B
  Where B.$b$ Not In (Select A From A Where A.version = $v_A - 1$)
  And A.version = $v_B - 1$ And Not Exists(B.$z$)
```

The first query selects all eagerly migrated entities in the current version. The second subquery selects all entities of $B$ in the previous version that already contain a $B.Z$ in version $v_B - 1$. Due to the Ignore approach, the value is not affected by the operation. The third subquery selects the value of $B.Z$ that is still stored in a corresponding $A.X$, similar to the query Heterogeneity Class 2. If there are multiple matches, we use the first match. The fourth subquery checks for entities with a matching partner in kind $A$ whereby the matching partner does not have the property $A.X$. In this case, we substitute a Null value.

---

4 PostgreSQL Flavor. The Distinct On(...) clause checks for the first match and ignores any additional ones.
In this query, we additionally have to exclude results from the previous subquery which is done using the sub-subquery. Otherwise, if an entity of B in version \(v_B - 1\) has one matching partner in kind A in version \(v_A - 1\) with a property \(x\) and a matching partner in kind A in version \(v_A - 1\) without an property (which means that the entity of kind B is partner in a n:1 cardinality), the entity of kind B is in the result set twice. The last query selects the entities of B that do not have a matching partner (0:1 cardinalities) and substitute \texttt{NULL} values for corresponding \(B.z\) property values. The subqueries of the above given query are directly derived from the semantics of the \texttt{Move} operation in HC3. Each case in this definition of the evolution operation generates one subselect clause of the query.

**Backward Query Rewriting for the Evolution Operation Copy**  
Consider the version \(v\) which was generated from the version \(v - 1\) by applying a Copy operation:

**Schema evolution operation:** Copy \(A.X\) To \(B.Z\) Where \(A.A = B.B\)

**Query:** Select * From B

This query assumes the schema version \(v_B\) for properties of kind B. A *backward query rewriting* is also integrating entities that are still in version \(v_B - 1\).

**Query for Heterogeneity Class 1:**

Select * From B Where version = \(v_B\) 
Union Select B.*, A.X As Z From A, B Where A.A = B.B And 
A.version = \(v_A - 1\) And B.version = \(v_B - 1\)

For entities which are migrated eagerly, we can query the newest version. For entities which are migrated lazily – entities in version \(v_B - 1\) – we have to query the second-latest version as well.

**Query for Heterogeneity Class 2 and 3:**

Analogously to the previous reason, the Copy operation queries in HC2 and HC3 are equal to the queries of the \texttt{Move} operation in HC2 and HC3.

### 4 Related Work

The main focus in this paper is on combining heterogeneity and evolution in NoSQL databases for query rewriting. For this task, we consider several related work.

In database theory, there are various techniques to define dependencies between two relations. In the context of schema mapping *source-to-target tuple generating dependencies (ST-TGDs)* can be used to describe the dependencies between two databases (cf. [PS11], [AHV95]).
The CHASE algorithm is a fixed-point algorithm that migrates a database instance into another instance of database by applying the ST-TGDs (cf. [AHV95]).

Schema evolution with complex schema modification operations (SMOs), automated data migration operations, and automated rewriting of queries for relational databases, has been investigated by Moon et. al. in the PRISM project [MCZ10]. In [He17], several schema versions are being maintained within a single relational database. A language for bidirectional schema evolution and forwards and backwards delta code generation is defined. Rather than rewriting queries, the authors migrate the data on demand, between the schema versions. While the problem setting is similar, our solution differs insofar as we rewrite queries, rather than continuously migrate data between schema versions.

There are several tools for schema evolution of NoSQL databases. Most of them realize an eager migration, for instance, Mongeez, Flyway and Liquibase. In the context of NoSQL datastores, however, legacy entities in different schema versions may co-exist in the same data store, especially in case of lazy and hybrid data migration. The foundation of lazy NoSQL data migration has been proposed in [SKS13]. A similar approach is introduced in [SDH16], here the performance of lazy migration in NoSQL data stores has been studied and in [KSS16] first ideas for hybrid approaches and an estimation of their effort are given. The foundations on query rewriting in Darwin have been developed in [St17]. Here, operations are translated into disjunctive embedded dependencies and forward and backward mappings are defined. For the first heterogeneity classes, prototypical implementations have been made in this work.

Another approach for query rewriting is developed under the name EasyQ in [Ha18]. In this article, all variants that occur for each property are stored as so-called paths in a dictionary, so that each query is expanded. However, this method has the disadvantage that the result set is too large. The combination of all variants flows into each query result. As far as we know, the combination of input data set in different HCs, versioning and multi-type evolution operation has not been studied before. The first steps in this field are realized in the Darwin project but this is still an ongoing research task.

