Formal Semantics of NoSQL Evolution Operations for Different Data Heterogeneity Classes [1]

Technical Report

Mark Lukas Möller[2], Meike Klettke[3], Uta Störl[4]

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 Hetergeneity 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

[1] This Technical Report 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.

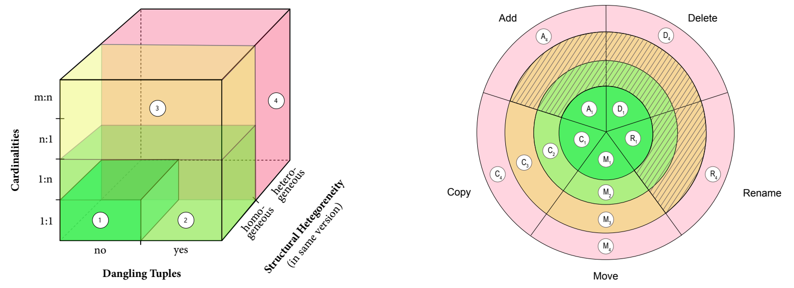
[2] University of Rostock, mark.moeller2@uni-rostock.de

[3] University of Rostock, meike.klettke@uni-rostock.de

[4] 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 1a visualizes three independent dimensions that have to be considered.



(a) Three dimensions of the four NoSQL HCs

(b) Heterogeneity Classes 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 1a). 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.

Heterogeneity Classes

HC1: In this class, the database can contain datasets in different structural versions, yet 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 matching conditions.

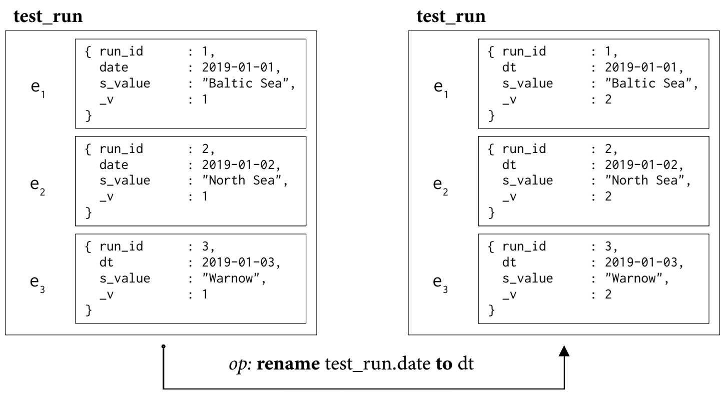
HC2: The second class encompasses HC1 and adds 1:n cardinalities. Dangling tuples can occur and have to be considered during data migration.

HC3: The third class extends HC2 to arbitrary cardinalities (1:1, 1:n, n:1, n:m).

HC4: The fourth class represents NoSQL databases that can have different structures within the same version. Here, optional properties can occur that may be available in some entities of a concrete version and missing in other entities of the same version.

NoSQL databases allows this heterogeneity even if an explicit schema, e.g., a JSON schema for a JSON database, is present.

We want to motivate with a small and simple example, why heterogeneity influences the process of data migration in NoSQL. In Figure 2, an excerpt of instances of a database is given which stores experiment data. In this excerpt, three entities in version 1 are given. The property `_v` is a system-handled property and contains the schema version of this entity. The entity's structure is heterogeneous (HC4), because two entities have an attribute `date` and the third has an attribute `dt` 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.



```
{ run_id      : 1,
  date        : 2019-01-01,
  s_value     : "Baltic Sea",
  _v          : 1
}
{ run_id      : 2,
  date        : 2019-01-02,
  s_value     : "North Sea",
  _v          : 1
}
{ run_id      : 3,
  dt          : 2019-01-03,
  s_value     : "Warnow",
  _v          : 1
}
```

```
{ run_id      : 1,
  dt          : 2019-01-01,
  s_value     : "Baltic Sea",
  _v          : 2
}
{ run_id      : 2,
  dt          : 2019-01-02,
  s_value     : "North Sea",
  _v          : 2
}
{ run_id      : 3,
  dt          : 2019-01-03,
  s_value     : "Warnow",
  _v          : 2
}
```

Fig. 2: Data Migration after a Rename Operation

Eager Migration

With an *eager migration* all entities are immediately updated and are stored in the NoSQL database in version 2. For example, we want to select all Players with a score greater than 100. In this case, the following sample query can be executed and assumes that all entities are in version 2.

```sql
SELECT run_id, date, s_value FROM test_run WHERE _v=2
```

The query returns all the entities from `2.1` to `2.3` as the result.

Lazy Migration

In case of *lazy migration*, all, some or none entities can still be present in version 1. In this approach, data is only migrated on-access. A query has to be rewritten for selecting the entities for migration. Applying the previous query is not enough. For query rewriting, (a) the migration approach, (b) the structural variants for heterogeneous datasets in a version and (c) cardinalities of multi-entity operations have to be considered.

To access all entities which are still in the previous version, the following query has to be executed:

```
SELECT run_id, dt, s_value FROM test_run WHERE _v=1)
UNION
SELECT run_id, date AS dt, s_value FROM test_run WHERE _v=1
```

Only with this query, we get the complete result set for datasets of HC4. To fetch all entities in all available versions, both queries are executed and the result sets are combined with a union operation.

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 multiple schema changes. It is favourable to have a component which keeps track of the different evolution operations and the heterogeneity of data and supports in query formulation since generating queries by hand for different kinds and versions with additional respect to schema heterogeneity is extremely error prone. We introduce the general idea of a query rewriting component in Section 4 while a detailed description is out of such a component is out of this paper's scope.

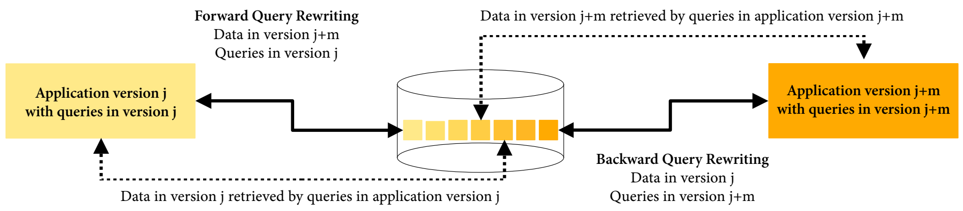

Forward Query Rewriting
Data in version j+m
Queries in version j
Data in version j+m retrieved by queries in application version j+m
Application version j with queries in version j
Application version j+m with queries in version j+m
Backward Query Rewriting
Data in version j
Queries in version j+m
Data in version j retrieved by queries in application version j

Fig. 3: Forward and Backward Query Rewriting

Forward and Backward Query Rewriting Query rewriting (QR) is applicable in two directions. In the *Backward Query Rewriting* approach, the application is aware of the most recent schema of the database while entities can be present in older versions due to a lazy migration approach. The task of the QR component is to lookup entities and their property values as if they are eagerly migrated. In Figure 3, this is shown on the right hand side. The left hand side gives the idea of *Forward Query Rewriting* which can be used to support legacy applications. When the database schema evolves, queries of an application might become broken. Instead of adapting the application, the QR component can lookup queries and try to virtually invert evolution operations as if entities in newer schema versions were not migrated in the past.

For both QR cases, it is mandatory to understand the semantics of the evolution operations and how they differ for heterogeneous datasets in contrast to homogenenous ones and how different cardinalities of multi-entity operations affect the evolution process. Therefore, we present an in-depth specification for evolution operations in different heterogeneity classes.

Contribution

In this technical report we will make the following contribution.

- We introduce four heterogeneity classes (HC1–HC4) 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 6 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.

Kind NoSQL data with an equal or similar set of properties is called a *kind*. A kind named A is defined by a *schema* S_A and a *set of entities* E_A, i.e., $\mathcal{K}_A = (S_A, E_A)$.

Schema **Entity** The schema S_A consists of a set of property names $S_A = \{A_1, \ldots, A_n\}$. The set of entities over S_A is defined as $E_A := \{e_1, \ldots, e_m\}$ where m is the number of entities. Each entity in E_A consists of up to n attributes called a_{i_j}, so $e_i = \{a_{i_j} \mid i \in \{1, \ldots, m\}, j \in \{1, \ldots n\}\}$. Here, i represents the index for the i-th entity of E_A and j is the j-th attribute of the entity.

Attribute Each attribute a_{i_j} consists of an attribute name and an attribute value, i.e., $a_{i_j} = (A_{i_j} : v_{i_j}) \in S_{A_i} \times \mathcal{D}(A_i)$ whereby $S_{A_i} \subseteq S_A$ and $\mathcal{D}(A_i) \subseteq \mathcal{D}(A)$. Thus, $S_{A_i} \times \mathcal{D}(A_i)$ represents the property domain. The property value v_{i_j} is either a null value, a boolean, a string, a number, an array, or it can contain nested properties, as is typical in NoSQL applications. If v_{i_j} contains the nested property w, the value of w is accessible by the *path expression* $v_{i_j}.w$. In case of nested properties, all paths are assumed to be available in the schema as well.

