Evolving Gracefully: Building Robust and Self-Adaptive Data Cleaning Pipelines for Schema Evolution and Uncertainty

Kevin M. Kramer
FernUniversität in Hagen
Hagen, Germany
kevin.kramer@fernuni-hagen.de

Valerie Restat
FernUniversität in Hagen
Hagen, Germany
valerie.restat@fernuni-hagen.de

Uta Störl
FernUniversität in Hagen
Hagen, Germany
uta.stoerl@fernuni-hagen.de

ABSTRACT

The lifecycle of data cleaning pipelines is accompanied by diverse forms of data and software evolution. Oftentimes, these changes are introduced upstream without communicating them to downstream data consumers which creates uncertainty. Evolution and uncertainty lead to substantial human involvement and thereby, high maintenance costs for long-running data cleaning pipelines. A significant factor contributing to this situation is the robustness of operators, *i.e.*, if and how operators are affected by certain types of change and which consequences this might entail for the whole pipeline. In the present work we investigate and define the robustness of data cleaning operators towards schema evolution. To this end, we categorize data cleaning operations based on how they interact with the data on a structural level. Given these categories and the different cases of structural change, a decision tree is created which enables a systematic understanding of robustness for data cleaning pipelines towards schema evolution. Based on these theoretical findings, we present concepts and techniques that work towards a vision of self-adaptive data cleaning pipelines.

VLDB Workshop Reference Format:
Kevin M. Kramer, Valerie Restat, and Uta Störl. Evolving Gracefully: Building Robust and Self-Adaptive Data Cleaning Pipelines for Schema Evolution and Uncertainty. VLDB 2025 Workshop: 14th International Workshop on Quality in Databases (QDB'25).

1 INTRODUCTION

Imagine the following scenario: A big retail company utilizes an end-to-end data processing pipeline to generate reports, create dashboards and run analysis. Each day one data batch is processed, encompassing the retail data from that day. This pipeline is split into different segments, *e.g.* data integration, data cleaning, analysis. Each segment is maintained by one or many different teams, with little to no communication between them. Now imagine, you are part of the data cleaning team. Your goal is to produce high quality data for downstream data consumers. Initially, you created a data cleaning pipeline for that purpose consisting of operators for dealing with missing values, for handling outliers and so on. You created that pipeline based on the initial schema, thereby mapping your implementation to the data. One day you get notified – either by a monitoring system or some very angry colleague waiting for your data downstream – that your pipeline has crashed. You check the problem and realize that it happened due to upstream schema changes, *i.e.*, schema evolution. The mapping you created no longer works and it is your job to fix it. Obviously, you need to fix the problem right away, because everyone downstream is depending and waiting on you. But before you can start, you need to reschedule all activities and appointments you had planned for the day.

When talking to experts from industry the problems described in the scenario are said to be very common. They become especially apparent in the realm of data cleaning, since data cleaning pipelines are an integral part of end-to-end data processing systems. The goal of these pipelines is to transform erroneous upstream data into high-quality data which can be used downstream for gaining knowledge through analysis [1, 5, 27, 36]. Even though this step is essential, oftentimes these pipelines are only considered to be a means to an end before the actual work can start. This view stands in contrast to the fact that data cleaning makes up 60% of the work of a data scientist[1].

One major problem is missing communication between the teams of different pipeline segments. Upstream changes are not known to downstream teams until they arrive which is too late. This circumstance produces *uncertainty* about when, what and to which extent change might occur. This situation can be interpreted as having little to no *socio-technical congruency* [4, 24]. To tackle this challenge some companies utilize data governance, for example in the form of data contracts [22].

Uncertain changes have the capacity to produce errors which might result in runtime failure or introduce the potential for erroneous results. A very common type of change is *schema evolution* [2, 31, 34]. Dealing with schema changes during the lifecycle of a data cleaning pipeline is usually done manually as described in the introductory scenario. This entails multiple issues which contribute to high maintenance cost for these types of pipelines. Ideally, these pipelines would be set up in a way, which makes them functionally robust towards schema evolution while still ensuring the highest possible data quality. In some cases adaptation of the pipeline becomes necessary, to achieve these goals and enable the pipeline to evolve gracefully. Ideally, this would happen autonomously through self-adaptation.

The contributions of this paper are thus:

- The definition of robustness of data cleaning operators towards schema evolution.
- A categorization of data cleaning operators, based on how they interact with the schema.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

[1] CrowdFlower Data Science Report (2016) https://visit.figure-eight.com/rs/416-ZBE-142/images/CrowdFlower_DataScienceReport_2016.pdf

- The creation of a decision tree that reveals which combinations of schema evolution and operators are robust and when self-adaptation is required.
- The vision of self-adaptive data cleaning pipelines that can deal with schema changes automatically.

The remainder of this paper is structured as follows. Section 2 presents related work. In Section 3 we introduce a running example which is used throughout the rest of the paper. We present our definition of operator robustness w.r.t. schema evolution in Section 4. We also propose a categorization for data cleaning operators in this section. Section 5 includes a conceptual workflow and a decision tree for dealing with schema evolution. Building on these findings, we present concepts and techniques to achieve a vision of self-adaptive data cleaning pipelines in Section 6. Section 7 summarizes the findings and provides an outlook for future research possibilities.