5 Summary and Future Work

In lazy data migration, datasets are only updated on demand. Consequently, NoSQL databases contain datasets in different schema versions. For querying NoSQL data, we have to apply query rewriting techniques so that queries against the latest version of the schema are rewritten for querying datasets in previous schema versions. For that, we have to use the inverse schema evolution operations describing the changes between two successive structure versions.

In this article, we have shown that query rewriting can be applied straightforward in case of NoSQL databases in HC1. More complicated are query rewriting operations in case of
dangling tuples in join conditions between two entity types and heterogeneous datasets within the same version (HC3).

Without any additional knowledge about the NoSQL heterogeneity class of the input data, we merely expect, that the datasets are in NoSQL HC3 and thus apply the query rewriting approach considering all structural variants. In case that the datasets are in NoSQL HC1, query rewriting is much easier. Consequently, information about the NoSQL heterogeneity class can significantly improve performance of this process. In NoSQL databases with a rigid schema management, we can guarantee that datasets are in NoSQL HC1. If there are NoSQL databases that did not yet use a schema management component, we can realize a schema extraction for deriving the structures and the valid heterogeneity forms of available databases. Such a schema extraction approach is part of the research prototype Darwin [KSS15].

The overall aim of the Darwin project is the development of a migration adviser for supporting users to choose the optimal migration strategy for a certain application. Based on different NoSQL database characteristics, it shall recommend either eager or lazy migration or a hybrid approach. In this paper we have shown, that (beside other factors like monetary costs, latency, and volume of data) even the Heterogeneity Class of the NoSQL data influences this choice. In case of HC1, a lazy migration can be applied. In datasets in HC3, eager data migration is more advantageous than lazy or hybrid approaches. In the next step, we will extend Darwin so that the different heterogeneity classes are taken into account during query rewriting. We will also consider the heterogeneity classes when developing the migration adviser in order to select a suitable migration strategy.

Acknowledgements The article is published in the scope of the project "NoSQL Schema Evolution und Big Data Migration at Scale" which is funded by the Deutsche Forschungsgemeinschaft (DFG) under the number 385808805. A special thanks goes to Stefanie Scherzinger, Andrea Hillenbrand, Dennis Marten, Tanja Auge, and Hannes Grunert for their support, comments on this work, and several discussions.
References


A Appendix

A.1 Move Overwrite Semantics in Heterogeneity Class 3

Move Overwrite A.X To B.Z Where A.K = B.F

global precond : \{ X \in S_A[v_a] ; Z \in S_B[v_b] \}

\[ S_A(X', K', A_3, \ldots, A_n)[v_a] \rightarrow S_A(K', A_3, \ldots, A_n)[v_{a+1}] \]

\[ S_B(F', Z, B_2, \ldots, B_m)[v_b] \rightarrow S_B(Z, F', B_2, \ldots, B_m)[v_{b+1}] \]

\[ \forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F \]

\[ \begin{align*}
\text{case } : X \in^* e_i[v_a] & : \\
& \begin{cases}
\text{case precond : } \{ X \in^* e_i[v_a] \land Z \notin^* e_j[v_b] \} \\
& (e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a]) \\