Example. Let us consider a database of a research institute storing data of experiments. We regard three different kinds storing information about projects, metadata, and the sensor values for each test run. The kind *projects* stores information about a project and is defined as $\mathcal{K}_{projects} = \{S_{projects}, E_{projects}\}$. The according schema is $S_{projects} = \{$“p_id”, “station_name”, “funder”, “budget”$\}$. A set of possible entities could read as follows:

```
E_projects = {
 {("p_id": 1), ("station_name": "Ocean"), ("funder": "DFG"), ("budget": "5 Mil")},
 {("p_id": 2), ("station_name": "Baltic Sea")}
}
```

Here, $E_{projects}$ represents the set with two entities of this kind where the first entity has two optional properties. Then, the first property of the first entity is $a_{1_1} = (A_{1_1} : v_{1_1}) = (\text{“p\_id”} : 1)$. All property names are available in the schema.

Example 2. Consider a kind for calibration data with nested data. The following entity is part of this kind:

```
E_calibdata = {{("gyroscope": {"offset": {"x"} : 10}, {"y": 2}}})
```

For this example, “gyroscope”, “gyroscope.offset”, “gyroscope.offset.x”, and “gyroscope.offset.y” are part of $S_{calibdata}$.

$\in^*$-Operator 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 $\in^*$ 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:

$$\pi_A := S_{A_i} \times \mathcal{D}(A_i) \longrightarrow S_{A_i} \text{ with } (A_{i_j}, v_{i_j}) \mapsto A_{i_j}.$$

Based on this projection we can define the $\in^*$ operator.

$$X \in^* e_i :\Leftrightarrow \exists a_{i_j} \in e_i : X \in \pi_A(a_{i_j})$$
$$X \in^* E_A :\Leftrightarrow \forall e_i \in E_A : X \in^* e_i$$

Reconsider the previous example. Here, “funder” $\in^*$ e_1 is *True* while “funder” $\in^*$ e_2 evaluates to *False*. Checking the presence of a property named “manager” evaluates to *False* for all entities.

Versioning Due to migration, we consider the same kind at different points of time. Therefore, we introduce a notation to state the *version* 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 $\mathcal{K}_A$ and $\mathcal{K}_B$.

Preconditions, Postconditions

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

Key Characteristics

- Structurally homogeneous data
- No dangling tuples
- 1:1 cardinalities

HC1 covers the simplest cases. All datasets are regular which entails that all entities of a kind have the same internal schema. Homogeneity enables to provide a reversible semantics for the 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, only a relaxed inverse can be specified, i.e. the schema can be reconstructed but not the values (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.

2.1.1 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. Formally, the operation is defined as:

Semantics of Add in HC1

▷ Add A.X = d

$$precond : \{X \notin S_{A[v_A]}\}$$
$$S_A(A_1, \ldots, A_n)_{[v_A]} \rightarrow S_A(X, A_1, \ldots, A_n)_{[v_A+1]}$$
$$\forall e_i \in E_A : (e_i(a_1, \ldots, a_n)_{[v_A]} \rightarrow e_i((X : d), a_1, \ldots, a_n)_{[v_A+1]})$$
$$postcond : \{X \in S_{A[v_A+1]} \wedge \forall e_i \in E_{A[v_A+1]} : X \in^* e_i\}$$

First, this evolution operation checks if the precondition *precond* is fulfiled, which specifies that the name of the property is currently not available in the explicit schema of kind $\mathcal{K}_A$ at version v_A. Because we previously defined for the internal schema of the entities that $a_{i_j} = (A_{i_j} : v_{i_j}) \in S_{A_i} \times D(A_i)$ 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 *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 $X \in^* e_i$.

Null Value (⊥)

Beside the possibility to add a property with a default value, it is also possible to add a property without default value, e.g. `Add A.X`. In this case, instead of $(X : d)$, the property $(X : \bot)$ is added.

It may be necessary to reverse the `Add` operation, e.g. for Forward Query Rewriting. For this, a reverse operation has to be defined. Semantically, any reverse operation should restore the schema of the previous version and should reconstruct the instances as far as possible, too. Actually, data is not re-migrated to a previous version. Conceptually, a view over the old schema is created which transparently looks up the corresponding properties across newer schema versions. We denote the backward direction of any evolution operation *op* with op^{-1}, e.g. $Add^{-1} A.X$.

Reversing the `Add` operation is easy. The added attribute is removed.

Reverse Semantics of Add in HC1

▷ **Add^{-1} A.X**

$$precond : \{X \in S_{A[v_{t+1}]} \wedge \forall e_i \in E_{A[v_t+1]} : X \in^* e_i\}$$
$$S_A(X, A_1, \ldots, A_n)_{[v_{t+1}]} \rightarrow S_A(A_1, \ldots, A_n)_{[v_t]}$$
$$\forall e_i \in E_A : (e_i((X : \bot), a_1, \ldots, a_n)_{[v_{t+1}]} \rightarrow e_i(a_1, \ldots, a_n)_{[v_t]})$$
$$postcond : \{X \notin S_{A[v_t]}\}$$

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

2.1.2 The Rename Operation

This operation renames a property of all entities of a kind. As a precondition, the property to be renamed (“old property”) has to be existent, while the new property name is not allowed to be present in advance of the operation in any kind. Formally, we can define the operation as follows:

▷ **Rename A.X To Z**

Semantics of Rename in HC1

$$precond : \{X \in S_{A[v_A]}, Z \notin S_{A[v_A]}\}$$
$$S_A(X, A_2, \ldots, A_n)_{[v_A]} \rightarrow S_A(Z, A_2, \ldots, A_n)_{[v_A+1]}$$
$$\forall e_i \in E_A : (e_i((X : x), a_2, \ldots, a_n)_{[v_A]} \rightarrow e_i((Z : x), a_2, \ldots, a_n)_{[v_A+1]})$$
$$postcond : \{X \notin S_{A[v_A+1]}, Z \in S_{A[v_A+1]}\}$$

On the schema level, the property name X is replaced by the property name Z. On the entity level, the property $(X : x)$ is modified. After the operation, the property was changed to $(Z : x)$ – The property name changed from X to Z while the property value x was preserved. After the `Rename` operation, no entity contains the property X anymore but each entity contains Z as stated in the postcondition.

2.1.3 The Delete Operation

The `Delete` operation a the property from all entities of a kind. To execute the operation, the property is required to be present in advance of the operation.

▷ **Delete A.X**

Semantics of Delete in HC1

$$precond : \{X \in S_{A[v_A]}\}$$
$$S_A(X, A_2, \ldots, A_n)_{[v_A]} \rightarrow S_A(A_2, \ldots, A_n)_{[v_{A+1}]}$$
$$\forall e_i \in E_A : e_i((X : x), a_2, \ldots, a_n)_{[v_A]} \rightarrow e_i(a_1, \ldots, a_n)_{[v_{A+1}]}$$
$$postcond : \{X \notin S_{A[v_{A+1}]} \wedge \forall e_i \in E_{A[v_A+1]} : X \notin^* e_i\}$$

In contrast to the other operations in Heterogeneity Class 1, a loss of information ocurred 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. At least, it is possible to reconstruct the schema. Instead of the original values, the lost property values are substituted with a $\perp$ value.

▷ **Delete^{-1} A.X**

Reverse Semantics of Delete in HC1

$$precond : \{X \notin S_{A[v_A]}\}$$
$$S_A(A_2, \ldots, A_n)_{[v_{A+1}]} \rightarrow S_A(X, A_2, \ldots, A_n)_{[v_A]}$$
$$\forall e_i \in E_A : e_i(a_1, \ldots, a_n)_{[v_{A+1}]} \rightarrow e_i((X : \perp), a_2, \ldots, a_n)_{[v_A]}$$
$$postcond : \{X \in S_{A[v_{A+1}]}\}$$

The ability to reconstruct the but not the values is comparable to a relaxed inverse in the context of the CHASE algorithm (c.f. [Fa11]).

2.1.4 The Move Operation

The `Move` operation is a multi-type operation which moves a property from the entities of a kind entities of a different kind based on a matching condition. In HC1, we assume an uncomplicated 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. Accordingly, 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. Furthermore this restriction creates some implicit assumptions, for instance that both kinds have the same amount of entities.

