Keeping NoSQL Databases up to date – Semantics of Evolution Operations and their Impact on Data Quality

Mark Lukas Möller[1], Meike Klettke[1], and Uta Störl[2⋆]

[1] University of Rostock, Rostock, Germany
{mark.moeller2,meike.klettke}@uni-rostock.de
[2] Darmstadt University of Applied Sciences, Darmstadt, Germany
uta.stoerl@h-da.de

Abstract. Evolving a NoSQL database schema regularly involves migrating datasets into new schema versions. NoSQL databases store datasets in different heterogeneity levels (HCs) that can be characterized by their degree of regularity and cardinality of various entity types. In this article, we present the semantics of NoSQL evolution operations and their corresponding data migration operations while distinguishing different NoSQL HCs. One use-case of NoSQL evolution operations is improvement of actuality and completeness of data which is especially relevant in terms of the ever-expanding volume of data.

Keywords: NoSQL Schema Evolution · Schema Evolution Operation · Data Heterogeneity Classes · Data Quality

1 Introduction

In agile software development environments source code is changed frequently which also can include changes of the data in a database. In order to deal with schema changes, schema evolution operations adapt the data to the new structure. While for relational databases schema evolution has been studied in detail in the past [13], these approaches cannot be directly transferred to NoSQL since characteristics like data heterogeneity have to be taken into account.

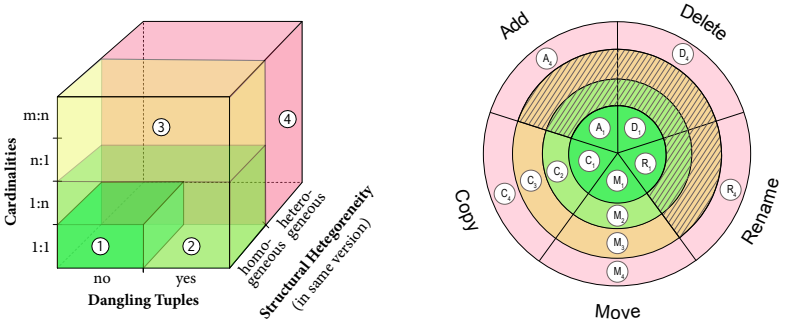
The majority of NoSQL database systems can be used for storing datasets with different characteristics:

1. *No or limited schema control:* In NoSQL, neither schema information nor semantical constraints have to be defined before the actual storing of the datasets. Thus, datasets with different structures can be stored even within the same collection and may lead to heterogeneous data.
2. *Regularity of data:* Oftentimes NoSQL databases are generated by applications or object mappers resulting in data structures that are checked in

terms of data consistency. In these cases, well-structured data is stored in NoSQL databases that at least have an implicit schema.

3. *Versioned datasets:* In other applications, regular datasets are generated with a certain structure, yet this structure changes frequently over time. Consequently, the NoSQL database becomes heterogeneous since it contains datasets in different versions within the same collection.

In all datasets that are used over long periods of time, we have to enable their evolution. In order to transform pre-existing stored data into a new structure, efficient schema evolution operations are required that can cope with problems of heterogeneity and cardinalities and that update and cleanse the data to ensure a high level of *data quality*.



(a) Three dimensions of the four NoSQL HCs

(b) Heterogeneity Classes of the Evolution Operations

Fig. 1. NoSQL Heterogeneity Classes

First, we are introducing different degrees of NoSQL heterogeneity. Figure 1(a) visualizes the three dimensions that have to be considered. The first dimension (x-axis in Figure 1(a)) describes the existence of dangling tuples. Our evolution language includes two multi-type operations, move and copy. Both operations specify matching conditions between entities. In this context, dangling tuples are termed as entities without a matching partner regarding a multi-type operation.

The second dimension describes the cardinalities between kinds that are affected by multi-type operations. Becauses NoSQL databases do not check semantic constraints in advance, it is required to differentiate whether all properties have a matching partner or dangling tuples exist. If matching partners exist, it is important to determine the number of partners - referred to as cardinality.