2 RELATED WORK

Data cleaning is essential to ensure data quality and thus also the quality of analyses and predictions following it. For this reason, data cleaning is a widely researched topic in academia and practice and a variety of different data cleaning methods exist. A description of the complete data cleaning process and various methods can be found in [13]. Many different tools exist for cleaning data. The combination of Raha and Baran, for example, provides an end-to-end data cleaning pipeline [20]. Another example is the combination of HoloDetect [11] and HoloClean [25]. An overview of further tools is provided in [10]. However, the concepts described in the present paper are generally applicable to data cleaning pipelines, regardless of the specific tool.

Schema evolution is very common given long-running software systems [31, 34]. The change between two schemas can be expressed by means of *schema modification operations* (SMO). The most basic SMOs are `ADD`, `DELETE` and `RENAME` which add a new property or delete or rename an existing one respectively [7, 12, 30]. Several approaches exist for schema and software co-evolution [32, 33]. These systems focus on deliberate schema evolution, *i.e.*, the changes to the schema are performed by the user under full certainty. This stands in contrast to our use-case, where uncertainty plays a vital role. Even though these systems propose valuable information on how adaptation of software can be achieved in parallel to schema evolution, the human-in-the-loop stands in contrast to our vision of autonomous data cleaning pipelines.

Self-adaptive systems (SAS) have a long history of research. In their paper Kephart et al. defined the *MAPE-K loop* which stands for monitor, analyze, plan, execute, knowledge [14]. Even though additions to this model have been proposed [23], it still provides a strong theoretical basis for SAS. These systems extend a normal software system by adaptation logic. This logic allows specific system functionality to autonomously react to changing circumstances. Control-theoretical software adaptation is one approach for self-adaptivity, where functionality is provided by two components. *Sensors* which can be mapped to the monitor and analyze parts of the MAPE-K model, notice and interpret change. *Actuators* propagate the adaptation and adjust the system. They represent the plan and execute steps of the MAPE-K model. Both components utilize knowledge of the system and its circumstances [9]. Different types of adaptation mechanisms exist, providing various granularities for readjusting a software system [35]. In the present work we focus on the analysis (Section 5), plan and execute (Section 6) steps of the MAPE-K loop.

To our knowledge, self-adaptive data cleaning pipelines have not been the subject of research besides a set of conceptual requirements which we proposed in [18].

3 RUNNING EXAMPLE

In this section we present a running example. Similar to the introductory example, we employ a data preparation pipeline that performs batch processing of data on a daily basis. Figure 1 provides an overview of the system. Given a data cleaning pipeline consisting of several operators, *i.e.*, algorithms that perform specific cleaning tasks and a set of consecutive data batches, we account for schema evolution that may occur between batches in the form of SMOs. It is uncertain if and to what extent these changes might occur between batches. In a first step, this paper focuses on the three basic SMOs: `ADD`, `DELETE` and `RENAME`. Moreover, inferring the correct set of SMOs is not the focus of the present work. This task falls into the realm of schema extraction / inference [3, 6, 15] and schema versioning [2, 16].

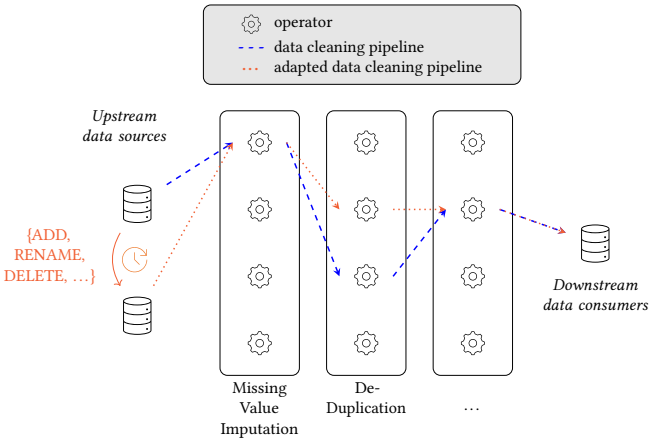

Figure 1: Overview – data batches are cleaned in a pipeline consisting of several operators. Between data batches schema evolution occurs, resulting in a set of SMOs. If operators are not robust against these changes, this can lead to failures or suboptimal data quality.

For a better understanding, the data used in the running example in the rest of the paper is introduced below. The use case consists of sensor data that measures air pollution and temperature. This data includes errors such as missing values, outliers, functional dependencies and others for which data cleaning is required. Figure 2 shows an example excerpt of the data. This contains, for example, outliers in `Temperature` (*123*) and missing values in `Pollution` (*null*). Other possible errors would be the violation of the functional dependency between `Sensor-ID` and `Location` or duplicates. To repair these errors, data cleaning operators are required, such as an operator for the *Missing Value Imputation* of `Pollution`.

```
{
  {
  "Sensor-ID": "936A92",
  "Time": "25-07-02 08:00",
  "Location": "countryside",
  "Pollution": 32.3,
  "Temperature": 123
  },
  {
  "Sensor-ID": "936A92",
  "Time": "25-07-02 08:30",
  "Location": "countryside",
  "Pollution": null,
  "Temperature": 24
  },
  {
  "Sensor-ID": "A92639",
  "Time": "25-07-02 08:08",
  "Location": "city",
  "Pollution": 83.8,
  "Temperature": 29
  },
  ...
}
```


Figure 2: Excerpt of sensor data. Missing values can be seen in `Pollution` and outliers in `Temperature`.