For Heterogeneity Class 1, we define the semantics of the `Move` operation as follows:

Semantics of Move in HC1

▷ **Move A.X To B.Z Where A.K = B.F**

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

$$\forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F :$$
$$(e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A]} \wedge e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B]}$$
$$\rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A+1]} \wedge e_j((Z : x), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B+1]})$$
$$postcond : \{X \notin S_{A[v_A+1]}, Z \in S_{B[v_B+1]}\}$$

In the `Move` operation, we specify the source and target kinds, in the example $\mathcal{K}_A$ and $\mathcal{K}_B$ 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 $\mathcal{K}_A$ 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, it is possible to simply "move back" the affected property with the same matching condition in consequence of the given bijectivity (each source entity has exactly one matching partner and vice versa).

2.1.5 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:

Semantics of Copy in HC1

▷ **Copy A.X To B.Z Where A.K = B.F**

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

$$\forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F :$$
$$(e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A]} \wedge e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B]}$$
$$\rightarrow e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A+1]} \wedge e_j((Z : x), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B+1]})$$
$$postcond : \{X \in S_{A[v_A+1]}, Z \in B_{S[v_B+1]}\}$$

To invert the Copy operation, it is required to to delete the property from the schema information and from the entity of the target kind (here $\mathcal{K}_B$). Because the copied property is still present in the source kind after the „forward“ operation, there is no loss of information after the operation. The reverse copy operation can be described as:

Reverse Semantics of Copy in HC1

▷ **Copy^{-1} A.X To B.Z Where A.K = B.F**

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

$$\forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F :$$
$$(e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A+1]} \wedge e_j((Z : x), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B+1]})$$
$$\rightarrow e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A]} \wedge e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B]})$$
$$postcond : \{X \in S_{A[v_A]}, Z \notin B_{S[v_B]}\}$$

Until here, we presented the definitions for five basic operations on homogeneous data and 1:1 matches for Multi-Entity operations. In the following sections, we show the impact of more complex heterogeneity classes on the semantics, their definitions and consequences.

2.2 Heterogeneity Class 2

Key Characteristics

- Structurally homogeneous data
- Dangling tuples are possible
- 1:n cardinalities

Heterogeneity Class 2 extends HC1 by 1:n cardinalities. Datasets of a kind in the same version are still homogeneous and without optional properties. Because cardinalities only affect multi-entity-operations, the semantics for the single-type operations remain the same while the semantics for `Move` and `Copy` changes.

The semantics of Heterogeneity Class 2 for backward query rewriting is very similar. Nevertheless, two main changes have to be considered:

- Due to multiple matches on the target side, it is not possible to increment the version numbers directly as in the semantics of HC1. Otherwise, they are incremented as often as an entity of the source kind matches with an entity of the target kind. Hence, a more transactional expression is required.
- It is necessary to define how properties behave which do not have a matching partner on the target kind (1:n match with n=0). For those entities, the property is simply removed.

2.2.1 The Move Operation

Operation Stages

From the effect, the `Move` operation in Heterogeneity Class 2 does the same as in HC1 but with 1:n cardinalities. Consider the difference in the semantics. The lines in the definition represent *operation stages*. A following step is only executed when the stage before is finished.

In the first stage (text lines 3–7) the case for entities of the source kind with at least one matching partner is considered. The property is moved as in HC1 with the difference that the version number is not incremented yet.

In the second step (lines 8–9), the semantics describes how to deal with source kinds without a matching partner. In this case, the property which is affected from the move operation is simply removed.

The third step increments the version numbers for all entities of the source and the target kind.

Semantics of Move in HC2

▷ **Move A.X To B.Z Where A.K = B.F**

$$precond : \{X \in S_{A[v_A]}, Z \notin S_{B[v_B]}\}$$

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

$\forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F :$
$$(e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A]} \wedge e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B]}$$
$$\rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A]} \wedge e_j((Z : x), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B]})$$

$\forall e_i \in E_A : \nexists e_j \in E_B : e_i.K = e_j.F$
$$(e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]})$$

$$e_{i[v_A]} \rightarrow e_{i[v_A+1]}$$
$$e_{j[v_B]} \rightarrow e_{j[v_B+1]}$$

$$postcond : \{X \notin S_{A[v_A+1]}, Z \in S_{B[v_B+1]}\}$$

2.2.2 The Copy Operation

A similar semantics is provided for the `Copy` operation. Instead of removing the property for entites of the source kind without a matching partner, the property remains.

Semantics of Copy in HC2

▷ **Move A.X To B.Z Where A.K = B.F**

$$precond : \{X \in S_{A[v_A]}, Z \notin S_{B[v_B]}\}$$

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

$\forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F :$
$$(e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A]} \wedge e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B]}$$
$$\rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A]} \wedge e_j((Z : x), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B]})$$

$$e_{i[v_A]} \rightarrow e_{i[v_A+1]}$$
$$e_{j[v_B]} \rightarrow e_{j[v_B+1]}$$

$$postcond : \{X \notin S_{A[v_A+1]}, Z \in S_{B[v_B+1]}\}$$

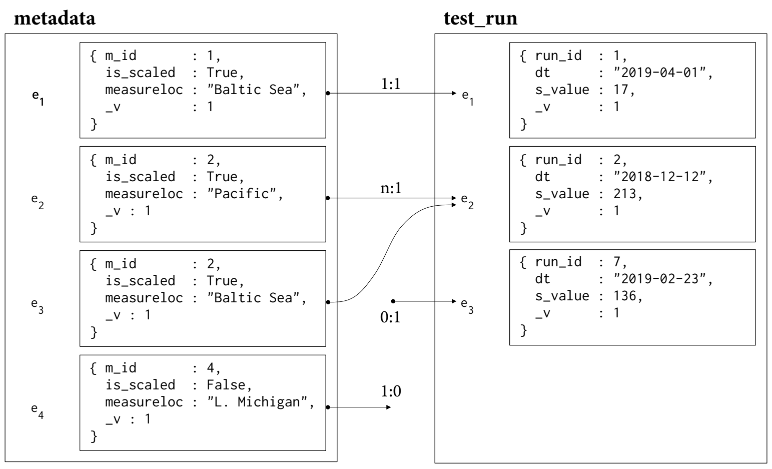
2.3 Heterogeneity Class 3

Key Characteristics

- Structurally homogeneous data
- Dangling tuples are possible
- m:n cardinalities

In this heterogeneity class, structurally homogeneous data within the same version with arbitrary cardinalities are considered. Entities without join partner on the source side or on the target side can occur and entail dangling tuples as well as side effects of these problems. The single-type operations are the same as before while the semantics of the multi-entity operation is changing again.

Arbitrary Cardinalities can cause problems regarding the `Move` and `Copy` operation as shown in Figure 4. Let us slightly adapt our running example. We introduce the kind "metadata" with the four properties "m_id", "is_scaled", "measureloc" and "_v". A `Move` operation is executed which moves the property "measureloc" from $\mathcal{K}_{metadata}$ to $\mathcal{K}_{test\_run}$ for properties where the matching condition "metadata.m_id = test_run.run_id" holds.



```
{ m_id       : 1,
  is_scaled  : True,
  measureloc : "Baltic Sea",
  _v         : 1
}
```

```
{ m_id       : 2,
  is_scaled  : True,
  measureloc : "Pacific",
  _v : 1
}
```

```
{ m_id       : 2,
  is_scaled  : True,
  measureloc : "Baltic Sea",
  _v : 1
}
```

```
{ m_id       : 4,
  is_scaled  : False,
  measureloc : "L. Michigan",
  _v : 1
}
```

```
{ run_id  : 1,
  dt      : "2019-04-01",
  s_value : 17,
  _v      : 1
}
```

```
{ run_id  : 2,
  dt      : "2018-12-12",
  s_value : 213,
  _v      : 1
}
```

```
{ run_id  : 7,
  dt      : "2019-02-23",
  s_value : 136,
  _v      : 1
}
```

Move [Overwrite|Ignore] metadata.measureloc **To** test_run.measureloc
Where meatata.m_id = test_run.run_id

Fig. 4: Possible problems of cardinalities when executing the Move operation

In this context, we see several problems that we need to solve. First, the value of `e`$_2$ of $\mathcal{K}_{test\_run}$ has to be determined since there are two possible values – "Pacific" and "Baltic Sea". Then, for the entity `e`$_2$ of $\mathcal{K}_{test\_run}$, no metadata is 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 `e`$_2$ of $\mathcal{K}_{test\_run}$ in Figure 4 – conflicht resolution strategies 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.

2.3.1 The Move Operation

In Figure 5, 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 HC3, 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 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_i, 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 seconds case describes what happens when there is already such a property with the name Z on the target side. This can happen if n:1 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 4, the value of `A.2` would be 130 after the first match. Afterwards, 130 ist 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 in 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 of the same argumentation as in HC2.

