Heterogeneity in NoSQL Databases — Challenges of Handling schema-less Data

Mark Lukas Möller[1], Dominique Hausler[1], Sebastian Strasser[1], Tanja Auge[1] and Meike Klettke[1]

[1]*Faculty of Computer Science and Data Science, University of Regensburg, Germany*

Abstract

The schema flexibility of database management systems is often seen as an advantage, because it makes it easy to store all kinds of different data. Schema-less database systems (such as JSON or graph databases) allow structurally different data to be stored in the same database. It also allows storing different variants of data or data evolving over time. However, their use is much more complicated compared to relational data. In this paper, we show the *impact of heterogeneity* on two data processing steps: query execution and evolution operations including their composition and data transformation. From this, we will derive three proposals: (i) storing data with a partial schema management, (ii) vertical partitioning, and (iii) the usage of multi-model databases. In all cases, regular and irregular parts are distinguished in order to mitigate these effects of heterogeneity.

Keywords

NoSQL databases, graph databases, heterogeneity, schema-less systems, query execution, schema evolution, schema management, vertical partitioning, multi-model databases, position paper

1. Introduction

For a long time, the relational data model [1] and its regular structure with predefined fixed schema and regular tuples was the standard model in databases. Subsequent data models (XML, JSON, and graph data) have been developed with the aim to store *heterogeneous data sets* in the same database. For this, the database management system allows either *flexible schemas*, schema definitions which are only *optional*, or even *completely schema-less data*. In the latter case, no schema constraints are checked by the database management system.

The benefits of these systems (XML, JSON and graph databases) were highly valued because they can store all kinds of data. The disadvantages are equally obvious: when using schema-less or schema-flexible databases, all structural variants of the data must be considered in all database components and also their downstream applications that access the database. This leads to additional challenges in a lot of database tasks. Significant effort was put into solving challenges arising from the flexible schema imposed by NoSQL databases, e.g. schema design [2] and

LWDA'23: Lernen, Wissen, Daten, Analysen. October 09–11, 2023, Marburg, Germany

dominique.hausler@ur.de (D. Hausler); sebastian.strasser@ur.de (S. Strasser); tanja.auge@ur.de (T. Auge); meike.klettke@ur.de (M. Klettke)

0000-0001-5726-7134 (M. L. Möller); 0009-0004-2381-133X (D. Hausler); 0009-0001-8848-1368 (S. Strasser); 0009-0006-2150-9713 (T. Auge); 0000-0003-0551-8389 (M. Klettke)

CEUR Workshop Proceedings (CEUR-WS.org)

modeling [3], data integration [4], and data querying [5]. In our own work regarding XML databases and JSON data collections [6, 7, 8, 9] and our ongoing work in graph databases, we have seen that many procedures already developed for the relational world are *far more complicated to implement for schema-flexible database management systems*.

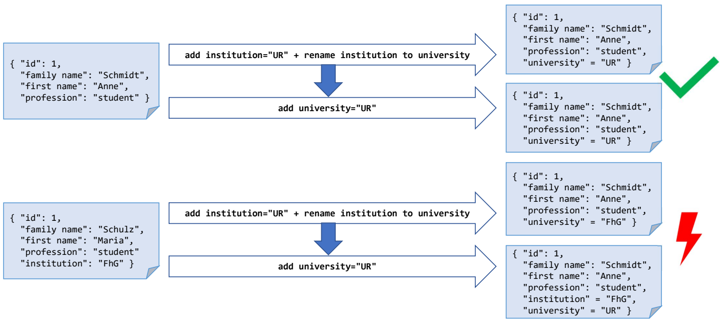
When thinking about optimizing schema modifications, we had an initial presumption that *Schema Modification Operators (SMOs)* similar to those suggested in [10] can be composed. For example, if a new attribute A is inserted into a table t and later renamed to B, it has the same effect as if it has been inserted at once with the attribute name B. In the relational world, for a table t which does not contain an attribute A and B, we can guarantee that the following two operations on the left-hand side generate the same table as the operation on the right-hand side:

```
(alter table t add column A) + (alter table t rename column A to B)
 →  alter table t add column B
```

Based on this experience from relational databases, we developed a common *SMO composition rule* for JSON databases whereby E is an entity type and A and B are properties in E:

add $E.A$ + rename $E.A$ to B → add $E.B$

Applying these SMOs (`add` & `rename`) to a JSON document, they are translated into data migration operations. Because of *schema-lessness*, i.e., the lack of schema, we have to consider the different variants of datasets. Figure 1 represents this example for an entity type `Students` and the properties `institution` and `university`. Here, we see that the step-wise execution of the two SMOs `add` and `rename` produces a different output than a composed execution. This means that the composition rule does not hold for heterogeneous datasets.