These operators are combined into a data cleaning pipeline, as shown in Figure 1. The creation of this pipeline is not part of the paper. This is still an open research question for which various initial approaches exist [19, 21, 36]. In [29], we propose an approach called FONDUE in which the search space of possible operators is first reduced as far as possible by rule-based optimization and the integration of best practices. The most optimal pipeline is then determined with the help of cost-based optimization. The quality of the pipeline is determined based on the resulting data quality, determined by a tool such as CheDDaR [8].

In this paper, we assume that – as described in the introductory example – we have already created the pipeline with all the required operators, based on the initial schema. Now new batches arrive every day and run through this pipeline. In this process, changes in the shape of the SMOs can occur between the individual batches.

In Figure 3, we consider two batches: batch 1 and batch 2, which arrive on consecutive days. An example is shown for each of the three SMOs `ADD` (Figure 3a), `RENAME` (Figure 3b) and `DELETE` (Figure 3c). For the purpose of clarity, only one SMO is shown at a time. In practice, several SMOs can of course occur between two batches.

4 ROBUSTNESS TOWARDS SCHEMA EVOLUTION

In this section we develop our concept of robustness of data cleaning operators and pipelines towards schema evolution. First, we provide some terminology and intuition. Second, we present our definition of operator robustness. Third, we present a categorization of data cleaning operators w.r.t. their interaction with the schema, which significantly moderates their robustness. Finally, we explain how *contextualization* plays a vital role and can be used not only to increase robustness, but also data quality.

batch 1


```
{
  {
  "Sensor-ID": "936A92",
  "Time": "25-07-03 08:00",
  "Location": "countryside",
  "Pollution": 32.3,
  "Temperature": 123
  },
  ...
}
```


batch 2


```
{
  {
  "Sensor-ID": "BF9827",
  "Time": "25-07-04 08:00",
  "Location": "city",
  "Pollution": 81.4,
  "Temperature": -26,
  "Manufacturer": "M1"
  },
  ...
}
```


(a) `ADD Manufacturer`

batch 1


```
{
  {
  "Sensor-ID": "936A92",
  "Time": "25-07-02 08:00",
  "Location": "countryside",
  "Pollution": 32.3,
  "Temperature": 123
  },
  ...
}
```


batch 2


```
{
  {
  "ID": "BF9827",
  "Time": "25-07-03 08:00",
  "Location": "city",
  "Pollution": 81.4,
  "Temperature": -26
  },
  ...
}
```


(b) `RENAME Sensor-ID TO ID`

batch 1


```
{
  {
  "Sensor-ID": "936A92",
  "Time": "25-07-02 08:00",
  "Location": "countryside",
  "Pollution": 32.3,
  "Temperature": 123
  },
  ...
}
```


batch 2


```
{
  {
  "Sensor-ID": "BF9827",
  "Time": "25-07-03 08:00",
  "Pollution": 81.4,
  "Temperature": -26
  },
  ...
}
```


(c) `DELETE Location`

Figure 3: Examples for each of the SMOs `ADD`, `RENAME`, `DELETE` between batch 1 (left) and batch 2 (right).

Terminology:

- **Data Cleaning Operator**: As described in the running example, an operator is an algorithm that performs a specific cleaning task, like *Missing Value Imputation* of `Pollution`. Operators work on different structural levels.
- **Structural Level**: A dataset consists of different structural levels. These levels encompass *single value*, *property*, *record*, and *kind*. We use the unified terms for structural elements as described in [17], since we do not limit our work to relational data.
- **Schema**: The specific structure of a dataset is described by the schema.

As stated in the introductory example, change over time in the form of schema evolution is certain. Depending on the data cleaning

pipeline the impact of such change might be a crash, incorrect results or suboptimal data quality. Accordingly, robustness can be functional, *e.g.* the pipeline does not crash and continues to process data, but it also entails the data quality dimension. This is sensible, since producing *high quality data* is the primary goal of a data cleaning pipeline. Another goal which is especially true for pipelines with a long lifecycle under uncertainty, is *minimizing human involvement*. This non-functional requirement is particularly important to reduce the running cost of such pipelines. These two goals are strongly interconnected in the provided use case. When examining a system as shown in Figure 1, the potential for failing these goals becomes apparent. A pipeline which produces the best possible data quality, but is also fragile towards change stemming from its complexity, will crash more often under schema evolution. In contrast, a general purpose data cleaning pipeline, implemented with flexibility towards change, will never achieve optimal data quality. We derive the following definition:

> ***Operator Robustness towards Schema Evolution***
> If a data cleaning operator remains functional and delivers the highest possible data quality, even after the data it works on have been affected by schema evolution, we call this operator robust towards schema evolution.

The first step to achieving schema robust operators and pipelines, is to investigate the ways in which operators interact with the schema. In the following, we describe common data cleaning operators and the structural levels on which they work. In the present work, we focus on the levels property and record, since most common data cleaning operators work on these structures. In order to be able to differentiate more precisely in the context of robustness, we further divide the property-based operators into the following *structural categories*: *Single Property (SP)*, *Multiple Properties (MP)*, and *Property plus Context (PC)*. Record-based operators do not require any further distinctions and thus represent a single category. Figure 4 shows an overview of this classification. The operators examined and the structural categories on which they work are presented in Table 1 – a more detailed explanation follows below. Even though the list is clearly not complete, it represents a well-defined basis for non-domain-specific data cleaning tasks [26]. It becomes apparent that property-based operators are the most common. Accordingly, these types of operators should be considered first when developing solutions for the given problem.