The last dimension regards the heterogeneity of entities of the same version. Here we distinguish between datasets in which all entities of the same version have homogeneous or heterogeneous structures (z-axis in Figure 1(a)). We derive different *heterogeneity classes (HCs)* per schema evolution operations, starting

from the most structured datasets and 1:1 cardinalities up to unstructured datasets and arbitrary cardinalities.

HC1: In this class, the operation affects datasets in the same or different structural versions (e.g., when *lazy migration approaches* are used), yet all datasets in the same version have exactly the same structure. Multi-type operations presume 1:1 cardinalities only and there are no dangling tuples allowed between two kinds of matching conditions.

HC2: The second class extends HC1 by 1:n cardinalities. Therefore, it is required to deal with dangling tuples.

HC3: The third class encompasses HC2 with arbitrary cardinalities. Additional strategies are required for determining property values of entities affected by multi-entity operations with n:m cardinalities.

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

A schema evolution operation against a NoSQL database must be able to cope with all variants of input datasets. The article makes the following contribution.

- We have already introduced four different heterogeneity classes (HC1-HC4) for NoSQL. Based on these heterogeneity classes, we define the operational semantics and data migration for a NoSQL evolution language in Section 3. We show that for certain HCs the evolution operations can be simplified.
- We discuss the impact on schema evolution operations on the data quality in Section 4, namely data actuality, data completeness, and data consistency.

2 Foundations

Our NoSQL evolution language contains three single-type operations, add, delete and rename, and two multi-type operations, move and copy. The operations are defined for the evolution of the schema and entailed data migration operations can be derived. The schema evolution and data migration operations are used to bring entities into the latest structural version. Firstly, we introduce the semantic foundations.

Data with an equal or similar set of properties is called a an *entity-type* or a *kind*. A kind named A consists of a schema and of a set of entities and is defined as $\mathcal{K}_A = (S_A, E_\mathrm{A})$.

The schema S_A is defined as a set of *property-names*, $S_A = \{A_1, \ldots, A_n\}$. The set of entities E_A of $\mathcal{K}_A$ over the schema S_A is defined as $E_\mathrm{A} := \{e_1, \ldots, e_m\}$ whereby m represents the number of entities and where each entity e_i in E_A consists of up to n *properties* (also referred to as *attributes*) called a_{i_j} with $i \in (1, \ldots, m)$ and $j \in (1, \ldots, n)$. Formally, $e_i = \{a_{i_j} \mid j \in \{1, \ldots n\}\}$.

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

Example. To illustrate the definitions, let us consider an example for the representation of a subset of a research project database which stores information about research stations, the name of the funder of the project, and the budget. The kind is called *project* and is defined as $\mathcal{K}_{project} = \{S_{project}, E_{\text{project}}\}$, whereby $S_{project} = \{\text{"p\_id"}, \text{"station\_name"}, \text{"funder"}, \text{"budget"}\}$. E_{project} is the set of entities that contains two entities (e_1 and e_2) of the kind *project*. A valid set of data E_{project} is:

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

For the evolution operations, it is required to check whether an entity contains a property with a certain name, regardless of its value. Because properties are stored as a tuple and not as a set, the operator $\in^*$ is defined which evaluates if there is a property available for a given entity or not. For this purpose, we define a projection operation that projects onto the property name: $\pi_{\mathsf{A}} := S_{A_i} \times \mathcal{D}_{A_i} \rightarrow S_{A_i}$ with $(A_{i_j}, v_{i_j}) \mapsto A_{i_j}$. Based on this projection, the $\in^*$ operator is defined. $X \in^* e_i :\Leftrightarrow \exists a_{i_j} \in e_i : X \in \pi_{\mathsf{A}}(a_{i_j})$, and $X \in^* E_{\text{A}} :\Leftrightarrow \forall e_i \in E_{\text{A}} : X \in^* e_i$.

Reconsider the previous example. Here, "station_name" $\in^*$ e_1 is True while "location" $\in^*$ e_2 is False.

The Dot-Notation is introduced for reading the value of a given property name and is particularly needed in order to express matching conditions for multi-entity operations. The following notation is introduced:
$\forall X \in^* e_i : e_i.X := \pi_{\mathsf{v}}(a_{i_j})$ with $\pi_{\mathsf{v}} := S_{A_i} \times \mathcal{D}_{A_i} \rightarrow \mathcal{D}_{A_i}$ with $(A_{i_j}, v_{i_j}) \mapsto v_{i_j}$.