add institution="UR" + rename institution to university
add university="UR"
add institution="UR" + rename institution to university
add university="UR"

```
{ "id": 1,
  "family name": "Schmidt",
  "first name": "Anne",
  "profession": "student" }
```

```
{ "id": 1,
  "family name": "Schmidt",
  "first name": "Anne",
  "profession": "student",
  "university" = "UR" }
```

```
{ "id": 1,
  "family name": "Schmidt",
  "first name": "Anne",
  "profession": "student",
  "university" = "UR" }
```

```
{ "id": 1,
  "family name": "Schulz",
  "first name": "Maria",
  "profession": "student"
  "institution": "FhG" }
```

```
{ "id": 1,
  "family name": "Schmidt",
  "first name": "Anne",
  "profession": "student",
  "university" = "FhG" }
```

```
{ "id": 1,
  "family name": "Schmidt",
  "first name": "Anne",
  "profession": "student",
  "institution" = "FhG",
  "university" = "UR" }
```

Figure 1: Composition of two SMOs on a heterogeneous dataset

The example shows that in NoSQL databases we always have to consider the possibility of heterogeneity. In this paper, we present the *impact of heterogeneity* through query execution and evolution operations including their composition based on our experiences.

We will use *four different heterogeneity classes (HCs)*, ranging from very regular (**HC1**) to very irregular (**HC4**) (see Section 2). We will show how database subtasks have to be adapted to these heterogeneity classes in Section 3. We start by querying heterogeneous data and show why it can deliver unwanted results. Next, we define the semantics of database evolution operations for the different HCs. Then, we describe why the composition of evolution operations becomes more complicated the higher the heterogeneity level. Finally, we briefly explain data transformation and query rewriting for the different heterogeneity classes. Based on these observations, we propose different *data storage solutions* for hybrid data consisting of homogeneous and heterogeneous parts (Section 4). Finally, we close with some ideas for future work.

2. Heterogeneity Classes

Due to the inherent schema of NoSQL data, we can distinguish different levels of heterogeneity. Data can be present in a completely homogeneous structure where each entity consists of the same properties. However, it is also possible for properties to only occur in some entities. In this case, the properties are considered as optional and the entities of the entity type are called heterogeneous. We introduced four *heterogeneity classes (HCs)* which influence the complexity of SMOs [7]:

HC1: All entities of a database are homogeneous which means they have the same implicit structure. **Note:** In the case of *multi-type operations*, i.e., using data from two entity types, only 1:1 or 1:n relationships occur without dangling tuples.

HC2: Multi-type operations with 1:1 and 1:n relationships can create *dangling tuples*, i.e., join operations with datasets without a join partner.

HC3: Multi-type operations with n:1 and n:m relationships can create dangling tuples.

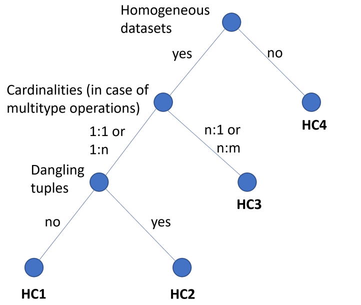


Figure 2: Classification of the different Heterogeneity Classes

HC4: Databases can contain entities with different structures, i.e., two entities from the same entity type can have different properties. **Note:** This heterogeneity class is the most flexible one. Every possible variant of the data must be considered in all operations.

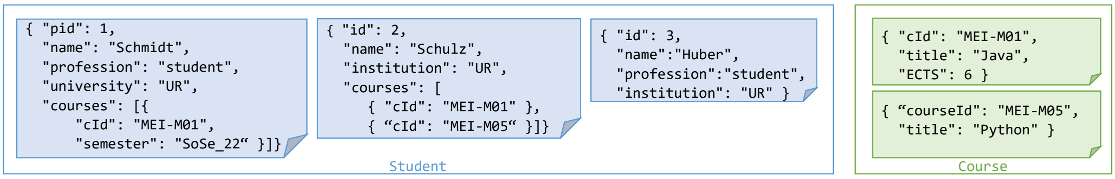
If we do not have further information, we always have to assume **HC4** because there are *neither schema constraints nor semantic constraints* which can be guaranteed. In contrast, if **HC1** can be assumed, querying and evolving databases is much easier. In Section 3, we will demonstrate these two cases.

3. Handling of Heterogeneous Data

When databases are stored without a fixed schema, they can vary structurally from entity to entity. Also, checking for semantic constraints is not part of the data insertion process. Thus, in database management systems containing foreign key references, so-called *dangling tuples* can occur. In this section, we will briefly outline a few cases that illustrate the impact of structural heterogeneity and dangling tuples when using or querying data.