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

Semantics of Move Overwrite in HC3

▷ **Move Overwrite A.X To B.Z where A.K = B.F**

$$global\ precond : \{X \in S_{A[v_A]}, Z \notin B_{S[v_B]}\}$$

$$S_A(X, K, A_3, \dots, A_n)_{[v_A]} \rightarrow S_A(K, A_3, \dots, A_n)_{[v_A+1]}$$
$$S_B(F, B_2, \dots, B_m)_{[v_B]} \rightarrow S_B(Z, F, B_2, \dots, B_m)_{[v_B+1]}$$

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

$$case : Z \notin^* e_{j[v_B]} \begin{cases} case\ precond : \{Z \notin^* e_{j[v_B]}\} \\ \quad (e_i((X : x), (K : k), a_{i_3}, \dots, a_{i_n})_{[v_A]} \\ \quad \wedge e_j((F : k), b_{j_2}, \dots, b_{j_m})_{[v_B]} \\ \quad \rightarrow e_i((X : x), (K : k), a_{i_3}, \dots, a_{i_n})_{[v_A]} \\ \quad \wedge e_j((Z : x), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_B]}) \\ case\ postcond : \{Z \in^* e_{i[v_B]}\} \end{cases}$$

$$case : Z \in^* e_{j[v_B]} \begin{cases} case\ precond : \{Z \in^* e_{j[v_B]}\} \\ \quad (e_i((X : x), (K : k), a_{i_3}, \dots, a_{i_n})_{[v_A]} \\ \quad \wedge e_j((Z : x'), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_B]} \\ \quad \rightarrow e_i((X : x), (K : k), a_{i_3}, \dots, a_{i_n})_{[v_A]} \\ \quad \wedge e_j((Z : x), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_B]}) \\ case\ postcond : \{Z \in^* e_{i[v_B]}\} \end{cases}$$

$$e_i((X : x), (K : k), a_{i_3}, \dots, a_{i_n})_{[v_A]} \rightarrow e_i((K : k), a_{i_3}, \dots, a_{i_n})_{[v_A+1]}$$
$$e_{j[v_B]} \rightarrow e_{j[v_B+1]}$$

$(\forall e_i \in E_A : \nexists e_j \in E_B : e_i.K = e_j.F) \vee (\forall e_j \in E_B \nexists e_i \in E_A : e_j.F = e_i.K) :$

$$(e_i((X : x), (K : k), a_{i_3}, \dots, a_{i_n})_{[v_a]} \rightarrow e_i((K : k), a_{i_3}, \dots, a_{i_n})_{[v_a+1]})$$
$$(e_j((F : k), b_{j_2}, \dots, b_{j_m})_{[v_b]} \rightarrow e_j((Z : \bot), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_b+1]})$$

$$global\ postcond : \{X \notin S_{A[v_a+1]}, Z \in S_{B[v_b+1]}\}$$

Fig. 5: Definition of the Move Overwrite Operation, HC2

name Z and with $\bot$ as the property value is added to the entities of the target kind. This ensures schema homogeneity.

The Ignore Conflict Resolution Strategy

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:

$$case : Z \in^* e_{j[v_B]} \begin{cases} case\ precond : \{Z \in^* e_{j[v_B]}\} \\ (e_i((X:x),(K:k),a_{i_3},\dots,a_{i_n})_{[v_A]} \\ \wedge e_j((Z:x'),(F:k),b_{j_2},\dots,b_{j_m})_{[v_B]} \\ \rightarrow e_i((X:x),(K:k),a_{i_3},\dots,a_{i_n})_{[v_A]} \\ \wedge e_j((\mathbf{Z}:\mathbf{x'}),(F:k),b_{j_2},\dots,b_{j_m})_{[v_B]}) \\ case\ postcond : \{Z \in^* e_{i[v_B]}\} \end{cases}$$

Since in both approaches both cases have the same postcondition and there may be overwritten or ignored property values, it is impossible to find an exact query inverse. In case more sophisticated function are applied for handling n:1 and n:m matches, for instance the aggregate functions *sum* or *avg*, we neither cannot inverse the operations. However, because we assumed structural homogeneity in this Heterogeneity Class, at least finding of a relaxed inverse is possible.

2.3.2 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:

Semantics of Copy Overwrite in HC3

▷ **Copy Overwrite A.X To B.Z where A.K = B.F**

$$global\ precond : \{X \in S_{A[v_A]}, Z \notin B_{S[v_B]}\}$$

$$S_A(X,K,A_3,\dots,A_n)_{[v_A]} \rightarrow S_A(X,K,A_3,\dots,A_n)_{[v_A+1]}$$
$$S_B(F,B_2,\dots,B_m)_{[v_B]} \rightarrow S_B(Z,F,B_2,\dots,B_m)_{[v_B+1]}$$

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

$$case : Z \notin^* e_{j[v_B]} \begin{cases} case\ precond : \{Z \notin^* e_{j[v_B]}\} \\ (e_i((X:x),(K:k),a_{i_3},\dots,a_{i_n})_{[v_A]} \\ \wedge e_j((F:k),b_{j_2},\dots,b_{j_m})_{[v_B]} \\ \rightarrow e_i((X:x),(K:k),a_{i_3},\dots,a_{i_n})_{[v_A]} \\ \wedge e_j((Z:x),(F:k),b_{j_2},\dots,b_{j_m})_{[v_B]}) \\ case\ postcond : \{Z \in^* e_{i[v_B]}\} \end{cases}$$

$$case : Z \in^* e_{j[v_B]} \begin{cases} case\ precond : \{Z \in^* e_{j[v_B]}\} \\ (e_i((X:x),(K:k),a_{i_3},\dots,a_{i_n})_{[v_A]} \\ \wedge e_j((Z:x'),(F:k),b_{j_2},\dots,b_{j_m})_{[v_B]} \\ \rightarrow e_i((X:x),(K:k),a_{i_3},\dots,a_{i_n})_{[v_A]} \\ \wedge e_j((Z:x),(F:k),b_{j_2},\dots,b_{j_m})_{[v_B]}) \\ case\ postcond : \{Z \in^* e_{i[v_B]}\} \end{cases}$$

$$e_{i[v_A]} \rightarrow e_{i[v_A+1]}$$
$$e_{j[v_B]} \rightarrow e_{j[v_B+1]}$$

$(\forall e_i \in E_A : \nexists e_j \in E_B : e_i.K = e_j.F) \vee (\forall e_j \in E_B \nexists e_i \in E_A : e_j.F = e_i.K) :$

$$e_{i[v_A]} \rightarrow e_{i[v_A+1]}$$
$$(e_j((F:k),b_{j_2},\dots,b_{j_m})_{[v_b]} \rightarrow e_j((Z:\perp),(F:k),b_{j_2},\dots,b_{j_m})_{[v_b+1]}$$

$$global\ postcond : \{X \notin S_{A[v_a+1]}, Z \in S_{B[v_b+1]}\}$$

The necessary changes for the `Ignore` approach of the `Copy` operation are comparable to the `Move` operation. Here, the semantics block for the second case can be replaced with the following:

$$case : Z \in^* e_{j[v_B]} \left\{ \begin{array}{l} case\ precond : \{Z \in^* e_{j[v_B]}\} \\ (e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A]} \\ \wedge e_j((Z : x'), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B]} \\ \rightarrow e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A]} \\ \wedge e_j((\mathbf{Z} : \mathbf{x}'), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B]}) \\ case\ postcond : \{Z \in^* e_{i[v_B]}\} \end{array} \right.$$

2.4 Heterogeneity Class 4

Key Characteristics

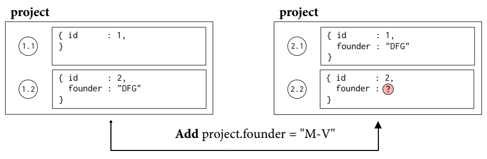
- Heterogeneous data
- Dangling tuples are possible
- m:n cardinalities

Operations of HC4 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 6. Here, a simple `Add` operation is executed. The first entity is updated. When the property “founder” is added to the second entity it is necessary to decide if the value of “founder” is either ignored and the property value “DFG” is preserved, or if the value is overwritten. Again, we introduce for our approach the additional keywords `Overwrite` and `Ignore` for specifying the conflict resolution strategies. In contrast to previous heterogeneity classes, this problem also affects single type operations (like the `Add` operation) or even for matches with a 1:1 cardinality.

Optional Property Notation ($\overset{?}{\in}$)

For heterogeneous data sets we need to specify optionality in the semantics. We denote optional attributes with a question mark. For example, $X \overset{?}{\in} S_A$ 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 $X \overset{?}{\in} S_A$.



```
{ id      : 1,
}

{ id      : 2,
  founder : "DFG"
}

{ id      : 1,
  founder : "DFG"
}

{ id      : 2,
  founder : ?
}
```

Fig. 6: Possible problem with heterogenous data with the example operation `Add Player.points =25` .