In the example, e_1.station_name evaluates to *"Ocean"* and e_2.station_name evaluates to *"Baltic Sea"*, while e_1.location throws an exception.

Due to migration and encompassed different schema versions, the same kind is inspected at different points in time. For this, a notation of a *version* is introduced in the form of in square brackets. For instance, $S_{A[10]} = \{A_1, \ldots, A_n\}_{[10]}$ describes the schema of kind A at schema version 10. In the abstract notation for the evolution and migration operation, $[v_A]$ and $[v_B]$ is used for the version information of the kinds $\mathcal{K}_A$ and $\mathcal{K}_B$.

Generally, S_A can be derived by iterating over all entities of E_{A} and collect all attribute names. Nevertheless, S_A is stored as well to support a query rewriting approach presented in [9].

3 Semantics of the Evolution Operations

In this section, we define the semantics of the evolution operations on regular structures and structured datasets, and we will extend them to irregular structures and heterogeneous datasets. The evolution operations were introduced for the first time as EBNF and as a NoSQL programming language in [14] and since that time continously extended. The chosen evolution operations *add*, *rename*, *delete*, *move* and *copy* represent a set of frequent schema evolution operations

in open source applications (c.f. [3]). The effort for data migration increases accordingly to the HC. In order to define the concrete heterogeneity classes, *pre- and postconditions* are used to determine the regularity of the data. The pre- and postconditions are inspired by the *Hoare triple*. These conditions 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 postconditions are guaranteed.

Hereafter we define the single-type operation add and the multi-type operation move. The semantics for the complete evolution language is given in [10].

3.1 Heterogeneity Class 1

Operations in HC1 assumes that in a dataset all entities of a kind have the same schema within the same schema version. Hence, there is no possibility to have datasets with optional properties. For multi-type operations, this class can only cope with matches of 1:1 cardinalities.

Each of the operations evolves the schema and migrates the entities into the new version. On the instance level, the operation modifies the data structure and updates affected instances. The effects of the evolution operations have to be defined on the schema level and the instance level. Evolution operations are defined as *rules* whereby the left side of a rule describes the schema/instances before the operation while the right side describes the schema/instances after the operation. All rules consist of a precondition which needs to hold before the operation. If the condition is not fulfilled, the operation is not executed. The postconditions are fulfilled after the operation and will become important for the chaining of operations and for the examination of Data the Quality.

The Add Operation This operation adds a property to all entities of a kind. The operation specifies the *kind*, the *new property name* and additionally, the *default property value*. In HC1, the add operation is defined as:

$\triangleright$ **add A.X = d**

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

First, the operation verifies that the precondition is fulfilled which states that the name of the property is not allowed to be available in the schema of $\mathcal{K}_A$ in the version v_A. The second line describes the schema evolution of $\mathcal{K}_A$. In version v_A, schema S_A consists of n properties $A_1, \ldots, A_n$. After the operation in version $v_A + 1$, the schema consists of $n + 1$ properties including the added property named X. The third line describes the instance level modification of each entity of $\mathcal{K}_A$. Each entity consists of the properties a_1 to a_n and additionally the new property $(X : d)$ whereby X is the name of the added property and d is the

default value. After the modification of the schema and the entity migration, the postcondition holds which states that property name X is part of S_A in version $v_A + 1$. As a variant of the given semantics, it is possible to add a property without default value: add A.X. In this case, the property $(X : \perp)$ is added whereby $\perp$ represents a Null value.

The Move Operation The multi-type operation move transfers a property from the entities of one kind (termed as *source kind*) to entities of a different kind (termed as *target kind*). To execute a multi-type operation, a *matching condition* between both kind is mandatory. In HC1, the matching cardinality is assumed as 1:1, which entails bijectivity so that every entity of the source kind has exactly one match with an entity of the target kind, and vice versa. This also 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 does not have a matching partner. Consequently, multi-entity operations in HC1 are restricted to kinds with the same amount of entities.