Using and Querying Homogeneous and Heterogeneous Data To demonstrate the effects of heterogeneity, let's look at a concrete example implemented in MongoDB[1]. As seen in Figure 3, the university database contains entities with different structures. For instance, the attributes `cId` and `courseId` of the `Student` document with `"id":2` both describe identical attributes,

[1] MongoDB: https://www.mongodb.com/



```
{ "pid": 1,
  "name": "Schmidt",
  "profession": "student",
  "university": "UR",
  "courses": [{
      "cId": "MEI-M01",
      "semester": "SoSe_22“ }]}
```

```
{ "id": 2,
  "name": "Schulz",
  "institution": "UR",
  "courses": [
     { "cId": "MEI-M01" },
     { “cId": "MEI-M05“ }]}
```

```
{ "id": 3,
  "name":"Huber",
  "profession":"student",
  "institution": "UR" }
```

```
{ "cId": "MEI-M01",
  "title": "Java",
  "ECTS": 6 }
```

```
{ “courseId": "MEI-M05",
  "title": "Python" }
```

Figure 3: Example for heterogeneous datasets (`Student` and `Course` documents)

but have different property[2] names. Another case of heterogeneity is the optionality of the attribute `profession`. There is also a lack of `courses` in `Student` with `"id":3`. All in all, the database can be classified as **HC4** following the taxonomy defined in Section 2.

```
db.getCollection("Student").find(
    { institution: "UR" }
)
```

Query 1: Querying the `Student` documents with `institution="UR"` (in MongoDB)

```
db.Student.aggregate([{
    $lookup: {
        from: "Course",
        localField: "courses.cId",
        foreignField: "cId",
        as: "courses"}
    }])
```

Query 2: Joining the collections `Student` and `Course` by property `cId` (in MongoDB)

Example 1: Query 1 filters all `Students` by the condition `institution="UR"`. The results, shown in Query result 2, contains the `Student` documents with `"id":2` and `"id":3`. Due to variation in the attribute names (`institution` or `university`), the `Student` document with `"pid":1` is not in the result. Users unaware of the heterogeneity in their data could expect an output of all three entities.

```
{ id: 2,
  name: "Schulz",
  institution: "UR",
  courses: [
    { cId: "MEI-M01" },
    { cId: "MEI-M05" }]}
```

```
{ id: 3,
  name: "Huber",
  profession: "student",
  institution: "UR" }
```

Query result 1: Results of Query 1

Example 2: Query 2 shows a join between the `Student` and `Course` documents. It yields one course for `Student` documents each with `"pid":1` and `"id":2`. This is a result of performing the lookup function which corresponds to a left outer join with the condition `Student.courses.cId` = `Course.cId`. Additionally, the `Student` document with `"id":2` contains a second course with `"cId":"MEI-M05"`. This attribute has no joining partner in the `Course` document. Because of the incorrect attribute name, `courseId` represents a dangling tuple.

[2] *Attributes* and *properties* are used interchangeably as they describe the same element in different database management systems.

The result of the join between `Student` and `Course` documents are summarized in the Query result 2. In this result, the `Student` document with `"id":3` has a `courses` attribute containing `"courseId":"MEI-M05"`. This unexpected result is caused by the structural heterogeneity of the `Course` document. The `lookup` function sets the value of the `localField` (in our example the `Student` document with `"id":3`) and the `foreignField` (the `Course` document with `cId`) to `NULL`. In the query execution, both values are `NULL` [11] and are therefore joined.

To avoid this unwanted effect and address the problem of missing values being set to `NULL`, one could introduce a match stage before executing the lookup function. This filters for `Student` documents containing a `courses` attribute and is implemented in MongoDB as follows: `$match:{"courses":{"$exists":true,"$ne":[]}}`.

```
{ pid: 1,
  name: "Schmidt",
  profession: "student",
  institution: "UR",
  courses: [{
    cId: "MEI-M01",
    title: "Java",
    ECTS: 6 }]}
```

```
{ id: 2,
  name: "Schulz",
  institution: "UR",
  courses: [{
    cId: "MEI-M01",
    title: "Java",
    ECTS: 6 }]}
```

```
{ id: 3,
  name: "Huber",
  profession: "student",
  institution: "UR",
  courses: [{
    title: "Python",
    courseId: "MEI-M05" }]}
```

Query result 2: Results of Query 2

Both example queries show cases where schema and integrity constraints cannot be guaranteed and **HC4** must be assumed. This leads to individual documents not being found by the queries; concretely: the `Student` document with `"pid":1` in Query 1 or the `Course` document with `"courseId":"MEI-M05"` in Query 2. This is the case for both divergent property names and dangling tuples. The heterogeneity of the data must be considered not only in queries, but also in evolution. In the following, we will emphasize the difficulties heterogeneity brings in this context.

Semantics of Evolution Operations In databases which are used over a long time, *evolution operations* like `add`, `delete`, `rename`, `move`, `copy`, `split` and `merge` must be performed. These operations are suggested in several approaches for relational databases [12], XML [13], JSON databases [14, 15, 16], and for multi-model systems in [17].