2.4.1 The Add operation

In HC4, we often have to distinguish a couple of cases. After the `Add` operation, we can formulate postconditions for the schema level, no matter if we used the `Overwrite` or the `Ignore` approach. We start with the `Overwrite` approach.

Semantics for Add Overwrite in HC4

▷ **Add Overwrite A.X = d**

$$global\ precond : \{X \overset{?}{\in} S_A\}$$

$$S_A(X?, A_2, \ldots, A_n)_{[v_t]} \rightarrow S_A(X, A_2, \ldots, A_n)_{[v_{t+1}]}$$

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

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

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

$$global\ postcond : \{X \in S_{A[v_{t+1}]}, \forall e_i \in E_{A[v_{t+1}]} : X \in e_i\}$$

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 $\perp$.

Note about optionality in HC4

Please note that in HC4 *all* properties are considered as optional that do not directly affect the operation (here: $A_2, \ldots, A_n$). For better readability of the operations, we annotate optionality for properties which have impact on the operation only (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:

Semantics for Add Ignore in HC4

▷ **Add Ignore A.X = d**

$$global\ precond : \{X \overset{?}{\in} S_A\}$$

$$S_A(X?, A_2, \dots, A_n)_{[v_t]} \rightarrow S_A(X, A_2, \dots, A_n)_{[v_{t+1}]}$$

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

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

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

$$global\ postcond : \{X \in S_{A[v_{t+1}]}, \forall e_i \in E_{A[v_{t+1}]} : X \in^* e_i\}$$

In both cases, it is not trivial to invert the `Add` operation. It is unclear if the added property can be removed as in HC1 because we do not have the information whether the property was present before the information or not. Nevertheless, to be able to specify a reverse operation, the property will be removed without respect to the presence before the operation.

2.4.2 The Delete operation

The definition of the Delete operation in Heterogeneity Class 4 is relatively simple. The property is removed 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:

Semantics of Delete in HC4

▷ **Delete A.X**

$$global\ precond : \{X \overset{?}{\in} S_A\}$$

$$S_A(X?, A_2, \dots, A_n)_{[v_t]} \rightarrow S_A(A_2, \dots, A_n)_{[v_{t+1}]}$$

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

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

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

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

2.4.3 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 4, 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:

Semantics of Rename Overwrite in HC4

▷ **Rename Overwrite A.X To Z**

$$global\ precond : \{X \overset{?}{\in} S_A, Z \overset{?}{\in} S_A\}$$

$$S_A(X?, Z?A_3, \dots, A_n)_{[v_t]} \rightarrow S_A(Z?, A_3, \dots, A_n)_{[v_{t+1}]}$$

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

$$case : X \in e_{i[v_t]} \wedge Z \notin e_{i[v_t]} \begin{cases} case\ precond : \{X \in e_{i[v_t]} \wedge Z \notin e_{i[v_t]}\} \\ \quad e_i((X : x), a_{i_3}, \dots, a_{i_n})_{[v_t]} \\ \quad \rightarrow e_i((Z : x), a_{i_3}, \dots, a_{i_n})_{[v_{t+1}]} \\ case\ postcond : \{X \notin e_{i[v_t]} \wedge Z \in e_{i[v_t]}\} \end{cases}$$

$$case : X \in e_{i[v_t]} \wedge Z \in e_{i[v_t]} \begin{cases} case\ precond : \{X \in e_{i[v_t]} \wedge Z \in e_{i[v_t]}\} \\ \quad e_i((X : x), (Z : z), a_{i_3}, \dots, a_{i_n})_{[v_t]} \\ \quad \rightarrow e_i((Z : x), a_{i_3}, \dots, a_{i_n})_{[v_{t+1}]} \\ case\ postcond : \{X \notin e_{i[v_t]} \wedge Z \in e_{i[v_t]}\} \end{cases}$$

$$case : X \notin e_{i[v_t]} \wedge Z \in e_{i[v_t]} \begin{cases} case\ precond : \{X \notin e_{i[v_t]} \wedge Z \in e_{i[v_t]}\} \\ \quad e_i((Z : z), a_{i_3}, \dots, a_{i_n})_{[v_t]} \\ \quad \rightarrow e_i((Z : z), a_{i_3}, \dots, a_{i_n})_{[v_{t+1}]} \\ case\ postcond : \{X \notin e_{i[v_t]} \wedge Z \in e_{i[v_t]}\} \end{cases}$$

$$case : X \notin e_{i[v_t]} \wedge Z \notin e_{i[v_t]} \begin{cases} case\ precond : \{X \notin e_{i[v_t]} \wedge Z \notin e_{i[v_t]}\} \\ \quad e_i(a_{i_3}, \dots, a_{i_n})_{[v_t]} \\ \quad \rightarrow e_i((Z : \perp), a_{i_3}, \dots, a_{i_n})_{[v_{t+1}]} \\ case\ postcond : \{X \notin e_{i[v_t]} \wedge Z \in e_{i[v_t]}\} \end{cases}$$

$$global\ postcond : \{X \notin S_{A[v_{t+1}]}, Z \in S_{A[v_{t+1}]}\}$$

Because it is not known whether the source (X) or the target property (Z) name is present before the operation, both properties are considered to be optional.

The conditions of the first case are equal to those of 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 `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 NULL value analogously to the `Move` and `Copy` operation in HC2. With respect to the application context, it might make more sense not to introduce a NULL value but to remain the entity unchanged. This could be covered by introducing another keyword, e.g. `Default Null` or `Default None` which is evaluated if there is no property after the operation. Due to the broad variety of possiblities, we restric our semantics to the Overwrite and Ignore approach in the scope of this paper.

Semantics of Rename Ignore in HC4

The `Rename Ignore` approach works quite similar to the introduced `Rename Overwrite` approach. Here, the semantics remains the same with the difference of the second case which is defined as follows.

$$case : X \in e_{i[v_t]} \wedge Z \in e_{i[v_t]} \begin{cases} case\ precond : \{X \in e_{i[v_t]} \wedge Z \in e_{i[v_t]}\} \\ \quad e_i((X:x),(Z:z),a_{i_3},\dots,a_{i_n})_{[v_t]} \\ \quad \rightarrow e_i((Z:z),a_{i_3},\dots,a_{i_n})_{[v_{t+1}]} \\ case\ postcond : \{X \notin e_{i[v_t]} \wedge Z \in e_{i[v_t]}\} \end{cases}$$

If the new and the old property name both exist in advance of the operation, we keep the existing value instead of overwriting it.

2.4.4 The Move Operation

The multi entity operations `Move` and `Copy` are the most difficult cases in HC4. All possible cardinalities have to be considered as before. Due to schema homogeneity, even for 1:1 cardinalities several cases need to be are distinguished.

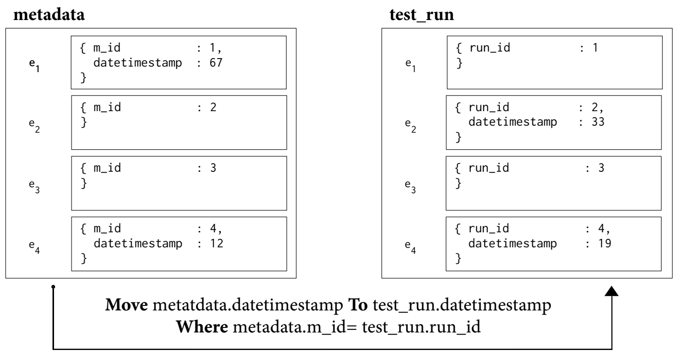

metadata
test_run
e1
{ m_id : 1, datetimestamp : 67 }
e2
{ m_id : 2 }
e3
{ m_id : 3 }
e4
{ m_id : 4, datetimestamp : 12 }
e1
{ run_id : 1 }
e2
{ run_id : 2, datetimestamp : 33 }
e3
{ run_id : 3 }
e4
{ run_id : 4, datetimestamp : 19 }
Move metatdata.datetimestamp To test_run.datetimestamp
Where metadata.m_id= test_run.run_id

Fig. 7: Emerging cases due to schema heterogeneity in case of matches with 1:1 cardinalities

Figure 7 depicts all the cases which can occur in the 1:1 case for the `Move` operation. The first match depicts the case where “datetimstamp” is present in the entity $\mathcal{K}_{metadata}$ but not in the entity of $\mathcal{K}_{test\_run}$ and can be easily moved. The second case describes that “datetimstamp” is not present in the entity of $\mathcal{K}_{metadata}$ but in the entity $\mathcal{K}_{test\_run}$. Here, the already existing value for “datetimestamp” in the entity of $\mathcal{K}_{test\_run}$ is preserved. The third case describes the case that “datetimestamp” is not present in any of both entities and the last case depicts the case where the property is part of both entities. It is required that all 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 $S_{metadata}$) does not contain X (“datetimestamp”) anymore and S_B (S_{test_run}) definitely contains Z (“datetimestamp”). 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 $\perp$ value on the target side ($\mathcal{K}_B$), respectively.