Single Property (SP): Operators which only and exclusively work on a single property and which cannot utilize additional contextual information in order to enhance the quality of their repairing task. Only the *Wrong Datatype Handling* operator from Table 1 falls under this category. To the best of our knowledge, there is no sensible way of increasing data quality through the use of more context.

Multiple Properties (MP): This category represents operators which need more than one property for their basic functionality. An operator for handling functional dependencies is part of this category. A functional dependency is always defined for at least two properties, *i.e.*, if one of these properties is deleted by means of

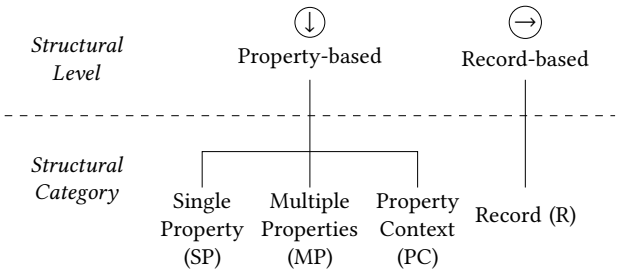

Figure 4: Relation between *Structural Level* and *Structural Category*

Table 1: Data Cleaning Operators and the Structural Categories they work on
↓ = Property-based , → = Record-based
(SP = Single Property, MP = Multiple Properties, PC = Property plus Context, R = Record)

Operator	Structural Category
Missing Value Imputation	↓ PC
Functional Dependency Restoration	↓ MP
De-Duplication	→ R
Outlier Handling	↓ PC
Set Violation Handling	↓ PC
Interval Violation Handling	↓ PC
Wrong Datatype Handling	↓ SP
Uniqueness Violation Handling	↓ PC

schema evolution, the operation becomes pointless, since the constraint expressed through the functional dependency is gone. These operators do not benefit from additional contextual properties[2].

Property plus Context (PC): Operators falling under this category generally work with a single property, but they can benefit from context provided by additional properties in order to increase the data cleaning quality. We term these operators to be *context-sensitive*. Consequently, these operators are especially important when considering robustness and data quality at the same time. We provide more detail about context-sensitive operators at the end of this section.

Record (R): Operators working record-based work orthogonal to property-based operators. They slice through the dataset horizontally instead of vertically and are therefore not affected by the schema or schema evolution in the same way as property-based operators. In our table, only de-duplication belongs to this category.

The structural categories provide a perspective on how operator-data-interaction is mediated through the schema. Looking at robustness towards schema evolution from the perspective of data quality, context-sensitive operators become especially important. This makes *contextualization* through additional properties an significant building block for our work. Contextualization is the mechanism affecting operators belonging to the *property plus context* category. This concept is especially important because it bridges the gap

[2]Not to be confused with additional properties resulting in n-ary functional dependencies.

between purely functional robustness and data quality. Consider the data from the running example, as seen in Figure 2: `Pollution` values obviously differ a lot by `Location` (*e.g. countryside* vs. *city*). Now let's take an operator for missing value imputation of the property `Pollution` (*property to be cleaned*). This can be done using only the mean value of `Pollution`. However, since the values differ so greatly by location, the operator also uses the `Location` property (as a *context property*) to insert the mean value grouped by location. Accordingly, data quality can be increased if contextual information is provided by other properties, for example as shown in the form of grouping. Another example would be contextual outliers which might only be detectable and thus repairable if context from another property is considered. Take for instance the temperature in the running example. Given the whole property `Temperature`, an outlier might be defined as something exceeding the normal range of temperatures. Contextualizing this operation with the month from the `Time` property and using the resulting groups' ranges to define month-based outliers, leads to significantly better data quality.

Contextualization
Contextualization is the use of additional context-properties in the cleaning process of operators from the PC category to achieve higher data quality.

Given the definitions of robustness, the categories of data cleaning operators and how they interact with the schema and the definition of contextualization, we can now investigate a workflow to achieve robust data cleaning pipelines.

5 CONCEPTUAL ROBUSTNESS WORKFLOW

In this section we present a solution to achieve robust data cleaning pipelines. In order to accomplish this, we develop a conceptual workflow and a decision tree. Our solution is presented step-by-step in the following.

The first step is to deal with uncertainty. Even though uncertainty about change is present, we can identify different cases and categorize them. In this way, we can reduce a potentially very large margin for change. In Step `I` of Figure 5 we can see the reduction from the infinite space of all possible SMOs to a specific set, that affected the current data batch. As stated earlier, generating this set is not within the scope of this paper, still, we want to provide some intuition for this step: In [16], an approach is presented that generates schema version graphs and derives SMOs based on them. However, this can lead to ambiguities. In our running example in Figure 3b, it is not clear whether we are dealing with a `RENAME Sensor-ID TO ID` operation or a sequence of a DELETE and an ADD operation (`DELETE Sensor-ID`, `ADD ID`). These ambiguities must either be resolved by the user [37] or by analyzing the data [16]. One challenge in our batch processing scenario is that the old data is no longer available. Suitable approaches such as using data profiles or similar must be used here. This will be the focus of our future research. In addition, only one change (one SMO) per property per schema version is considered in [16]. The generalization to multiple SMOs per property is also an open research question.