& \land (e_j((F : k), b_{j_1}, \ldots, b_{j_m})[v_b]) \\
& \rightarrow (e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a]) \\
& \land (e_j((Z : z), (F : k), b_{j_1}, \ldots, b_{j_m})[v_b])
\end{cases}
\end{align*} \]

\[ \text{case postcond : } \{ X \in^* e_i[v_a] \land Z \in^* e_j[v_b] \} \]

\[ e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a] \rightarrow e_i((K : k), a_{i_1}, \ldots, a_{i_n})[v_{a+1}] \]

\[ e_j((Z : z), (F : k), b_{j_1}, \ldots, b_{j_m})[v_b] \rightarrow e_j((Z : z), (F : k), b_{j_1}, \ldots, b_{j_m})[v_{b+1}] \]

\[ e_i[v_a] \rightarrow e_i[v_{a+1}] \]

\[ e_j[v_b] \rightarrow e_i[v_{b+1}] \]

\[ \left( \forall e_i \in E_A : \exists e_j \in E_B : e_i.K = e_j.F \right) \lor \left( \forall e_j \in E_B : \exists e_i \in E_A : e_j.F = e_i.K \right) ;
\]

\[ \begin{align*}
\text{case } : Z \notin^* e_i[v_a] & : \\
& \begin{cases}
\text{case precond : } \{ X \notin^* e_i[v_a] \land Z \in^* e_j[v_b] \} \\
& (e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a]) \\
& \land (e_j((F : k), b_{j_1}, \ldots, b_{j_m})[v_b]) \\
& \rightarrow (e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a]) \\
& \land (e_j((Z : z), (F : k), b_{j_1}, \ldots, b_{j_m})[v_b])
\end{cases}
\end{align*} \]

\[ \text{case postcond : } \{ X \notin^* e_i[v_a] \land Z \notin^* e_j[v_b] \} \]

\[ \left( e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a] \right) \land \left( e_j((F : k), b_{j_1}, \ldots, b_{j_m})[v_b] \right) \rightarrow \left( e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a] \right) \land \left( e_j((Z : z), (F : k), b_{j_1}, \ldots, b_{j_m})[v_b] \right) \]

\[ e_i[v_a] \rightarrow e_i[v_{a+1}] \]

\[ e_j[v_b] \rightarrow e_j[v_{b+1}] \]

\[ \left( \forall e_i \in E_A : \exists e_j \in E_B : e_i.K = e_j.F \right) \lor \left( \forall e_j \in E_B : \exists e_i \in E_A : e_j.F = e_i.K \right) ;
\]

\[ \begin{align*}
\text{case } : X \notin^* e_i[v_a] & : \\
& \begin{cases}
\text{case precond : } \{ X \notin^* e_i[v_a] \land Z \notin^* e_j[v_b] \} \\
& (e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a]) \\
& \land (e_j((Z : z), (F : k), b_{j_1}, \ldots, b_{j_m})[v_b]) \\
& \rightarrow (e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a]) \\
& \land (e_j((Z : z), (F : k), b_{j_1}, \ldots, b_{j_m})[v_b])
\end{cases}
\end{align*} \]

\[ \text{case postcond : } \{ X \notin^* e_i[v_a] \land Z \notin^* e_j[v_b] \} \]

\[ e_i[v_a] \rightarrow e_i[v_{a+1}] \]

\[ e_j[v_b] \rightarrow e_j[v_{b+1}] \]

\[ \left( \forall e_i \in E_A : \exists e_j \in E_B : e_i.K = e_j.F \right) \lor \left( \forall e_j \in E_B : \exists e_i \in E_A : e_j.F = e_i.K \right) ;
\]

\[ \begin{align*}
\text{case } : Z \in^* e_i[v_a] & : \\
& \begin{cases}
\text{case precond : } \{ X \in^* e_i[v_a] \land Z \notin^* e_j[v_b] \} \\
& (e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a]) \\
& \land (e_j((F : k), b_{j_1}, \ldots, b_{j_m})[v_b]) \\
& \rightarrow (e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a]) \\
& \land (e_j((F : k), b_{j_1}, \ldots, b_{j_m})[v_b])
\end{cases}
\end{align*} \]

\[ \text{case postcond : } \{ X \in^* e_i[v_a] \land Z \notin^* e_j[v_b] \} \]

\[ \left( e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a] \right) \land \left( e_j((F : k), b_{j_1}, \ldots, b_{j_m})[v_b] \right) \rightarrow \left( e_i((X : x), (K : k), a_{i_1}, \ldots, a_{i_n})[v_a] \right) \land \left( e_j((F : k), b_{j_1}, \ldots, b_{j_m})[v_b] \right) \]