The definition of the `Move` operation for HC4 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` operation for HC4 is given in the Appendix of this article, too.

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 3 shows *forward and backward query rewriting* for such applications.

We explain the query rewriting procedure and the application of the semantics for a given example. The 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. We show how to rewrite data with a lazy migration approach for data which rely in two different versions. In the following chapter, we will derive a rewriting component for multiple versions.

Generally, only read-only queries are considered. If there is a writing query, all affected entities are migrated into the latest version.

3.1 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:

For the given HC, Schema evolution operation and original Query, the Query is modified to:

```
SELECT * FROM A WHERE version = vA
UNION
SELECT *, d AS X FROM A WHERE version = vA − 1.
```

For datasets in Heterogeneity Class 4, the query rewriting is much more complicated. Here we get the following rewritten query:

Query for Heterogeneity Class 4 (conflict resolution strategy: Ignore):

For the given HC, Schema evolution operation and original Query, the Query is modified to:

```
SELECT * FROM A WHERE version = vA
UNION
SELECT * FROM A WHERE version vA − 1 AND Exists(A.x)
UNION
SELECT *, d AS X FROM A WHERE version = vA AND NOT Exists(A.x)
```

Exists Keyword

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, similar as in the semantics.

3.1.1 Query for Heterogeneity Class 4 (conflict resolution strategy: Overwrite):

For the given HC, Schema evolution operation and original Query, the Query is modified to:

```
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`.

3.2 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 an `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 – 4:

For the given HC, Schema evolution operation and original Query, the Query is modified to:

```
SELECT * FROM A WHERE version = v_A
UNION
SELECT < PropertyList \ {X} > From A Where version = v_A − 1.
```

Rewriting the `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 `PropertyList`) without the property X. The semantics for HC1 and HC4 is identical except for the precondition and there is no difference in query rewriting for different heterogeneity classes.

3.3 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 `Rename` operation:

Schema evolution operation: `Rename A.X To Z`

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:

For the given HC, Schema evolution operation and original Query, the Query is modified to:

```
SELECT * FROM A WHERE version = vA
UNION
SELECT < 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 `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.

Query for Heterogeneity Class 4 (Conflict Resolution Strategy: Ignore):

For the given HC, Schema evolution operation and original Query, the Query is modified to:

```
SELECT * FROM A WHERE version = vA
UNION
SELECT < PropertyList \ {X} >, X AS Z FROM A WHERE version = vA − 1 AND
↪  Exists(A.X) AND NOT Exists(A.Z)
UNION
SELECT < PropertyList \ {x} >, Null AS Z FROM A WHERE version = vA − 1 AND NOT
↪  Exists(A.X) AND NOT Exists(A.Z)
```

```
UNION
SELECT * FROM A WHERE version = vA − 1 AND NOT Exists(A.X) AND EXISTS(A.Z)
UNION
SELECT < PropertyList \ {x} > FROM A WHERE version = vA − 1 AND Exists(A.X) AND
↪   Exists(A.Z)
```

Query for Heterogeneity Class 3 (Conflict Resolution Strategy: Overwrite):

For the given HC, Schema evolution operation and original Query, the Query is modified to:

```
SELECT * FROM A WHERE version = vA
UNION
SELECT < PropertyList \ {X} >, X AS Z FROM A WHERE version = vA − 1 AND
↪   Exists(A.X) AND NOT Exists(A.Z)
Union
SELECT < PropertyList \ {X} >, Null AS Z FROM A WHERE version = vA − 1 AND NOT
↪   Exists(A.X) AND NOT Exists(A.Z)
UNION
SELECT * FROM A WHERE version = vA − 1 AND NOT Exists(A.X) AND Exists(A.Z)
UNION
SELECT < PropertyList \ {X, Z} >, X AS Z FROM A WHERE version = vA − 1 AND
↪   Exists(A.X) AND 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.

3.4 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:

For the given HC, Schema evolution operation and original Query, the Query is modified to:

```
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 ist 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 3 (conflict resolution strategy: Ignore):

For the given HC, Schema evolution operation and original Query, the Query is modified to:

```
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[5]. 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. As a kind of semantic sugar, we introduce the `FIRST MATCH ONLY` to prescind the `Distinct on` clause which is only Postgres related. Other variants to join the first matching partner only are possible, too[6], and have to be adapted to the used database system.

[5] PostgreSQL Flavor. The `Distinct On(...)` clause checks for the first match and ignores any additional ones.
[6] https://www.periscopedata.com/blog/4-ways-to-join-only-the-first-row-in-sql [2019-04-20]

Query for Heterogeneity Class 4 (conflict resolution strategy: Ignore):

For the given HC, Schema evolution operation and original Query, the Query is modified to:

```
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 B.* 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 B.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 partners in kind A whereby the matching partner does not have the property `A.X`. In this case, we substitute a NULL value. 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 NULL values for corresponding $B.z$ property values. The subqueries of the above given query are directly derived from the semantics of the `Move` operation in HC3. Each case in this definition of the evolution operation generates one subselect clause of the query.

3.5 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:

For the given HC, Schema evolution operation and original Query, the Query is modified to:

```
SELECT * FROM B WHERE version = vB
UNION
SELECT B.*, A.X AS Z FROM A, B WHERE A.A = B.B AND A.version = vA − 1 AND B.version
↪   = vB − 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 – 4:

Analogously to the previous reason, the `Copy` operation queries in HC2 and HC3 are equal to the queries of the `Move` operation in HC2 and HC3.

4 Outlook: Conceptual Model of a Query Rewriting Component

We introduced a semantics for schema evolution operations in different heterogeneity classes. The queries of the previous section were generated by hand. It is desirable to have a component which rewrites queries automatically. While developing a query rewriting component is out of the scope of this paper, we want to give an outlook of the conceptual model of such a component.

Lazy migrated entities can be present in several versions in a database. Therefore, the component needs metadata information which operation affected which kind in which schema version. Additionally, it is necessary to keep track of all versions of a kind.

The QR component needs to take different kinds into account when multi-entity-operations were applied. If data needs to be looked up in several versions, a query is generated for each version. A recursive approach is conceivable which generates query for a version by adapting the query of the previous version. This entails a high amount of subquery generation of both the source and the target kind of multi-entity operations are present in several versions due to a lazy migration approach. If multiple kinds need to be taken into account because a property was moved or copied across several kinds, this entails a rewriting cascade.

Conceptually, the metadata for such a QR can be modelled as a graph whereby the nodes represent the kinds and the edges represent the operations. The edge labels are parametrized with information about the version of the source and target kind, the name of the source and target kind and the parametrized operation itself.

We plan to develop and implement a QR routine based on this graph-based model and based on the semantics in the near future.

5 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.

6 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. A tool for determining the heterogeneity class of a NoSQL database exists.

References

[AHV95] Abiteboul, Serge; Hull, Richard; Vianu, Victor: Foundations of Databases. Addison-Wesley, 1995.

[Fa11] Fagin, Ronald; Kolaitis, Phokion G.; Popa, Lucian et al.: Schema Mapping Evolution Through Composition and Inversion. In: Schema Matching and Mapping, Data-Centric Systems and Applications. Springer, 2011.

[Ha18] Hamadou, Hamdi Ben; Ghozzi, Faïza; Péninou, André et al.: Towards Schema-independent Querying on Document Data Stores. In: Proc. of the 20th EDBT/10th ICDT (Joint Conference), Vienna. 2018.

[He17] Herrmann, Kai; Voigt, Hannes; Rausch, Jonas; Behrend, Andreas; Lehner, Wolfgang: Living in Parallel Realities — Co-Existing Schema Versions with a Bidirectional Database Evolution Language. In: Proc. SIGMOD '17. 2017.

[KSS15] Klettke, Meike; Störl, Uta; Scherzinger, Stefanie: Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In: Proc. 16. BTW, Hamburg, Germany. volume 241 of LNI. GI, 2015.

[KSS16] Klettke, Meike; Störl, Uta; Shenavai, Manuel: NoSQL schema evolution and big data migration at scale. In: 2016 IEEE International Conference on Big Data, BigData 2016, Washington DC. IEEE, 2016.

[MCZ10] Moon, Hyun Jin; Curino, Carlo A.; Zaniolo, Carlo: Scalable Architecture and Query Optimization for Transaction-time DBs with Evolving Schemas. In: Proc. SIGMOD'10. 2010.

[PS11] Pichler, Reinhard; Skritek, Sebastian: The complexity of evaluating tuple generating dependencies. In: Database Theory - ICDT 2011, 14th International Conference, Uppsala, 2011. ACM, 2011.

[Ro92] Roddick, John F.: Schema Evolution in Database Systems - An Annotated Bibliography. SIGMOD record, 21(4):35–40, 1992.

[SDH16] Saur, Karla; Dumitras, Tudor; Hicks, Michael W.: Evolving NoSQL Databases without Downtime. In: ICSME. IEEE Computer Society, 2016.

[SKS13] Scherzinger, Stefanie; Klettke, Meike; Störl, Uta: Managing Schema Evolution in NoSQL Data Stores. Proc. DBPL, CoRR, abs/1308.0514, abs/1308.0514, 2013.

[St17] Stenzel, Julian: Query Rewriting in NoSQL-Datenbanksystemen. Master's thesis, University of Applied Sciences Darmstadt, 2017.

A Appendix

A.1 Move Overwrite Semantics in Heterogeneity Class 4

Semantics of Move Overwrite in HC4

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

$$\mathit{global\ precond} : \{X \overset{?}{\in} S_{A[v_a]}, Z \overset{?}{\in} S_{B[v_b]}\}$$

$$S_A(X?, K?, A_3?, \ldots, A_n?)_{[v_a]} \rightarrow S_A(K?, A_3?, \ldots, A_n?)_{[v_a+1]}$$
$$S_B(F?, B_2?, \ldots, B_m?)_{[v_b]} \rightarrow S_B(Z, F?, B_2?, \ldots, B_m?)_{[v_b+1]}$$

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

$$\mathit{case} : X \in^* e_{i[v_a]} \left\{ \begin{array}{l}
\mathit{case} : Z \notin^* e_{j[v_b]} \left\{ \begin{array}{l}
\mathit{case\ precond} : \{X \in^* e_{i[v_a]} \wedge Z \notin^* e_{j[v_b]}\} \\
\quad (e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\
\quad\quad \wedge e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \\
\quad \rightarrow e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\
\quad\quad \wedge e_j((Z : x), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]}) \\
\mathit{case\ postcond} : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\}
\end{array} \right. \\
\mathit{case} : Z \in^* e_{j[v_b]} \left\{ \begin{array}{l}
\mathit{case\ precond} : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \\
\quad (e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\
\quad\quad \wedge e_j((Z : z), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \\
\quad \rightarrow e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\
\quad\quad \wedge e_j((\mathbf{Z} : \mathbf{x}), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]}) \\
\mathit{case\ postcond} : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\}
\end{array} \right. \\
\; e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a+1]} \\
\; e_{j[v_b]} \rightarrow e_{j[v_b+1]} \\
\; \mathit{case\ postcond} : \{X \notin^* e_{i[v_a+1]} \wedge Z \in^* e_{j[v_b+1]}\}
\end{array} \right.$$

$$\mathit{case} : X \notin^* e_{i[v_a]} \left\{ \begin{array}{l}
\mathit{case} : Z \in^* e_{j[v_b]} \left\{ \begin{array}{l}
\mathit{case\ precond} : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \\
\quad (e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\
\quad\quad \wedge e_j((Z : z), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \\
\quad \rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\
\quad\quad \wedge e_j((Z : z), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]}) \\
\mathit{case\ postcond} : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\}
\end{array} \right. \\
\mathit{case} : Z \notin^* e_{j[v_b]} \left\{ \begin{array}{l}
\mathit{case\ precond} : \{X \notin^* e_{i[v_a]} \wedge Z \notin^* e_{j[v_b]}\} \\
\quad (e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\
\quad\quad \wedge e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \\
\quad \rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\
\quad\; x \wedge e_j((Z : \perp), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]}) \\
\mathit{case\ postcond} : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\}
\end{array} \right. \\
e_{i[v_a]} \rightarrow e_{i[v_a+1]} \\
e_{j[v_b]} \rightarrow e_{j[v_b+1]} \\
\mathit{case\ postcond} : \{X \notin^* e_{i[v_a+1]} \wedge Z \in^* e_{j[v_b+1]}\}
\end{array} \right.$$

$(\forall e_i \in E_A : \nexists e_j \in E_B : e_i.K = e_j.F) \vee (\forall e_j \in E_B \nexists e_i \in E_A : e_j.F = e_i.K) :$

$$(e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a+1]})$$
$$(e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a+1]})$$
$$(e_j((Z : z), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \rightarrow e_j((Z : z), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b+1]})$$
$$(e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \rightarrow e_j((Z : \perp), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b+1]})$$

$$\mathit{global\ postcond} : \{X \notin S_{A[v_a+1]}, Z \in S_{B[v_b+1]}\}$$

A.2 Move Ignore Semantics in Heterogeneity Class 4

Semantics of Move Ignore in HC4

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

$$global\ precond : \{X \overset{?}{\in} S_{A[v_a]}, Z \overset{?}{\in} S_{B[v_b]}\}$$

$$S_A(X?, K?, A_3?, \ldots, A_n?)_{[v_a]} \rightarrow S_A(K?, A_3?, \ldots, A_n?)_{[v_a+1]}$$

$$S_B(F?, B_2?, \ldots, B_m?)_{[v_b]} \rightarrow S_B(Z, F?, B_2?, \ldots, B_m?)_{[v_b+1]}$$

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