Given the set of SMOs we form the set of all permutations of tuples, consisting of one of each operator of the data cleaning pipeline and one of each SMO in Step `II` of the workflow. This set of tuples serves as input for a decision tree in Step `III`. Figure 6 depicts the tree. The leaves provide decisions what to do with a specific operator-SMO-combination. An operator is either robust towards a distinct schema modification, or it needs to be adapted. This adaptation ensures not only functionality, but also the highest possible data quality. Each tuple is processed until the set of tuples is empty. In the `ADD`-case it might be necessary to add new operators to the pipeline in order to clean newly added, erroneous properties. This is outside the scope of the present work and an open research question we plan to tackle in the future. This topic is strongly tied to generating pipelines for whole datasets, as was presented in Section 3.

In the following, the different cases of the decision tree are explained in detail w.r.t. robustness and adaptation.

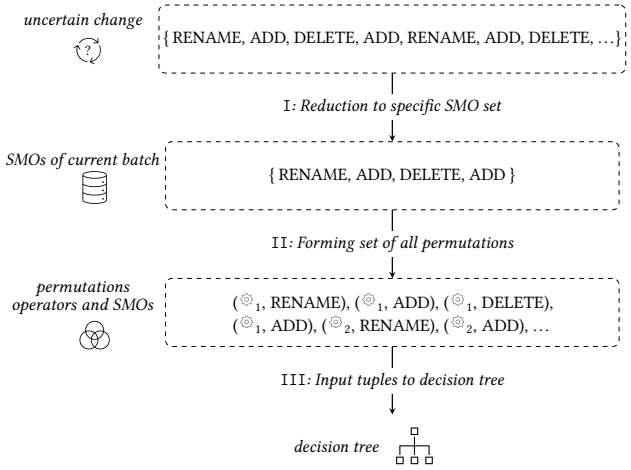

Figure 5: Conceptual workflow – uncertain change in the form of all possible SMOs affecting the schema are reduced to the set of actual SMOs which were applied to the schema between batches (I). All combinations of operators and SMOs are formed (II) and serve as input for the decision tree (III).

The first case of robustness is achieved when the operator is not property-based ①. For instance, as presented in Table 1 an operator for detecting duplicates works record-wise and will always be schema robust, because it works horizontally. The next node splits the decision tree in half, based on whether the operator is indifferent, *i.e.*, not directly affected by the SMO in the tuple. We first look at the lower left half of the tree. If the operator is indifferent, and the SMO is not `ADD`, then the operator is robust ②. If the SMO is indeed `ADD`, but the operator is not context-sensitive, then it is robust again ③. If it is context-sensitive, but the newly added property cannot be used as additional context, the operator is robust ④. In the case that it can benefit from additional contextualization in order to increase data quality, adaptation needs to happen in order to achieve robustness ⑤.

At this point, we present examples for the cases thus far, in order to ease comprehension.