The HCs of the data affect how complex the evolution operations become. This heterogeneity classes hold for all schema-less databases. In the following, we define them for JSON database. If we can start from data in **HC1** (regular, relational-like data, no dangling tuple), an operation `add` is defined as follows:

$$\textbf{HC1}\text{: add } E.A = x \quad \forall e_i \in E : \{..\} \longrightarrow \{..,''A'' : ''x''\}.$$

It specifies that a new property A with value x is added to all entities e_i of entity type E. For a detailed definition, we refer to [7].

If we execute the same operation on a dataset in **HC4**, we need to distinguish two cases:

$$\textbf{HC4}\text{: add } E.A = x : \begin{cases} A \in e_i : & \{..,''A'' : ''a'',..\} \longrightarrow \{..,''A'' : ''a'',..\}, \\ A \notin e_i : & \{..\} \longrightarrow \{..,''A'' : ''x''\}. \end{cases}$$

The attribute A can either already exist in the entity or not. Therefore, both cases must be considered. There are two different semantics of the operation: *overwrite* and *ignore*. In case a property is present and we are adding a property with the same key, then *overwrite* — as the name already says — overwrites the value in the database. When following the *ignore* semantics, the operation would be rejected keeping the previous value. In the definition of `add` $E.A$ in **HC4** from above, the *ignore* semantics is given.

In a similar way, we define a `rename` operation. Whereas in **HC1** only one case suffices, for **HC4** we have to distinguish four cases:

$$\textbf{HC1}\text{: }\texttt{rename}\ E.A \text{ to } B \quad \forall e_i \in E : \{..,"A":"x",..\} \longrightarrow \{..,"B":"x",..\}.$$

$$\textbf{HC4}\text{: }\texttt{rename}\ E.A \text{ to } B : \begin{cases} A \in e_i \wedge B \notin e_i : & \{..,"A":"x",..\} \longrightarrow \{..,"B":"x",..\}, \\ A \in e_i \wedge B \in e_i : & \{..,"A":"a","B":"b"..\} \\ & \longrightarrow \{..,"A":"a","B":"b",..\}, \\ A \notin e_i \wedge B \in e_i : & \{..,"B":"x",..\} \longrightarrow \{..,"B":"x",..\}, \\ A \notin e_i \wedge B \notin e_i : & \{..\} \longrightarrow \{..\}. \end{cases}$$

We want to mention that the definition for **HC1** corresponds to the first case of the definition for **HC4**. Again, we differ between the *ignore* and *overwrite* semantics. In the definition of `rename` $E.A$ to $E.B$ in **HC4** from above, the *ignore* semantics is used which avoids overwriting already available properties. For using *overwrite*, which is overwriting the already available properties, the definition of `rename` $E.A$ to B has to be adapted. For details we refer to [7].

Most evolution approaches also offer so-called *multi-type operations* like `move`, `copy`, `split`, and `merge` for a complex restructuring. Here, we also see large differences in the definitions of semantics for different heterogeneity classes. We pick the `move` operation for **HC1** and **HC4** as an example. We define `move` $E_1.A$ to $E_2.B$ where $E_1.C_1 = E_2.C_2$ for **HC1**:

$$\begin{aligned} &\forall e_i \in E_1, \forall e_j \in E_2 : e_i : \{..,"A":"x","C_1":"c",..\}, e_j : \{..,"C_2":"c",..\} \\ &\longrightarrow e_i : \{..,"C_1":"c",..\}, e_j : \{..,"A":"x","C_2":"c",..\}. \end{aligned}$$

The definition of the `move` operation also looks a lot more complex when assuming the datasets are in **HC4**. Following are the different cases to be considered:

- $A \in e_i$ vs. $A \notin e_i$
- $C_1 \in e_i$ vs. $C_1 \notin e_i$
- $B \in e_j$ vs. $B \notin e_j$
- $C_2 \in e_j$ vs. $C_2 \notin e_j$

Combining these conditions, $2^4 = 16$ different cases are resulting. All cases where a join condition is not valid that means ($C_1 \notin e_i \vee C_2 \notin e_j$) do not need to be considered furthermore. We only have to distinguish and define four different cases in the semantics for the `move` operation:

- ($A \in e_i, B \in e_j, C_1 \in e_i, C_2 \in e_j$),
- ($A \in e_i, B \notin e_j, C_1 \in e_i, C_2 \in e_j$),
- ($A \notin e_i, B \in e_j, C_1 \in e_i, C_2 \in e_j$),
- ($A \notin e_i, B \notin e_j, C_1 \in e_i, C_2 \in e_j$).

For the definition of all evolution operations in all heterogeneity classes, we again refer to [7].