In HC1, the semantics of the move operation is defined as follows:

▷ **move A.X To B.Z where A.K = B.F**

$$
\begin{gathered}
precond : \{X \in S_{A[v_A]}, Z \notin S_{B[v_B]}\} \\
S_A(X, K, A_3, \ldots, A_n)_{[v_A]} \rightarrow S_A(K, A_3, \ldots, A_n)_{[v_A+1]} \\
S_B(F, B_2, \ldots, B_m)_{[v_B]} \rightarrow S_B(Z, F, B_2, \ldots, B_m)_{[v_B+1]} \\
\\
\forall e_i \in E_A, e_j \in E_B, e_i.K = e_j.F : \\
(e_i((X : x), (K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A]} \wedge e_j((F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B]} \\
\rightarrow e_i((K : k), a_{i_3}, \ldots, a_{i_n})_{[v_A+1]} \wedge e_j((Z : x), (F : k), b_{j_2}, \ldots, b_{j_m})_{[v_B+1]}) \\
postcond : \{X \notin S_{A[v_A+1]}, Z \in S_{B[v_B+1]}\}
\end{gathered}
$$

Beside the matching condition, the source and target kinds as well as the property names are specified. Here, these are $\mathcal{K}_A$ with the property X and $\mathcal{K}_B$ with property Z. The move operations implicitly realizes a rename operation if the property names of the source and target kinds are different. In the where clause, the matching condition is explicitly specified.

Before the operation, S_A of $\mathcal{K}_A$ contains the property name X, while S_B of the $\mathcal{K}_B$ does not. 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. During the operation, all entities e_i and e_j are modified. The property $(X : x)$ is not present anymore in any entity of $\mathcal{K}_A$ while $(Z : x)$ is part of each entity of $\mathcal{K}_B$. The same symbol x on the left and on the right hand side of the rule indicate the same property value – the value is transferred without a modification from the source kind to the target by the operation. The matching condition between both kinds is represented by the same property value k ($(K : k)$ for e_i and $(F : k)$ for e_j) as well.

3.2 Heterogeneity Classes 2 and 3

In heterogeneity classes 2 and 3, we assume structurally homogeneous data within the same version, however, cardinalities are extended to 1:n in HC2 and to m:n in HC3. Thus, it is necessary to deal with dangling tuples and multi matches. Since HC2 and HC3 are inherited in HC4 in terms of their characteristics, we will explain the properties and challenges of these HCs in the next section. Furthermore, both HCs are discussed in detail in [10].

3.3 Heterogeneity Class 4

Evolution operations in HC4 cover the most complicated NoSQL databases considering all structural variants. In this HC, schema heterogeneity and multi-entity operation of arbitrary cardinalities are included.

An example for challenges in HC4 is given in Figure 2. Here, an add operation is executed and affects two entities of $\mathcal{K}_{project}$. For the property value of funder of the entity with the id : 2 it is required to decide whether the value of funder is either overwritten with the default value or preserved. The semantics is extended by introducing the additional keywords overwrite and ignore for implementing *conflict resolution strategies*.

For heterogeneous data, it is required to denote optionality in the semantics, especially in the preconditions, since it is not known whether a certain property occurs in all entities of a kind. Optional properties are labeled with a question mark. For example, $X \overset{?}{\in} S_A$ states that X is an optional property in the schema of kind A and *can* or *cannot* appear in an entity. This requires to deal with both cases in the semantics. On the schema level, the notation $S_A(X?)$ is used analogously.

The Add operation The definition of the add operation is given below. Here, the overwrite approach is used which adds the property and the specified default value to entities without that property. For entities that already contain the property before of the operation, their affected property values are overwritten by the operation's default value.

In contrast to HC1, it is distinguished between the *global conditions* which hold for the schema and all entities affected by the evolution operation, and *case conditions* which only hold for a subset of the entities affected by the operation.