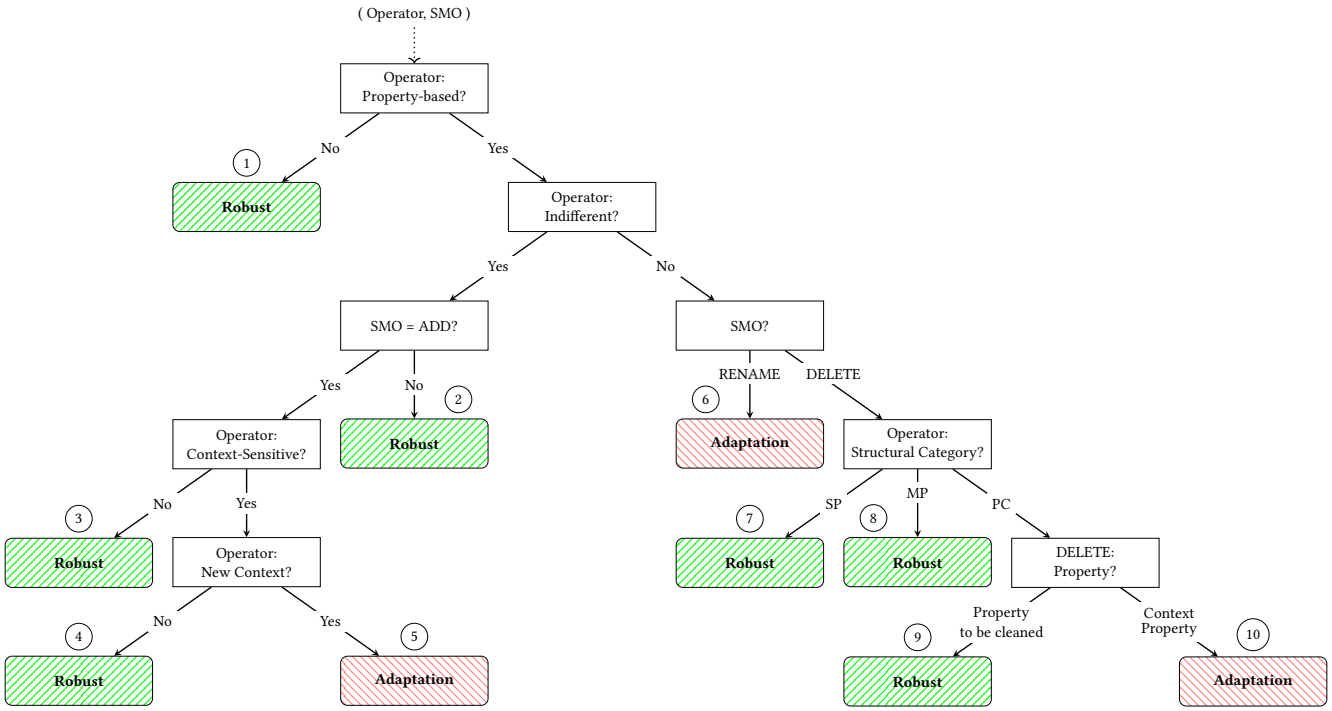

Figure 6: Decision tree – encapsulating the different cases of schema robustness. For each input consisting of an operator SMO tuple, a decision is made as to whether the operator is robust to the SMO (green, north east lines) or whether an adaptation is required (red, north west lines).

① De-duplication operator working record-based.
② Outlier handling operator working on property `Temperature` is not affected by the SMO `RENAME Sensor-ID TO ID` (see Figure 3b).
③ Operator for handling a wrong datatype, *i.e.*, cannot utilize additional context.
④ While an operator for missing value imputation of property `Pollution` is context-sensitive, the added property `Manufacturer` (see Figure 3a) is not suitable.
⑤ Building on example ④, this time the new property is `Location`, which can be used to increase the data quality achieved by the missing value imputation operator.

We now look at the lower right part of the decision tree, *i.e.*, the operator is affected by the SMO. If the SMO is `RENAME` an adaptation needs to be performed ⑥. If the SMO is `DELETE` and the operator is either single ⑦ or multi ⑧ property based, then it is robust. If it is context-sensitive, robustness depends on whether the property under cleaning or the property providing additional context is deleted. The former results in a robust operator ⑨, the latter needs adaptation, to deal with the new circumstances ⑩.

We now present examples for the remaining cases.

⑥ Any Operator working on property `Sensor-ID` is affected by the SMO `RENAME Sensor-ID TO ID`, as can be seen in Figure 3b.
⑦ An operator for handling a wrong datatype simply becomes inactive when the property under cleaning is deleted.
⑧ A functional dependency between two properties becomes obsolete when one of them is deleted, resulting in a robust operator. Take for instance the functional dependency `Sensor-ID` → `Location`, as shown in Figure 2. If either one of them is deleted, an MP-type operator such as one dealing with functional dependencies is robust.
⑨ A context-sensitive operator for imputing missing values in property `Pollution` uses additional context provided by `Location`. If `Pollution` (property to be cleaned) is deleted, the operator is robust, because it becomes obsolete. This example is shown in Figure 7.
⑩ Building on example ⑨, this time `Location` (context property) is deleted, which was used to calculate a group mean used for imputation. The operator needs to adapt, *e.g.* make use of a fallback in order to deal with the removed context property. Again, this example is presented in Figure 7.

We have shown in which cases of schema evolution specific types of data cleaning operators are robust or not. It has become clear that for some operators (especially context-sensitive ones) the adaptation ranges from mere functionality of the pipeline (*e.g.* fallback to the mean of `Pollution` when `Location` is deleted) to ensuring data quality (*e.g.* by adapting the operator with another context property). This emphasizes our definition of robustness in Section 4. Consequently, we deduce that adaptation is a spectrum ranging from assuring functionality to satisfying the highest possible data quality. At the functionality end, the spectrum covers concepts and techniques that ensure pipelines and operators

continue to work under a changing schema. At the data quality end, it includes self-adaptation capabilities that enable pipelines to readjust themselves to maintain the highest possible data quality. The following section presents practical examples from across this adaptation spectrum.

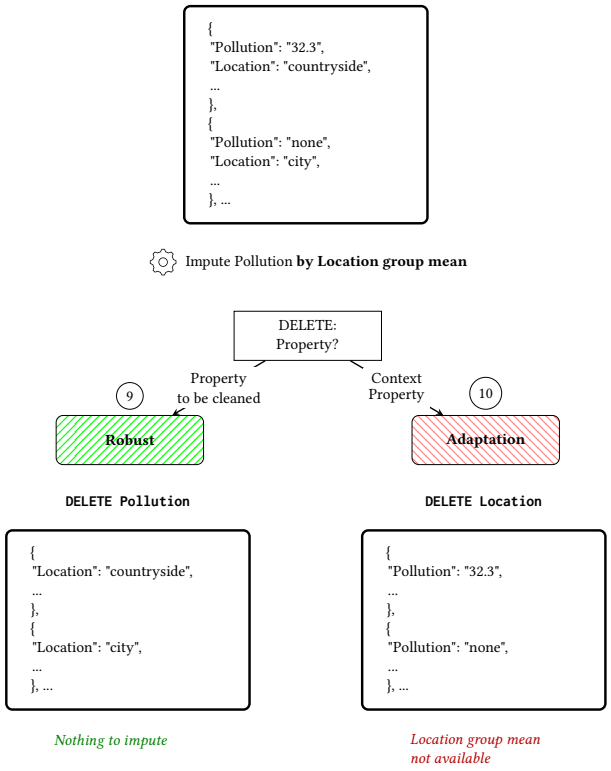

Figure 7: An operator imputes missing values for property `Pollution`. Context is provided by the property `Location` to increase data quality. Depending on whether `Pollution` or `Location` is deleted, the operator is either robust or needs to adapt.