Composition of Operations for Heterogeneous Data When multiple evolution operations need to be performed (e.g., legacy data that has to be updated over several versions), an optimisation shall be applied. An obvious method is the *composition of operations*. In Figure 1, a first example for a composition of an add and a subsequent rename operation was already given. There are many other possible compositions that can be performed if we have the data in **HC1**. Some examples are:

$$\begin{aligned}
\texttt{add}\ E.A + \texttt{rename}\ E.A \text{ to } B &\longrightarrow \texttt{add}\ E.B \\
\texttt{add}\ E.A + \texttt{delete}\ E.A &\longrightarrow \emptyset \\
\texttt{add}\ E.A + \texttt{rename}\ E.A \text{ to } B + \texttt{delete}\ E.B &\longrightarrow \emptyset \\
\texttt{rename}\ E.A \text{ to } B + \texttt{rename}\ E.B \text{ to } C &\longrightarrow \texttt{rename}\ E.A \text{ to } C \\
\texttt{move}\ E_1.A \text{ to } E_2 \text{ on } E_1.K = E_2.F + \texttt{rename}\ E_2.A \text{ to } B &\longrightarrow \texttt{move}\ E_1.A \text{ to } E_2.B \text{ on } E_1.K = E_2.F \\
\texttt{rename}\ E.A \text{ to } B + \texttt{delete}\ E.B &\longrightarrow \texttt{delete}\ E.A \\
\texttt{copy}\ E_1.A \text{ to } E_2 \text{ on } E_1.K = E_2.F + \texttt{delete}\ E_2.A &\longrightarrow \emptyset
\end{aligned}$$

Figure 4: Composition of evolution operations

We can distinguish three abstract classes for compositions: (i) adding and renaming can be composed to a modified add operation, (ii) several rename operations can be combined, and (iii) rename operations followed by a delete can be composed to a modified delete operation.

In case we can assume **HC1**, the *composed operations* (right-hand side in the equations of Figure 4) have the same effect as a *step-wise execution* (left-hand side). Data migration operations are generated for each evolution operation. Consequently, reducing the number of these evolution operations *significantly improves performance*. Let's assume a NoSQL database with 10000 entities and five evolution operations. In the step-wise execution, we have to apply the data migration operations on $5 \cdot 10000$ entities. If we combine the operations into two evolution operations, then we need to perform the corresponding data migration operations on only $2 \cdot 10000$ entities. In [9] we have shown that using an operation caching improves the run-time significantly. Therefore, a composition of evolution operations is recommended.

In [8], we have defined which compositions are possible for all combinations of evolution operations (add, delete, rename, move, and copy). This article was published some years before we defined the HCs for NoSQL databases. We have used *pre- and postconditions* instead. For example, for the composition of the operations add $E.A$ + rename $E.A$ to B (see Figure 1), we have set the preconditions $A \in E$ and $B \notin E$. Preconditions define the presence or absence of properties and guarantee certain dataset characteristics. so that the composition to add $E.B$ in the example can be done. Each composition can only be applied if all preconditions hold.

The postconditions define the conditions that hold after the execution of the evolution operation. When applying several evolution operations, these conditions define the status of the dataset after each operation execution. They are needed to determine how evolution operations can be combined. Otherwise, i.e., in heterogeneity classes other than **HC1**, no composition is possible and the evolution operations have to be applied step-wise instead.

If we do not apply preconditions in the composition, we have to assume **HC4**. Thus, all structural variants has to be defined in the composition. We want to explain this for the following evolution operations:

$$\begin{aligned}
&\texttt{move}\ E_1.A \text{ to } E_2 \text{ on } E_1.K = E_2.F + \texttt{rename}\ E_2.A \text{ to } B \\
&\longrightarrow \texttt{move}\ E_1.A \text{ to } E_2.B \text{ on } E_1.K = E_2.F.
\end{aligned}$$

In the prior paragraph, we have defined the `move` operation and the `rename` operation for **HC4**. For both operations, we distinguished four cases. If we compose both operations then we have to define $4 \cdot 4 = 16$ cases. We want to mention that besides the large effort to define all cases separately, a composition is not possible in all cases. This short example shows how high the effort for composition is in **HC4**.

Database Transformation and Query Rewriting Database heterogeneity has to be considered in all database tasks. Two further examples focus on the transformation of heterogeneous data into other formats and query rewriting.

Data transformation has to consider all variants of the data. We assume that the database has heterogeneous structures in the source. If we define a mapping between source and target format, the transformations must be defined for all variants. For instance, if we use the *local as view (LaV) approach* — originally developed in [18] for relational data —, this leads to incomparably larger transformation scripts. These scripts have to contain all variants of the input data.

If we assume data in different formats, we have to use *query rewriting*. This is necessary either in case of versioned databases or in data integration where we query against the global schema and translate the queries into the structure of the local schemas. For both tasks, the number of different cases that have to be defined depends on the heterogeneity class [6].

4. Databases Designs for Heterogeneous Data

In summary, as expected, all database tasks are more complex to perform for heterogeneous databases. Nevertheless, we cannot avoid heterogeneity; in countless applications, it is visible that not all data is structured in a regular way. And finally, the various developments of schemaless database management systems show this need. How can one now manage this balancing act between regular and irregular data? Our conclusion from the application examples listed in this article is very simple: we propose *hybrid methods* storing a maximum amount of data under schema control. For the rest of the data, we need the ability to store irregular data. Several applications following this hybrid approach have been presented in literature. They deal with the conjunction of relational databases with document stores [19] or graph databases [20].