\[ e_i[v_a] \rightarrow e_i[v_{a+1}] \]

\[ e_j[v_b] \rightarrow e_j[v_{b+1}] \]

\[ \text{global precond : } \{ X \in S_A[v_{a+1}] ; Z \in S_B[v_{b+1}] \} \]
A.2 Move Ignore Semantics in Heterogeneity Class 3

Move Ignore A.X To B.Z Where A.K = B.F

global precond : \{ X \in S_{A[\nu]}, Z \in S_{B[\nu]} \}

\[
\begin{align*}
& S_A(X^?, K^?, A_1^?, \ldots, A_n^?[, \nu]_A) \rightarrow S_A(K^?, A_1^?, \ldots, A_n^?[, \nu+1]_A) \\
& S_B(F^?, B_2^?, \ldots, B_m^?[, \nu]_B) \rightarrow S_B(Z, F^?, B_2^?, \ldots, B_m^?[, \nu+1]_B)
\end{align*}
\]

For all \(e_i \in E_A, e_j \in E_B, e_i, K = e_j, F:

\[
\begin{align*}
\text{case : } Z \notin e_j[\nu] & \rightarrow \text{case precond : } \{ X \notin e_i[\nu] \wedge Z \notin e_j[\nu] \} \\
& \quad | (e_i((X : x), (K : k), a_1, \ldots, a_n)_[\nu]) \\
& \quad \quad \wedge e_j((F : k), b_2, \ldots, b_m)_[\nu] \\
& \quad \quad \rightarrow (e_i((X : x), (K : k), a_1, \ldots, a_n)_[\nu]) \\
& \quad \quad \wedge e_j((Z : z), (F : k), b_2, \ldots, b_m)_[\nu] \\
& \quad \quad \text{case postcond : } \{ X \notin e_i[\nu] \wedge Z \notin e_j[\nu] \}
\end{align*}
\]

\[
\begin{align*}
\text{case : } X \notin e_i[\nu] & \rightarrow \text{case precond : } \{ X \notin e_i[\nu] \wedge Z \notin e_j[\nu] \} \\
& \quad | (e_i((X : x), (K : k), a_1, \ldots, a_n)_[\nu]) \\
& \quad \quad \wedge e_j((Z : z), (F : k), b_2, \ldots, b_m)_[\nu] \\
& \quad \quad \rightarrow (e_i((X : x), (K : k), a_1, \ldots, a_n)_[\nu]) \\
& \quad \quad \wedge e_j((Z : z), (F : k), b_2, \ldots, b_m)_[\nu] \\
& \quad \quad \text{case postcond : } \{ X \notin e_i[\nu] \wedge Z \notin e_j[\nu] \}
\end{align*}
\]

\[
\begin{align*}
\text{case : } Z \notin e_j[\nu] & \rightarrow \text{case precond : } \{ X \notin e_i[\nu] \wedge Z \notin e_j[\nu] \} \\
& \quad | (e_i((X : x), (K : k), a_1, \ldots, a_n)_[\nu]) \\
& \quad \quad \wedge e_j((Z : z), (F : k), b_2, \ldots, b_m)_[\nu] \\
& \quad \quad \rightarrow (e_i((X : x), (K : k), a_1, \ldots, a_n)_[\nu]) \\
& \quad \quad \wedge e_j((Z : z), (F : k), b_2, \ldots, b_m)_[\nu] \\
& \quad \quad \text{case postcond : } \{ X \notin e_i[\nu] \wedge Z \notin e_j[\nu] \}
\end{align*}
\]

\[
\begin{align*}
& (\forall e_i \in E_A) \exists e_j \in E_B : e_i, K = e_j, F \quad \forall (e_j \in E_B) \exists e_i \in E_A : e_j, F = e_i, K : \\
& (e_i((X : x), (K : k), a_1, \ldots, a_n)_[\nu]) \rightarrow (e_i((K : k), a_1, \ldots, a_n)_[\nu+1]) \\
& (e_i((K : k), a_1, \ldots, a_n)_[\nu]) \rightarrow (e_i((K : k), a_1, \ldots, a_n)_[\nu+1]) \\
& (e_j((Z : z), (F : k), b_2, \ldots, b_m)_[\nu]) \rightarrow (e_j((Z : z), (F : k), b_2, \ldots, b_m)_[\nu+1]) \\
& (e_j((F : k), b_2, \ldots, b_m)_[\nu]) \rightarrow (e_j((F : k), b_2, \ldots, b_m)_[\nu+1])
\end{align*}
\]

\[
\begin{align*}
\text{global postcond : } \{ X \notin S_{A[\nu+1]}, Z \notin S_{B[\nu+1]} \}
\end{align*}
\]
A.3 Copy Overwrite Semantics in Heterogeneity Class 3

Copy Overwrite A.X To B.Z Where A.K = B.F

\[ \forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F : \]

\[ S_A(X, K, a_i, \ldots, a_n) \rightarrow S_B(X, K, a_i, \ldots, a_n) \]  \[ S_B(F, B, a_i, \ldots, B_m) \rightarrow S_B(Z, F, B, a_i, \ldots, B_m) \]

\[ (\forall e_i \in E_A : \exists e_j \in E_B : e_i.K = e_j.F) \lor (\forall e_j \in E_B : \exists e_i \in E_A : e_i.F = e_j.K) : \]

\[ (e_i \rightarrow e_i[v_a + 1]) \land (e_j \rightarrow e_j[v_b + 1]) \land (e_i\{v_a\} \rightarrow e_i[v_a + 1]) \land (e_j\{v_b\} \rightarrow e_j[v_b + 1]) \]