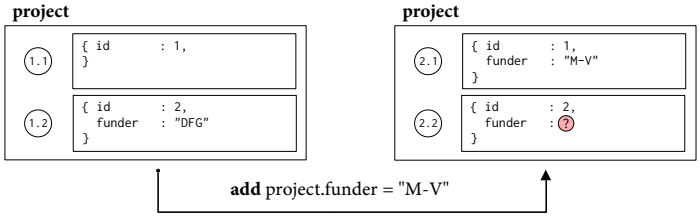


Fig. 2. Execution of the add operation on heterogeneous data in HC4

The definition of the evolution operation is divided into two cases: The first case defines the operation for all datasets in which X is not available. A property named X is added with the default value d. The second case defines the operation for the datasets that already contain X. The existing value of the property X is overwritten with the default value d. Analogously to HC1, this operation also can be defined without a default value.

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 an improved readability, the denotation for optionality is only given for properties that are affected by the evolution operation (here: X).

▷ **add overwrite A.X = d**

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

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

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

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

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

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

The Move Operation The definition of the move operation is more difficult because it has to be defined for two kinds (source and target). It is necessary to cope with both heterogeneity and arbitrary cardinalities in the semantics, whereby even 1:1 matches entail complex problems. Let us extend the introduced example by a second kind called metadata which at least consists of a property called m_id. Some entities of $\mathcal{K}_{metadata}$ contain the property station_name as well. The database administrator wants to evolve the database schema by moving station_name from $\mathcal{K}_{metadata}$ to $\mathcal{K}_{project}$. Since project consists of the property station_name as well, determining the property value is not trivial. Figure 3 depicts the cases that can occur with a matching cardinality of 1:1 for the move operation in HC4. The first match describes the case where station_name is available in the corresponding entity of $\mathcal{K}_{metadata}$, but not in $\mathcal{K}_{project}$, and can be moved easily. The second case describes where station_name is not available in $\mathcal{K}_{metadata}$ yet in $\mathcal{K}_{project}$. For both introduced conflict resolution strategies the pre-existing value for station_name is preserved. The third case describes the

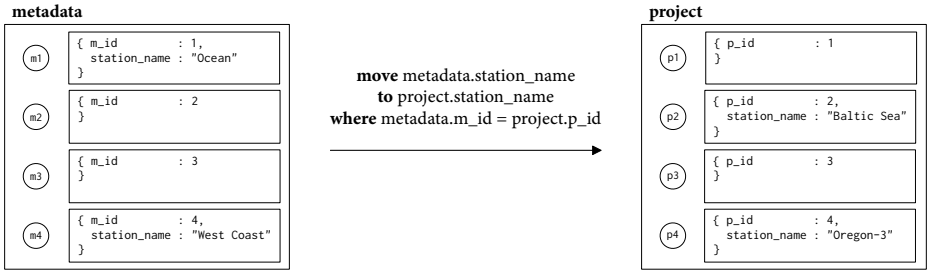


Fig. 3. Emerging cases of the move operation with 1:1 matching cardinalities and heterogeneous data

case that station_name is neither available in $\mathcal{K}_{metadata}$ nor in $\mathcal{K}_{project}$, here, a property with an empty value will be introduced. The last case delineates that station_name is part of both entities. For the last case, the value of station_name depends on the conflict resolution strategy. All cases are required to be handled by the semantics of the move operation in HC4.

On the schema level, it is established that the operation datetimestamp is removed from the source kind, while the property datetimestamp is contained in the target kind.

For all entities of the source kind without a matching partner, the property is removed and for all entities of the target kind without a matching partner the entity is assigned with a property of a Null value.

The formal semantics of the move overwrite operation for HC4 is given in the Appendix of this paper. The semantics of all other single-type and multi-type operations of the NoSQL evolution language and their different conflict resolution approaches are described in [10].

4 Increased Data Quality through Schema Evolution

Quality of data entails several characteristics such as data completeness, data actuality, and data consistency (c.f. [17]). Schema evolution can be applied for refreshing the datasets and in parallel increasing the data quality and in some cases decreasing the HC. Both will be sketched in the following.

Data Actuality The main focus of the evolution lies on updating datasets and migrating them into the latest version. In our previous work, we have introduced methods for an eager data migration (immediately after introducing a new version), lazy migration (on demand, if datasets are accessed) or by using hybrid strategies [6], [8]. In all cases, datasets are transformed into the actual schema version. This enables that legacy datasets can be updated, transformed into the current structure and guarantees data actuality.