$$case : X \in^* e_{i[v_a]} \left\{ \begin{array}{l} case : Z \notin^* e_{j[v_b]} \left\{ \begin{array}{l} case\ precond : \{X \in^* e_{i[v_a]} \wedge Z \notin^* e_{j[v_b]}\} \\ \quad (e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\ \quad\quad \wedge e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \\ \quad \rightarrow e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\ \quad\quad \wedge e_j((Z : x), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]}) \\ case\ postcond : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \end{array} \right. \\ case : Z \in^* e_{j[v_b]} \left\{ \begin{array}{l} case\ precond : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \\ \quad (e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\ \quad\quad \wedge e_j((Z : z), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \\ \quad \rightarrow e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\ \quad\quad \wedge e_j((\mathbf{Z} : \mathbf{z}), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]}) \\ case\ postcond : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \end{array} \right. \\ e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a+1]} \\ e_{j[v_b]} \rightarrow e_{j[v_b+1]} \\ case\ postcond : \{X \notin^* e_{i[v_a+1]} \wedge Z \in^* e_{j[v_b+1]}\} \end{array} \right.$$

$$case : X \notin^* e_{i[v_a]} \left\{ \begin{array}{l} case : Z \in^* e_{j[v_b]} \left\{ \begin{array}{l} case\ precond : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \\ \quad (e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\ \quad\quad \wedge e_j((Z : z), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \\ \quad \rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\ \quad\quad \wedge e_j((Z : z), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]}) \\ case\ postcond : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \end{array} \right. \\ case : Z \notin^* e_{j[v_b]} \left\{ \begin{array}{l} case\ precond : \{X \notin^* e_{i[v_a]} \wedge Z \notin^* e_{j[v_b]}\} \\ \quad (e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\ \quad\quad \wedge e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \\ \quad \rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \\ \quad\quad x \wedge e_j((Z : \bot), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]}) \\ case\ postcond : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \end{array} \right. \\ e_{i[v_a]} \rightarrow e_{i[v_a+1]} \\ e_{j[v_b]} \rightarrow e_{j[v_b+1]} \\ case\ postcond : \{X \notin^* e_{i[v_a+1]} \wedge Z \in^* e_{j[v_b+1]}\} \end{array} \right.$$

$(\forall e_i \in E_A : \nexists e_j \in E_B : e_i.K = e_j.F) \vee (\forall e_j \in E_B \nexists e_i \in E_A : e_j.F = e_i.K) :$

$$(e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a+1]})$$

$$(e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a]} \rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_a+1]})$$

$$(e_j((Z : z), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \rightarrow e_j((Z : z), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b+1]})$$

$$(e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b]} \rightarrow e_j((Z : \bot), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_b+1]})$$

$$global\ postcond : \{X \notin S_{A[v_a+1]}, Z \in S_{B[v_b+1]}\}$$

A.3 Copy Overwrite Semantics in Heterogeneity Class 4

Semantics of Copy Overwrite in HC4

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

$$\mathit{global\ precond} : \{X \overset{?}{\in} S_{A[v_a]}, Z \overset{?}{\in} S_{B[v_b]}\}$$

$$S_A(X?, K?, A_3?, \dots, A_n?)_{[v_a]} \rightarrow S_A(X?, K?, A_3?, \dots, A_n?)_{[v_a+1]}$$
$$S_B(F?, B_2?, \dots, B_m?)_{[v_b]} \rightarrow S_B(Z, F?, B_2?, \dots, B_m?)_{[v_b+1]}$$

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

$$\mathit{case} : X \in^* e_{i[v_a]} \begin{cases} \mathit{case} : Z \notin^* e_{j[v_b]} \begin{cases} \mathit{case\ precond} : \{X \in^* e_{i[v_a]} \wedge Z \notin^* e_{j[v_b]}\} \\ (e_i((X : x), (K : k), a_{i_3}, \dots, a_{i_n})_{[v_a]} \\ \quad \wedge e_j((F : k), b_{j_2}, \dots, b_{j_m})_{[v_b]} \\ \rightarrow e_i((X : x), (K : k), a_{i_3}, \dots, a_{i_n})_{[v_a]} \\ \quad \wedge e_j((Z : x), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_b]}) \\ \mathit{case\ postcond} : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \end{cases} \\ \mathit{case} : Z \in^* e_{j[v_b]} \begin{cases} \mathit{case\ precond} : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \\ (e_i((X : x), (K : k), a_{i_3}, \dots, a_{i_n})_{[v_a]} \\ \quad \wedge e_j((Z : z), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_b]} \\ \rightarrow e_i((X : x), (K : k), a_{i_3}, \dots, a_{i_n})_{[v_a]} \\ \quad \wedge e_j((\mathbf{Z} : \mathbf{x}), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_b]}) \\ \mathit{case\ postcond} : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \end{cases} \\ e_{i[v_a]} \rightarrow e_{i[v_a+1]} \\ e_{j[v_b]} \rightarrow e_{j[v_b+1]} \\ \mathit{case\ postcond} : \{X \notin^* e_{i[v_a+1]} \wedge Z \in^* e_{j[v_b+1]}\} \end{cases}$$

$$\mathit{case} : X \notin^* e_{i[v_a]} \begin{cases} \mathit{case} : Z \in^* e_{j[v_b]} \begin{cases} \mathit{case\ precond} : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \\ (e_i((K : k), a_{i_3}, \dots, a_{i_n})_{[v_a]} \\ \quad \wedge e_j((Z : z), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_b]} \\ \rightarrow e_i((K : k), a_{i_3}, \dots, a_{i_n})_{[v_a]} \\ \quad \wedge e_j((Z : z), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_b]}) \\ \mathit{case\ postcond} : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \end{cases} \\ \mathit{case} : Z \notin^* e_{j[v_b]} \begin{cases} \mathit{case\ precond} : \{X \notin^* e_{i[v_a]} \wedge Z \notin^* e_{j[v_b]}\} \\ (e_i((K : k), a_{i_3}, \dots, a_{i_n})_{[v_a]} \\ \quad \wedge e_j((F : k), b_{j_2}, \dots, b_{j_m})_{[v_b]} \\ \rightarrow e_i((K : k), a_{i_3}, \dots, a_{i_n})_{[v_a]} \\ \quad x \wedge e_j((Z : \perp), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_b]}) \\ \mathit{case\ postcond} : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \end{cases} \\ e_{i[v_a]} \rightarrow e_{i[v_a+1]} \\ e_{j[v_b]} \rightarrow e_{j[v_b+1]} \\ \mathit{case\ postcond} : \{X \notin^* e_{i[v_a+1]} \wedge Z \in^* e_{j[v_b+1]}\} \end{cases}$$

$(\forall e_i \in E_A : \nexists e_j \in E_B : e_i.K = e_j.F) \vee (\forall e_j \in E_B \nexists e_i \in E_A : e_j.F = e_i.K) :$

$$(e_{i[v_a]} \rightarrow e_{i[v_a+1]})$$
$$(e_j((Z : z), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_b]} \rightarrow e_j((Z : z), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_b+1]})$$
$$(e_j((F : k), b_{j_2}, \dots, b_{j_m})_{[v_b]} \rightarrow e_j((Z : \perp), (F : k), b_{j_2}, \dots, b_{j_m})_{[v_b+1]})$$

$$\mathit{global\ postcond} : \{X \notin S_{A[v_a+1]}, Z \in S_{B[v_b+1]}\}$$

A.4 Copy Ignore Semantics in Heterogeneity Class 4

Semantics of Copy Overwrite in HC4

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

$$global\ precond : \{X \overset{?}{\in} S_{A[v_a]}, Z \overset{?}{\in} S_{B[v_b]}\}$$

$$S_A(X?, K?, A_3?, \ldots, A_n?)_{[v_a]} \rightarrow S_A(X?, K?, A_3?, \ldots, A_n?)_{[v_a+1]}$$
$$S_B(F?, B_2?, \ldots, B_m?)_{[v_b]} \rightarrow S_B(Z, F?, B_2?, \ldots, B_m?)_{[v_b+1]}$$

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

$$
\begin{cases}
case : X \in^* e_{i[v_a]} \begin{cases}
case : Z \notin^* e_{j[v_b]} \begin{cases}
case\ precond : \{X \in^* e_{i[v_a]} \wedge Z \notin^* e_{j[v_b]}\} \\
\ (e_i((X:x),(K:k),a_{i_3},\ldots,a_{i_n})_{[v_a]} \\
\quad \wedge e_j((F:k),b_{j_2},\ldots,b_{j_m})_{[v_b]} \\
\ \rightarrow e_i((X:x),(K:k),a_{i_3},\ldots,a_{i_n})_{[v_a]} \\
\quad \wedge e_j((Z:x),(F:k),b_{j_2},\ldots,b_{j_m})_{[v_b]}) \\
case\ postcond : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\}
\end{cases} \\
case : Z \in^* e_{j[v_b]} \begin{cases}
case\ precond : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \\
\ (e_i((X:x),(K:k),a_{i_3},\ldots,a_{i_n})_{[v_a]} \\
\quad \wedge e_j((Z:z),(F:k),b_{j_2},\ldots,b_{j_m})_{[v_b]} \\
\ \rightarrow e_i((X:x),(K:k),a_{i_3},\ldots,a_{i_n})_{[v_a]} \\
\quad \wedge e_j((\mathbf{Z}:\mathbf{z}),(F:k),b_{j_2},\ldots,b_{j_m})_{[v_b]}) \\
case\ postcond : \{X \in^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\}
\end{cases} \\
\ e_{i[v_a]} \rightarrow e_{i[v_a+1]} \\
\ e_{j[v_b]} \rightarrow e_{j[v_b+1]} \\
\ case\ postcond : \{X \notin^* e_{i[v_a+1]} \wedge Z \in^* e_{j[v_b+1]}\}
\end{cases} \\
case : X \notin^* e_{i[v_a]} \begin{cases}
case : Z \in^* e_{j[v_b]} \begin{cases}
case\ precond : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\} \\
\ (e_i((K:k),a_{i_3},\ldots,a_{i_n})_{[v_a]} \\
\quad \wedge e_j((Z:z),(F:k),b_{j_2},\ldots,b_{j_m})_{[v_b]} \\
\ \rightarrow e_i((K:k),a_{i_3},\ldots,a_{i_n})_{[v_a]} \\
\quad \wedge e_j((Z:z),(F:k),b_{j_2},\ldots,b_{j_m})_{[v_b]}) \\
case\ postcond : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\}
\end{cases} \\
case : Z \notin^* e_{j[v_b]} \begin{cases}
case\ precond : \{X \notin^* e_{i[v_a]} \wedge Z \notin^* e_{j[v_b]}\} \\
\ (e_i((K:k),a_{i_3},\ldots,a_{i_n})_{[v_a]} \\
\quad \wedge e_j((F:k),b_{j_2},\ldots,b_{j_m})_{[v_b]} \\
\ \rightarrow e_i((K:k),a_{i_3},\ldots,a_{i_n})_{[v_a]} \\
\quad x \wedge e_j((Z:\perp),(F:k),b_{j_2},\ldots,b_{j_m})_{[v_b]}) \\
case\ postcond : \{X \notin^* e_{i[v_a]} \wedge Z \in^* e_{j[v_b]}\}
\end{cases} \\
e_{i[v_a]} \rightarrow e_{i[v_a+1]} \\
e_{j[v_b]} \rightarrow e_{j[v_b+1]} \\
case\ postcond : \{X \notin^* e_{i[v_a+1]} \wedge Z \in^* e_{j[v_b+1]}\}
\end{cases}
\end{cases}
$$

$(\forall e_i \in E_A : \nexists e_j \in E_B : e_i.K = e_j.F) \vee (\forall e_j \in E_B \nexists e_i \in E_A : e_j.F = e_i.K) :$

$$(e_{i[v_a]} \rightarrow e_{i[v_a+1]})$$
$$(e_j((Z:z),(F:k),b_{j_2},\ldots,b_{j_m})_{[v_b]} \rightarrow e_j((Z:z),(F:k),b_{j_2},\ldots,b_{j_m})_{[v_b+1]})$$
$$(e_j((F:k),b_{j_2},\ldots,b_{j_m})_{[v_b]} \rightarrow e_j((Z:\perp),(F:k),b_{j_2},\ldots,b_{j_m})_{[v_b+1]})$$

$$global\ postcond : \{X \notin S_{A[v_a+1]}, Z \in S_{B[v_b+1]}\}$$