These *hybrid approaches* can be implemented in three different ways (see Figure 5): (a) within one NoSQL or graph database with partial schema control, (b) as a set of connected NoSQL or graph databases (some with schema control and the others without), or (c) in a multi-model database combining one relational database and NoSQL databases.

JSON and Graph databases with partial schema control. NoSQL management systems like JSON and graph database systems added the opportunity to check schema constraints (visualized in Figure 5(a)). A prominent example is MongoDB where users can specify a JSON schema for a specific MongoDB collection [21]. With this specification, document validation against a predefined schema is possible [22].

Neo4J – as an example for a graph database – follows a similar approach, making it possible to generate a partial schema control by defining an individual schema. Cypher offers existence, type, and uniqueness constraints, which are available for properties of either nodes or relationships.

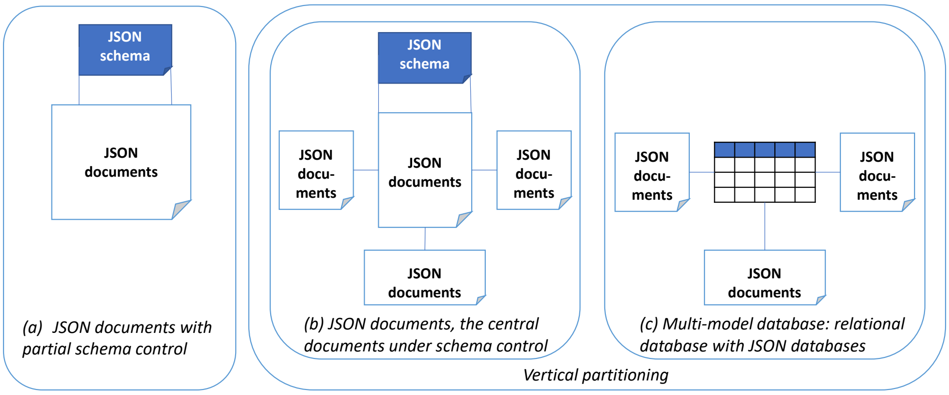
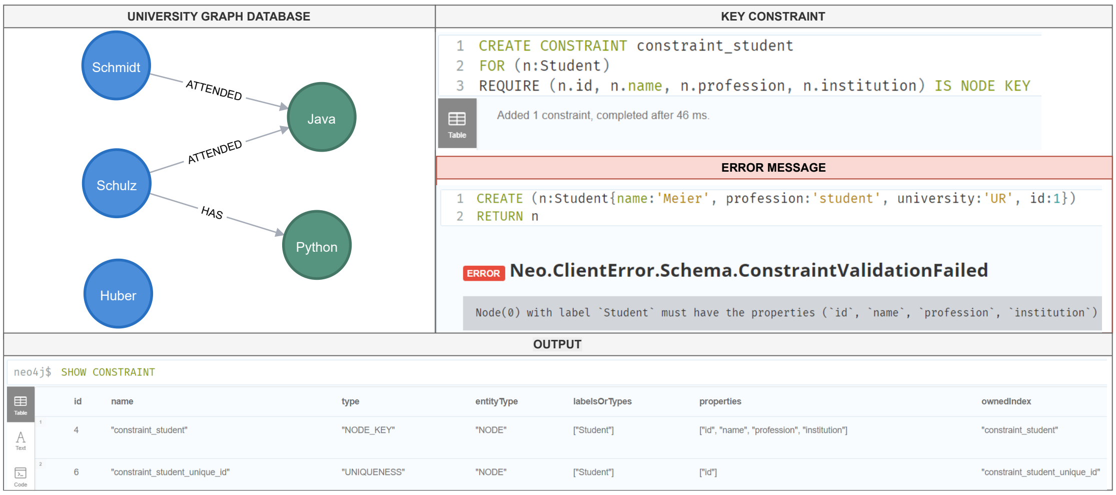

JSON schema
JSON documents
(a) JSON documents with partial schema control
JSON schema
JSON docu-ments
JSON documents
JSON docu-ments
JSON documents
(b) JSON documents, the central documents under schema control
JSON docu-ments
JSON docu-ments
JSON documents
(c) Multi-model database: relational database with JSON databases
Vertical partitioning

Figure 5: Overview on the three proposed hybrid storage methods

Furthermore, key constraints can be defined for whole entities such as nodes or relations [23]. The constraints are checked upon creation or manipulation of data.