\[ \forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F : \]

\[ (e_i \rightarrow e_i[v_a + 1]) \land (e_j \rightarrow e_j[v_b + 1]) \land (e_i\{v_a\} \rightarrow e_i[v_a + 1]) \land (e_j\{v_b\} \rightarrow e_j[v_b + 1]) \]

\[ \forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F : \]

\[ (e_i \rightarrow e_i[v_a + 1]) \land (e_j \rightarrow e_j[v_b + 1]) \land (e_i\{v_a\} \rightarrow e_i[v_a + 1]) \land (e_j\{v_b\} \rightarrow e_j[v_b + 1]) \]

\[ (e_i \rightarrow e_i[v_a + 1]) \land (e_j \rightarrow e_j[v_b + 1]) \land (e_i\{v_a\} \rightarrow e_i[v_a + 1]) \land (e_j\{v_b\} \rightarrow e_j[v_b + 1]) \]
A.4 Copy Ignore Semantics in Heterogeneity Class 3

Copy Overwrite A.X To B.Z Where A.K = B.F

\[ \forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F : \]
\[ \text{global postcond : } \{ X \in S_{A[v_a]}, Z \in S_{B[v_b]} \} \]
\[ S_A(X^?, K^?, A_1^?, \ldots, A_n^?)_{[v_a]} \rightarrow S_A(X^?, K^?, A_1^?, \ldots, A_n^?)_{[v_a+1]} \]
\[ S_B(F^?, B_2^?, \ldots, B_m^?)_{[v_b]} \rightarrow S_B(Z^?, F^?, B_2^?, \ldots, B_m^?)_{[v_b+1]} \]

\[ \forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F : \]
\[ \text{case : } Z \not\in e_j[v_{vb}] \]
\[ \text{case precond : } \{ X \not\in e_i[v_{va}] \wedge Z \not\in e_j[v_{vb}] \} \]
\[ (e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})_{[v_a]} \]
\[ \land e_j((F : k), b_{j1}, \ldots, b_{jm})_{[v_b]} \rightarrow e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})_{[v_a]} \]
\[ \land e_j((Z : z), (F : k), b_{j1}, \ldots, b_{jm})_{[v_b]} \rightarrow e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})_{[v_a]} \]
\[ \text{case postcond : } \{ X \not\in e_i[v_{va}] \wedge Z \not\in e_j[v_{vb}] \} \]

\[ \text{case : } X \not\in e_i[v_{va}] \]
\[ \text{case precond : } \{ X \not\in e_i[v_{va}] \wedge Z \not\in e_j[v_{vb}] \} \]
\[ (e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})_{[v_a]} \]
\[ \land e_j((F : k), b_{j1}, \ldots, b_{jm})_{[v_b]} \rightarrow e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})_{[v_a]} \]
\[ \land e_j((Z : z), (F : k), b_{j1}, \ldots, b_{jm})_{[v_b]} \rightarrow e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})_{[v_a]} \]
\[ \text{case postcond : } \{ X \not\in e_i[v_{va}] \wedge Z \not\in e_j[v_{vb}] \} \]

\[ \text{case : } Z \not\in e_j[v_{vb}] \]
\[ \text{case precond : } \{ X \not\in e_i[v_{va}] \wedge Z \not\in e_j[v_{vb}] \} \]
\[ (e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})_{[v_a]} \]
\[ \land e_j((F : k), b_{j1}, \ldots, b_{jm})_{[v_b]} \rightarrow e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})_{[v_a]} \]
\[ \land e_j((Z : z), (F : k), b_{j1}, \ldots, b_{jm})_{[v_b]} \rightarrow e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})_{[v_a]} \]
\[ \land x \land e_j((Z : z), (F : k), b_{j1}, \ldots, b_{jm})_{[v_b]} \rightarrow e_i((X : x), (K : k), a_{i1}, \ldots, a_{in})_{[v_a]} \]
\[ \text{case postcond : } \{ X \not\in e_i[v_{va}] \wedge Z \not\in e_j[v_{vb}] \} \]

\[ \forall e_i \in E_A, e_j \in E_B : e_i.K = e_j.F \vee (\forall e_j \in E_B \exists e_i \in E_A : e_i.F = e_j.K) : \]
\[ (e_i[v_{va}] \rightarrow e_i[v_{va}+1]) \]
\[ (e_j((Z : z), (F : k), b_{j1}, \ldots, b_{jm})_{[v_b]} \rightarrow e_j((Z : z), (F : k), b_{j1}, \ldots, b_{jm})_{[v_b+1]}) \]
\[ (e_j((F : k), b_{j1}, \ldots, b_{jm})_{[v_b]} \rightarrow e_j((F : k), b_{j1}, \ldots, b_{jm})_{[v_b+1]}) \]

\[ \text{global postcond : } \{ X \not\in S_{A[v_{va}+1]} \wedge Z \in S_{B[v_{vb}+1]} \} \]