Data Completeness and Data Consistency The NoSQL evolution operations presented in this paper never increase the heterogeneity of the databases. After execution data migration operations the databases always remain in the same HC as the source datasets. Even further, the operations can be used to increase regularity and completeness of the NoSQL databases, and in some cases to reduce the heterogeneity class and so improve the data quality. In the following we will present this in more detail for the heterogeneity classes.

Reconsidering the given semantics, it is evident that data in heterogeneity class 1 or 2 always remain in this class due to the restrictions of the pre- and postconditions, the heterogeneity and the matching conditions. For both HCs there are no optional properties and operations always affect all entities of a kind. Concluding, it is impossible to transform data without optional properties into schema-heterogeneous data. For multi-type operations with the same matching condition, the cardinality remains the same, even for chained operations.

In HC3, the same argumentation holds for optional properties. Regarding cardinalities, data in HC3 also remains in this HC for two multi-entity operations with the same matching condition. Nevertheless, the conflict resolution approaches provide an advantage. Consider two kinds with a n:1 relation (encompassed in HC3), e.g. two entities of $\mathcal{K}_{metadata}$ (caused by duplicates) belong to a single entity of $\mathcal{K}_{project}$. Selecting data from both kinds using a join operation normally returns two result rows. By evolving the database and moving all properties from the entities of $\mathcal{K}_{metadata}$ to $\mathcal{K}_{project}$ using overwrite or ignore results in a concrete property value for all properties moved to the entity of $\mathcal{K}_{project}$. Nevertheless, depending on the application, it might be a downside that for both strategies because a subset of property values is lost after the move or copy operation. A better solution can be the generation of an array of values to collect the values of all matching partners while decreasing heterogeneity.

For data in HC4, it can be possible to transform this data into lower heterogeneity classes. Only HC4 copes with optional properties. Consider $\mathcal{K}_{project}$ from the example on page 4 where the only optional property budget is not existent in each entity. After an add operation (with an arbitrary conflict resolution approach) on $\mathcal{K}_{project}$, all entities have a homogeneous schema. Hence, evolution operations can be used this way in order to increase the schema-homogeneity of NoSQL datasets.

5 Related Work

The main aspect of this paper deals with the semantics for NoSQL schema evolution operations and data migration for different heterogeneity classes. Additionally, we presented the impact of evolution operations on data quality. In this section, we present approaches and concepts related to ours.

In [7], the authors present an approach for schema mapping. Similar to our semantics, a mapping consists of a source and a target schema, and a set of formulas of some logic over both schemas. The used formalism to describe database dependencies are *Tuple-generating dependencies (TGDs)* (see also [12], [1]).

In [5], several schema versions are being maintained within a single relational database. In that publication a language for bidirectional schema evolution and forwards and backwards delta code generation is defined to support multiple versions of an application while maintaining only one database with co-existing schema versions.

Schildgen presents in [15] the language *NotaQL* to transform NoSQL data and uses this language to overcome different kinds of heterogeneity.

Data quality is a long studied field in relational database theory and covers a broad field of characteristics, such as data homogeneity, data correctness, and data completeness. An overview of data integration steps and tools in practice is given in [4]. Naumann describes in [11] research directions and challenges of data quality and classifies different data profiling subtasks. The aspects of the duplicate elimination/coping redundancy can be part of a data cleansing process (c.f. [2]). Our presented semantics eliminates multiple values for properties that are affected by schema evolution operations by using the overwrite or ignore approach. This avoids a duplication of records with only one different property value. In contrast to other data cleansing approaches, we are focusing on transformation of NoSQL dataset into the current version and in parallel increasing the regularity of the databases. The transformation process is described by the evolution operations.

In our research project *Darwin* [16], we realized the evolution for MongoDB, Cassandra and CouchDB for the single- and multi-type operations in HC1 and HC2.

6 Summary and Future Work

In agile development environments, data structures are often changed which necessitates the definition of schema evolution operations. For efficient schema evolution, schema evolution operations were defined that take the characteristics of NoSQL data, such as data heterogeneity, into account.