Constraints can be used to avoid heterogeneity of data. In Figure 6 a Neo4J example is shown. Here, a key constraint `constraint_student` on the `Student` nodes with properties `id`, `name`, `profession` and `institution` is defined. The graph is based on the dataset of Figure 3. When violating the constraint, an error message is generated. In Neo4J the command `SHOW CONSTRAINTS` can be used to show all defined constraints.



```
CREATE CONSTRAINT constraint_student
FOR (n:Student)
REQUIRE (n.id, n.name, n.profession, n.institution) IS NODE KEY
```

Added 1 constraint, completed after 46 ms.

```
CREATE (n:Student{name:'Meier', profession:'student', university:'UR', id:1})
RETURN n
```

ERROR Neo.ClientError.Schema.ConstraintValidationFailed

```
Node(0) with label `Student` must have the properties (`id`, `name`, `profession`, `institution`)
```

```
neo4j$ SHOW CONSTRAINT
```

	id	name	type	entityType	labelsOrTypes	properties	ownedIndex
1	4	"constraint_student"	"NODE_KEY"	"NODE"	["Student"]	["id", "name", "profession", "institution"]	"constraint_student"
2	6	"constraint_student_unique_id"	"UNIQUENESS"	"NODE"	["Student"]	["id"]	"constraint_student_unique_id"

Figure 6: Constraint definition and checking for the university graph database (in Neo4J)

Connected JSON or Graph databases. In this storage method, we propose *vertical partitioning*, i.e., the splitting of attributes into groups and the distribution of these groups into different tables [24]. In our case, the regular and irregular parts of a database are stored separately in NoSQL (JSON or graph) databases. For the regular components, a NoSQL database with schema control is used. This can be seen as the *kernel* of the dataset. The other portions contain the heterogeneous parts of the datasets which could be imagined as satellites surrounding the kernel. The connection between the different databases is realized by references (see Figure 5(b)). The same approach can also be used for *graph databases*.

When looking at Figure 6, one example for this approach would be to save essential information of a `Student` node as displayed in the constraint called `constraint_student` under schema control matching the kernel. Additional optional information for each `Student` node (e.g., a photo or her credit points) would be stored separately in the graph database.

Multi-model database. The storage method visualized in Figure 5(c) is quite similar to the previous one. Instead of storing the regular parts in a NoSQL database under schema control (see Figure 5(b)), a relational database is used. This relational database has the built-in feature *schema-first* which guarantees schema constraints and offers the opportunity to define further semantic constraints. Similarly to the approach shown in Figure 5(b), the heterogeneous parts are kept in JSON or graph databases. The connections between the datasets are realized by references or other inter-model linkage constraints.

5. Conclusion and Future Work

In recent years, the need to store heterogeneous data has become increasingly apparent. This resulted in the development of different generations of database systems that allow the heterogeneity of data (XML, JSON and graph databases). However, we conclude that one should keep these heterogeneous parts as small as possible and store everything else relationally. With such an approach, we can achieve a balance in data storage between the desirability of regularly storing large chunks of homogeneous data and the need to store heterogeneous data externally.

NoSQL databases like wide-column stores (Cassandra [25]) and JSON databases (MongoDB [21]) allow a (partial) schema definition. Therefore, they can be used for hybrid approaches defined in Section 4. As graph databases such as Neo4J [23] allow the definition of semantic constraints, they could also make use of the suggested storage methods.

In the past, reverse engineering approaches have been developed to derive the structures [16, 26, 27, 28] and semantic constraints [29] from NoSQL data. There is still a need for such components. Schema management for graphs is also an active research topic [30]. Derivation of semantic constraints from graph data and multi-model data is part of our future work.

Another interesting research direction is the consideration of hybrid databases in schema optimization. The challenge is to distinguish between data that is better suited to be stored in relational databases and data you are better off storing in nonrelational databases. Here, different factors like storage size or query performance have to be considered.

Acknowledgments

The article is published in the scope of the project “NoSQL Schema Evolution und Big Data Migration at Scale” which is funded by the Deutsche Forschungsgemeinschaft(DFG) under the grant no. 385808805.

References

[1] E. F. Codd, Extending the Database Relational Model to Capture More Meaning, ACM Trans. Database Syst. 4 (1979) 397–434.

[2] M. Shah, A. Kothari, S. Patel, Influence of Schema Design in NoSQL Document Stores, in: Mobile Computing and Sustainable Informatics, Springer Singapore, 2022, pp. 435–452.

[3] A. Abubakar Imam, S. Basri, R. Ahmad, J. Watada, M. González-Aparicio, M. A. Almomani, Data Modeling Guidelines for NoSQL Document-Store Databases, International Journal of Advanced Computer Science and Applications 9 (2018).

[4] S. Amghar, S. Cherdal, S. Mouline, Data Integration and NoSQL Systems: A State of the Art, in: BDIoT, ACM, 2019, pp. 16:1–16:6.

[5] H. B. Hamadou, F. Ghozzi, A. Péninou, O. Teste, Towards Schema-independent Querying on Document Data Stores, in: DOLAP, volume 2062 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018.