6 VISION OF SELF-ADAPTIVE DATA CLEANING

In this section we take a look at the adaptation spectrum and its practical characteristics. The spectrum is shown in Figure 8. As with all self-adaptive systems following the MAPE-K framework [23], we first need information about if and what type of change happened. In our case, this information is about schema evolution, as represented by the set of SMOs. As stated earlier, gaining this information is not within the scope of the present work. The next steps are to analyze the changes and develop a plan. Our workflow and the decision tree help with analysis, but a plan for achieving the specified adaptation goal, in our case functionality and data quality, still needs to be produced. Lastly, the plan needs to be executed. During all phases, the system has access to knowledge, *e.g.* different older versions of the schema and their associated pipelines, metadata in the form of data profiles or operator configurations. We developed a preliminary solution based on a graph representation, implemented in Neo4j[3] in order to store and access this knowledge. In the following, we focus on the planning and execution phases.

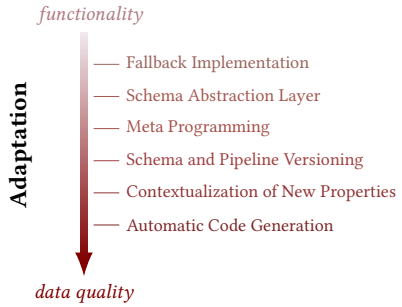

Figure 8: Adaptation spectrum – ranging from *functionality* to *data quality*

It is generally true, that the adaptation solutions presented here become increasingly complex the more they try to exceed the functionality end and try to assure or even increase data quality. Going from top to bottom on the spectrum, we present different techniques and provide examples to illustrate our ideas.

Fallback Implementation: A data cleaning operator, highly customized for a specific schema will fail more easily under evolution. Therefore, it is advisable to implement fallback alternatives to make the operator more self-adaptive.

Schema Abstraction Layer: Instead of hard-coding the schema information, *e.g.* property names into the operator source code, an abstraction layer can be used. If the schema changes, this layer can be changed, making the system more flexible and adaptable. This abstraction may also provide further functionality, such as aliases. For instance, if the name of a property changes to one which was used before and is known to the system, it is recognized as an alias and it can be normally accessed by all associated operators. Tools like Apache Avro[4] provide such functionality.

Meta Programming: Utilizing meta programming in the form of decorators or meta classes can be a valuable approach for adaptation. For example, operators can be marked with a decorator which enables them to validate arguments containing schema information against inferred schema evolution. Other forms are imaginable, *e.g.* constructing very flexible operator meta classes which can react to a specific set of changes. The main limitation of this approach is that this only works for known, predefined types of change. Our own preliminary findings support this claim.

Schema and Pipeline Versioning: Using a database for storing metadata, especially a versioned history of the schema [16] and the associated pipelines, can make the system more adaptable. For instance, if a new batch arrives with a different schema compared to the last one, such a database can be checked and if the schema is known, the corresponding pipeline can be reinstated, instead of adapting the current one. To achieve such functionality in an efficient manner, it is advisable to use an abstraction layer for pipeline definitions, as we have presented with ALPINE [28].

[3]https://www.neo4j.com/
[4]https://avro.apache.org/

Contextualization of New Properties: As stated in Section 4, a functioning operator may not produce the highest quality data. We presented a case in which an operator can benefit from additional contextual properties to increase its cleaning quality. This is the first type of adaptation which is truly dynamic. Investigating whether or not a newly added property can be used as context for an existing operator cannot be predetermined.

Automatic Code Generation: Pipelines can react to any type of change through new code. Techniques exist for applying or generating such code automatically. Code can be changed almost arbitrarily through semantic patches, as long as change can be represented by a pattern. An example of a system using semantic patching for schema evolution is DeBinelle [32]. Another example is the use of model-driven development in which the functionality of a system is abstracted into a model first. Second, specific mapping rules translate the model into code[5]. This split of the system into abstract model and specific implementation allows for easier adaptation, since the whole process can be done on the model level.

In this section we showed different approaches for adaptation which can be used to make data cleaning pipelines self-adaptive. Utilizing these concepts and techniques not only makes the pipelines more fail proof, but also ensures the highest possible data quality.

7 CONCLUSION AND OUTLOOK

In this paper we developed the concept of robustness of data cleaning operators and pipelines towards schema evolution. We presented a categorization of data cleaning operators w.r.t. their interaction with the schema. We created a decision tree which defines exact cases for data cleaning operators affected by schema evolution. This tree can be used during the analysis phase of the MAPE-K-loop. Furthermore, we presented a vision on how to achieve self-adaptive capabilities. Our contribution is the first of its kind and lays the foundation towards gracefully evolving, self-adaptive data cleaning pipelines. Our approach transforms uncertainty into clarity for the basic SMOs `ADD`, `DELETE` and `RENAME` while ensuring the highest possible data quality.

In future work, we plan to look at other SMOs including multi-type SMOs such as `COPY`, `MOVE`, and `SPLIT`. We also want to tackle the challenge of inferring the set of SMOs that occurred between batches. Furthermore, we want to investigate context-sensitive operators, since they have the potential to increase data quality through self-adaptation. We also plan to test and benchmark the various adaptation mechanisms presented in the last section.