In this article, we introduced NoSQL heterogeneity classes which relate to the complexity of operations. We presented as a subset of our schema evolution language the semantics for the single-type operation add and the multi-type move for different HCs. We have shown the complexity of the operations in different heterogeneity classes and why evolving the schema allows to improve data quality under certain conditions. Storing completely unstructured and heterogeneous data is very uncommon, even in the NoSQL world – applications often require a certain schema for reading and processing data. Hence, datasets are stored homogeneously. Data in higher heterogeneity classes require sophisticated evolution and migration operations. In this article, we have shown that the presented semantics is able to migrate into a lower heterogeneity class when certain requirements are met.

In the future, we plan to extend the semantics by introducing further schema evolution operations. The current operations have been chosen due to an analysis of schema changes in open-source applications like Wikipedia (c.f. [3]). Further

operations such as split and merge are possible and useful as well. We plan to estimate and benchmark the impact of schema heterogeneity and low data quality for various scenarios, such as schema evolution or query rewriting in environments where data is lazily migrated as examined in [9].

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

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995), http://webdam.inria.fr/Alice/
2. Bench-Capon, T.J.M., Soda, G., Tjoa, A.M. (eds.): DEXA '99, Florence, Italy, Proc. LNCS, vol. 1677. Springer (1999)
3. Curino, C., Moon, H.J., Tanca, L., Zaniolo, C.: Schema Evolution in Wikipedia - Toward a Web Information System Benchmark. In: Proc. ICEIS'08 (2008)
4. Haas, L.M.: Beauty and the beast: The theory and practice of information integration. In: ICDT. Springer LNCS, vol. 4353, pp. 28–43. Springer (2007)
5. Herrmann, K., Voigt, H., Rausch, J., et al.: Living in Parallel Realities — Co-Existing Schema Versions with a Bidirectional Database Evolution Language. In: Proc. SIGMOD (2017)
6. Klettke, M., Störl, U., Shenavai, M., et al.: NoSQL Schema Evolution and Big Data Migration at Scale. In: IEEE Big Data 2016, Washington DC. IEEE (2016)
7. Kolaitis, P.G.: Schema mappings, data exchange, and metadata management. In: PODS. pp. 61–75. ACM (2005)
8. Möller, M.L.: Datenevolutions- und Migrationsstrategien in NoSQL-Datenbanken. In: Grundlagen von Datenbanken. CEUR Workshop Proc., vol. 2126 (2018)
9. Möller, M.L., Klettke, M., Hillebrand, A., et al.: Query Rewriting for Continuously Evolving NoSQL Databases (2019), accepted for ER2019, Salvador, Brazil
10. Möller, M.L., Klettke, M., Störl, U.: Formal Semantics of NoSQL Evolution Operations under different Heterogeneity Levels (2018), Tech. Report, Rostock University
11. Naumann, F.: Data profiling revisited. SIGMOD Record **42**(4), 40–49 (2013)
12. Pichler, R., Skritek, S.: The Complexity of Evaluating Tuple Generating Dependencies. In: ICDT 2011, Uppsala, 2011. ACM (2011)
13. Roddick, J.F.: Schema Evolution in Database Systems - An Annotated Bibliography. SIGMOD record **21**(4), 35–40 (1992)
14. Scherzinger, S., Klettke, M., Störl, U.: Managing schema evolution in nosql data stores. Proc. DBPL **CoRR, abs/1308.0514, abs/1308.0514** (2013)
15. Schildgen, J., Deßloch, S.: Heterogenität überwinden mit der Datentransformationssprache NotaQL. Datenbank-Spektrum **16**(1), 5–15 (Mar 2016)
16. Störl, U., Müller, D., Tekleab, A., et al.: Curating variational data in application development. In: ICDE. pp. 1605–1608. IEEE Computer Society (2018)
17. Strong, D.M., Lee, Y.W., Wang, R.Y.: Data Quality in Context. Commun. ACM **40**(5), 103–110 (1997)

A Appendix

Move Overwrite Semantics in Heterogeneity Class 4

▷ **move overwrite A.X to B.Z where A.K = B.F**

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

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

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

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

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

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

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