[6] M. L. Möller, M. Klettke, A. Hillenbrand, U. Störl, Query Rewriting for Continuously Evolving NoSQL Databases, in: ER, volume 11788 of *LNCS*, Springer, 2019, pp. 213–221.

[7] M. L. Möller, M. Klettke, U. Störl, Keeping NoSQL Databases Up to Date - Semantics of Evolution Operations and their Impact on Data Quality, in: LWDA, volume 2454 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2019, pp. 25–37.

[8] M. Klettke, U. Störl, M. Shenavai, S. Scherzinger, NoSQL schema evolution and big data migration at scale, in: IEEE BigData, IEEE Computer Society, 2016, pp. 2764–2774.

[9] U. Störl, A. Tekleab, M. Klettke, S. Scherzinger, In for a Surprise When Migrating NoSQL Data, in: ICDE, IEEE Computer Society, 2018, p. 1662.

[10] C. Curino, H. J. Moon, C. Zaniolo, Graceful database schema evolution: the PRISM workbench, Proc. VLDB Endow. 1 (2008) 761–772.

[11] MongoDB, Inc., MongoDB documentation – $lookup (aggregation), https://www.mongodb.com/docs/manual/reference/operator/aggregation/lookup/, 2023. Accessed: 2023-07-31.

[12] K. Herrmann, H. Voigt, A. Behrend, J. Rausch, W. Lehner, Logical Data Independence in the 21st Century - Co-Existing Schema Versions with InVerDa, CoRR abs/1608.05564 (2016).

[13] M. Polák, M. Chytil, K. Jakubec, V. Kudelas, P. Piják, M. Necaský, I. Holubová, Data and Query Adaptation Using DaemonX, Comput. Informatics 34 (2015) 99–137.

[14] S. Scherzinger, M. Klettke, U. Störl, Managing Schema Evolution in NoSQL Data Stores, in: DBPL, 2013.

[15] A. H. Chillón, D. S. Ruiz, J. G. Molina, Towards a Taxonomy of Schema Changes for NoSQL Databases: The Orion Language, in: ER, volume 13011 of *LNCS*, Springer, 2021, pp. 176–185.

[16] P. Suárez-Otero, M. J. Mior, M. J. S. Cabal, J. Tuya, CoDEvo: Column family database evolution using model transformations, J. Syst. Softw. 203 (2023) 111743.

[17] P. Koupil, J. Bártík, I. Holubová, *MM-evocat:* A Tool for Modelling and Evolution Management of Multi-Model Data, in: CIKM, ACM, 2022, pp. 4892–4896.

[18] J. D. Ullman, Information integration using logical views, Theoretical Computer Science 239 (2000) 189–210.

[19] G. Ongo, G. P. Kusuma, Hybrid Database System of MySQL and MongoDB in Web

Application Development, in: ICIMTech, 2018, pp. 256–260.

[20] H. Vyawahare, P. Karde, V. Thakare, A Hybrid Database Approach Using Graph and Relational Database, in: RICE, 2018, pp. 1–4.

[21] MongoDB, Inc., MongoDB documentation – Specify JSON Schema Validation, https://www.mongodb.com/docs/manual/core/schema-validation/specify-json-schema/, 2023. Accessed: 2023-07-13.

[22] F. Pezoa, J. L. Reutter, F. Suárez, M. Ugarte, D. Vrgoc, Foundations of JSON Schema, in: WWW, ACM, 2016, pp. 263–273.

[23] Neo4J, Inc., Neo4J Docs – Constraints, https://neo4j.com/docs/cypher-manual/current/constraints/, 2023. Accessed: 2023-07-24.

[24] S. B. Navathe, S. Ceri, G. Wiederhold, J. Dou, Vertical Partitioning Algorithms for Database Design, ACM Trans. Database Syst. 9 (1984) 680–710.

[25] Apache Cassandra, Cassandra Docs – Defining Database Schema, https://cassandra.apache.org/doc/latest/cassandra/data_modeling/data_modeling_schema.html, 2023. Accessed: 2023-07-28.

[26] D. S. Ruiz, S. F. Morales, J. G. Molina, Inferring Versioned Schemas from NoSQL Databases and Its Applications, in: ER, volume 9381 of *LNCS*, Springer, 2015, pp. 467–480.

[27] M. Klettke, H. Awolin, U. Störl, D. Müller, S. Scherzinger, Uncovering the evolution history of data lakes, in: IEEE BigData, IEEE Computer Society, 2017, pp. 2462–2471.

[28] P. Koupil, S. Hricko, I. Holubová, A universal approach for multi-model schema inference, J. Big Data 9 (2022) 97.

[29] S. Klessinger, M. Klettke, U. Störl, S. Scherzinger, Extracting JSON Schemas with tagged unions, in: DEco@VLDB, volume 3306 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 27–40.

[30] A. Bonifati, The Quest for Schemas in Graph Databases (keynote), in: DOLAP, volume 3369 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023, pp. 1–2.