REFERENCES

[1] Fatemeh Ahmadi et al. 2024. Accelerating the Data Cleaning Systems Raha and Baran through Task and Data Parallelism. In *VLDB Workshops*. VLDB.org.
[2] Zouhaier Brahmia et al. 2024. A Literature Review on Schema Evolution in Databases. *Computing Open* (2024).
[3] Zouhaier Brahmia et al. 2024. Schema Versioning in Databases: A Literature Review. *Computing Open* (2024).
[4] Marcelo Cataldo et al. 2008. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *ESEM*. ACM.
[5] Xu Chu et al. 2016. Data Cleaning: Overview and Emerging Challenges. In *SIGMOD Conference*. ACM.
[6] Pavel Contos et al. 2020. JSON Schema Inference Approaches. In *ER (Workshops)*. Springer.
[7] Carlo Curino et al. 2008. Graceful database schema evolution: the PRISM workbench. *Proc. VLDB Endow.* (2008).
[8] Indra Diestelkämper et al. 2025. CheDDaR: Checking Data - Data Quality Report. In *BTW*. GI.
[9] Antonio Filieri et al. 2015. Software Engineering Meets Control Theory. In *SEAMS@ICSE*. IEEE Computer Society.
[10] Mazhar Hameed et al. 2020. Data Preparation: A Survey of Commercial Tools. *SIGMOD Rec.* (2020).
[11] Alireza Heidari et al. 2019. HoloDetect: Few-Shot Learning for Error Detection. In *SIGMOD Conference*. ACM.
[12] Irena Holubová et al. 2019. Evolution Management of Multi-model Data - (Position Paper). In *Poly/DMAH@VLDB*. Springer.
[13] Ihab F. Ilyas et al. 2019. *Data Cleaning*. ACM.
[14] Jeffrey O. Kephart et al. 2003. The Vision of Autonomic Computing. *Computer* (2003).
[15] Meike Klettke et al. 2015. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *BTW*. GI.
[16] Meike Klettke et al. 2017. Uncovering the evolution history of data lakes. In *IEEE BigData*. IEEE Computer Society.
[17] Pavel Koupil et al. 2022. MM-infer: A Tool for Inference of Multi-Model Schemas. In *EDBT*. OpenProceedings.org.
[18] Kevin Kramer. 2023. Towards Evolution Capabilities in Data Pipelines. In *GvDB*. CEUR-WS.org.
[19] Sanjay Krishnan et al. 2019. AlphaClean: Automatic Generation of Data Cleaning Pipelines. *CoRR* (2019).
[20] Mohammad Mahdavi et al. 2021. Semi-Supervised Data Cleaning with Raha and Baran. In *CIDR*. www.cidrdb.org.
[21] Felix Neutatz et al. 2021. From Cleaning before ML to Cleaning for ML. *IEEE Data Eng. Bull.* (2021).
[22] Sarah Oppold et al. 2025. Data Contracts to Leverage (De-)centralized Data Management in Manufacturing Industries: An Experience Report. In *BTW*. GI.
[23] Ana Petrovska et al. 2023. A Theoretical Framework for Self-Adaptive Systems: Specifications, Formalisation, and Architectural Implications. In *SAC*. ACM.
[24] Binish Raza et al. 2021. Socio-Technical Congruence as an Emerging Concept in Software Development: A Scientometric Analysis and Critical Literature Review. *IEEE Access* (2021).
[25] Theodoros Rekatsinas et al. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* (2017).
[26] Valerie Restat et al. 2022. GouDa - Generation of universal Data Sets: Improving Analysis and Evaluation of Data Preparation Pipelines. In *DEEM@SIGMOD*. ACM.
[27] Valerie Restat et al. 2024. Towards an End-to-End Data Quality Optimizer. In *ICDEW*. IEEE.
[28] Valerie Restat et al. 2025. ALPINE: Abstract Language for Pipeline Integration and Execution. In *BTW*. GI, Bonn.
[29] Valerie Restat et al. 2025. FONDUE - Fine-Tuned Optimization: Nurturing Data Usability & Efficiency. *J. Big Data* (2025).
[30] Stefanie Scherzinger et al. 2013. Managing Schema Evolution in NoSQL Data Stores. In *DBPL*.
[31] Stefanie Scherzinger et al. 2020. An Empirical Study on the Design and Evolution of NoSQL Database Schemas. In *ER*. Springer.
[32] Stefanie Scherzinger et al. 2021. DeBinelle: Semantic Patches for Coupled Database-Application Evolution. In *ICDE*. IEEE.
[33] Robert E. Schuler et al. 2022. Managing Database-Application Co-Evolution in a Scientific Data Ecosystem. In *e-Science*. IEEE.
[34] Robert E. Schuler et al. 2023. Database Evolution, by Scientists, for Scientists: A Case Study. In *e-Science*. IEEE.
[35] Stepan Shevtsov et al. 2018. Control-Theoretical Software Adaptation: A Systematic Literature Review. *IEEE Trans. Software Eng.* (2018).
[36] Shafaq Siddiqi et al. 2023. SAGA: A Scalable Framework for Optimizing Data Cleaning Pipelines for Machine Learning Applications. *Proc. ACM Manag. Data* (2023).
[37] Uta Störl et al. 2022. Darwin: A Data Platform for Schema Evolution Management and Data Migration. In *EDBT/ICDT Workshops*. CEUR-WS.org.

[5] https://eclipse.dev/emf/